Per Bjesse and Anna Slobodová (Editors)

Proceedings of the 11th Conference on

Formal Methods in Computer Aided Design (FMCAD 2011)

In cooperation with
ACM Special Interest Group on Programming Languages
ACM Special Interest Group on Software Engineering

Technical co-sponsorship of IEEE Council on
Electronic Design Automation

AT&T Executive Education and Conference Center
Austin, Texas, USA
October 30 – November 2, 2011

TABLE OF CONTENTS **i**

Preface – Per Bjesse and Anna Slobodová **iii**

***Verifying Concurrent Programs* (Invited Tutorial) – Aarti Gupta** **1**

***Self-Timing: a Step Beyond Synchrony* (Invited Tutorial) – Ivan Sutherland** **2**

***IC3: Where Monolithic and Incremental Meet* (Invited Tutorial) – Fabio Somenzi and Aaron Bradley** **3**

***Planning for End-to-End Formal using Simulation-based Coverage* (Invited Tutorial) – Prashant Aggarwal, Darrow Chu, Vijay Kadamby and Vigyan Singhal** **9**

***Specification Based Testing with QuickCheck* (Invited Tutorial) – John Hughes** **17**

***Theorem Proving for Verification: The Early Days* (Keynote) – J Strother Moore** **18**

Interpolants from Z3 Proofs – Kenneth Mcmillan **19**

Effective Word-Level Interpolation for Software Verification – Alberto Griggio **28**

Accelerating MUS Extraction with Recursive Model Rotation (Short Paper) – Anton Belov and Joao Marques-Silva **37**

Pseudo-Boolean Solving by Incremental Translation to SAT (Short Paper) – Panagiotis Manolios and Vasilis Papavasileiou **41**

Automated Specification Analysis Using an Interactive Theorem Prover – Harsh Raju Chamarthi and Panagiotis Manolios **46**

Proving and Explaining the Unfeasibility of Message Sequence Charts for Hybrid Systems – Alessandro Cimatti, Sergio Mover and Stefano Tonetta **54**

Post-Silicon Fault Localisation Using Maximum Satisfiability and Backbones (Short Paper) – Charlie Shucheng Zhu, Georg Weissenbacher and Sharad Malik **63**

Algebraic Approach to Arithmetic Design Verification (Short Paper) – Mohamed Basith, Tariq Ahmad, Andre Rossi and Maciej Ciesielski **67**

Time-Bounded Analysis of Real-Time Systems – Sagar Chaki, Arie Gurfinkel and Ofer Strichman **72**

Timing Analysis of Interrupt-Driven Programs under Context Bounds – Jonathan Kotker, Dorsa Sadigh and Sanjit A. Seshia **81**

Automated Error Localization and Correction for Imperative Programs – Robert Koenighofer and Roderick Bloem **91**

Optimal Redundancy Removal without Fixedpoint Computation – Mike Case, Jason Baumgartner, Hari Mony and Bob Kanzelman **101**

Approximate Reachability With Combined Symbolic And Ternary Simulation – Michael Case, Jason Baumgartner, Hari Mony and Robert Kanzelman 109

Learning Conditional Abstractions – Bryan Brady, Randal Bryant and Sanjit A. Seshia 116

Efficient Implementation of Property Directed Reachability – Niklas Een, Alan Mishchenko and Robert Brayton 125

Incremental Formal Verification of Hardware – Hana Chockler, Alexander Ivrii, Arie Matsliah, Shiri Moran and Ziv Nevo 135

An Incremental Approach to Model Checking Progress Properties – Aaron Bradley, Fabio Somenzi, Zyad Hassan and Yan Zhang 144

Hardware Model Checking: Status, Challenges, and Opportunities (Discussion Panel) – Muralidhar Talupur 154

Realtime Regular Expressions for Analog and Mixed-Signal Assertions – John Havlicek and Scott Little 155

Formal Analysis of Fractional Order Systems in HOL – Umair Siddique and Osman Hasan 163

Static Scheduling of Latency Insensitive Designs with Lucy-n (Short Paper) – Louis Mandel, Florence Plateau and Marc Pouzet 171

A Theory of Abstraction for Arrays – Steven German 176

Parameterized Verification of Deadlock Freedom in Symmetric Cache Coherence Protocols – Brad Bingham, Jesse Bingham and Mark Greenstreet 186

Scaling Probabilistic Timing Verification of Hardware Using Abstractions in Design Source Code – Jayanand Asok Kumar, Lingyi Liu and Shobha Vasudevan 196

Pervasive Formal Verification in Control System Design (Discussion Panel) – Lee Pike 206

Hybrid Verification of a Hardware Modular Reduction Engine – Jun Sawada, Peter Sandon, Viresh Paruthi, Jason Baumgartner, Michael Case and Hari Mony 207

Desynchronization: Design For Verification – Sudarshan Srinivasan and Raj Katti 215

Hunting deadlocks efficiently in microarchitectural models of communication fabrics – Freek Verbeek and Julien Schmaltz 223

Author Index 232

Preface

The International Conference on Formal Methods in Computer-Aided Design, FMCAD, is a series of conferences on the theory and application of formal methods to the computer-aided design and verification of hardware and systems. The eleventh conference in the series, FMCAD 2011, was held in Austin, Texas, USA, October 30th – November 2nd.

In the past, FMCAD took place in the United States on even years and its sister conference CHARME was held in Europe on odd years. In 2006, these two conferences merged to form an annual conference with a unified international community. The merged conference inherited the name FMCAD, and is now held yearly. It provides a leading international forum for researchers and practitioners in academia and industry to present and discuss novel methods, technologies, theoretical results and tools for formal reasoning about computing systems.

This year, the conference received in-cooperation status with ACM under the Special Interest Group on Programming Languages and the Special Interest Group on Software Engineering. It also received technical sponsorship from the IEEE Council on Electronic Design Automation.

Three additional events were co-located with the conference this year: the ACL2 Workshop, the Design and Implementation of Formal Tools and Systems (DIFTS) Workshop, and the Hardware Model Checking Competition (HWMCC).

The FMCAD 2011 conference received 72 submissions. Each submission was reviewed by at least four reviewers, and some submissions received five or six reviews. After a long decision process that involved often vigorous discussions by Program Committee members and subreviewers, 26 submissions were eventually selected for presentation at the conference — 21 as regular papers and 5 as short papers. The accepted papers covered topics ranging from model checking and solver technology to design for verification. Moreover, they addressed a broad spectrum of abstraction levels ranging from analog and mixed-signal, and real-time systems to traditional synchronous hardware and C code.

Besides reviewed submissions, our program was enriched by five invited tutorial speakers. Aarti Gupta, a senior researcher at NEC, talked about “Verifying Concurrent Programs”. John Hughes, a Professor at Chalmers University of Technology and CEO of QuviQ, gave a talk about “Specification Based Testing with QuickCheck”. Vigyan Singhal, President and CEO of Oski Technology, presented “Planning for End-to-End Formal using Simulation-based Coverage”. Fabio Somenzi, Professor at University of Colorado Boulder, gave a presentation on a recently developed decision method for model checking in his talk “IC3: Where Monolithic and Incremental Meet”. Ivan Sutherland, the 1988 Turing Award winner, gave a talk titled “Self-Timing: a Step Beyond Synchrony”.

The Keynote speaker for the conference, J Moore, delved into the question of “The Role of Human Creativity in Mechanized Verification”. He argued that “by highlighting the minor decisions that represent major breakthroughs in the problem, we serve our science better because we identify the key problems yet to be solved”.

Two panel sessions contributed to the vibrancy of the conference. Lee Pike from Galois moderated “Pervasive Formal Verification in Control System Design” with panelists Darren Cofer (Rockwell Collins), Eric Feron (Georgia Institute of Technology), Tom Hawkins (Eaton Corp.), and Hakan Yazarel (Toyota). Murali Talupur of Intel Corp. was the moderator of the panel “Hardware Model Checking: Status, Challenges and Opportunities”. The panelists were Pranav Ashar (RealIntent), Jason Baumgartner (IBM), Bob Brayton (UC Berkeley), and Erik Seligman (Intel).

The 2011 Proceedings of FMCAD are available through the ACM Digital Library, at IEEE Xplore Digital Library, or as a free download from the FMCAD website.

We would like to sincerely thank our industrial sponsors for their financial support of FMCAD 2011: Centaur Technology, IBM Corp., Intel Corp., Jasper Design Automation, and NEC Laboratories America, Inc. We would also like to acknowledge the continuous support of FMCAD Inc.

We owe a large debt to this year's organizing committee, composed of David Rager (Local Arrangements), Barbara Jobstmann (Tutorials), and Viktor Kuncak (Publication). We would also like to thank the members of the FMCAD Steering Committee — Jason Baumgartner, Aarti Gupta, Warren Hunt, Panagiotis Manolios, and Mary Sheeran — for their kind advice during the conference preparation process. A big thanks goes to all members of the Program Committee who, with the help of many subreviewers, did a stellar job not only of selecting this year's exciting program, but also of providing feedback to the authors to help them improve their papers for publication. Last, but not least, the conference would not be possible without all the authors who submitted papers and all the attendees.

Per Bjesse and Anna Slobodová (chairs)

Conference Organization

Program Co-Chairs

Per Bjesse, *Synopsys, USA*
Anna Slobodová, *Centaur Technology, USA*

Local Arrangements Chair

David Rager, *University of Texas at Austin, USA*

Tutorial Chair

Barbara Jobstman, *CNRS/Verimag, France*

Publication Chair

Viktor Kuncak, *EPFL, Switzerland*

Panel Chairs

Lee Pike, *Galois, Inc., USA*
Murali Talupur, *Intel, Corp., USA*

Program Committee

Nina Amla, *NSF, USA*
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*
Alessandro Cimatti, *FBK-irst, Italy*
Koen Claessen, *Chalmers University of Technology, Sweden*
Rolf Drechsler, *University of Bremen, Germany*
Bruno Dutertre, *SRI international, USA*
Ganesh Gopalakrishnan, *University of Utah, USA*
Aarti Gupta, *NEC Laboratories America, USA*
Alan Hu, *University of British Columbia, Canada*
Chung-Yang Ric Huang, *National Taiwan University, Taiwan*
Barbara Jobstmann, *CNRS/Verimag, France*
Kevin Jones, *City University London, UK*
Gerwin Klein, *NICTA, Sydney, Australia*
Daniel Kroening, *University of Oxford, UK*
Thomas Kropf, *Robert Bosch GmbH, and University of Tuebingen, Germany*
Viktor Kuncak, *EPFL, Switzerland*

Oded Maler, *Verimag, France*
Panagiotis Manolios, *Northeastern University, USA*
Ken McMillan, *Microsoft Research, USA*
Tom Melham,*University of Oxford, UK*
Doron Peled, *Bar Ilan University, Israel*
Lee Pike,*Galois, Inc., USA*
Kavita Ravi, *AAAS Science and Technology Policy Fellow, USA*
Sandip Ray, *Intel Corp., USA*
Julien Schmaltz, *Open University of the Netherlands, The Netherlands*
Peter-Michael Seidel, *AMD, USA*
Natasha Sharygina, *Universita' della Svizzera Italiana, Switzerland*
Satnam Singh, *Microsoft Research, Cambridge, UK*
Anna Slobodova, *Centaur Technology, USA*
Sofiene Tahar, *Concordia University, Canada*
Murali Talupur, *Intel Corp., USA*
Helmut Veith, *Vienna University of Technology, Austria*
Karen Yorav, *IBM Haifa Research Lab, Israel*

Steering Committee

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

External Reviewers

Naeem Abbasi, Allon Adir, Behzad Akbarpour, Francesco Alberti, Eli Arbel, Eugene Asarin , Magnus Björk, Victor Braberman, Roberto Bruttomesso, Richard Bubel, Krishnendu Chatterjee, Chun-Nan Chou, Jordi Cortadella, Matthias Daum, Aldric Degorre, Niklas Een, Stephan Eggersglüß, Azadeh Farzan, Grigory Fedyakovich, Goeschwin Fey, Alexander Finder, Arthur Flatau, Sicun Gao, Eugene Goldberg, Jean Goubault-Larrecq, David Greenaway, Andreas Griesmayer, Daniel Grosse, Arie Gurfinkel, Ali Habibi, Liana Hadarean, Finn Haedicke, Hyojung Han, Kevin Harer, Nannan He, Joe Hendrix, Georg Hofferek, Andreas Holzer, Alexander Ivrii, Swen Jacobs, Himanshu Jain, Mitesh Jain, Visar Januzaj, Hoonsang Jin, Attila Jurecska, Alexander Kaiser, Hyondeuk Kim, Mike Kishinevsky, Alfred Koelbl, Robert Koenighofer, Rafal Kolanski, Laura Kovacs, Sava Krstic, Hoang Le, Chih-Chun Lee, Guodong Li, Ann Lillieström, Liya Liu, Florian Lonsing, Claire Maiza, Arie Matsliah, Tarek Mhamdi, Andrea Micheli, Michał Moskal, Sergio Mover, Amir Nahir, Kedar Namjoshi, Iman Narasamdya, Rajeev Narayanan, Ziv Nevo, Dejan Nickovic, Michael Norrish, Sam Owre, Vasilis Papavasileiou, David Penry, Dmitry Pidan, Kairong Qian, Ajitha Rajan, Zvonimir Rakamaric, Yusi Ramadian, Heinz Riener, Simone Fulvio Rollini, Marco Roveri, Hassen Saidi, Sawada Jun, Bas Schaafsma, Nassim Seghir, Martina Seidl, Ondrej Sery, Divjyot Sethi, Thomas Sewell, Gil Shurek, Radu Siminiceanu, Konrad Slind, Nick Smallbone, Mohamed Yousri Soliman, Fabio Somenzi, Sudarshan Srinivasan, Wilfried Steiner, Andreas Steininger, Rob Sumners, Philippe Suter, Kai-Fu Tang, Michael Tautschnig, Richard Trefler, Shihheng Tsai, Aliaksei Tsitovich, Tomas Vojnar, Christian Von Essen, Georg Weissenbacher, Robert Wille, Simon Winwood, Bo-Han Wu, Cheng-Yin Wu, Chi-An Wu, Jun Yuan

Verifying Concurrent Programs

(Tutorial Talk)

Aarti Gupta
NEC Laboratories America, Inc.

ABSTRACT

The proliferation of multi-core hardware has led to widespread use of concurrent programs. However, these programs are notoriously difficult to get right and to debug for developers. Even for automated verification, it is a big challenge to reason about subtle synchronization between communicating threads or processes, combined with an exponential number of interleavings. This tutorial will focus on the main ideas that have been used to handle synchronization and interleavings in automatic verification techniques for concurrent programs. These ideas have been applied in many settings, inspired by successful verification efforts for finite state systems on one hand, and for sequential programs on the other.

We will start by describing model checking efforts on concurrent programs based on pushdown system (PDS) models. PDS-based model checking and dataflow analysis have been successfully used for sequential program verification. However, extending these techniques to a system of interacting PDSs is challenging, due to undecidability of basic reachability checking when recursive programs interact using certain kinds of synchronization. Existing methods get around this by using various abstractions or restricting the patterns of synchronization allowed.

While PDSs can naturally model programs, the large model sizes for real programs and the complexity of model checking prohibit their application in practice. Instead, practical model checkers operate over finite state abstractions, typically by inlining procedures (without precisely modeling recursion). Often, they target standard concurrency-related bugs such as dataraces, deadlocks, atomicity violations, etc. Some pioneering efforts used explicit state model checking, with partial order reduction to reduce the number of interleavings. More recently, the success of SAT/SMT solvers has been leveraged for symbolic exploration in bounded settings, where memory consistency axioms implicitly encode the allowable set of interleavings.

On the program analysis front, capturing thread interference plays a key role. A general sequentialization technique has been proposed to lift any sequential program analyzer to bounded context analysis of multi-thread programs. Here, thread interference along a bounded schedule is represented using nondeterministic values on shared data, on which consistency is checked later according to individual threads. For unbounded analysis, techniques have been proposed to utilize automatically generated invariants (via abstract interpretation) to refine an over-approximation of thread interferences, or to incrementally propagate them starting from an under-approximation, until a fixpoint. The ultimate abstraction is to view interference from other threads as the environment, motivating efforts based on thread-modular and compositional verification.

Finally, we will describe trace-methods that utilize given dynamic test executions as a starting point to systematically explore alternate interleavings. Dynamic partial order reduction and preemptive context bounding techniques are based on controlling the scheduler to dynamically explore other interleavings. Another interesting direction is predictive analysis, where a predictive model is derived from a given trace and explored to predict violations in alternate interleavings of the same events.

The tutorial will highlight the progress made along these fronts, and the many challenges that remain in practical settings.

Self-Timing: a Step Beyond Synchrony

(Tutorial Talk)

Ivan Sutherland
Portland State University

ABSTRACT

Each part of a self-timed system starts work as soon as all its inputs are available, taking whatever time it needs to do its job and signaling when it is done. Each part waits for its predecessors to finish. The parts operate concurrently but not synchronously. Self-timed systems use local timing signals rather than a global clock

Self-timing eliminates the rigidity and energy consumption of a global clock. Self-timed systems operate over a wide range of power supply voltage. They automatically go slower at reduced voltage, saving energy both from lower voltage and from reduced speed. Their power-saving properties are making them increasingly attractive.

The design challenge posed by self-timing is the subject of this tutorial. Self-timed systems need the usual proofs of logical correctness. In addition, because they can take advantage of average rather than worst-case delay, analysis of circuit delay based on data statistics will be important. Of course, it's essential to check the timing of local signals, but commercial timing tools all check timing against a global "clock" signal that's absent from self-timed systems. Instead, self-timing focuses attention on the relative timing of pairs of signals. Self-timed systems pose all the elusive problems of concurrency: deadlock, non-determinism, and arbitration.

SHORT BIOGRAPHY

Ivan Sutherland and Marly Roncken started the Asynchronous Research Center (ARC) at Portland State University in 2009. Ivan was previously a Fellow at Sun Microsystems for nearly 25 years. Ivan holds an ACM Turing Award and is a member of the National Academy of Engineering and the National Academy of Sciences.

IC3: Where Monolithic and Incremental Meet

Fabio Somenzi
Dept. of Electrical, Computer, and Energy Engineering
University of Colorado at Boulder
Email: fabio@colorado.edu

Aaron R. Bradley
Summit Charter Middle School
Email: arbrad@cs.stanford.edu

***Abstract*—IC3 is an approach to the verification of safety properties based on relative induction. It is incremental in the sense that instead of focusing on proving one assertion, it builds a sequence of small, relatively easy lemmas. These lemmas are in the form of clauses that are derived from counterexamples to induction and that are inductive relative to reachability assumptions. At the same time, IC3 progressively refines approximations of the states reachable in given numbers of steps. These approximations, also made up of clauses, are among the assumptions used to support the inductive reasoning, while their strengthening relies on the inductive clauses themselves. This interplay of the incremental and monolithic approaches lends IC3 efficiency and flexibility and produces high-quality property-driven abstractions. In contrast to other SAT-based approaches, IC3 performs very many, very inexpensive queries. This is another consequence of the incrementality of the algorithm and is a key to its ability to be implemented in highly parallel fashion.**

I. Introduction

This paper discusses the IC3 technique for model checking safety properties. It is meant as a companion to [13]. Section II illustrates the approach on examples, while the rest of this introduction and Section III put the algorithm in its historical and ideological context by showing its relation to other methods for finite-state verification.

A. Induction

Induction is fundamental to the verification of safety properties [1], [2]. The only question is how it should be applied.

Consider a finite state system, $S : (\overline{i},\ \overline{x},\ I(\overline{x}),\ T(\overline{i},\overline{x},\overline{x}'))$, consisting of primary inputs $\overline{i}$, state variables $\overline{x}$, a propositional formula $I(\overline{x})$ describing the initial configurations of the system, and a propositional formula $T(\overline{i},\overline{x},\overline{x}')$ describing the transition relation. Primed state variables $\overline{x}'$ represent the next state.

Suppose that one wants to prove that every reachable state satisfies state assertion $P(\overline{x})$. Beginner's luck might allow one to proceed as follows:

- Show that the initial configuration of the system satisfies P: $I(\overline{x}) \Rightarrow P(\overline{x})$, where $\Rightarrow$ corresponds to implication. That is, all states that satisfy the initial condition I also satisfy P.
- Show that a P-state can only be followed by another P-state: $P(\overline{x}) \wedge T(\overline{i},\overline{x},\overline{x}') \Rightarrow P(\overline{x}')$.

These two steps—sometimes called *initiation* and *consecution*, respectively—comprise induction over S.

B. Monolithic and Incremental Methods

Outside of a classroom, such a direct application of induction is bound to fail. The development of safety model checking has essentially been the study of what one should do when, as usual, it does fail. In *Temporal Verification of Reactive Systems: Safety* [3], Manna and Pnueli write,

> We present two solutions to this problem, which can be summarized by the following strategies:
> 1) Use a stronger assertion, or
> 2) Conduct an incremental proof, using previously established P-invariants.

They go on to endorse the latter approach when engaging in manual or computer-aided verification:

> We strongly recommend [an incremental proof] whenever applicable. Its main advantage is that of *modularity*.

The former approach, however, is the one that has been most pursued from an algorithmic point of view in the context of hardware model checking. The formal basis for this approach is the following. If

- $I(\overline{x}) \Rightarrow F(\overline{x})$
- $F(\overline{x}) \wedge T(\overline{i},\overline{x},\overline{x}') \Rightarrow F(\overline{x}')$
- $F(\overline{x}) \Rightarrow P(\overline{x})$

then P is an invariant of S. In words, if F is inductive over S and implies P, then both F and P are invariants.

Traditional model checkers, based on BDDs [4] or SAT [5], explicitly compute post-conditions to compute the strongest possible strengthening of P, namely the reachable set of states, or pre-conditions to compute the weakest possible strengthening of P, namely all states except those that can lead to a violation of P. Bounded model checkers (BMC) exploit the finiteness of the state graph to enable a complete approach based on unrolling T and searching, with a SAT solver, for a counterexample trace [6]. An alternative to relying on a property of the state graph is to strengthen consecution simply by considering multiple time steps at once: k-induction assumes that P holds over multiple time steps to increase the likelihood that P holds in the next time step [7]. BDD-based algorithms that compute backward reachability can also be interpreted as computing increasingly strong consecutions: the number of iterations required for the fixpoint computation to converge to the weakest possible invariant that implies P gives the number of time steps to be considered to turn P into an inductive assertion.

Finally, one can abstract the post-condition in order to ease the computation, as in abstract interpretation [8]. Even better, one can abstract it with respect to the property, as in interpolation-based model checking, in which interpolants are derived from failed BMC queries [9].

We refer to these methods as *monolithic* because they spend all of their resources in computing one inductive assertion. Furthermore, their success is fundamentally tied to the reasoning engines—either the BDD package or the SAT solver. The representation of states reachable from the initial ones or states that can reach the target ones often entails prohibitively large BDDs. BMC, k-induction, and interpolant-based model checking fail when the SAT solver is overwhelmed by the number of unrollings of T.

One must then wonder whether an *incremental* approach, which is so successful for humans, might not be a bad idea as the basis for an algorithm. An incremental approach would compute many inductive assertions that all together strengthen P. It would thus have the modularity that Manna and Pnueli highlight—each assertion need only refer to an aspect of S—as well as the potential of not taxing the reasoning engines quite so much. Moreover, the incremental approach would be property directed, like the interpolant-based method: each intermediate assertion would arise to eliminate some hypothesized error.

The formal basis for an incremental approach is the following. Consider a sequence $\varphi_1(\overline{x}), \ldots, \varphi_n(\overline{x})$ of assertions such that

- every assertion is satisfied by the initial states: for each j, $I(\overline{x}) \Rightarrow \varphi_j(\overline{x})$,
- each assertion obeys consecution under the assumption that its predecessors hold: for each j,

$$\bigwedge_{1 \le k \le j} \varphi_k(\overline{x}) \;\wedge\; T(\overline{i}, \overline{x}, \overline{x}') \;\Rightarrow\; \varphi_j(\overline{x}') \;,$$

- and all together they imply P:

$$\bigwedge_{1 \le j \le n} \varphi_j(\overline{x}) \;\Rightarrow\; P(\overline{x}) \;.$$

If P also satisfies initiation, then it is an invariant of S. In this version of consecution (the second condition), we say that φ_j is inductive *relative to* $\varphi_1, \ldots, \varphi_{j-1}$.

In the incremental approach, one might as well assume P. If

- P is satisfied by the initial states: $I(\overline{x}) \Rightarrow P(\overline{i})$,
- every assertion is satisfied by the initial states: for each j, $I(\overline{x}) \Rightarrow \varphi_j(\overline{x})$,
- each assertion obeys consecution under the assumption that its predecessors and P hold: for each j,

$$\bigwedge_{1 \le k \le j} \varphi_k(\overline{x}) \;\wedge P(\overline{x}) \wedge T(\overline{i}, \overline{x}, \overline{x}') \;\Rightarrow\; \varphi_j(\overline{x}') \;,$$

- and P is inductive relative to the assertions,

$$\bigwedge_{1 \le j \le n} \varphi_j(\overline{x}) \;\wedge P(\overline{x}) \wedge T(\overline{i}, \overline{x}, \overline{x}') \;\Rightarrow\; P(\overline{x}') \;,$$

then P is an invariant of S.

Bradley and Manna proposed the first incremental safety model checking algorithm [10], [11]. It discovers inductive subclauses of the negation of states that lead, not necessarily directly, to violations of P. Such clauses eliminate the states from which they are derived while generalizing to eliminate many other states as well. Each clause is an assertion φ_j that is indeed typically inductive only relative to prior assertions but not on its own. As expected, deriving the clauses is relatively easy: the employed SAT solver solves many, often hundreds or thousands, of queries per second, in stark contrast to BMC, k-induction, and the interpolant method. An unexpected benefit is that this instance of the incremental approach is effectively parallelizable—and easily so. This characteristic has carried through in subsequent work.

Besides modularity and reduced labor, the incremental approach has one more benefit: induction-based generalization is a powerful mechanism for property-directed abstraction. Induction tends to find semantic relationships among states rather than simply adjacency, or structural, relationships, as in traditional model checking. The clause that eliminates a state s may well eliminate states that are far, or even disconnected, from s in the state graph. When induction is applied throughout the analysis rather than being the goal of a monolithic propagation, it abstracts the system in a property-directed fashion.

Unfortunately, this algorithm suffers from a common pitfall of incremental methods. Manna and Pnueli write:

> There are cases in which the conjunction $\varphi_1 \wedge \varphi_2$ is inductive, but it is not the case that φ_1 is inductive and φ_2 is inductive relative to φ_1.

In the context of the algorithm, a state s can be encountered such that $\neg s$ does not contain a subclause that is inductive relative to known information. In such situations, the algorithm falls back on state enumeration until sufficient information is acquired to resume inductive clause construction. Yet when such a situation does not occur, the algorithm is extremely effective [11].

This weakness of the incremental method is not an issue for manual or computer-assisted verification, as the human can provide an insight. But in an algorithmic context, one typically limits the form of assertions in order to control computational costs [8]. Is an algorithmic incremental method thus doomed from the start?

C. IC3: A Monolithic-Incremental Hybrid

While an incremental method may be limited in the form of its assertions, Bradley eventually realized that the constructed clauses need not be truly inductive. The machinery of induction can be applied just as well when stronger information is assumed—information that is not necessarily valid for the entire state space. In particular, stepwise assumptions—assertions that hold for some number of timesteps rather than for all time—could be combined with relative inductive clause generation to yield a hybrid monolithic-incremental method in

which *relatively inductive clauses are guaranteed to exist* if P is invariant. IC3 is the result of this insight [12], [13].

IC3 is incremental in that it finds inductive subclauses of the negations of states, just as the first approach does—except that these clauses are now inductive relative to certain assumptions. Its use of SAT solvers is thus similar: hundreds to thousands of queries are solved per second. Additionally, the clauses are the right compromise between effort and information content, so that they can be traded effectively among parallel processes.

IC3 is monolithic in that it computes over-approximations to the sets of states reachable in one step, two steps, etc., until it converges upon an inductive strengthening assertion. Each major iteration propagates the clauses that comprise the timestep approximations forward in time as much as possible. These over-approximations are the information relative to which new clauses are generated.

Hence, IC3 alternates between an incremental mode, in which it uses states that lead, not necessarily directly, to violations of P to discover new relatively inductive clauses, and a monolithic mode, in which it propagates clauses forward across time steps. Models on which the original method [10] devolves into enumerating states cause IC3 to go through more major iterations, yielding long sequences of stepwise over-approximations. Models on which the original method succeeds are just as easy, and often easier, for IC3, and result in short sequences of stepwise over-approximations before the final inductive strengthenings are formed. And many other models cause IC3 to adapt either a more monolithic or a more incremental strategy at various stages. The power of IC3 is that it can quickly deduce lemmas for certain aspects of a model while working harder—and, at times, more monolithically—for other lemmas that require more clauses.

II. Examples

This section presents IC3 by way of two examples. The objective is to show the nature of the algorithm. Certain optimizations omitted from this exposition are essential in practice for good performance.

A. A Passing Property

Figure 1 shows the state transition graph of a system S with no primary inputs and state variables $\overline{x} = \{x_1, x_2\}$ such that

$$\begin{aligned} I(\overline{x}) &= \neg x_1 \wedge \neg x_2 \\ T(\overline{x}, \overline{x}') &= (x_1 \vee \neg x_2 \vee x_2') \wedge (x_1 \vee x_2 \vee \neg x_1') \\ &\quad \wedge (\neg x_1 \vee x_1') \wedge (\neg x_1 \vee \neg x_2') \wedge (x_2 \vee \neg x_2') \\ P(\overline{x}) &= \neg x_1 \vee x_2 \ . \end{aligned}$$

Each state in the figure is annotated with its encoding. The incoming arrow designates q_0 as initial, while the shaded state (q_3) violates P. Inspection of Fig. 1 reveals that the only reachable state of S is q_0 and that $S \models P$. This example is not meant to highlight the efficiency of IC3. On the contrary, it provides the opportunity for a rather extensive tour of the algorithm in spite of its simplicity. (The reader is however cautioned that interesting aspects of IC3, like its ability to quickly compute long counterexamples, or to find large sets of mutually inductive clauses, are better understood via the algorithm's fundamental intentions. This section only provides a stepping stone in that direction.)

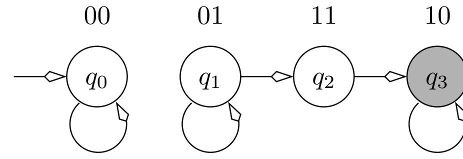


Fig. 1. The state transition graph of a simple system.

The initial check performed by IC3 establishes that there are no counterexamples of length 0 or 1. Therefore, the over-approximations (or stepwise assumptions)

$$\begin{aligned} F_0 &= I = \neg x_1 \wedge \neg x_2 \\ F_1 &= P = \neg x_1 \vee x_2 \end{aligned}$$

satisfy the fundamental IC3 invariants for $k = 1$:

$$\begin{aligned} &I \Rightarrow F_0 && \\ &F_i \Rightarrow F_{i+1} && 0 \leq i < k \\ &F_i \Rightarrow P && 0 \leq i \leq k \\ &F_i \wedge T \Rightarrow F'_{i+1} && 0 \leq i < k \ . \end{aligned}$$

Together, these invariants assert the "reasonableness" of the stepwise assumptions. In particular, since no counterexample of length up to k exists, all states reachable in at most k steps are P-states. Taking F_k to be P is therefore a valid over-approximation. If IC3 eventually increases k to 2, it is because it has established that there are no counterexamples of length up to 2. In general, if IC3 increases k from n to $n + 1$, it is because it has established that there are no counterexamples of length up to $n + 1$. It does so by proving that there are no counterexamples-to-induction (CTI) states that are reachable in at most n steps from some initial state. For that, it checks whether $F_n \wedge T \Rightarrow P'$ can be violated.

The check $F_1 \wedge T \Rightarrow P'$ produces $s = x_1 \wedge x_2$ as CTI. (Note that this check is equivalent to $P \wedge T \Rightarrow P'$, the inductive step of a simple inductive proof.) If $S \models P$, a CTI must be unreachable from the initial states. If $\neg s \wedge F_1 \wedge T \Rightarrow \neg s'$, unreachability is proved. If, however, the implication does not hold, the CTI may still be unreachable (as in this case) and IC3 tries to learn something useful about it: specifically, it tries to bound the length of a counterexample that goes through the CTI. Hence, $\neg s = \neg x_1 \vee \neg x_2$ is checked for inductiveness relative to the various F_i's. It is not inductive relative to F_1 because of the transition between q_1 and q_2. It is, however, inductive relative to F_0. (Otherwise, P would not hold.)

The inductiveness check has established that the CTI is not reachable in one step. Therefore, it would be possible to remove it from F_1 by adding the clause $\neg s$ to it. However, removing one CTI at a time is not practical for all but the simplest systems. Instead, IC3 looks for more states, be they CTIs or not, that, like the one at hand, are not reachable in

one step and such that they are all described by a subclause of $\neg s$. That is, IC3 tries to generalize $\neg s$.

Generalization of $\neg s$ is thus attempted at level 0. The algorithm may find either $\neg x_1$ or $\neg x_2$ as subclauses of $\neg s$, because both satisfy both initiation and consecution. In fact, the conjunction of either clause with F_0 yields F_0 itself, from which no state violating either $\neg x_1$ or $\neg x_2$ may be reached. For the execution of the algorithm, however, which clause is the result of generalization makes a difference. Suppose $\neg x_2$ is found. Then the update of F_1 produces

$$F_1 = (\neg x_1 \vee x_2) \wedge \neg x_2 \ \ ,$$

which is equivalent to F_0. While this observation suffices to prove termination, IC3 first checks whether $F_1 \wedge T \Rightarrow P'$; that is, it checks whether the strengthening of F_1 has gotten rid of the CTI. Since the answer is positive, it increases k to 2, instantiates $F_2 = \neg x_1 \vee x_2$, and then propagates $\neg x_2$ from F_1. That is, it adds $\neg x_2$ to F_2 because $F_1 \wedge T \Rightarrow \neg x_2'$. The addition causes F_1 and F_2 to be identical and terminates the proof because $F_1 = (\neg x_1 \vee x_2) \wedge \neg x_2$ has been shown to be inductive ($I \Rightarrow F_1$ and $F_1 \wedge T \Rightarrow F_1'$) and is known to imply P. (F_1 is initially P and can only get stronger through the run of IC3.)

If, instead of $\neg x_2$, the generalization of $\neg x_1 \vee \neg x_2$ produces $\neg x_1$, the update of the reachability over-approximations results in

$$F_1 = (\neg x_1 \vee x_2) \wedge \neg x_1 \ \ ,$$

which is equivalent to $\neg x_1$. This F_1 is not as strong as in the previous case, and in particular does not exclude q_1, but it is still sufficient to satisfy $F_1 \wedge T \Rightarrow P'$. Therefore, IC3 sets k to 2, instantiates $F_2 = \neg x_1 \vee x_2$ and tries to strengthen it by propagating clause $\neg x_1$ from F_1. However,

$$F_1 \wedge T \not\Rightarrow \neg x_1' \ \ ,$$

because of the transition from q_1 to q_2; hence, no strengthening takes place. State $s = x_1 \wedge x_2$ is found once again as a CTI. The difference from the previous iteration is that it is now known that no counterexample of length less than 3 may go through it. IC3 then tries to prove that no counterexample of length 3 exists. The next step is therefore finding i such that

$$(\neg x_1 \vee \neg x_2) \wedge F_i \wedge T \Rightarrow (\neg x_1' \vee \neg x_2') \ \ .$$

Since $F_2 = P$ and F_0 has not changed, the answers for $i = 2$ and $i = 0$ are already known. It remains to ascertain whether F_1 is strong enough to support $\neg s$. Once again, the transition between q_1 and q_2 causes the answer to be negative. Therefore, $\neg s$ is inductive at level 0, but not at level 1. Generalization of this clause also proceeds as in the previous iteration and may result in either literal being dropped. If $\neg x_2$ is found, then its addition to F_1 makes it inductive, so that both $\neg x_1$ and $\neg x_2$ are propagated to F_2 causing termination.

If, on the other hand, $\neg x_1 \vee \neg x_2$ is generalized to $\neg x_1$, then no changes to F_1 result and no clause propagation ensues. Since F_2 has not changed, the CTI has not been removed. To guarantee termination, IC3 identifies a predecessor of $s = q_2$ that is an F_1 state, but not an F_0 state. The only choice is $t = \neg x_1 \wedge x_2$. If this state is proved unreachable, progress is made. More generally, if all predecessors of s in F_1 are shown to be unreachable in at most one step, then s is not reachable in at most two steps and hence there is no counterexample of length up to 3 through it.

IC3 therefore recurs on t to find which is the least i (if any) such that

$$\neg t \wedge F_i \wedge T \Rightarrow \neg t' \ \ .$$

Since $\neg t$ is itself inductive (q_1 in Fig. 1 has no incoming transitions from other states) $i = 2$. Since x_1 does not satisfy initiation, the only generalization of $\neg t$ is $\neg x_2$. The addition of this clause to both F_1 and F_2 makes them identical and causes termination.

In this case, F_1 is exact at termination. That is, F_1 describes exactly the states reachable in at most one step from the initial states. Oftentimes, though, the ability to prove properties quickly stems from the ability to keep the over-approximations loose. This is one reason why IC3 does not decompose the initial condition into a set of strong clauses that can be propagated.[1]

In contrast to IC3, the approach of [10] focuses on removing each CTI by generalizing its negation to an inductive clause. For the system of Fig. 1, this entails generalizing $s = \neg x_1 \vee \neg x_2$ by checking whether it contains a subclause d such that

$$d \wedge P \wedge T \Rightarrow d' \ \ .$$

The solution is in this case $d = \neg x_2$. Once this clause is discovered, it is possible to prove that $\neg x_2 \wedge P \wedge T \Rightarrow P'$, which in turn proves $S \models P$. However, if the encoding of the states is changed so that $q_2 = x_1 \wedge \neg x_2$ and $q_3 = x_1 \wedge x_2$, then the negation of the CTI $\neg s = \neg x_1 \vee x_2$ has no inductive generalization and the approach of [10] falls back on removing the CTI alone from further consideration. While this is hardly a disadvantage when there are only four states, it is the main weakness of that method. IC3 is also affected by the change of encoding, in that $\neg s = \neg x_1 \vee x_2$ can only be generalized to $\neg x_1$, but relatively inductive clauses can always be found.

B. A Failing Property

Figure 2 shows the transition graph of a system U with no primary inputs and state variables $\overline{x} = \{x_1, x_2, x_3\}$ defined by

$$\begin{aligned} I(\overline{x}) &= \neg x_1 \wedge \neg x_2 \wedge \neg x_3 \\ T(\overline{x}, \overline{x}') &= (x_1 \vee \neg x_2') \wedge (\neg x_1 \vee x_2') \\ &\quad (x_2 \vee \neg x_3') \wedge (\neg x_2 \vee x_3') \\ P(\overline{x}) &= \neg x_1 \vee \neg x_2 \vee \neg x_3 \ \ . \end{aligned}$$

State q_i has code i. For example, q_3 is $\neg x_1 \wedge x_2 \wedge x_3$. As in Fig. 1, the shaded state violates property P.

[1] Implementations rely on pre-analysis of the model that easily discovers most state variables that can only take one value. Using any remaining literals from the initial condition typically lengthens the analysis because it overconstrains the early over-approximations.

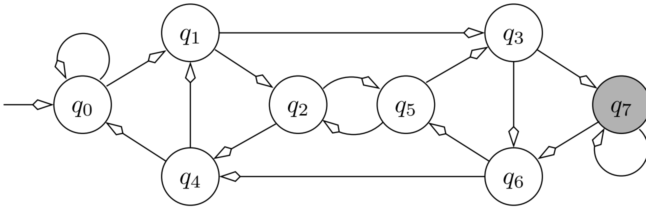


Fig. 2. Transition graph for a system with a failing property.

Having checked that there are no counterexamples of length up to 1, IC3 sets $k = 1$ and chooses

$$F_0 = I = \neg x_1 \wedge \neg x_2 \wedge \neg x_3$$
$$F_1 = P = \neg x_1 \vee \neg x_2 \vee \neg x_3$$

as stepwise assumptions. Checking whether $F_1 \wedge T \Rightarrow P'$ yields $s = \neg x_1 \wedge x_2 \wedge x_3$ as CTI. Inductiveness of $\neg s$ is established at level 0 and the generalization of $\neg s$ is $\neg x_2$. After the strengthening of F_1,

$$F_1 = P \wedge \neg x_2 \ ,$$

$F_1 \wedge T \Rightarrow P'$. Therefore, k increases to 2 and $F_2 = P$ is instantiated. No clause is propagated from F_1 to F_2. Therefore, the same CTI as before is found when $F_2 \wedge T \Rightarrow P'$ is tested. Since $\neg s \wedge F_1 \wedge T \not\Rightarrow \neg s'$, inductiveness is again established at level 0 and the generalization is again $\neg x_2$. Nothing changes in the stepwise assumptions, and the CTI remains an F_2 state. IC3 therefore looks for a predecessor of s that is in F_1. The choice is between $\neg x_1 \wedge \neg x_2 \wedge x_3$ and $x_1 \wedge \neg x_2 \wedge \neg x_3$. The former is immediately shown to be a successor of the initial state because its negation is not inductive even at level 0. Therefore the minimum-length counterexample q_0, q_1, q_3, q_7 is found.

If, instead of $\neg x_1 \wedge \neg x_2 \wedge x_3$, IC3 chooses $t = x_1 \wedge \neg x_2 \wedge \neg x_3$ as F_1 predecessor of the CTI, $\neg t$ is proved inductive at level 1 because the two predecessors of t are not in F_1. Generalization of $\neg t$ produces $\neg x_1$, which is added to both F_1 and F_2 eliminating $x_1 \wedge \neg x_2 \wedge \neg x_3$ from both. This forces the choice of $\neg x_1 \wedge \neg x_2 \wedge x_3$ as F_1 predecessor of the CTI and leads to the same counterexample as before. It should be noted how the refinement of the stepwise assumptions acted as guidance in the search for the counterexample.

IC3 does not guarantee counterexamples of minimum length. While k cannot increase beyond the length of a shortest counterexample, IC3 may find a counterexample well before k matches its length. This ability proves an important advantage when the transition relation is such that refining the stepwise assumptions beyond a certain point becomes difficult. This may be the case when the CTIs and the states that should be removed from one of the F_i's to get out of an impasse have codes that are different enough that the generalized inductive clauses do not cover the "problem" states.

When refinement of the stepwise assumptions proves difficult, IC3 often finds that the negation of the target state (CTI or one of its predecessors) is inductive at the level immediately preceding that of the target state. It then chooses a predecessor at the same level, producing a path with several states for one F_i until either the path eventually crosses into F_{i-1} or new clauses are generated that cause a refinement of the stepwise assumptions. Under these circumstances IC3 may still discover a deep counterexample even though k is small.

III. Discussion

A. What Problem is IC3 Trying to Solve?

Interpolation and k-induction address the practical incompleteness of BMC. The latter combines BMC with a consecution check:

$$P \wedge \bigwedge_{i=0}^{k-1} (T^{(i)} \wedge P^{(i)}) \Rightarrow P^{(k)} \ .$$

When that check fails, k is increased, corresponding to a further unrolling of T. In practice, k can be prohibitively large.

The interpolant method goes further: it suggests forming over-approximate stepwise reachability sets F_i using a fixed unrolling. It addresses the failure of the following implication by increasing k:

$$F_i \wedge \bigwedge_{i=0}^{k-1} (T^{(i)} \wedge P^{(i)}) \Rightarrow P^{(k)} \ .$$

Because the implication does not hold, no interpolant exists that lies between the i-step over-approximation F_i and the k-step unrolling leading to a violation of P. The interpolant method thus increases k for the next round, yielding better over-approximations F_i.

Hence, neither k-induction nor the interpolant method drop the regime of unrolling that BMC introduced. While they attempt to reduce the number of necessary unrollings, their completeness—both practically and theoretically—is still fundamentally tied to unrolling.

IC3 entirely sidesteps the need for unrolling and thus sets out on a new trail than that blazed by BMC. When confronted with a problem similar to the one in interpolation (though lacking any unrolling), that is, the failure of the implication

$$F_i \wedge T \Rightarrow P' \ ,$$

it refines the i-step over-approximation F_i itself—and typically earlier stepwise over-approximations—in order to make the refined implication come closer to holding. It accomplishes this refinement by incrementally generating stepwise-relative inductive clauses in reaction to the CTI that the implication's failure reveals. In the end, the sequence of over-approximate stepwise assertions F_i can be seen as a possible outcome of the interpolant method—though derived in a fundamentally different manner.

B. The Incremental Method: Beyond IC3

The purely incremental method fails when the space of assertions is too poor to provide lemmas for all possible situations. In the case of safety model checking, clauses are too weak to be the basis of a robust algorithm. IC3 provides a stronger framework in which to use a weak, but expressively

complete, assertion domain. However, a pure incremental approach can work on its own in other settings.

In this conference, we present an incremental approach to model checking LTL properties of systems [14]. The fundamental insight is that *SCC-closed regions* of the state graph, which are a fundamental characterization used in BDD-based techniques [15], can be discovered through induction. Hence, inductive assertions, as discovered by IC3, are the intermediate lemmas of this approach. Unlike the relationship between error states and clauses in safety model checking, every hypothesized error—which we call a *skeleton*—that does not correspond to an actual error has a corresponding inductive proof. Thus, the algorithm is purely incremental, and it enjoys the usual benefits: modular reasoning, natural abstraction, and opportunities for parallelization.

Acknowledgments. This material is based on work supported in part by the National Science Foundation under grant No. 0952617 and by the Semiconductor Research Corporation under contract GRC 1859. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

REFERENCES

[1] R. W. Floyd, "Assigning meanings to programs," in *Symposia in Applied Mathematics*, vol. 19. American Mathematical Society, 1967, pp. 19–32.

[2] C. A. R. Hoare, "An axiomatic basis for computer programming," *Communications of the ACM*, vol. 12, no. 10, pp. 576–580, October 1969.

[3] Z. Manna and A. Pnueli, *Temporal Verification of Reactive Systems: Safety*. Springer-Verlag, 1995.

[4] 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.

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

[6] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu, "Symbolic model checking without BDDs," in *Fifth International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS'99)*, Amsterdam, The Netherlands, Mar. 1999, pp. 193–207, lNCS 1579.

[7] M. Sheeran, S. Singh, and G. Stålmarck, "Checking safety properties using induction and a SAT-solver," in *Formal Methods in Computer Aided Design*, W. A. Hunt, Jr. and S. D. Johnson, Eds. Springer-Verlag, Nov. 2000, pp. 108–125, lNCS 1954.

[8] P. Cousot and R. Cousot, "Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *POPL*. ACM Press, 1977, pp. 238–252.

[9] K. L. McMillan, "Interpolation and SAT-based model checking," in *Fifteenth Conference on Computer Aided Verification (CAV'03)*, W. A. Hunt, Jr. and F. Somenzi, Eds. Berlin: Springer-Verlag, Jul. 2003, pp. 1–13, lNCS 2725.

[10] A. R. Bradley and Z. Manna, "Checking safety by inductive generalization of counterexamples to induction," in *Formal Methods in Computer Aided Design (FMCAD'07)*, Austin, TX, 2007, pp. 173–180.

[11] A. R. Bradley, "Safety analysis of systems," Ph.D. dissertation, Stanford University, May 2007.

[12] ——, "k-step relative inductive generalization," CU Boulder, Tech. Rep., March 2010, http://arxiv.org/abs/1003.3649.

[13] ——, "SAT-based model checking without unrolling," in *Verification, Model Checking, and Abstract Interpretation (VMCAI'11)*, Austin, TX, 2011, pp. 70–87, lNCS 6538.

[14] A. R. Bradley, F. Somenzi, Z. Hassan, and Y. Zhang, "An incremental approach to model checking progress properties," in *Formal Methods in Computer Aided Design (FMCAD'11)*, Austin, TX, 2011.

[15] R. Bloem, H. N. Gabow, and F. Somenzi, "An algorithm for strongly connected component analysis in $n \log n$ symbolic steps," in *Formal Methods in Computer Aided Design*, W. A. Hunt, Jr. and S. D. Johnson, Eds. Springer-Verlag, Nov. 2000, pp. 37–54, lNCS 1954.

Planning for End-to-End Formal using Simulation-based Coverage

(Invited Tutorial)

Prashant Aggarwal
Oski Technology
Gurgaon, India
prashant@oskitech.com

Darrow Chu
Cadence Design Systems
San Jose, CA, USA
darrow@cadence.com

Vijay Kadamby
Cisco
San Jose, CA, USA
vkadamby@cisco.com

Vigyan Singhal
Oski Technology
Mountain View, CA, USA
vigyan@oskitech.com

***Abstract*—Model checking tools are gaining traction as a practical formal verification solution for industrial designs. However, the use of absraction models is key to overcoming complexity barriers in applying these tools. Coverage has been a useful metric to determine when simulation-based verification is complete. In this paper, we show how similar coverage metrics can be used to determine the completeness of a formal verification setup. We also show how coverage can be used to determine effectivness of different abstraction models are. This methodology can be used to set formal verification goals, and to measure the progress of the work, thereby placing formal verification in a chip design schedule. We use a real-world design with a large state space, and present quantitative coverage metrics to illustrate the methodology, and its benefits for faster run-time, faster discovery of bugs, and higher coverage.**

I. INTRODUCTION

During the last decade, formal verification tools have been increasingly more popular for the pre- and post-silicon verification of a diverse class of IC designs, varying from custom processor designs to general-purpose ASICs. While multiple formal verification technologies are used in the industry (e.g. model checking, theorem proving, C-vs-RTL sequential equivalence checking), model checking tools account for most of the usage, judging from the number of available commercial tools as well as verification users in place. Furthermore, major EDA vendors (Cadence, Mentor Graphics and Synopsys) as well as a few startups (Averant, Jasper, OneSpin and Real Intent) offer competitive solutions. In this paper, we will use the term model checking synonymously with formal verification.

The extent to which an ASIC design tapeout schedule depends on formal verification is greatly contingent upon the scope of verification addressed by formal. Most often, formal is used as a supplement to simulation, to prove some specific difficult-to-verify behavior, local embedded RTL assertions, or interface protocol checks between blocks. Less often is formal used for end-to-end verification to replace simulation, where formal verifies most or all the functionality of a design, and replaces simulation at that level – simulation may still be used at a higher chip-level or system-level verification. End-to-end formal usually requires almost the entire logic in the design to be analyzed by the formal tool, and poses significant complexity barriers.

Formal verification tool developers as well as users have long used abstraction techniques to overcome the computational complexity problem. Most tools deploy sophisticated abstraction-refinement algorithms under the hood [5], [19]. On top of that, formal users can deploy manually crafted abstractions [4], [6], [7], [11], [13] to further reduce the complexity of the proofs. In this paper, we will take a complex design with a large state space, and show how the use of abstraction models can help achieve end-to-end formal for this design.

Coverage metrics are widely used in simulation-based verification to improve the quality of the test suite and estimate the progress of the verification task [9], [18]. Coverage can help identify important gaps in the stimuli provided to the design-under-test, although it has a known limitation that coverage does not evaluate the quality of the simulation checkers. The same coverage metrics can be deployed for formal verification with the same limitation [16]. Besides identifying unintentional over-constraints in a formal environment, formal coverage can estimate the effectiveness of the abstraction techniques being deployed – for example, a set of abstraction techniques is useful, if it enables many more lines or expressions of code to be reachable in the same amount of CPU time.

In this paper, we use formal coverage metrics to quantitatively demonstrate that suitable abstraction models achieve convergence. We begin by introducing end-to-end formal verification in Section II, and the components required to build such an environment. We mention the role of abstraction techniques to solve end-to-end formal in Section III. Next, in Section IV we discuss how coverage is used for formal verification, and introduce a coverage-driven flow for formal verification. In Section V, we introduce the design we have. This design has a state space that is fairly large for a typical model checker to handle, more than 1 million flops. The design is an integral part of a large real-world ASIC switch. Section VI describes some of the constraints and checkers needed for formal verification, including the most important end-to-end data checker. In Section VII, we describe the abstraction models deployed to overcome complexity barriers. We present the coverage results in Section VIII.

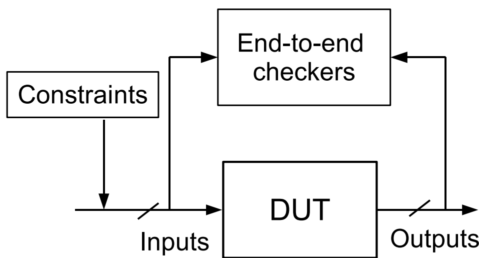


Fig. 1. End-to-end verification setup

II. End-to-End Formal

A. Checkers and Constraints

Besides reading the design-under-test (DUT), a model checker requires a set of checkers and constraints as inputs. The checkers and constraints can be written as properties in SystemVerilog Assertion language (SVA) [8]. However, often these checkers and constraints require supporting modeling code written in synthesizable SystemVerilog.

Checkers can vary widely in scope:

- Local checkers, also known as assertions. These checkers verify local properties of the design, and belong to one of the following:
  - Embedded RTL assertions. These assertions are local properties about the implementation details in the DUT, such as a state machine always stays one-hot encoded, or that a full FIFO is never written to. These assertions are typically written by the RTL designer, and embedded in the RTL code [21].
  - Interface assertions. These assertions encode the handshake protocol requirements for any of the interfaces of a design. These requirements can vary from a simple request-acknowledgement protocol to a more complex ARM AMBA AXI [15] or DDR2 protocol [6].
- End-to-end checkers (Fig. 1). These checkers primarily use a significant modeling code to encode a reference model for the required behavior of the design, by relating the correctness of the output data path of a design, given the transactions on the input datapath.

Bugs found through any of these checkers are useful. However, if formal is to be relied upon as a primary verification methodology for a design, simply verifying local checkers is not enough – a significant number of end-to-end checkers must be used to achieve adequate verification. Not surprisingly, proving the end-to-end checkers is usually computationally much more complex than local checkers, although there may be exceptions to this, and some local checkers may be difficult to prove too.

B. Complexity

The largest barrier to formal verification achieving the desired results is the complexity barrier faced by the tools. All known algorithms are worst-case exponential in the size of the cone-of-influence of the checks and constraints. For end-to-end formal verification, the model checking engine which is often the most effective is Bounded Model Checking (BMC) [1]. Although BMC can only find counterexamples, and not establish the full proof of any checks, the bounded proofs are good enough if the bounds are greater than the interesting corner-case behavior of the design, as judged by the verification or the design engineer.

Two complexity problems can interfere with BMC reaching acceptable proof bounds:

- the size of the logic in the cone-of-influence, including the number of flops as well as the combinational logic; and
- the state space diameter of the design, especially in presence of large counters, or sequentially deep logic.

The use of abstractions, discussed in the next section is the best strategy to overcome these complexity problems.

III. Abstraction Techniques

Abstraction techniques [3] are used to reduce the state space of the design, so that formal verification tools can solve a computationally easier problem. An abstraction is considered *sound* if does not reduce any design behavior, even if it adds to the design behaviors. We will only consider sound abstractions in this paper. Such abstractions can find proofs or failures faster. Every proof is guaranteed to be a proof on the original design. Each failure can be debugged to determine if it is a true counterexample due to an RTL bug, or a false counterexample due to an over-abstraction.

Examples of various abstraction techniques include:

1) Cut-points. Any internal logic in the design can be replaced by a cut-point, allowing that net to freely take a random value at any time [6], [12]. If a checker proves with such an abstraction, we can achieve significant reductions in run-time (of course, it also implies the need for additional checkers, since the proven checker is clearly independent of the excised logic).
2) Counter abstraction. Many designs have deep counters, for example, the initialization phase for DDR2 memory controllers last for hundreds of milliseconds, consuming millions of clock cycles. Many useful checks can be proved by abstracting the 2^n-state graph of an n-bit counter to a few states, e.g. 0, 1, *at-least-one*, *at-least-zero* [14].
3) Symmetric datatypes. Certain systems [7], [13] allow the users to specify that certain data types in the design are symmetric, and the values of this type are used only in certain symmetric ways (e.g. only compared for equality, or used as indices for arrays). This allows the system to reduce multiple symmetric proofs into a single one.
4) Data independence. When a design moves data across the design, and does not use the data contents for controlling the movement of the data, a few finite instantiations of data values are sufficient to establish the correctness of any checkers [20]. This abstraction

has been used to prove data correctness for many data transport hardware designs [11], [17].

5) Tagging. Often systems deal with a finite but large set of distinct data values [13]. Portions of such systems can be abstracted by simplifying the structure with respect to a specific or a symbolic tag.

Often, using an abstraction technique requires cut-pointing a section of the design, and adding constraints on the cut-points. The abstraction can be used to prove the desired checks. To complete the compositional proof [13] however, a second step is required – the constraints need to be converted into checks, and proven on the previously excised logic.

IV. Coverage

A. Coverage in Simulation

In simulation-based verification, coverage metrics are used heavily to determine when simulation is complete. The most common coverage metric is code coverage, including line, expression, FSM and toggle coverage. Line coverage, for example, computes what percentage of RTL statements in the DUT were exercised by a given set of tests. For example, consider:

```
1:  always @(posedge clk) begin
2:    if ((a && b) || c)
3:      e <= d1;
4:    else
5:      e <= d2;
6:  end
```

This example results in two line coverage targets, corresponding to lines 3 and 5. If a test causes `c` to be 1, the line 3 will be marked as covered. If no test in a test suite covers line 5, line coverage for the suite will be reported at 50%.

100% *judged* line coverage (given, say 99% *automated* coverage) is frequently a requirement for an ASIC tapeout – each line that is not automatically reported as covered in simulation, must be manually judged to be either redundant, or legacy code, or symmetric to another tested line. Tapeout would be delayed until more tests are written to cover the remaining lines. 100% line coverage does not imply an absence of bug. Still, line coverage helps measure the continuous progress of verification completeness in a dynamic chip design schedule, and often points to important coverage holes.

B. Formal Coverage Metrics

The same coverage metrics used in simulation can be applied to answer the question of whether the planned formal verification tasks are complete, or how much the formal verification tasks complement the simulation effort [16].

Simulation-based line (or expression) coverage metrics can be used to mean exactly the same in formal – given the constraints used and the proof depths reached in BMC (say, n cycles), report what percentage of line (or expression) targets are reachable in n cycles. For the example in Section IV-A, if `((a && b) || c)`, in line 2, is reachable in n cycles, this line would be reported as covered, and otherwise, not. Thus,

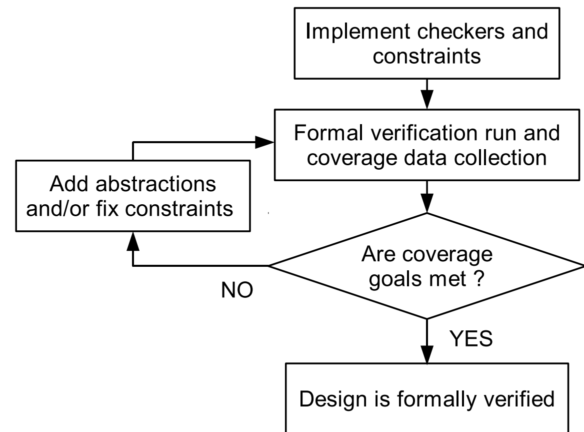


Fig. 2. Formal verification coverage flow

line coverage numbers would mean the same in simulation – whether a certain coverage target is exercised or not. And for formal, this would measure the quality of constraints (i.e. absence of over-constraints), as well as the BMC proof depths. Abstraction techniques, described in Section III, can help in achieving higher proof depths, improving the coverage results and thereby increasing the value of formal verification. and valid of formal verification. difference in the coverage results and the value of formal verification Commercial formal tools are beginning to support the measurement of formal coverage.

C. Formal Coverage Flow

Refer to Fig. 2 for the flow we use for a coverage-driven formal verification deployment. Like simulation, code coverage results are measured to identify missing gaps in the formal verification implementation. Abstraction models are used heavily to increase the coverage to acceptable levels on complex designs where formal would otherwise be infeasible.

Since we are using the same coverage metrics, we can even merge coverage results. It is often the case that one block is verified end-to-end with formal, and a larger block containing this block is verified with simulation. Even if the line coverage with formal is not 100% for the block, as long as the unified simulation and formal line coverage is 100%, verification is considered complete from the perspective of line coverage goals. This of course relies on an important assumption – *that the set of formal checkers is as complete as the set of simulation checkers*. Although formal coverage helps determine the quality of constraints as well as sequential depth reached, like simulation, coverage does not imply anything about the completeness of checkers. This has to be evaluated independently.

V. CellReformatter Design

The *Packet Rewrite Module* (PRM) design modifies incoming packets from multiple ports and reformats these packets before passing them on. Fig. 3 shows the sequence of operations on a packet when it passes through various stages of PRM. The four stages are Fragmentation (Stage #1),

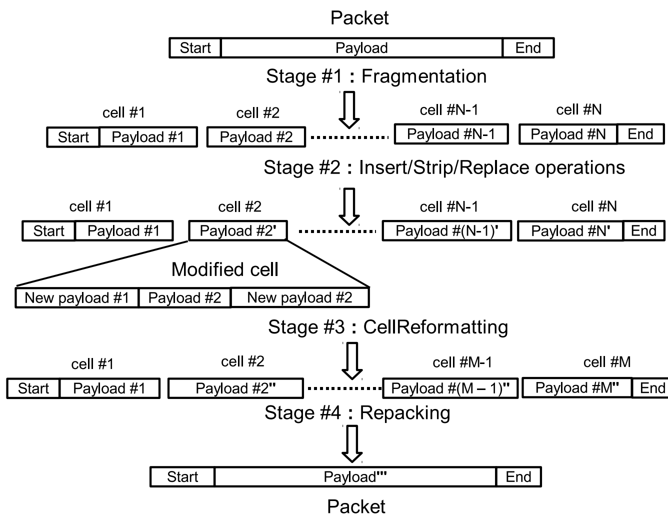


Fig. 3. Various stages of PRM

Insert/Strip/Replace operations on packet payload (Stage #2), CellReformatting (Stage #3) and Repacking (Stage #4).

A. Functional Specification

By the end of Stage #1, each packet is fragmented into single/multiple subpacket(s), called *cells*, depending upon the payload size. A cell has three main attributes: start of packet (SOP), end of packet (EOP) and number of payload bytes carried (ValidBytes). Some desired properties of the cells are:

1) The first and only the first cell has SOP as 1
2) The last and only the last cell has EOP as 1
3) A cell with EOP as 0 will have ValidBytes as 128
4) A cell will have ValidBytes greater than 0

e.g. As an example, suppose at the end of Stage #1, cell #1 has SOP as 1, EOP as 0 and ValidBytes as 128, cell #2 has SOP as 0 and EOP as 0 and ValidBytes as 128 and cell #N ($N = 3$) has SOP as 0, EOP as 1 and ValidBytes as 120.

Stage #2 modifies bytes of payload of a cell by performing insert, strip and replace operations. ValidBytes of each cell also gets modified accordingly. In Fig. 3, for the simplicity of illustration, we show that only cell #2 is being modified – i.e., the payloads of other cells do not undergo any change. Payload of cell #2 gets modified to payload #2' by insertion of two new payloads, one before, and one after the original payload, as depicted by Modified cell in the figure. In the actual design, Stage #2 can modify any or all N cells. With a combination of insert, strip and replace operations, ValidBytes of a modified cell can vary between 1 and 256. Suppose, in our example, after Stage #2, ValidBytes of cell #2 is 144, resulting in ValidBytes of 128, 144 and 120, respectively, for the three cells. Due to these modifications, a cell may not satisfy the desired properties on ValidBytes listed in the previous paragraph, at the end of Stage #2. The purpose of the next Stage #3, which constitutes our DUT, the CellReformatter design, is to rectify this.

CellReformatting (Stage #3) reformats the modified cells so that they satisfy the desired ValidBytes properties and can

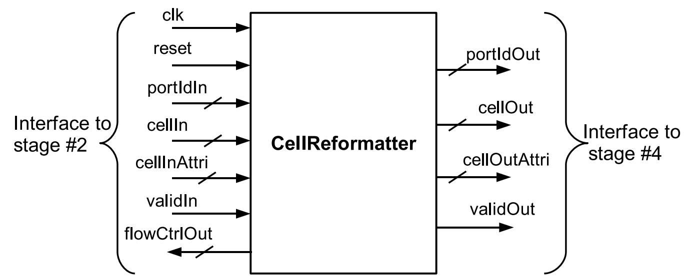


Fig. 4. Toplevel of CellReformatter

be repacked into a packet in the next stage. The number of cells for a packet at end of Stage #3 may be different than the number of cells at the beginning of the stage, depending upon reformatting. In our example, the payload of the non-EOP cell #2, at start of stage #3, does not satisfy the non-EOP ValidBytes property. So, in the CellFormatter stage, this cell gets reformatted to comprise of the first 128 bytes of the input cell. The remaining 16 ($= 144-128$) bytes are appended before the payload of cell #3, resulting a modified cell #3 of 128 bytes. The 8 ($= 120 + 16 - 128$) trailing bytes of the original cell #3 constitute a new cell #4.

Repacking (Stage #4) repacks the reformatted cells into a packet, that can be forwarded to port(s).

B. Micro-Architecture

CellReformatter supports reformatting of cells for packets from 56 different concurrent ports. Cells for a packet on one port may be interleaved with cells from other ports. This increases the design and verification complexity. Fig. 4 shows the interfaces of the CellReformatter design, the interface to Stage #2 on the left side, and the interface to the Stage #4 on the right side. *portIdIn* refers to incoming port. *cellIn* represents incoming cell, varying between 1 and 256 bytes long. *cellInAttri* is a structure consisting of cell attributes, including SOP, EOP, ValidBytes. *validIn* and *validOut* indicate the validity of inputs and outputs of CellReformatter respectively. Inputs are valid if they are transmitted when *validIn* is high. Similarly, outputs are valid if they arrive when *validOut* is high. *flowCtrlOut* is a feedback to Stage #2 to stop it from sending more cells for the relevant port. Thus this acts as a throttle and prevents the overflow of internal FIFO(s) for the port. *flowCtrlOut* is a 56-bit wide signal with each bit corresponding to a port.

Memory Design: CellReformatter has FIFOs for storing the reformatted cells (*dataFifo*) and its attributes (*statusFifo*). Each of *dataFifo* and *statusFifo* is implemented as an SRAM memory, with separate regions for different ports. The least-significant bit of *portId*, called *oddBank*, is used to determine which of the two banks is used, while the remaining higher-significant bits, called *streamId*, are used as memory address:

```
portId = {streamId, oddBank}
```

As shown in Fig. 5, the memory in each bank is logically divided into 28 *streamId*'s. Each bank of the *dataFifo* memory

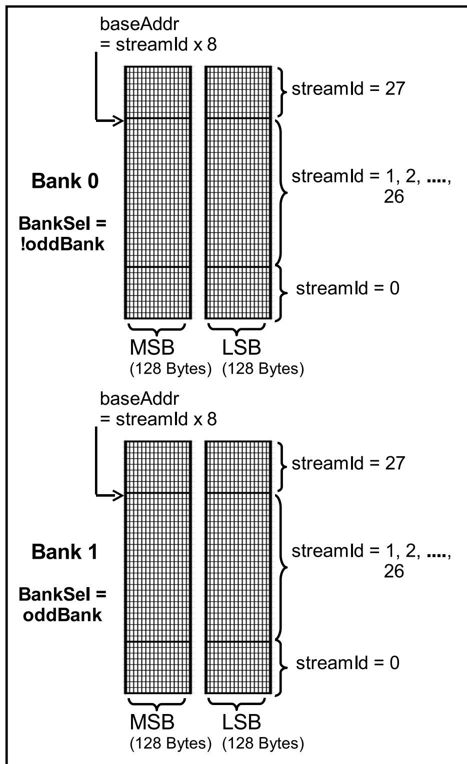


Fig. 5. Banked architecture of *dataFifo*

is further divided into two separate single-port SRAMs 128-bytes wide, called *MSB* and *LSB*. Further, each port occupies a depth of 8 entries in each of *MSB* and *LSB*. Note that in one clock at most 256 bytes will arrive from Stage #2 for a given port. Depending on where we wrote last for this port, this data will cause one or two writes into the *MSB* and/or the *LSB* section for that port. For the example in Section V-A, when cell #1 arrives, all of its 128 bytes are written into *LSB*, at depth of 0. When cell #2 arrives, 128 of its least significant bytes are written to *MSB* at depth of 0, and the remaining 16 bytes are written to *LSB* at depth of 1. Finally, when cell #3 arrives, its 112 ($= 128 - 16$) least significant bytes are shifted up by 16 bytes and written to *LSB* at depth of 1, and the remaining 8 ($= 120 - 112$) bytes are written to *MSB* at depth 1.

CellReformatter has another FIFO (*stateFifo*) for remembering the current write and read address pointers into *dataFifo* for a port. This FIFO is also implemented by a single-port two-bank SRAM memory.

Latency: The fastest end-to-end latency of CellReformatter is 6 clock cycles; the FIFO write operation has a 4-cycle latency and the FIFO read operation has a 3-cycle latency. A constraint on the design, that *oddBank* toggles every clock cycle, ensures that bank contention is avoided for simultaneous read and write operations.

C. Challenges to Formal

The major challenges to achieving convergence with formal are:

TABLE I
DESIGN SUMMARY OF CELLREFORMAT

Parameters	Values
Inputs	4,425
Outputs	3,488
Total flops	1,048,481

1) *Large number of flops*. Greater than 1 million storage elements (Table I) is enough to create a state space search problem that cannot be solved without the use of abstraction models. This large count is dominated by the number of flops needed for *dataFifo*: due to number of ports (56), number of per-port cells stored (16) and the size of each cell (128 bytes).
2) *High sequential depth due to latency*. No input port at input is allowed to appear more than once in 4 consecutive clock cycles. This constraint, along with the latency of CellReformatter and the FIFOs depths, implies that a high sequential depth is required for proofs.

VI. CHECKERS AND CONSTRAINTS

The CellReformatter design has following interface constraints:

1) For a port, between 2 cells at input with SOP as 1, there should be a cell with EOP as 1
2) For a port, between 2 cells at input with EOP as 1, there should be a cell with SOP as 1
3) For a port, the next valid cell after an EOP as 1 must have SOP as 1
4) For a port, input cell should have ValidBytes > 0
5) For a port, input cell should have ValidBytes < 256
6) The oddBank should toggle each cycle
7) A port at input should appear no more than once in 4 consecutive clock cycles

The interface checkers are as follows:

1) For a port, between 2 cells at output with SOP as 1, there should be a cell with EOP as 1
2) For a port, between 2 cells at output with EOP as 1, there should be
3) For a port, the next valid cell after an EOP as 1 must have SOP as 1
4) For a port, output cell should have ValidBytes > 0
5) For a port, output cell with EOP as 0 should have ValidBytes as 128

End-to-end checkers are written using a reference model that tracks the outstanding cells for a port, and also reformats them into 128-byte cell boundaries. Examples of end-to-end checkers:

1) For a port, the valid output (*validOut*) can be 1 only if there are outstanding cells in flight that have not been sent out

2) For a port, payload of a cell at the output should correspond to payload of expected cell in the reference model, computed based on payloads that arrived at the input in the past

Consider this last end-to-end checker, the most important checker for this DUT. The checker is written in SVA as:

```
property cellOutMatch_a;
 @(posedge clk) disable iff(reset)
  (validOut &&
   (portIdOut == watchedPort)) |->
  (cellOut[watchedByte][watchedBit] ==
   referenceBit);
endproperty
cellOutMatch_A:
 assert property(cellOutMatch_a);
```

We used the following symbolic variables in this checker:

1) *watchedPort*. This variable, varying between 0 and 55, represents the specific port that is being verified. While the design interleaves the inputs and outputs across multiple ports, in one trace, we can verify the outputs for a specific port.
2) *watchedByte*. This variable, varying between 0 and 127, represents the specific byte number in an output cell that is being verified in this trace.
3) *watchedBit*. This variable, varying between 0 and 7, represents the specific bit being verified in the *watchedByte* byte.

Since these variables are symbolic, all possible output data bits from all possible ports are verified with the end-to-end checker. In any given trace of execution, these variables can be kept constant with SVA constraints like the following:

```
property watchedPort_r:
 @(posedge clk) disable iff(reset)
  (##1 $stable(watchedPort));
watchedPort_R:
 assume property(watchedPort_r);
```

This end-to-end checker also depends on the predicted value of the output bit from the reference model, *referenceBit*. The reference model is implemented in SystemVerilog, and using the three watched symbolic variables, implementing a queue of watched bits in flight in the design. The value of *referenceBit* equals the bit at the top of the queue. We will discuss an abstraction in Section VII-B, that shows how to implement this reference model more efficiently.

VII. Abstraction Models

We have a design with more than 1 million state elements. This will lead to state space explosion with any existing formal verification tool. Abstractions are essential to achieve convergence on a design like this.

A. Memory Abstraction

The *dataFifo* memory stores up to 16 cells for every port, 8 cells in *LSB*, and 8 in *MSB*. The memory stores the reformatted

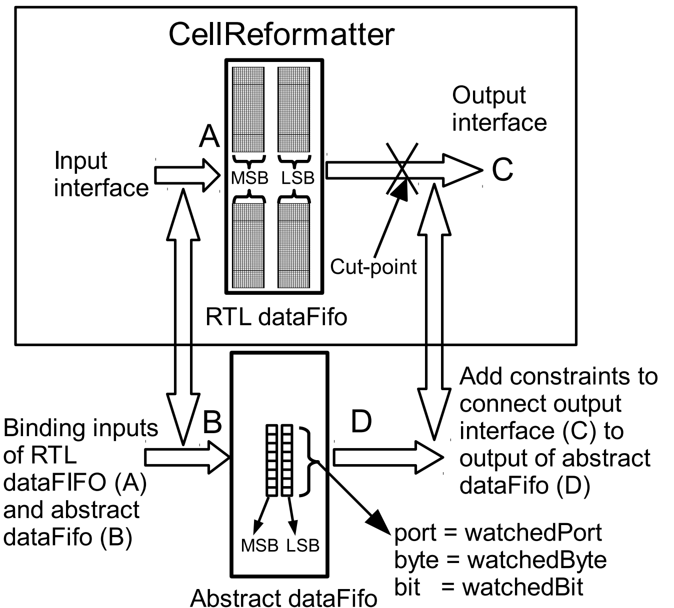


Fig. 6. Deploying memory abstraction for *dataFifo*

cells, after performing the necessary shifting, described in Section V-B. Since the main end-to-end checker (*cellOutMatch_A* in Section VI) uses symbolic watched variables for the port number and the verified bit in a cell, each flop in *dataFifo* is essential to establish the correctness of the proof. This places a tremendous burden on a formal verification tool.

Using the three 'watched' symbolic variables, we create an abstraction for *dataFifo*, shown in Fig. 6. This abstraction model contains only 16 flops, 8 for an abstraction of the *LSB* section of the memory banks, and 8 for an abstraction for the *MSB* section.

We tie the inputs of the abstract *dataFifo* to the inputs of the RTL *dataFifo* (implemented by the SystemVerilog bind construct). In addition, *watchedPort*, *watchedByte* and *watchedBit* are extra inputs to the abstract *dataFifo*.

When there is write to the RTL memory, if the write address input matches *watchedPort*, we pick the *watchedBit* bit from the *watchedByte* of the write data input to the memory, and store that in one of the 16 bits in the abstract memory (4 least significant bits of the write address input determine which of the 16 per-port cells was being written by the write command).

To enable the abstraction, we add cut-points at the read data outputs of the RTL *dataFifo*. Further, we add a constraint that if the read address input matches *watchedPort*, then *watchedBit* bit of *watchedByte* read data output byte equals the value stored in the i-th (of 16) abstract *dataFifo* bits (where i equals the 4 least significant bits in the read address input). This enables the read data for the watched bit to be faithful to what is in the RTL, and the remaining bits or read data output for a non-watched port to be arbitrary. But, since the checker is checking only the watched port and watched bit, the abstraction should not give a false negative.

Using this abstraction model, we have reduced 917,504 flops

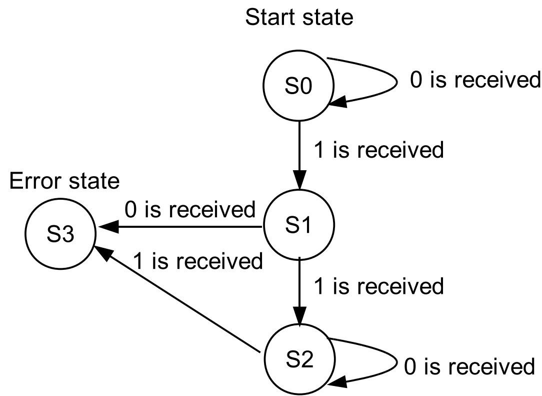


Fig. 7. State machine for pattern $0^{\star}110^{\omega}$ detection

in the RTL *dataFifo* that were in the cone-of-influence of the end-to-end checker to the 16 flops in the abstract *dataFifo*. More important, this abstraction does not introduce any false negatives with respect to the end-to-end checker. Similar abstraction models were built for *statusFifo* and *stateFifo*, albeit only with respect to *watchedPort*. See Table II for the reductions in the cone-of-influence; note that there are other peripheral flops in the memories because the memories have additional flops due to the latency, as well as some parity-checking flops.

Note that to complete the proof with abstractions using compositional reasoning, we also need to separately prove that the abstract *dataFifo* is a sound abstraction of the RTL *dataFifo*. We do this by removing the cut-points on the read data outputs, and reversing the constraints on the read data outputs to checkers, then proving them independently of the main end-to-end checker.

B. Data Independence Abstraction

The main end-to-end checker (*cellOutMatch_A* in Section VI) requires a reference model for the expected behavior of the *referenceBit* bit. Even after the memory abstraction in the previous section, we know that there are at least 16 bits in flight for the watched bit we want to track. However, this is just a lower bound, since there may be additional bits on the way to *dataFifo*, or on the way from *dataFifo*. Suppose there are at most n bits in flight we need to track; to implement the reference model with a FIFO, we will need at least n entries in the reference model FIFO.

Fortunately, we can use a variant of the data independence abstraction [20], to avoid the dependence on the unknown n, and more importantly to verify with more efficient state space. The data independence theorems state that for certain data-independent designs (when data is merely transported across the design, and not queried to make the routing decisions), a small set of finite data values is sufficient for end-to-end proofs. For our end-to-end checker, it is sufficient to prove the preservation of infinite streams of the form $0^{\star}110^{\omega}$ across the design. Each stream in this set has a finite but arbitrary number of 0's followed by two consecutive 1's, followed by an infinite sequence of 0's; for example, input sequences like $11000\cdots$, $011000\cdots$, and $000\cdots011000\cdots$. Note that given the three 'watched' symbolic variables, we need to apply this abstraction only to the consecutive bits that will be written to the abstract *dataFifo* from the previous section.

We add a constraint to the inputs of the DUT so that watched bits create this sequence by using the state machine in Fig. 7. We constrain the inputs so that the error state $S3$ is never reachable. Next, we use an identical state machine to verify the output watched bit from the DUT. We modify the checker so that the expected *referenceBit* is not allowed to be 0 is state $S1$, or to be 1 in state $S2$ – all other values are allowed for *referenceBit*.

Using this data indendence abstraction, we do not have to implement a reference FIFO, whose depth is design-dependent. We save additional flops in the cone-of-influence, and proofs run much faster.

VIII. Experimental Results

We used the Cadence® Incisive® Enterprise Verifier (IEV) tool [2] for this verification. Since the un-abstracted design has more than 1 million flops, hence it is not feasible to run formal without deploying the abstraction models described in Section VII.

The verification setup for the DUT consists of the CellReformatter RTL, checkers and constraints (using the necessary reference models), and the abstraction models described in Section VII. There are 23 checkers and 21 constraints. We found 15 bugs in the RTL design.

As expected, BMC was the most effective engine for verifying the main end-to-end checker. For the shortest possible packet, the data can be seen at the output of the design at a BMC proof depth of 7 clock cycles. However, the most interesting behavior of the design occurs when *dataFifo* is full before data is unloaded to the outputs. By understanding the design micro-architecture, including the latencies and memory depths, it was determined that a proof depth of 63 cycles is sufficient to hit this extreme behavior (the constraint that successive input data for the same port must be 4 cycles apart is responsible for much of this depth).

We use the IEV code coverage feature to report the amount of coverage hit to determine if the use of abstractions was successful in covering the design. Coverage results are reported in Table III. We notice that the expression coverage is 100% and the line coverage is almost 100% at a proof depth of 63. The missing coverage holes need to be judged and possibly waived by the design engineers. For the un-abstracted design, the BMC proof depths reached at similar run-times are close to 0, hence the corresponding coverage results are close to 0% (not surprising given the amount of state in the DUT).

The level of coverage reached is very much in line with the desired verification coverage, if we were verifying this design using simulation. We must remind the reader that the desired coverage result must be considered in conjunction with the confidence in the completeness of checkers. Unfortunately, as with simulation, formal code coverage by itself does not yet

TABLE II
COMPARISON OF RTL AND ABSTRACT MEMORIES

Memory	Flops in RTL	Flops in abstract memory
dataFifo	948,636	204
statusFifo	89,986	4,854
stateFifo	2,394	268

TABLE III
FORMAL COVERAGE RESULTS

Proof depth	Line coverage	Expression coverage
7	96.5%	100.0%
15	99.5%	100.0%
63	99.7%	100.0%

determine the completeness of checkers. However, we do know that with the use of the abstraction models, we were able to exercise almost all the RTL code. Without these abstraction models, we would not get much more than 0% coverage, and formal verification would not have been able to replace simulation on this design.

IX. CONCLUSION

In this work, we show how end-to-end formal can replace simulation efforts and provide faster verification with higher coverage. Without the use of abstraction models, formal verification is often infeasible for end-to-end verification. With the use of abstraction models, we can counter state space explosion, and reach acceptable levels of quantifiable code coverage metrics. These results can be integrated with simulation-based code coverage results on neighboring designs, or the rest of the system.

ACKNOWLEDGEMENT

The authors would like to thank Sandesh Borgaonkar, Anton Lopatinsky and Deepak Pant for their for help and support in making this work possible.

REFERENCES

[1] A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, Y. Zhu. Bounded model checking. *Advances in Computers*, 58, 2003.

[2] *Cadence Incisive Enterprise Verifier datasheet*. Cadence Design Systems, Inc.

[3] E. M. Clarke, O. Grumberg, D. E. Long. Model checking and abstraction. *ACM Trans. Program. Lang. Syst.*, 16(5), pp. 1512–1542, 1994.

[4] E. M. Clarke, O. Grumberg, H. Hiraishi, S. Jha, D. E. Long, K. L. McMillan, L. A. Ness. Verification of the Futurebus+ cache coherence protocol. *Formal Methods in System Design*, 6(2), pp. 217–232, 1995.

[5] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, H. Veith. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM* 50(5), pp. 752–794, 2003.

[6] A. Datta, V. Singhal. Formal Verification of a Public-Domain DDR2 Controller Design. In *Proc. VLSI Design*, pp. 475–480, 2008.

[7] C. N. Ip, D. L. Dill. Better verification through symmetry. *Formal Methods in System Design*, 9(1/2), pp. 41–75, 1996.

[8] *IEEE standard for SystemVerilog: unified hardware design, specification and verification language*. IEEE Std. 1800-2009.

[9] M. Kantrowitz, L. M. Noack. I'm done simulating; now what? Verification coverage analysis and correctness checking of the DECchip 21164 Alpha microprocessor. In *Proc. Design Automation Conf.*, pp. 325–330, 1996.

[10] S. Katz, O. Grumberg, D. Geist. Have I written enough properties? A method of comparison between specification and implementation. In *Proc. CHARME, LNCS 1703*, pp. 280–297, 1999.

[11] B. A. Krishna, A. Sullerey, A. Jain. Formal verification of an ASIC Ethernet switch block. In *Proc. FMCAD*, pp. 13–20, 2010.

[12] R. P. Kurshan. Formal verification in a commercial setting. In *Proc. Design Automation Conf.*, pp. 258–262, 1997.

[13] K. L. McMillan. Verificaton of an implementation of Tomasulo's algorithm by compositional model checking. In *Proc. CAV, LNCS 1497*, pp. 110-121, 1998.

[14] F. Pong, M. Dubois. A new approach for the verification of cache coherence protocols. *IEEE Trans. Parallel Distrib. Syst.* 6(8), pp. 773-787, 1995.

[15] C. Sayer, J. Sonander. Formal verification of AMBA 3 AXI bus systems. In *ARM Information Quarterly*, pp. 15-17, 4(2), 2005.

[16] V. Singhal, P. Aggarwal. Using Coverage to Deploy Formal in a Simulation World. In *Proc. CAV, LNCS 6806*, pp. 44-49, 2011.

[17] C. Stangier, U. Holtmann. Applying formal verification with Protocol Compiler. In *Proc. Euromicro Symp. Digital Systems Design*, pp. 165–169, 2001.

[18] S. Tasiran, K. Keutzer. Coverage metrics for functional validation of hardware designs. *IEEE Des. Test*, 18(4), pp. 36–45, 2001.

[19] C. Wang, G. D. Hachtel, F. Somenzi. *Abstraction refinement for large scale model checking*. Springer, 2006.

[20] P. Wolper. Expressing interesting properties of programs in propositional temporal logic. In *Proc. POPL '86*, pp. 184–193, 1986.

[21] P. Yeung. How to instrument your design with simple SystemVerilog assertions. *EE Times DesignLine*, January 26, 2011.

Specification Based Testing with QuickCheck

(Tutorial Talk)

John Hughes
Chalmers University of Technology and QuviQ AB

Abstract

QuickCheck is a tool which tests software against a formal specification, reporting discrepancies as minimal failing examples. QuickCheck uses *properties* specified by the developer both to generate test cases, and to identify failing tests. A property such as

$$\forall \texttt{xs} : \texttt{list}(\texttt{int}()).\ \texttt{reverse}(\texttt{reverse}(\texttt{xs})) = \texttt{xs}$$

is tested by generating random lists of integers, binding them to the variable xs, then evaluating the boolean expression under the quantifier and reporting a failure if the value is false. If a failing test case is found, QuickCheck "shrinks" it by searching for smaller, but similar test cases that also fail, terminating with a minimal example that cannot be shrunk further. In this example, if the developer accidentally wrote

$$\forall \texttt{xs} : \texttt{list}(\texttt{int}()).\ \texttt{reverse}(\texttt{xs}) = \texttt{xs}$$

instead, then QuickCheck would report the list $[0, 1]$ as the minimal failing case, containing as few list elements as possible, with the smallest absolute values possible. The approach is very practical: QuickCheck is implemented just a library in a host programming language; it needs only to execute the code under test, so requires no tools other than a compiler (in particular, no static analysis); the shrinking process "extracts the signal from the noise" of random testing, and usually results in very easy-to-debug failures.

First developed in Haskell by Koen Claessen and myself, QuickCheck has been emulated in many programming languages, and in 2006 I founded QuviQ to develop and market an Erlang version. Of course, customers' code is much more complex than the simple reverse function above, and requires much more complex properties to test it. The challenge in applying QuickCheck to real code is in finding ways to formulate properties that are simple enough for people to use easily, concise enough to make property-based testing cost-effective, and avoid the trap of replicating the mistakes of the implementation in the specification. To this end we have extended QuickCheck with state machine formalisms, and standardized serializability properties that can expose harmful race conditions in concurrent code. I will present examples using these formalisms, and discuss our experiences of applying property-based testing in the telecoms, automotive, and distributed database industries.

Short Biography

John Hughes began research in functional programming in 1980 as a D.Phil. student at the University of Oxford, graduating in 1983. He wrote *Why Functional Programming Matters* in 1985 while a post-doc at Chalmers University—a functional manifesto which is still one of the most widely read papers in the field. He took up a Chair at Glasgow University from 1985–1992, where he was a founder member of the Haskell design committee (and later its co-Chair for the Haskell 98 standard). In 1992 he moved to a Chair at Chalmers University, Gothenburg. He and Koen Claessen began work on QuickCheck in 1998—initially just for fun—and published the first paper on it in 2000. In 2006, he and Thomas Arts founded Quviq AB to commercialize the QuickCheck approach, and since then he has shared his time between Chalmers and Quviq. In 2010, the first QuickCheck paper received the ACM SIGPLAN award for the Most Influential Paper of ICFP 2000.

The Role of Human Creativity in Mechanized Verification

(Invited Talk)

J Strother Moore
University of Texas at Austin

ABSTRACT

In a presentation at FMCAD 1996 I decried industry's expectations that the creative insights of highly-paid, world-class hardware designers "should" be checkable by "push-button" tools. In the associated paper, my co-authors and I observed that unequivocal rejection of "lightweight" tools is impossible because of the role of "heavyweight thinking" in their use: problems that are impossibly large can often be rendered tractable by push-button means if the user is clever or persistent enough to create the right abstractions. The sensitivity of "tractability" to apparently minor modeling decisions is a well-known phenomenon for all of our tools. The decision not to model a certain bit or to avoid a certain form of definition, while appearing coincidental to the reader, may in fact be a crucial choice and we ought to highlight such decisions when we are aware of their importance. That we sometimes do not highlight them is not intellectual dishonesty but concern for clarity. Like mathematicians who revise a proof repeatedly for publication, the key insights are often lost as the presentation is polished. In this talk I again delve into the key question of the role of human creativity in mechanized verification. I argue that the more explicit we make that role, the better. Unlike mathematicians, we are fundamentally concerned with automating the methods of theorem discovery and proof. By highlighting the "minor decisions" that represent major breakthroughs in the problem, we serve our science better because we identify the key problems yet to be solved.

SHORT BIOGRAPHY

J Strother Moore holds the Admiral B.R. Inman Centennial Chair in Computing Theory at the University of Texas at Austin. He is the author of many books and papers on automated theorem proving and mechanical verification of computing systems. Along with Boyer he is a co-author of the Boyer-Moore theorem prover and the Boyer-Moore fast string searching algorithm. With Matt Kaufmann he is the co-author of the ACL2 theorem prover. Moore got his BS from MIT in 1970 and his PhD from the University of Edinburgh in 1973. Moore was a co-founder of Computational Logic, Inc., and served as its chief scientist for ten years. He and Bob Boyer were awarded the Current Prize in Automatic Theorem Proving by the American Mathematical Society in 1991 and they were awarded the Herbrand Award in 1999. In 2005, Boyer, Moore and Kaufmann won the ACM Software System Award, for the Boyer-Moore theorem prover. Moore served as chair of the UT Department of Computer Science from 2001 to 2009. Moore is a Fellow of the American Association for Artificial Intelligence, the ACM, and the National Academy of Engineering.

Interpolants from Z3 proofs

Kenneth L. McMillan Microsoft Research

***Abstract*—Interpolating provers have a number of applications in formal verification, including abstraction refinement and invariant generation. It has proved difficult, however, to construct efficient interpolating provers for rich theories. We consider the problem of deriving interpolants from proofs generated by the highly efficient SMT solver Z3 in the quantified theory of arrays, uninterpreted function symbols and linear integer arithmetic (AUFLIA) a theory that is commonly used in program verification. We do not directly interpolate the proofs from Z3. Rather, we divide them into small lemmas that can be handled by a secondary interpolating prover for a restricted theory. We show experimentally that the overhead of this secondary prover is negligible. Moreover, the efficiency of Z3 makes it possible to handle problems that are beyond the reach of existing interpolating provers, as we demonstrate using benchmarks derived from bounded verification of sequential and concurrent programs.**

I. Introduction

Interpolating provers have a number of applications in formal verification, including abstraction refinement [9] and invariant generation [17]. Given a valid implication $P \rightarrow Q$, an interpolating prover can produce an *interpolant* for the implication, that is, a formula I, expressed using the common vocabulary of P and Q, such that $P \rightarrow I$, $I \rightarrow Q$. There are various methods to accomplish this, but a common one is to extract the interpolant from a proof refuting $P \wedge \neg Q$, using an *interpolation calculus* [18]. Typically, one computes interpolants modulo a theory, for example, the theory of linear arithmetic over the integers. In this case, the interpolant is allowed to contain any interpreted symbols of the theory. The required proof may be obtained from a satisfiability modulo theories (SMT) solver, instrumented to produce proofs.

A significant practical difficulty with this approach is to obtain an efficient SMT solver that produces proofs in the required proof system. The theory solvers in a modern SMT solver are complex, in part due to the requirement of fast incremental operation to support backtracking, and in part due to the complexity of the theories themselves (for example, efficient solving of integer linear arithmetic constraints has long been a topic of research). Because of the difficulty of producing efficient proof-generating theory solvers, existing interpolating provers are typically less efficient than state-of-the-art SMT solvers, or do not support all of the desired theories. In practice, this inefficiency has been compensated somewhat by reducing the complexity of the input formulas, for example by considering only a single program execution path, as in [9], [19]. If interpolating solvers matching the performance of the best SMT solvers were available, however, it might be possible to use interpolation in a broader context, for example, considering more complex control flow, or perhaps concurrency.

In this paper, we consider the problem of deriving interpolants from proofs generated by the state-of-the-art SMT solver Z3 [8] in a rich theory, namely, the quantified theory of arrays and linear integer arithmetic (AUFLIA, according to the SMT-LIB nomenclature [2]).

Z3's proof calculus is complex, and rich enough to polynomially simulate proofs systems such as extended resolution that do not admit feasible interpolation[1]. Moreover, it allows "theory lemmas" that can introduce any validity of the theory without proof. Thus, for example, to refute a pair of complex formulas A and B, the proof system would allow a theory lemma that simply says $A \wedge B \rightarrow \text{FALSE}$. As a result, there is no reason in principle why a Z3 proof should contain sufficient information to construct an interpolant.

For this reason, we will take an approach that considers a Z3 proof as guide for construction of a proof by a secondary, less efficient, interpolating prover. We will translate Z3 proofs into a proof calculus that does admit feasible interpolation, with "gaps", or lemmas, that must be discharged by the secondary prover. This approach succeeds if the secondary prover can in practice discharge these lemmas in time that is small in relation to the time Z3 used to construct the original proof. A key test in this regard is the number of backtracks that the secondary solver must perform. If this is low, then the secondary solver need not have highly efficient incremental theory solvers. There is then no need to modify Z3 for the purpose of interpolant generation.

An additional benefit of this approach is that the secondary solver need not implement the entire theory. Our secondary solver implements only the *quantifier-free* theory of linear arithmetic and uninterpreted function symbols (QF_UFLIA). Quantifier instantiation is performed by Z3, as is instantiation of the axioms of the array theory. Thus, we can in principle use any of the available interpolating provers for QF_UFLIA as our secondary solver [3], [4].

To test these ideas, we use a collection of interpolation problems in AUFLIA, derived from bounded verification of sequential and concurrent programs using the Poirot tool [15]. Because these formulas have complex Boolean structure, they exploit the ability of Z3 to backtrack efficiently. We observe experimentally that interpolants can be efficiently derived from the Z3 proofs, while existing interpolating provers are unable to handle these formulas.

Related work Previous to this work there were no interpolating provers available for AUFLIA. A number of interpolating SMT solvers have been produced for subsets of this theory,

[1]Extended resolution proofs generalize resolution proofs by allowing resolution on arbitrary formulas, rather than just propositional atoms. This system is known, under cryptographic assumptions, not to admit feasible interpolation [13].

including Princess [3] (UFLIA), MathSAT4 [4] (QF_UFLRA) and SMTInterpol[2] (QF_LIA). Their performance is not comparable to Z3, as we will observe in in section V, using Z3 to instantiate the quantifiers and array axioms. Any solvers supporting QF_UFLIA can be used as the secondary solver in the present approach.

Interpolation has also been implemented in the first-order prover Vampire [10], however it is complete only in the ground case and applies only to rational (not integer) arithmetic. Moreover, it lacks an SMT solver's efficiency in combining Boolean and theory reasoning.

Interpolation in the theory of arrays has been handled in different ways. The method of [11] is based on discovery of local instantiations of the array axioms (a local predicate is expressed entirely in the vocabulary of A or the vocabulary of B). It is necessarily incomplete, but is guaranteed to produce quantifier-free interpolants for quantifier-free formulas. The present method is complete but may introduce quantifiers in the interpolants caused by non-local axiom instantiations. The method of [12] is similar to the present one in this respect. The primary difference is that it eagerly instantiates the array axioms, whereas here we rely on instantiations generated by Z3. Also, we should note that the present method is not specific to the array theory. It can handle any theory which Z3 handles by axiom instantiation (though it cannot in general handle axiom schemas).

In [5] an entirely different approach to arrays is taken, extending the signature of the array theory to allow quantifier-free interpolation. If such an approach were used in the secondary solver, we could safely discard the array axiom instances produced by Z3. In this way, the present method can either accommodate the weaknesses or exploit the strengths of the secondary interpolating prover.

A significant hurdle in interpolating proofs generated by SMT solvers is that interpolating proof calculi require the pivots of resolution steps to be local, but SMT solvers may for various reasons resolve on non-local predicates. In [6] a method is introduced to raise non-local pivots to the leaves of a resolution proof by re-ordering resolution steps. This method is worst-case exponential. Here we take a less general but linear-time approach that relies on knowledge of the structure of Z3 proofs. It is sufficient to raise resolutions on non-local pivots introduced by equational rewriting in Z3, which accounts for most cases of non-local resolution pivots. The remaining cases are handled by a different technique called "lemma extraction".

Overview of the paper In the next section, we cover some background including definitions and notations used in the paper. Section III introduces a simple proof calculus allowing feasible interpolation, while section IV describes our approach of translating Z3 proofs into this calculus. Section V then describes our experimental evaluation.

II. Background

We use standard first-order logic over a countable vocabulary Σ of function and predicate symbols, with associated arities. Function symbols with arity zero will be called constants. We will use t, u, v to represent first-order terms and ϕ, ψ, p, q and capital Roman letters to represent first-order formulas. We distinguish a finite subset Σ_I of Σ as *interpreted* symbols. In particular, we assume that Σ_I contains the binary predicate symbol $=$, representing equality. We assume a countable set $\mathcal{V}$ of variables, distinct from Σ. We will use x, y, z to represent variables. The vocabulary of a term or formula ϕ, denoted $L(\phi)$ is the set of *uninterpreted* function and constant symbols occurring in ϕ. If S is a vocabulary, we say $\mathcal{L}(S)$ is the set of first-order terms and formulas ϕ such that $L(\phi) \subseteq S$. We will also write $\mathcal{L}(\phi)$ for $\mathcal{L}(L(\phi))$ and $s \ll \phi$ to indicate that a symbol s occurs in ϕ.

A *theory* is a set of first-order formulas over Σ. We say a formula ϕ is valid relative to a theory $\mathcal{T}$ if every model of $\mathcal{T}$ is a model of ϕ, and we write this $\models_{\mathcal{T}} \phi$. We use capitol Greek letters Γ and Δ to stand for multisets of formulas. We will write a formula multiset as list of formulas and formula multisets. Thus, if Γ is a multiset of formulas and ϕ a formula, then Γ, ϕ represents $\Gamma \cup \{\phi\}$. We write $\wedge\Gamma$ for the conjunction of the formulas in Γ, $\vee\Gamma$ for the disjunction and $\neg\Gamma$ for the multiset of negations of formulas in Γ.

A *sequent* is written $\Gamma \vdash \Delta$, where Γ and Δ are multisets of formulas. Here, Γ is said to be the *antecedent* and Δ the *consequent*. We also call the elements of Γ *assumptions*. This sequent is *valid* if the conjunction of the formulas in Γ implies the disjunction of the formulas in Δ, given a background theory $\mathcal{T}$. That is, $\Gamma \vdash \Delta$ is valid if $\models_{\mathcal{T}} \bigwedge\Gamma \rightarrow \bigvee\Delta$. An empty antecedent or consequent will be represented by a blank. Thus $\vdash \phi$ means ϕ is valid, and $\phi \vdash$ means ϕ is a contradiction (implies the empty disjunction or FALSE). We will sometimes use calligraphic letters such as $\mathcal{J}$ to stand for sequents.

A formula or term is said to be *ground* if it contains no variables. A *position* π is a finite sequence of natural numbers, representing a syntactic position in a term or formula. If ϕ is a formula or term, then $\phi|_\pi$ represents the subformula or subterm of ϕ at position π. Thus, $\phi|_\epsilon$ is ϕ itself, $\phi|_i$ is the i-th argument of ϕ, $\phi|_{ij}$ is the j-th argument of the i-th argument, and so on. The notation $\phi[\psi]_\pi$ means ϕ with ψ substituted in position π.

An *interpolant* for a valid implication $A \rightarrow B$ is a formula I such that $A \rightarrow I$ and $I \rightarrow B$ are valid, and such that I is written using the vocabulary common to A and B, that is, $I \in \mathcal{L}(A) \cap \mathcal{L}(B)$. The Craig interpolation lemma [7] states that an interpolant always exists for a valid implication in first order logic (FOL). Validity in this definition may also be relative to a theory $\mathcal{T}$, though interpolants may not always exist in this case. When dealing with refutation systems, it is more convenient to speak of an interpolant for an unsatisfiable conjunction $A \wedge B$. An interpolant for the conjunction $A \wedge B$ is a formula $I \in \mathcal{L}(A) \cap \mathcal{L}(B)$ such that $A \rightarrow I$ and $B \rightarrow \neg I$ are both valid. In the sequel, let A and B be fixed formulas.

An *inference* is of the form

$$\frac{\mathcal{P}_1 \ \cdots \ \mathcal{P}_k}{\mathcal{C}}$$

where $\mathcal{P}_1 \cdots \mathcal{P}_k$ is a multiset of sequents called *premises*

[2] http://swt.informatik.uni-freiburg.de/research/tools/smtinterpol

(which we will often abbreviate $\{\mathcal{P}_i\}$) and $\mathcal{C}$ is a sequent called the *conclusion*. An inference is *sound* if validity of the premises implies validity of the conclusion. Generally, inferences are instances of *inference rules*, or patterns. Such a rule is sound when every instance matching the pattern, and satisfying any side conditions, is sound.

A *derivation tree* is a directed tree whose nodes are labeled with inferences. The premises of each node must contain the multiset of conclusions of its children. A derivation tree may be *open*, however, in the sense that some premises of inferences are not conclusions of any child. We call these unproved sequents the *premises* of the derivation tree. A tree with no premises is said to be *closed*. The conclusion of a derivation tree is the conclusion of its root node.

III. Interpolating proof calculi

We begin by introducing a very simple proof calculus and a corresponding *interpolation calculus* [18] that allows us to derive interpolants from proofs. Our eventual goal is to translate proofs from Z3 into this calculus.

We will say a formula is *local* when it is expressed either in the the vocabulary of A or in the vocabulary of B. A sequent is local when *all* its formulas are expressed in the the vocabulary of A, or *all* are expressed in the vocabulary of B. That is, $\Gamma \vdash \Delta$ is local when $\Gamma, \Delta \subseteq \mathcal{L}(A)$ or $\Gamma, \Delta \subseteq \mathcal{L}(B)$. We will say that a sequent is *strict* if each individual formula in the sequent is local, that is, if $\Gamma, \Delta \subseteq \mathcal{L}(A) \cup \mathcal{L}(B)$. Similarly, we will say that an inference or derivation tree is local (respectively strict) if all of its premises and conclusions are local (respectively strict). In writing proof rules, we will use the notation $\Gamma \vdash_l \Delta$ to indicate a local sequent and $\Gamma \vdash_s \Delta$ to indicate a strict sequent.

We should note that strictness is not an issue in purely propositional clausal proofs. Every clause in such a proof is necessarily strict because every propositional atom in it occurs in either A or B. In the current more general setting, non-strictness may occur either because of mixed terms within an atomic formula, or because the formulas in the sequent are not atomic.

The rules of our proof calculus $\mathcal{S}_P$ are as follows:

$$\textsc{Local}\ \frac{}{\Gamma \vdash_l \Delta}\ \ \models_{\mathcal{T}} \wedge\Gamma \rightarrow \vee\Delta$$

$$\textsc{Res}\ \frac{\Gamma \vdash_s \Delta, p \quad \Gamma \vdash_s \Delta', \neg p}{\Gamma \cup \Gamma' \vdash_s \Delta \cup \Delta'}$$

$$\textsc{Contra}(\Gamma)\ \frac{\Gamma, \Gamma' \vdash_s}{\Gamma' \vdash_s \neg\Gamma}$$

The first rule, Local, allows us to introduce any valid *local* sequent. As we will see, computing interpolations for local sequents is trivial. The second rule, Res, allows us to resolve two strict sequents on some pivot formula p. Note that resolving two local sequents might result in a strict but not local sequent, since the pivot p might be in both $\mathcal{L}(A)$ and $\mathcal{L}(B)$. Note also that the pivot p need not be an atomic formula. It is only required to be local. The third rule, Contra, allows us to move formulas Γ from the left- to the right-hand side of a strict sequent. That is, if assuming Γ entails a contradiction, then one of the formulas in Γ must be false. Notice that the rules of our system allow us to produce only strict sequents. The soundness of these rules is easily verified. Completeness is also easily shown for theories that have the Craig interpolation property, though this is not relevant to the current discussion.

Now, given a derivation of a sequent $A, B \vdash$, we would like to derive an interpolant for $A \wedge B$. We can do this using an interpolation calculus in the style of [18]. We sketch one such system here, though a detailed understanding of this system is not needed for what follows.

For any set Γ of formulas, we will write Γ_B for $\Gamma \cap \mathcal{L}(B)$ and Γ_A for $\Gamma \setminus \mathcal{L}(B)$ (note the asymmetry in these definitions). A sequent in the interpolation calculus (also called an *interpolation*) is of the form $(A, B) \vdash \Delta\ [\phi]$. The antecedent is a pair of formulas, A and B, the consequent is a multiset of formulas Δ and the formula ϕ acts as an interpolant for the sequent. The sequent is said to be *valid* when

1) A and $\neg\Delta_A$ imply ϕ,
2) B and $\neg\Delta_B$ imply $\neg\phi$, and
3) $\phi \in \mathcal{L}(A) \cap \mathcal{L}(B)$.

Another way to say this is that the interpolation is valid when ϕ is an interpolant for $A \wedge (\wedge\neg\Delta_A)$ and $B \wedge (\wedge\neg\Delta_B)$. Moreover, when Δ is the empty set, ϕ is an interpolant for $A \wedge B$. The set of interpolation rules $\mathcal{S}_I$, shown in Figure 1 is sound in the sense that they produce valid interpolations from valid interpolations. These rules can be interpreted roughly as follows. To interpolate a purely local sequent on the A side, we take the disjunction of the formulas of Δ that are in the common vocabulary of A and B. To interpolate a purely local sequent on the B side, we simply take True as the interpolant. If we resolve on an A-side formula, we take the disjunction of the interpolants, while resolving on the B side gives the conjunction. The rule for proof by contradiction has no effect on the interpolation.

Now suppose we have a derivation in system $\mathcal{S}_P$ of a sequent $A, B \vdash_s$. This is, we have proved that formulas A and B are inconsistent. We can transform this into a derivation of an interpolation $(A, B) \vdash [\phi]$ in the system $\mathcal{S}_I$. To do this, we replace each inference in the proof by a corresponding inference in $\mathcal{S}_I$, so that each sequent $\Gamma \vdash \Delta$ in the proof is replaced by an interpolation of the form $(A, B) \vdash \neg(\Gamma \setminus \{A, B\}),\ \Delta\ [\phi]$. That is, in the derived interpolation, the assumptions other than A and B are moved to the consequent side. As an example transformation step, if $\psi \in \mathcal{L}(A) \cap \mathcal{L}(B)$, and $\phi \in \mathcal{L}(A) \setminus \mathcal{L}(B)$, we have

$$\textsc{Local}\ \frac{}{A, \phi \vdash_l \psi} \quad \rightarrow \quad \textsc{LocalA}\ \frac{}{(A, B) \vdash \neg\phi \vee \psi\ [\psi]}$$

Because we move assumptions to the right in the translation, the Contra rule becomes particularly trivial. For example, we have:

$$\textsc{Contra}\ \frac{A, \psi \vdash_l}{A \vdash_l \neg\psi} \quad \rightarrow \quad \textsc{Contra}\ \frac{(A, B) \vdash \neg\psi\ [\phi]}{(A, B) \vdash \neg\psi\ [\phi]}$$

Note that our ability to replace each inference of $\mathcal{S}_P$ by a corresponding inference of $\mathcal{S}_I$ depends critically on the strictness and locality conditions in $\mathcal{S}_P$. For example, we can

$$\textsc{LocalA}\ \frac{}{(A,B)\vdash\Delta\ [\vee\Delta_B]}\ \wedge A\models_{\mathcal{T}}\vee\Delta,\ \Delta\subseteq\mathcal{L}(A)$$

$$\textsc{LocalB}\ \frac{}{(A,B)\vdash\Delta\ [\textsc{True}]}\ \wedge B\models_{\mathcal{T}}\vee\Delta,\ \Delta\subseteq\mathcal{L}(B)$$

$$\textsc{ResA}\ \frac{(A,B)\vdash\Delta,p\ [\phi]\quad(A,B)\vdash\Delta',\neg p\ [\phi']}{(A,B)\vdash\Delta\cup\Delta'\ [\phi\vee\phi']}\ p\in\mathcal{L}(A)\setminus\mathcal{L}(B)$$

$$\textsc{ResB}\ \frac{(A,B)\vdash\Delta,p\ [\phi]\quad(A,B)\vdash\Delta',\neg p\ [\phi']}{(A,B)\vdash\Delta\cup\Delta'\ [\phi\wedge\phi']}\ p\in\mathcal{L}(B)$$

$$\textsc{Contra}\ \frac{(A,B)\vdash\Delta\ [\phi]}{(A,B)\vdash\Delta\ [\phi]}$$

Fig. 1. Interpolation system $\mathcal{S}_I$.

always replace an instance of LOCAL by an instance of either LOCALA or LOCALB because the locality condition demands that $\Gamma\vdash\Delta$ is written in either $\mathcal{L}(A)$ or $\mathcal{L}(B)$.

IV. TRANSLATING Z3 PROOFS

Our goal in this section will be to convert proofs from Z3 into proofs in our simple proof calculus $\mathcal{S}_P$, and from there into our interpolation calculus $\mathcal{S}_I$ to obtain an interpolant.

Given a set of assumptions Γ that are inconsistent relative to a theory $\mathcal{T}$ that Z3 supports, it can produce a proof of a sequent $\Gamma\ \vdash$. However Z3's proof system is much richer than the simple one we have sketched. At present the system contains 38 documented rules. Many of these relate to particular theories that Z3 supports such as linear arithmetic and the theory of arrays. There is a also a rule, for example, for universal quantifier instantiation. The system also contains rules equivalent to our RES and CONTRA rules.

A very powerful rule in the Z3 system is the THLEMMA rule. This rule takes an arbitrary set of sequents as premises and can produce as a conclusion any sequent implied under one of Z3's theories. The theory solver may provide some hints as to how the proof should be performed, but in general complete proofs of theory lemmas are not provided.

To cope with this, our approach will be to construct a proof in $\mathcal{S}_P$ that is as detailed as possible, leaving unproved "lemmas" at the leaves of the derivation tree. To fit within our system, these lemmas must be strict. It will be the job of a secondary prover to provide interpolations for these lemmas. In the worst case, the proof might reduce to a single big lemma of the form $A,B\vdash$. In practice, though, we will observe that the lemmas tend to be small, and are easily handled by an interpolating prover much less efficient than Z3. Moreover, the lemmas never require quantifier instantiation or the theory of arrays, allowing us to use a secondary prover supporting only equality and integer arithmetic.

We approach the proof translation in several stages. The first stage, called *axiom elimination*, removes any non-local instances of axioms. In the next stage, *localization*, we find any possible applications of the LOCAL rule. Any closed sub-tree of the proof whose conclusion is local can be simply replaced by a single instance of the LOCAL rule. This typically removes a large fraction of the proof. In the last stage, *lemma extraction* we eliminate any inferences that are not available in $\mathcal{S}_P$. This is done by replacing sub-trees of the proof with lemmas to be interpolated by a secondary prover.

We now consider each of the proof translation stages in detail, beginning with the simplest, localization, and proceeding to lemma extraction and axiom elimination. We then cover a few additional optimizations. We will describe these transformations in terms of *replacement rules*, that is, substitutions of a sound derivation sub-tree by another sound derivation tree with the same premises and conclusion.[3]

A. Localization

Any local closed sub-tree of a derivation can be replaced by an instance of the LOCAL rule. We represent this by the following replacement rule:

$$*\ \frac{}{\Gamma\vdash_l\Delta}\quad\rightarrow\quad\textsc{Local}\ \frac{}{\Gamma\vdash_l\Delta}$$

We use the label $*$ here to indicate application of any number of sound inference rules. This rule says that any closed sub-tree using rules of the Z3 proof calculus whose conclusion is $\Gamma\vdash_l\Delta$ can be replaced with an instance of LOCAL with the same conclusion. A *maximal local sub-tree* is a local closed sub-tree that is not a sub-tree of any other local sub-tree. In the localization stage, we apply this replacement rule to all maximal local sub-trees.

B. Lemma extraction

Consider a sound (possibly open) sub-tree whose premises $\vdash_l p_1$ through $\vdash_l p_k$ are local and whose conclusion $\vdash_s\Delta$ is strict. This sub-tree demonstrates that the premises imply the conclusion, that is, the sequent $\vdash\neg p_1,\ldots,\neg p_k,\Delta$ is valid. We can thus introduce this as a lemma, using the following rule, which we add to $\mathcal{S}_P$ to allow introduction of any valid strict sequent:

$$\textsc{Lemma}\ \frac{}{\Gamma\vdash_s\Delta}\quad\models_{\mathcal{T}}\wedge\Gamma\rightarrow\vee\Delta$$

[3] We should also note that in the proof representation provided by Z3, the antecedents of sequents are not explicit. These can be reconstructed, however, by a preliminary pass over the proof structure.

Note that the LEMMA rule generalizes the LOCAL rule in that it allows a conclusion that is strict but not necessarily local.

Having introduced $\vdash \neg p_1, \ldots, \neg p_k, \Delta$ as a lemma, we can then resolve it with all the premises $\vdash p_i$ in turn to obtain the conclusion $\vdash \Delta$. This gives us a way to replace sub-trees with lemmas. This is important, as Z3 often sprinkles short segments of equality reasoning between resolution steps in its proofs. To express this transformation as a replacement rule, we will use the notation RES* to indicate multiple applications of the resolution rule. We then have the following replacement rule:

$$* \frac{\{\Gamma_i \vdash_s p_i\}}{\cup_i \Gamma_i \vdash_s \Delta} \quad \rightarrow$$

$$\text{RES}^* \frac{\text{LEMMA} \dfrac{}{\cup_i \Gamma_i \vdash_s \{\neg p_i\} \cup \Delta} \qquad \{\Gamma_i \vdash_s p_i\}}{\cup_i \Gamma_i \vdash_s \Delta}$$

We can make several improvements to this basic transformation. First, note that it requires all assumptions in the premises to be present in the conclusion. If this is not the case, we can rewrite a premise $\Gamma_i, \Gamma'_i \vdash_s p_i$ to $\Gamma_i \vdash_s (\wedge \Gamma'_i) \rightarrow p_i$, where Γ'_i are not assumptions in the conclusion, provided $(\wedge \Gamma'_i) \rightarrow p_i$ is local.

Moreover, assumptions in the conclusion can be dropped if they are not actually used in the sub-tree. The resulting lemma will still be valid. In fact, there is only one rule in the Z3 calculus that uses assumptions. This is the ASSUMP rule, introducing sequents of the form $\phi \vdash \phi$. If an assumption does not appear in an occurrence of ASSUMP within the sub-tree, it can be dropped from the lemma. Finally, we can use the LOCAL rule in the replacement instead of LEMMA if the lemma happens to be local, saving a lemma.

We will call the above transformation *lemma extraction*. Lemma extraction applies to any subtree that is strict, where the consequents of all premises are singletons[4]. We will call such a sub-tree *extractable*. Every node is contained in a unique minimum extractable subtree. This sub-tree can be found by moving up the tree to the first ascendant with a strict conclusion, then extending downward to the first descendant along each branch whose conclusion is strict and has a singleton consequent. Note that a minimum tree must exist containing any given node, because the conclusion of the root node of the tree is $A, B \vdash$ which is strict.

We wish to use lemma extraction to remove from the proof any inferences that do not occur in $\mathcal{S}_P$. The question is which sub-trees to transform into lemmas. Since we want the lemmas to be as small as possible, we will always extract minimal extractable subtrees.

We will say that an inference is *foreign* if it does not occur in $\mathcal{S}_P$. This can be because it uses a rule not present in $\mathcal{S}_P$, or because it does not meet the strictness condition. A derivation tree node is foreign if the inference labeling it is foreign. A foreign node is *maximal foreign* in a given derivation if it is not a strict descendant of any foreign node. In applying lemma extraction, we eliminate the minimal extractable sub-tree of some maximal foreign node. Note that this sub-tree contains the foreign node, but not always at the root. This process is repeated until no foreign nodes remain. Thus, lemma extraction proceeds from the root to the leaves of the proof tree, extracting the smallest possible lemmas. Of course it is conceivable that the root node is foreign, and the the minimal extractable sub-tree is the entire tree. In this case the entire proof reduces to one large lemma, and we have gained nothing. However, in practice we find that the extracted lemmas are quite small.

Finally, having introduced the LEMMA rule into our proof calculus, we require a corresponding interpolation rule:

$$\text{LEMMA} \frac{}{(A, B) \vdash \Delta \ [\phi]} \quad \dagger$$

The side condition † is that ϕ is an interpolant for $(\wedge \neg \Delta_A) \wedge A)$ and $(\wedge \neg \Delta_B) \wedge B$. This is just a statement of the condition for validity of the conclusion. We have no syntactic way of computing an interpolant for a lemma. Rather, we use the secondary interpolating prover to compute an interpolant ϕ for the formulas $((\wedge \neg \Delta_A) \wedge A)$ and $((\wedge \neg \Delta_B) \wedge B)$. Thus, each lemma we introduce by lemma extraction entails one call to the secondary prover.

C. Axiom elimination

Z3 uses a variety of axioms in its proofs. Instances of these axioms are introduced as conclusions of the form $\vdash \phi$ with no premises. If the secondary prover is unaware of these axioms (for example, it does not support the theory of arrays) then it is essential to capture the axiom instances in the Z3 proof using the LOCAL rule. Otherwise, the secondary prover may fail to prove a lemma.

Unfortunately, axiom instances are not always local. A prominent example of this is the axiom for universal quantifier instantiation:

$$\text{QUANTI} \frac{}{\vdash (\forall x.\ \phi) \rightarrow \phi[t/x]} \quad t \text{ is ground}$$

This says that a formula universally quantified over variable x implies the same formula under substitution of any ground term t for free instances of x. The difficulty with this rule is that t is an arbitrary ground term. Thus, the conclusion of the rule may not be local, even if ϕ is local.

We can, however, force an axiom instance to be local if it is truly needed, at the possible expense of adding quantifiers to the interpolant. To do this we add a fresh set of *localization symbols* $\mathcal{X}$ to Σ_I. That is, we take these symbols to be interpreted so they do not count as part of the vocabulary of a term and may always occur in interpolants. We assume a total, well-founded order $\prec$ on $\mathcal{X}$. We will write $s \doteq t$ to stand for an equation $s = t$ such that $s \in \mathcal{X}$ and t is a ground term such that for all symbols $s' \in \mathcal{X}$ occurring in t, $s' \prec s$. Such an equation will be called a *definition*. Note that the well-founded order prevents circular definitions.

[4]In fact it can be generalized to the case where the consequents of the premises are local multisets, though this has not been implemented and does not appear to be necessary in practice

We introduce the following rule to allow us to drop a definition no longer in use:

$$\textsc{Elim}\ \frac{s \doteq t, \Gamma \vdash_s \Delta}{\Gamma \vdash_s \Delta}\ \ s \not\preccurlyeq \Gamma, \Delta$$

Now consider an axiom $\phi[\cdot]$, with a placeholder to be filled by an arbitrary term of a given sort. Suppose that ϕ itself is local, but we are given an instance $\phi[t]$ that is not local. Let π be a highest local position in t. That is, π is a syntactic position in formula t such that $t|_\pi$ is local, but no higher position in t is local. We can eliminate this non-locality by choosing a fresh localization symbol s, defining $s \doteq t|_\pi$, and substituting s into position π in t. Note that for this to be legitimate, the symbol s must be greater in the order $\prec$ than any localization symbol occurring in $t|_\pi$.

We can apply this idea to localize axiom instances using replacements of the following form:

$$* \frac{\overline{\vdash \phi[t]} \quad \{\mathcal{J}_i\}}{\Gamma \vdash_s \Delta} \quad \to \quad \textsc{Elim}\ \frac{* \dfrac{\overline{\vdash \phi[t[s]_\pi]} \quad \{\mathcal{J}_i\}}{s \doteq t|_\pi, \Gamma \vdash_s \Delta}}{\Gamma \vdash_s \Delta}$$

That is, suppose we can prove some strict sequent $\Gamma \vdash_s \Delta$ from the axiom instance $\phi[t]$ and some other premises $\{\mathcal{J}_i\}$. If we *assume* the definition $s \doteq t|_\pi$, we can prove the same result from the alternative axiom instance $\phi[t[s]_\pi]$. This can be done by carrying the assumption up to the level of the axiom instance and applying substitution to yield the original formula $\phi[t]$. A definition elimination step is then used to remove the assumption. Note this inference is strict since the definition $s \doteq t|_\pi$ is constructed to be local. In this way we obtain the original conclusion $\Gamma \vdash_s \Delta$ from the altered axiom instance $\phi[t[s]_\pi]$.

By repeated applying this rule, we eventually reach the top position of t. At this point, we obtain $\phi[s]$, which is a local instance of the axiom. Thus it can be replaced with an instance of the Local rule. Note this procedure easily generalizes to axioms with multiple placeholders. We apply the above transformation to all the minimal closed strict sub-trees of the proof. The result is that all axiom instances are eliminated from the proof, hence the secondary prover need not be aware of these axioms.

Now, since we have introduced the Elim rule into our proof system, we must also introduce corresponding interpolation rules. These rules are as follows:

$$\textsc{ElimA}\ \frac{(A,B) \vdash s \doteq t, \Delta\ [\phi]}{(A,B) \vdash \Delta\ [\exists s.\ \phi]}\ s \not\preccurlyeq A, B, \Delta\ \ t \in \mathcal{L}(A) \backslash \mathcal{L}(B)$$

$$\textsc{ElimB}\ \frac{(A,B) \vdash s \doteq t, \Delta\ [\phi]}{(A,B) \vdash \Delta\ [\forall s.\ \phi]}\ s \not\preccurlyeq A, B, \Delta\ \ t \in \mathcal{L}(B)$$

Notice that eliminating a definition on the A side adds an existential quantifier to the interpolant, while eliminating a definition on the B side adds a universal. Also note that the side condition that s not occur in A or B is critical to the soundness of the rule. That is, if A implies ϕ and s does not occur in A, then A implies $\forall s.\ \phi$. Similarly, if A and $s = t$ imply ϕ and s does not occur in A, then A implies $\exists s.\ \phi$, with t providing the witness for the existential.

Finally, notice that in case multiple definitions are introduced, their order of elimination is the reverse of the order of introduction. Thus, definitions corresponding to larger terms produce the inner quantifiers.

In practice, we apply this transformation to three axioms: the quantifier instantiation axiom shown above, and the two standard axioms of the non-extensional array theory. In general, this method can be applied to any theory that is finitely axiomatizable in FOL, provided the prover provides the required axiom instances. However, it does not apply to axiom schemas (such as the congruence axiom schema for the theory of uninterpreted functions) because we cannot quantify over functions and predicates in FOL. Though the Elim rule introduces quantifiers in the interpolants, in practice these can often be eliminated using simple rules, for example, by replacing $\exists s.\ s = x \wedge \phi$ with $\phi[x/s]$.

D. Accounting for rewriting

One of the most common reasons that non-strict inferences occur in Z3 proofs is rewriting. That is, if resolution is only performed on predicates that occur in the original assumptions A and B, then only strict inferences can occur in a resolution tree. However, Z3 typically generates some non-local predicates by rewriting. That is, for some predicate p occurring in A or B, Z3 infers $p \Leftrightarrow p'$, where p' is not local, by rewriting p with some unconditional equations in A and B. The non-local predicate p' is then substituted for p, resulting in resolutions on non-local predicates. Since non-strict inferences result in larger lemmas, we would like to substitute the original p back in for p' in the proof to increase strictness.

There may be many possible ways to achieve this. We briefly sketch here one simple approach that has proved effective. We first scan the proof for any sequents of the form $\Gamma \vdash p \Leftrightarrow p'$, where $\Gamma \subseteq \{A, B\}$, p is local, and p' is not local. We can use this equivalence to push resolutions on p' towards the leaves of the derivation tree. From the equivalence $p \Leftrightarrow p'$, we can derived the two implications $p \to p'$ and $p' \to p$. Let $\textsc{Rpl}(p', p)$ be a shorthand for a derivation tree of the following form:

$$\textsc{Res}\ \frac{\Gamma \vdash \Delta, p' \qquad * \overline{\Gamma \vdash p' \to p}}{\Gamma \vdash \Delta, p}$$

That is, $\textsc{Rpl}(p, p')$ uses the implication $p \to p'$ to replace p with p'. Now we can replace any occurrence of resolution on p' using the following rule:

$$\textsc{Res}\ \frac{\Gamma \vdash \Delta, p' \quad \Gamma' \vdash \Delta', \neg p'}{\Gamma \cup \Gamma' \vdash \Delta \cup \Delta'} \quad \to$$

$$\textsc{Res}\ \frac{\textsc{Rpl}(p',p) \dfrac{\Gamma \vdash \Delta, p'}{\Gamma \vdash \Delta, p} \quad \textsc{Rpl}(\neg p', \neg p) \dfrac{\Gamma' \vdash \Delta', \neg p'}{\Gamma' \vdash \Delta', \neg p}}{\Gamma \cup \Gamma' \vdash \Delta \cup \Delta'}$$

That is, we eliminate a resolution on p' by replacing p' with the equivalent p and resolving on p. The resulting instances of Rpl can be pushed up the resolution tree by simply reordering resolutions. Here we show only one case (omitting

the RES labels to save space):

$$\dfrac{\dfrac{\Gamma \vdash \Delta, p', q \quad \Gamma' \vdash \Delta', \neg q}{\Gamma, \Gamma' \vdash \Delta, \Delta', p'} \quad * \dfrac{}{\Gamma, \Gamma' \vdash p' \to p}}{\Gamma, \Gamma' \vdash \Delta, \Delta', p} \quad \to$$

$$\dfrac{\dfrac{\Gamma \vdash \Delta, p', q \quad * \dfrac{}{\Gamma, \Gamma' \vdash p' \to p}}{\Gamma, \Gamma' \vdash \Delta, \Delta', p, q} \quad \Gamma' \vdash \Delta', \neg q}{\Gamma, \Gamma' \vdash \Delta, \Delta', p}$$

In this way, the resolutions on non-local atoms are pushed upward in the derivation tree until they meet a non-resolution inference. Since these resolutions are foreign they will eventually be eliminated by lemma extraction. By moving them upward in the derivation tree, we make the resulting minimal extractable sub-trees smaller and thus reduce the size of lemmas that must be proved by the secondary prover.

E. Accounting for sub-tree sharing

The proofs generated by Z3 are represented not as trees, but as DAG's. That is, in the proof representation it is possible (and in fact common) for two nodes to share children. Of course we must take care not to process shared sub-trees twice in the translation process. This is easily done for the localization step, which remains linear time in the proof size. Lemma extraction is quadratic on DAG-like proofs because the minimal extractable sub-trees of distinct foreign inferences can overlap. In practice, though, since these sub-trees tend to be small, this effect is insignificant. Axiom elimination is in principle also quadratic on DAG's, since the definitions needed to localize each axiom instance may need to be eliminated at many nodes in the DAG. If this is a problem in practice, it can be solved by placing all instances of ELIM at the root of the derivation tree. The method of Section IV-D is also linear time for DAG-like proofs.

F. Summary of interpolation procedure

To summarize, the translation from a Z3 proof of $A, B \vdash$ to an interpolant for $A \wedge B$ proceeds in the following steps. We first push resolutions on non-local atoms upward in the derivation by using proved equivalences with local atoms. Next we convert the axiom instances to local formulas. This involves introducing fresh defined symbols, which are later eliminated using the ELIM rule. The localization phase then eliminates all closed sub-trees with local conclusions (including axiom instances) using the LOCAL rule. Lemma extraction is then used to eliminate sub-trees that cannot be represented in $\mathcal{S}_P$. This phase introduces the LEMMA rule. The resulting proof is translated inference-by-inference into a derivation in the interpolation calculus $\mathcal{S}_I$. In this process, lemmas are interpolated by calls to the secondary prover. Quantifiers are introduced in translating the ELIM rule. The result is a derivation of $(A, B) \vdash [\phi]$ where ϕ is an interpolant for $A \wedge B$.

V. EXPERIMENTAL RESULTS

In this section, we describe some experiments to evaluate the efficiency of the above approach in practice.

Our implementation is written in C++, calling directly to Z3 via its API. We use version 2.19 of Z3 without modification. This is important because we do not wish to degrade the performance of Z3 in any way, except insofar as proof generation degrades performance. Except for proof generation, we use Z3 with default options. Our secondary prover is a simple SMT solver supporting QF_UFLIA and interpolation. It uses a standard Nelson/Oppen theory combination, with theory propagation. Linear arithmetic is handled by the Simplex algorithm, with a branch-and-cut approach for integer arithmetic. Interpolation is done using essentially the system of [18], with the addition of the DIV rule of [22] to handle Gomory cuts. In principle, however, any interpolating prover that handles QF_UFLIA can be used as the secondary prover.

For benchmarks, we need a set of problems that require the power of Z3, and at the same time are representative of a realistic application of interpolation. Unfortunately, existing benchmarks are either very simple or not realistic. Earlier evaluations, such as [14], have used either formulas involving a single program execution path, or synthetic benchmarks derived from arbitrarily partitioning formulas derived from SMT-LIB benchmarks into conjuncts A and B. The former are inappropriate because by construction they are too simple to test the performance of the solver, while the latter are inappropriate because of the arbitrary partitioning. Since the performance of our method depends on locality in the proof, a realistic partitioning is essential for evaluation. Moreover, we would like to evaluate the method on problems for which interpolation is actually relevant.

For these reasons, we instead use a set of benchmark interpolation problems derived from bounded verification of safety properties of sequential and concurrent programs. These formulas are generated by the tool Poirot [15]. This tool unwinds the loops in a program and in-lines procedure calls up to some determined bound. The result is a conjunction of formulas in AUFLIA, each of which represents the semantics of a single procedure instance, plus one additional constraint representing a standard background theory and containing quantifiers. The procedure instances form a tree, such that the children of any node represent the procedures called within that node. The formulas may represent an under-approximation of the program behavior, in which case the leaf procedures are replaced by the summary FALSE, or an over-approximation, in which case the leaves are replaced by the summary TRUE. The conjunction of the formulas is satisfiable when the given safety property fails in the given over- or under-approximation.

For our benchmarks, we use the under-approximations, which are typically unsatisfiable. We choose the sub-tree rooted at an arbitrary procedure instance as the A formula, and the remainder of the conjuncts as the B formula. An interpolant for this pair is a formula involving only symbols that represent the pre-state and post-state of this particular procedure instance. Note that this can in principle be a large set of symbols, since it can include symbols representing any global variables referenced in the procedure or any of its transitive callees. This can include symbols representing the state of the heap.

We can think of the interpolant for $A \wedge B$ as a potential

summary for the given procedure. It is guaranteed by the procedure instance and is sufficient to prove the given property in the given calling context. However, since the unwinding is approximate, this summary is also approximate. Nonetheless, it is possible that such approximate summaries can be used to construct true inductive summaries, as in [21], or to derive predicates for predicate abstraction, as in [9] or that they can themselves be used as approximations of procedures in further unwinding of the call tree.

For our purposes, the interest of these benchmarks is that, because they represent a large space of possible program executions, they cannot be easily solved by existing interpolating provers. To evaluate our method using these problems, we will measure two quantities: the overhead incurred in the interpolation process, relative to the run time of Z3, and the relative performance of our method compared to three existing provers. The former is easy to measure. The latter is made difficult by the fact that no interpolating provers exist that can handle the full AUFLIA theory.

To work around this problem, we will make things easier for the existing provers by providing them with the necessary quantifier instantiations and array axiom instances. We can do this by first applying axiom elimination to the Z3 proof, then applying lemma extraction to the entire proof tree, less the axiom instances. The result is an interpolation problem in QF_UFLIA. It should be kept in mind that the performance of a prover on this problem puts a lower bound on performance on the original problem, since the prover is relieved of the need to handle quantifier instantiation and the array theory.

The results are summarized in Table I. All run times are using one core of a 4-core 3.06 GHz Intel Xeon processor. Memory usage is limited to 2.5GB. The first column gives a name for the benchmark, the subscripts indicating different under-approximations and sub-trees. The "mouser", "serial" and "fdc" examples are safety properties of Windows device drivers from the Windows Static Driver Verifier [1]. The "ndisprot" and "wmm" examples are derived from threaded programs via the Lal/Reps construction [16], the latter using a weak memory model. The next two columns give the size of the A and B formulas in number of procedure instances (not considering those approximated by the summary FALSE). The next column shows the size of the Z3 proof in number of inferences. The next three columns show the Z3 run time, the interpolation time (including execution of the secondary prover) and the fractional overhead introduced by interpolation. The next column shows the number of lemmas produced.

Finally, the last three columns show the run times of three existing interpolating provers on the full problems plus quantifier and array axiom instantiations generated by Z3. Run times longer than 1800s are notated > 1800. Memory exhaustion is indicated by MEM. The MathSAT4 solver [4] supports quantifier-free linear rational and integer difference bound arithmetic. Though it does not support full LIA, we still find that it can handle the smaller problems (meaning these problems have no models in the LRA or difference bound theories). It is, however, one to two orders of magnitude slower than Z3 on these problems (note Z3 is handling quantifiers and array axioms, while MathSAT4 is not). On the larger problems, MathSAT4 exhausts memory. In two cases marked CRASH, MathSAT4 crashed. The Princess prover [3] handles UFLIA. Though in principle it can handle quantifiers, we nonetheless eliminated the quantifiers from the input formulas. Despite this, Princess failed to solve any problem within 1800s. We also tested the SMTInterpol solver, which supports QF_LIA, but this tool exhausted memory on all problems.[5]

We can make two general observations from these data. First, the overhead of interpolation relative to proof production in Z3 is small, and in fact is smaller on the larger proofs. This is in spite of the fact that the secondary prover is far less sophisticated than Z3. By dividing the proof into relative small lemmas, we have lessened the burden on the secondary prover to the point that run time is dominated by Z3.

Second, these problems are out of range for existing interpolating provers. Even with assistance provided by Z3 in instantiating quantified formulas and axioms, the best of the existing provers can handle only the smaller problems. By exploiting a state-of-the-art SMT solver, we have obtained a multiple order-of-magnitude performance improvement.

VI. Conclusion

In this work we have described an interpolating prover that is simultaneously as efficient as state-of-the-art SMT solvers and that handles the rich theory required by program verification. This was accomplished by using an efficient *proof-generating* SMT solver as a guide to a less efficient interpolating prover. By dividing the proof generated by Z3 into small lemmas, we create interpolation problems small enough for the interpolating prover to handle efficiently. In this way, we obtain a heuristically efficient interpolation procedure without requiring Z3 to produce proofs in a restricted system that allows feasible interpolation. In fact, the system does not depend on the specific set of proof rules used by Z3, with the exception of a few, such as RES and CONTRA. Thus, the Z3 proof system can potentially be expanded without any modification to the interpolation system. Moreover, any interpolating prover can be used as the secondary prover. This may allow a variety of interpolation methods to be used.

Evaluation on a set of benchmarks derived from program verification seems to indicate that the performance of an efficient solver such as Z3 can expand the range of application of interpolating provers beyond what was previously possible. This might in turn support new classes of interpolation-based algorithms for verification. One such class might be algorithms that analyze whole programs rather than program paths.

An interesting question to address in the future is how this method affects the quality of interpolants produced. Some methods have been proposed that, in effect, search the space of available proofs for one producing an interpolant satisfying certain criteria, with the goal of preventing the interpolants from diverging with deeper unwindings [11], [20]. It seems possible that using larger lemmas may allow greater flexibility to the secondary prover in constructing high quality interpolants. Thus a trade-off of performance and interpolant

[5]The benchmarks and scripts to run the provers are available at `http://www.kenmcmil.com/z3interp`.

Problem	Procedures		Proof	Time (s)		interp/Z3	Lemmas	Time (s)		
	A	B	size	Z3	interp			MathSAT4	Princess	SMTInterpol
mouserA$_1$	22	12	15864	0.098	0.010	0.102	5	0.986	> 1800	MEM
mouserA$_2$	1	34	24270	0.421	0.011	0.026	0	1.804	> 1800	MEM
mouserA$_3$	1	38	23331	0.232	0.008	0.034	0	1.718	> 1800	MEM
serial$_1$	111	23	69006	3.309	0.042	0.013	11	115.947	> 1800	MEM
serial$_2$	1	138	70341	3.375	0.039	0.012	0	121.928	> 1800	MEM
mouserB$_1$	456	12	253078	28.0	0.345	0.012	162	MEM	> 1800	MEM
mouserB$_2$	454	12	249548	29.4	0.276	0.009	176	MEM	> 1800	MEM
mouserB$_3$	1	468	269550	26.2	0.183	0.007	21	MEM	> 1800	MEM
fdc$_1$	148	5	115090	3.78	0.107	0.028	91	MEM	> 1800	MEM
fdc$_2$	1	153	114109	3.67	0.101	0.028	0	MEM	> 1800	MEM
fdc$_3$	1	155	115420	3.28	0.073	0.022	16	MEM	> 1800	MEM
ndisprot$_1$	1	29	31468	0.460	0.089	0.193	283	CRASH	> 1800	MEM
ndisprot$_2$	1	71	133863	5.61	0.208	0.037	0	CRASH	> 1800	MEM
wmm$_1$	1	2	15657	0.082	0.014	0.170	20	0.313	> 1800	MEM

TABLE I
RESULTS OF INTERPOLATION EXPERIMENTS ON POIROT FORMULAS.

quality might be achieved by adjusting the proof translation process.

Acknowledgments Thanks to Shaz Qadeer and Akash Lal for providing the formulas generated by Poirot, to Alberto Griggio for assistance with MathSAT4, to Philipp Rümmer for assistance with Princess, and to the anonymous reviewers for thorough and insightful comments.

REFERENCES

[1] Thomas Ball, Byron Cook, Vladimir Levin, and Sriram K. Rajamani. Slam and Static Driver Verifier: Technology transfer of formal methods inside Microsoft. In *IFM*, pages 1–20, 2004.

[2] Clark Barrett, Morgan Deters, Albert Oliveras, and Aaron Stump. Design and results of the 3rd annual satisfiability modulo theories competition (SMT-Comp 2007). *International Journal on Artificial Intelligence Tools*, 17(4):569–606, 2008.

[3] Angelo Brillout, Daniel Kroening, Philipp Rümmer, and Thomas Wahl. Beyond quantifier-free interpolation in extensions of Presburger arithmetic. In *VMCAI*, pages 88–102, 2011.

[4] Roberto Bruttomesso, Alessandro Cimatti, Anders Franzén, Alberto Griggio, and Roberto Sebastiani. The MathSAT 4 SMT solver. In *CAV*, pages 299–303, 2008.

[5] Roberto Bruttomesso, Silvio Ghilardi, and Silvio Ranise. Rewriting-based quantifier-free interpolation for a theory of arrays. In *RTA*, pages 171–186, 2011.

[6] Roberto Bruttomesso, Simone Rollini, Natasha Sharygina, and Aliaksei Tsitovich. Flexible interpolation with local proof transformations. In *ICCAD*, pages 770–777, 2010.

[7] W. Craig. Three uses of the Herbrand-Gentzen theorem in relating model theory and proof theory. *J. Symbolic Logic*, 22(3):269–285, 1957.

[8] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *TACAS*, pages 337–340, 2008.

[9] T. A. Henzinger, R. Jhala, R. Majumdar, and K. L. McMillan. Abstractions from proofs. In *POPL*, pages 232–244. ACM, 2004.

[10] Krystof Hoder, Laura Kovács, and Andrei Voronkov. Interpolation and symbol elimination in vampire. In *IJCAR*, pages 188–195, 2010.

[11] Ranjit Jhala and Kenneth L. McMillan. A practical and complete approach to predicate refinement. In *TACAS*, volume 3920 of *LNCS*, pages 459–473. Springer, 2006.

[12] Deepak Kapur, Rupak Majumdar, and Calogero G. Zarba. Interpolation for data structures. In *SIGSOFT FSE*, pages 105–116, 2006.

[13] J. Krajíček and P. Pudlák. Some consequences of cryptographical conjectures for S_2^1 and EF. *Information and Computation*, 140(1):82–94, January 1998.

[14] Daniel Kroening, Jérôme Leroux, and Philipp Rümmer. Interpolating quantifier-free presburger arithmetic. In *LPAR (Yogyakarta)*, pages 489–503, 2010.

[15] Akash Lal, Shaz Qadeer, and Shuvendu Lahiri. Corral: A whole-program analyzer for Boogie. Technical Report MSR-TR-2011-60, Microsoft Research, May 2011.

[16] Akash Lal, Tayssir Touili, Nicholas Kidd, and Thomas W. Reps. Interprocedural analysis of concurrent programs under a context bound. In *TACAS*, pages 282–298, 2008.

[17] K. L. McMillan. Interpolation and SAT-based model checking. In *CAV*, volume 2725 of *LNCS*, pages 1–13. Springer, 2003.

[18] K. L. McMillan. An interpolating theorem prover. *Theor. Comput. Sci.*, 345(1):101–121, 2005.

[19] K. L. McMillan. Lazy abstraction with interpolants. In *CAV*, volume 4144 of *LNCS*, pages 123–136. Springer, 2006.

[20] Kenneth L. McMillan. Quantified invariant generation using an interpolating saturation prover. In *TACAS*, pages 413–427, 2008.

[21] Kenneth L. McMillan. Lazy annotation for program testing and verification. In *CAV*, pages 104–118, 2010.

[22] P. Pudlák. Lower bounds for resolution and cutting plane proofs and monotone computations. *J. Symbolic Logic*, 62(2):981–998, June 1997.

Effective Word-Level Interpolation for Software Verification

Alberto Griggio [*]
Embedded Systems Unit – FBK-IRST – Trento, Italy.

***Abstract*—We present an interpolation procedure for the theory of fixed-size bit-vectors, which allows to apply effective interpolation-based techniques for software verification without giving up the ability of handling precisely the word-level operations of typical programming languages. Our algorithm is based on advanced SMT techniques, and, although general, is optimized to exploit the structure of typical interpolation problems arising in software verification. We have implemented a prototype version of it within the MATHSAT SMT solver, and we have integrated it into a software verification framework based on standard predicate abstraction. Our experimental results show that our new technique allows our prototype to significantly outperform other systems on programs requiring bit-precise modeling of word-level operations.**

I. Introduction and Related Work

Since the seminal paper of McMillan [1], (Craig) interpolation has been recognized to be a substantial tool for formal verification. In particular, one of its most successful applications is in the context of software verification based on counterexample-guided abstraction-refinement (CEGAR), where interpolants of quantifier-free formulas in suitable theories are computed for automatically refining abstractions in order to rule out spurious counterexamples [2], [3].

Most programming languages use a fixed amount of bits for representing values of primitive data types, such as integers. However, most interpolation-based software verification tools represent primitive types using mathematical integers or rational numbers, encoding program operations into e.g. a combination of linear arithmetic and uninterpreted functions. This results in loss of precision, which might not only lead to the generation of false alarms, in which correct programs are classified as incorrect, but also, and worse, to failures in detecting bugs. As a simple example, the code fragment `if (x > 0 && y > 0) { assert(x + y > 0); }` is wrongly classified as safe if variables are modeled using unbounded integers.

One of the main reasons for not using a more accurate modeling of program operations is the lack of effective interpolation procedures for the theory of bit-vectors (BV), that allow bit-precise representation of operations while retaining the advantages of reasoning at the word-level structure of problems. Although a significant amount of work has been done on interpolation procedures for several important theories (including theories of equality, linear arithmetic, data structures) [4], [5], [6], [7], [8], [9], [10], [11], interpolation for bit-vectors has received very little attention so far. To the best of our knowledge, the only complete interpolation algorithm for BV is based on naïvely mapping a bit-level propositional interpolant into BV (by replacing propositional variables with bit-extraction terms and Boolean connectives with bit-wise operations), which however completely destroys the word-level structure of the original problem, thus defeating all the benefits of reasoning at a level of abstraction higher than that of single bits. The first partial solution for this problem was proposed in [12], where an algorithm is given for constructing a word-level interpolant from a bit-level proof of unsatisfiability. The approach, however, was limited to equality logic only. A different direction is explored in [13], where a rewrite-based procedure for a fragment of BV is presented. The procedure is incomplete in general, but the authors show that their specialised rewrite rules are often enough for successfully verifying programs, in particular in the domain of device drivers.

In this paper, we present a novel, complete interpolation procedure for BV which tries to retain as much as possible the word-level structure of the input problem. Our approach is based on lazy/DPLL(T) SMT techniques for interpolation [7], and generates interpolants from DPLL(T)-proofs of unsatisfiability by combining a layered hierarchy of different interpolation procedures for conjunctions of BV-constraints, of increasing power and complexity, with a standard Boolean interpolation algorithm. Although general, the BV-interpolation layers are optimized for the kind of formulas arising in software verification, exploiting "definitional" equalities and interpolation procedures for linear integer arithmetic, and falling back to a bit-level algorithm when none of the specialised techniques can be applied.

We have implemented a prototype version of our procedure within the MATHSAT [14] SMT solver, and we have integrated it into a software verification framework based on standard predicate abstraction and interpolation-based refinement. Our prototype significantly outperforms other systems on programs requiring a bit-precise modeling of word-level operations, which could not be verified when using linear arithmetic and uninterpreted functions instead of BV.

Paper Outline. We introduce some background concepts in Sec. II. In Sec. III we describe a simple bit-level interpolation algorithm for BV. Our new procedure is described and

[*] Supported by Provincia Autonoma di Trento and the European Community's FP7/2007-2013 under grant agreement Marie Curie FP7 - PCOFUND-GA-2008-226070 "progetto Trentino", project ADAPTATION.

discussed in Sec. IV, and experimentally evaluated in Sec. V. We conclude in Sec. VI with directions for future work.

II. Background

A. Terminology and Notation

We work in the setting of standard first-order logic. We denote formulas with φ, ψ, A, B, I, variables with x, y, z, terms with s, t, predicates with p, q, possibly adding sub- and/or superscripts. As usual in the SAT and SMT community, we call 0-arity predicates *Boolean variables*. In the following, we only deal with quantifier-free formulas, in which all variables are implicitly existentially quantified. We use the standard definitions of theory, model, satisfiability, validity. If ψ is a logical consequence of φ in a theory T, we write $\varphi \models_T \psi$. If φ is unsatisfiable in T, we write $\varphi \models_T \bot$. A (fixed-width) bit-vector, or word, is a list of bits of fixed size n. We denote bit-vector terms of size n with $t_{[n]}$. We use the definition of the theory of bit-vectors, BV, given e.g. in [15]. BV-operators include sub-word selection $t_{[n]}[i:j]$ (where $0 \leq j \leq i \leq n$), concatenation $t_{1[n]} :: t_{2[m]}$, arithmetic operations (addition $+$, subtraction $-$, multiplication $\cdot$, signed and unsigned division $/_s$ and $/_u$, and signed and unsigned remainder $\%_s$ and $\%_u$), shifts ($<<$ and $>>$), and bitwise operations (bitwise *not* $\sim$, bitwise *and* $\&$, *or* $|$ and *xor* ^). BV-predicates include equality $=$ and signed and unsigned inequalities $\leq_s$ and $\leq_u$.

B. Interpolation for Software Verification

Given an ordered pair of formulas (A, B) in a theory T, a (Craig) interpolant is a formula I that satisfies the following constraints: (i) $A \models_T I$; (ii) $B \wedge I \models_T \bot$; and (iii) all the uninterpreted (in T) symbols occurring in I occur in both A and B (i.e., they are *AB-common*). Interpolants have important applications in software verification, and in particular in software model checking based on counterexample-guided abstraction-refinement (CEGAR) [16]. When using predicate abstraction, predicates for automatic abstraction refinement can be extracted from interpolants generated from formulas representing (sets of) spurious counterexamples (i.e. program paths leading to an error location which are feasible in the abstract space but infeasible in the concrete program) [2]. Similarly, interpolants from spurious counterexamples can also be used for directly representing and refining program abstractions without the need of computing predicate abstractions [3]. Both techniques proved to be quite effective, and are now implemented within many model checking tools (e.g. [17], [13], [18], [19], [20]).

C. Bit-Vectors in SMT

For many important theories T, the currently most-popular approach for checking the satisfiability of a formula φ in T, SMT(T), is the so-called "lazy" or "DPLL(T)" approach [21], in which a DPLL-based SAT solver is used for enumerating truth-assignments for the propositional skeleton of φ, which are then checked for T-satisfiability by a decision procedure for conjunctions of constraints in T (T-solver).

However, bit-vectors are an exception to this trend. Although several different algorithms have been proposed in recent years (see e.g. [15] for a survey), most current state-of-the-art SMT(BV)-solvers (e.g. [22], [23], [24], [25]) are based on (i) preprocessing the input BV-formula by applying several word-level simplification techniques followed by (ii) eagerly encoding the result of such preprocessing into a purely-propositional formula ("bit-blasting"), which is then given to an efficient SAT solver. In a nutshell, bit-blasting consists of encoding each bit-vector $t_{[n]}$ using n Boolean variables $p_0^t, \ldots, p_{n-1}^t$ representing its bits, and then translating each BV operation into an equivalent Boolean circuit, possibly introducing fresh auxiliary Boolean variables.

III. Simple Interpolation for BV

From the purely theoretical point of view, computing interpolants in the theory of bit-vectors is an easy problem. It is solved by a conceptually-simple algorithm, based on bit-blasting, which exploits the availability of off-the-shelf (and efficient) algorithms for interpolation for propositional logic (e.g. [1]).

Given a pair of BV-formulas A and B, an interpolant I for (A, B) can be generated from a propositional interpolant as follows.

- First, A and B are converted via bit-blasting into two purely-Boolean formulas A^p and B^p. For interpolation, it is important that A and B are bit-blasted using disjoint sets of auxiliary variables (see Sec. II-C).
- A^p and B^p are then converted to CNF[1] and given to an interpolating SAT solver, which checks the satisfiability of $A^p \wedge B^p$ and computes a propositional interpolant I^p for (A^p, B^p).
- By construction, I^p contains only variables that occur in both A^p and B^p, and so it can not contain auxiliary Boolean variables introduced by bit-blasting or CNF conversion. Each variable p_j^t in I^p then corresponds to a single bit j of a bit-vector term $t_{[n]}$ that occurs in both A and B. An interpolant I for (A, B) can therefore be obtained from I^p by replacing each variable p_j^t with the bit-extraction $t_{[n]}[j:j]$, and each Boolean connective with its corresponding bitwise operator (i.e. $\neg$ with $\sim$, $\wedge$ with $\&$ and $\vee$ with $|$).

This procedure is simple both to define and to implement. It has, however, an obvious and major drawback: it *completely destroys the word-level structure of the problem*, since it only generates interpolants as Boolean combinations of individual bits. Clearly, this completely defeats the benefits of reasoning at a higher level of abstraction, for instance by making it very difficult to apply effective word-level simplification techniques which are crucial for the efficiency of current state-of-the-art SMT solvers for BV [22], [26], [25], or to extract useful high-level information which can be effectively exploited in software verification, like "good" word-level predicates for abstraction refinement.

[1] CNF conversion might introduce more auxiliary "label" variables. As before, it is important that the sets of label variables for A^p and B^p are disjoint.

IV. A LAYERED APPROACH TO BV INTERPOLATION

The above reasons make the simple bit-level interpolation procedure of the previous section not very appealing in practice. Rather, we would like to obtain an interpolation procedure that retains as much as possible the word-level structure of formulas. At the same time, we would also like to keep the performance benefits of bit-blasting, which is still the dominant technique for SMT(BV), adopted by the most efficient state-of-the art solvers ([22], [26], [23], [24], [25]).

In the rest of the section, we present our solution to this problem. Its main idea is that of reducing the problem of interpolant generation for BV-formulas with an arbitrary Boolean structure to that of computing interpolants for *conjunctions of BV-constraints*, which can then be interpolated using a *layered approach*, by applying a hierarchy of different techniques which try to retain as much as possible the word-level structure of the input problem.

This reduction to dealing only with conjunctions of constraints is standard in interpolation for SMT when using the lazy/DPLL(T) approach, in which interpolants can be extracted from proofs of unsatisfiability consisting of a Boolean skeleton, to which a propositional interpolation algorithm is applied, and a set of T-inconsistent conjunctions of constraints, corresponding to negations of the T-lemmas occurring in the proof, which are handled by T-specific interpolation procedures [4], [7]. However, SMT solvers based on bit-blasting typically follow the eager approach, for which such reduction is harder to achieve, and to the best of our knowledge has been done only for equality logic [12]. Here, we exploit the combination of bit-blasting and DPLL(T), which allows us to generate proofs of unsatisfiability which can be easily partitioned into Boolean and T-specific parts while still retaining as much as possible the performance advantage of SAT encodings for solving BV-formulas. After having generated such proofs, we then tackle interpolation for the conjunctions of BV-constraints corresponding to the negated BV-lemmas in the proof using a layered hierarchy of four different techniques of increasing power and complexity.

A. Lazy bit-blasting in DPLL(T)

Lazy bit-blasting is a simple technique for integrating a decision procedure for BV based on SAT encoding within an SMT solver based on the DPLL(T) approach. It is the default strategy used by the MATHSAT SMT solver [14], a state-of-the-art solver for BV.[2]

The main idea of lazy bit-blasting is that of using two (DPLL-based) independent SAT solvers, $\text{DPLL}_{\text{Bool}}$ and DPLL_{BV}, organized in a hierarchy. $\text{DPLL}_{\text{Bool}}$ corresponds to the "DPLL" part of the standard DPLL(T) approach, whereas DPLL_{BV} takes the role of the T-solver. More precisely, when solving a BV-formula φ, $\text{DPLL}_{\text{Bool}}$ is used to reason on the Boolean skeleton of (the CNF conversion of) φ, like in the usual DPLL(T) approach, whereas DPLL_{BV} is used for checking the consistency of truth-assignments of BV-atoms enumerated by $\text{DPLL}_{\text{Bool}}$. This is done by exploiting the capability of modern SAT solvers of *reasoning under assumptions* [27], [28]. DPLL_{BV} is initialized by adding to it, for each BV-atom a occurring in φ, the clauses resulting from the bit-blasting of the formula $(l_a \leftrightarrow a)$, where l_a is a fresh Boolean variable, which we call the *label* for a. Notice that this means that the set of clauses in DPLL_{BV} is always satisfiable. When DPLL_{BV} is asked to check the consistency of a set of BV-literals $L_1, \ldots, L_n$ generated by $\text{DPLL}_{\text{Bool}}$, the corresponding labels $l_1, \ldots, l_n$ are added as temporary assumptions to DPLL_{BV} (if L_i is a negative literal, $\neg l_i$ is added as assumption instead of l_i). If the resulting formula becomes unsatisfiable, then it is possible to compute the (typically small) subset of assumptions $l_j, \ldots, l_k$ (some of which possibly negated) which is responsible for the inconsistency (see e.g. [28]). From this set, a BV-conflict set $L_j, \ldots, L_k$ is computed, whose negation $\neg L_j \vee \ldots \vee \neg L_k$ is a BV-lemma that is given back to $\text{DPLL}_{\text{Bool}}$ as usual in DPLL(T).

B. BV Interpolation via EUF layering

A good "side-effect" of using lazy bit-blasting is that it enables the use of *layering* of theory solvers. In particular, since BV-constraints are not bit-blasted at the main DPLL level, truth assignments can be checked using a solver for equality and uninterpreted functions (EUF) before invoking DPLL_{BV}. In this way, "cheap" conflicts that are due to the violations of equality axioms can be handled efficiently, without resorting to the potentially-expensive SAT checks in DPLL_{BV}. In such cases, interpolants can be computed by efficient existing algorithms for EUF, starting from the proofs of unsatisfiability generated by the EUF solver [4], [9]. This is therefore the first layer of our procedure.

Example 1: Consider the BV-interpolation problem:

$$\begin{aligned} A &\stackrel{\text{def}}{=} (x_{1[32]} = 3_{[32]}) \wedge (x_{3[32]} = x_{1[32]} \cdot x_{2[32]}) \\ B &\stackrel{\text{def}}{=} (x_{4[32]} = x_{2[32]}) \wedge (x_{5[32]} = 3_{[32]} \cdot x_{4[32]}) \wedge \\ &\quad \neg(x_{3[32]} = x_{5[32]}). \end{aligned}$$

In order to detect the unsatisfiability of $A \wedge B$, it is not necessary to take the precise semantics of the BV multiplication operation $\cdot$. In fact, a solver for EUF, which treats $\cdot$ as an uninterpreted function, is enough to construct a proof of unsatisfiability for $A \wedge B$. From such proof, the BV-interpolant $I \stackrel{\text{def}}{=} (x_{3[32]} = 3_{[32]} \cdot x_{2[32]})$ can be computed using an efficient algorithm for EUF interpolation. ◇

C. BV Interpolation via Equality Substitution

Equalities can still be exploited even when EUF is not enough for detecting unsatisfiability. As an example, consider an interpolation problem for an inconsistent pair (A, B) of formulas in which A is of the form $(x = e) \wedge \varphi$, x does not occur in e and x is the only non-common symbol between A and B. Then, it is easy to see that the formula obtained by replacing x with e everywhere in φ (denoted $\varphi[x \mapsto e]$) is an interpolant

[2]The latest version MATHSAT was the winner of the 2011 SMT competition on the "BV+uninterpreted functions" (QF_UFBV) and the "incremental BV" (QF_BV application) categories, and performed better than the winner of 2010 on the "plain BV" (QF_BV) category (see http://smtcomp.org/2011/).

for (A, B). Similarly, if it is B to be of the form $(x = e) \wedge \psi$, then $\neg\psi[x \mapsto e]$ is also an interpolant for (A, B). These two examples are just special cases of the well-known general algorithm for computing interpolants via quantifier elimination (for theories for which this is possible): roughly speaking, given an inconsistent pair (A, B) of formulas, an interpolant can be computed by performing the existential elimination of all the non-common variables either from A or from $\neg B$.[3] In general, however, existential quantification for BV can be quite expensive, it may require bit-blasting (and a consequent loss of word-level structure), and it may cause a blow-up in the size of the formula.

The idea of our second technique is that of detecting situations in which interpolation via existential elimination amounts to *performing substitutions using equalities*.

More in detail, given an inconsistent conjunction of BV-constraints partitioned into A and B, we remove from A a positive equality $(x = e)$ in which x is a variable that does not occur in e and x is not AB-common. We then replace x with e in the rest of the constraints of A, and repeat the process until either all the non-AB-common variables have been eliminated, or a fixpoint is reached. If the result A' of this procedure contains no non-AB-common variable, then we can return A' as an interpolant. Otherwise, we try eliminating non-common variables from B, obtaining B', and if this operation succeeds, we return $\neg B'$ as an interpolant. Notice that this procedure requires no satisfiability checks, and is therefore very cheap.

Example 2: Consider the BV-interpolation problem:

$$A \stackrel{\text{def}}{=} (0_{[32]} \leq_s (0_{[24]} :: x_{1[8]}) - 1_{[32]}) \wedge (x_{2[8]} = x_{1[8]})$$
$$B \stackrel{\text{def}}{=} (x_{3[8]} = (-(0_{[24]} :: x_{2[8]}))[7:0]) \wedge (x_{3[8]} = 0_{[8]}).$$

The unsatisfiability of $A \wedge B$ cannot be determined with the EUF layer alone. However, using equality substitution, we can easily compute an interpolant for (A, B). The only non-AB-common symbol in A is the variable $x_{1[8]}$, which can be eliminated by exploiting the equality $(x_{2[8]} = x_{1[8]})$, thus generating the BV-interpolant $I \stackrel{\text{def}}{=} (0_{[32]} \leq_s (0_{[24]} :: x_{2[8]}) - 1_{[32]})$. ◇

D. BV Interpolation via LIA Encoding

In the third layer of our procedure, we try to reduce the problem of generating interpolants for BV to the computation of interpolants in linear arithmetic over the integers (LIA).

In principle, the idea is similar to the reduction to propositional logic described in Sec. III: given an unsatisfiable conjunction of BV-constraints partitioned into A and B, the algorithm consists of: (i) generating two LIA-formulas A_{LIA} and B_{LIA} using the encoding described in [29], (ii) building an interpolant I_{LIA} for $(A_{\text{LIA}}, B_{\text{LIA}})$ using any off-the-shelf efficient interpolation algorithm for LIA (e.g. [10], [30], [11]), and finally (iii) "translating back" I_{LIA} in order to obtain a BV-interpolant I for (A, B). In practice, however, reduction to LIA presents several difficulties that do not occur in the case of reduction to propositional logic:

- First, from the theoretical point of view the problem of obtaining a BV-interpolant I from a LIA-interpolant I_{LIA} is non-trivial. In particular, in the translation of arithmetic operations and predicates from LIA to BV, issues like overflow or signed/unsigned semantics should be properly taken into account. (We shall give examples of some of the problems that may arise later in this section, after having given some details of the encoding of BV into LIA.)
- Moreover, from the practical point of view, encoding of BV constraints into LIA might result in *very challenging* SMT(LIA)-formulas, which might be out of reach of current state-of-the-art SMT(LIA)-solvers, even for BV-problems that current SMT(BV)-solvers can easily handle. This might happen especially when encoding BV-operations that require a "mixed LIA/bit-blasting" approach, like e.g. multiplication of two variables [29].

We address the above two issues by taking an *optimistic approach*. First, in the encoding of BV constraints into LIA, we abstract away all the operations that require a mixed LIA/bit-blasting approach, in order to reduce the likelihood of generating difficult SMT(LIA)-formulas, by simply encoding them with integer variables. Further, we set bounds (dependent on the size of the input problem) to the resources available (time and memory) for solving and constructing the proof of unsatisfiability of the SMT(LIA)-formula $A_{\text{LIA}} \wedge B_{\text{LIA}}$ resulting from the encoding of the BV-interpolation problem (A, B). If the formula $A_{\text{LIA}} \wedge B_{\text{LIA}}$ turns out to be satisfiable (because of the abstraction) or its unsatisfiability can not be determined within the resource bounds, we resort to the last layer of our procedure, described in Sec. IV-E. Otherwise, we compute an interpolant I_{LIA} for $(A_{\text{LIA}}, B_{\text{LIA}})$, and we translate it back to a BV-formula I using an *optimistic naïve approach* that essentially disregards overflow and signed/unsigned issues. We then check whether I is actually an interpolant for (A, B), by testing whether the BV-formula $(A \wedge \neg I) \vee (B \wedge I)$ is unsatisfiable,[4] again using bounded resources. If the check succeeds, then we return I as an interpolant. Otherwise, we resort to the last layer of our procedure.

In the rest of this section, we provide some details of the encoding and the construction of a BV-interpolant from a LIA-interpolant, giving also examples of why this might fail.

Encoding of BV constraints into LIA. We use the LIA encoding of BV constraints described, e.g., in [29]. Each BV term $t_{[n]}$ of n bits is encoded as a LIA variable x_t, together with the constraint

$$(0 \leq x_t) \wedge (x_t \leq 2^n - 1). \quad (1)$$

(In what follows, we shall denote (1) as $x_t \in [0, 2^n)$.) Each BV-operation/predicate is encoded as a Boolean combination of LIA constraints, possibly introducing some auxiliary LIA

[3] It can be observed that this algorithm always generates the strongest or the weakest interpolant (wrt. logical implication) for (A, B), the former when starting from A, the latter when starting from $\neg B$.

[4] As described later in this section, I contains only AB-common symbols by construction.

Selection: $t_{[i-j+1]} \stackrel{\text{def}}{=} t_{1[n]}[i:j]$ becomes $(x_t = m) \wedge (x_{t_1} = 2^{i+1}h + 2^j m + l) \wedge l \in [0, 2^i) \wedge m \in [0, 2^{i-j+1}) \wedge h \in [0, 2^{n-i-1})$, where h, m, l are fresh.
Concatenation: $t_{[n+m]} \stackrel{\text{def}}{=} t_{1[n]} :: t_{2[m]}$ becomes $(x_t = 2^m x_{t_1} + x_{t_2})$.
Addition: $t_{[n]} \stackrel{\text{def}}{=} t_{1[n]} + t_{2[n]}$ becomes $(x_t = x_{t_1} + x_{t_2} - 2^n\sigma) \wedge (0 \leq \sigma) \wedge (\sigma \leq 1)$, where σ is fresh.
Multiplication by constant: $t_{[n]} \stackrel{\text{def}}{=} t_{1[n]} \cdot k$ becomes $(x_t = k \cdot x_{t_1} - 2^n\sigma) \wedge (0 \leq \sigma) \wedge (\sigma \leq k)$, where σ is fresh.
Left-shift by constant: $t_{[n]} \stackrel{\text{def}}{=} t_{1[n]} << k$ becomes $(x_t = 2^k x_{t_1[n-k-1:0]})$, where $x_{t_1[n-k-1:0]}$ is the LIA variable for the selection $t_1[n-k-1:0]$

Equality: $(t_{1[n]} = t_{2[n]})$ becomes $(x_{t_1} = x_{t_2})$.
Unsigned $\leq$: $(t_{1[n]} \leq_u t_{2[n]})$ becomes $(x_{t_1} \leq x_{t_2})$.
Signed $\leq$: $(t_{1[n]} \leq_s t_{2[n]})$ becomes $ITE(((x_{t_1} \leq 2^{n-1}-1) \wedge (x_{t_2} \leq 2^{n-1}-1)) \vee ((x_{t_1} \geq 2^{n-1}) \wedge (x_{t_2} \geq 2^{n-1})), (x_{t_1} \leq x_{t_2}), (x_{t_1} \geq 2^{n-1}))$, where $ITE(c,t,e)$ is a shorthand for $(c \rightarrow t) \wedge (\neg c \rightarrow e)$.

Fig. 1. LIA encoding of some BV operations and predicates.

and Boolean variables. Some examples are shown in Fig. 1.[5] As already mentioned before, for performance reasons we abstract BV-terms t requiring a mixed LIA/bit-blasting encoding [29] (i.e. non-linear multiplication/division/remainder, bit-wise operations between two non-constant terms, and shift by a non-constant term) by simply using the corresponding LIA-variable x_t, without additional constraints (other than (1)). As in the bit-blasting approach (see Sec. III), we assume that when generating an interpolant for a pair of formulas (A, B), A and B are encoded using disjoint sets of auxiliary variables.

Constructing a BV-interpolant from a LIA-interpolant. Current interpolation algorithms for LIA allow to produce interpolants in an extension of LIA with either divisibility predicates [10] or with the floor function $\lfloor\cdot\rfloor$ [11].Once a LIA-interpolant I_{LIA} (in either of the two extended signatures) for the encoded pair of formulas $(A_{\text{LIA}}, B_{\text{LIA}})$ has been generated, we translate it back to a *candidate BV-interpolant* I for the original pair (A, B), by replacing LIA-variables with the corresponding BV-variables, LIA-numbers with their BV-encoding, and LIA-operations with the corresponding BV-operations. More formally, we proceed as follows.

1) Let β be a (partial) mapping from LIA-terms/predicates to BV-terms/predicates. β is initialized by setting, for every LIA-variable x_t occurring in I_{LIA}, $\beta(x_t)$ to the corresponding BV-term t. Notice that, similarly to the case of interpolation via bit-blasting, I_{LIA} contains no auxiliary variables, since A and B were encoded using disjoint sets of such variables.
2) Integer constants k occurring in I_{LIA} are mapped to BV constants using a 2's complement representation. The size of the target BV-constant $\beta(k)$ is determined by examining the context in which k occurs. In particular, if k is the argument of a binary LIA-term or predicate $t \bowtie k$, we encode k with a bit-vector of the same width n as $\beta(t)$, [6] simply truncating if k does not fit in n bits.
3) LIA-additions and multiplications are mapped to BV-additions and multiplications respectively, without considering potential overflow issues. (If the bit-width of $\beta(t_1)$ and $\beta(t_2)$ are different, we extend the shortest of the operands by padding it with zeros).
4) For floor terms $\lfloor \frac{t}{k} \rfloor$, where k is a positive constant,[7] $\beta(\lfloor \frac{t}{k} \rfloor)$ is set to $\beta(t)/_u\beta(k)$.
5) LIA-equalities $(t_1 = t_2)$ and inequalities $(t_1 \leq t_2)$ are mapped to BV-equalities $(\beta(t_1) = \beta(t_2))$ and unsigned inequalities $(\beta(t_1) \leq_u \beta(t_2))$ respectively.
6) For divisibility predicates $k|t$, where k is a positive constant, $\beta(k|t)$ is set to the BV-equality $(\beta(t)\%_u\beta(k) = 0_n)$, where n is the bit-width of $\beta(t)$.
7) We construct I from I_{LIA} by replacing each LIA-atom a occurring in I_{LIA} with $\beta(a)$.

Example 3: Consider the BV-interpolation problem:

$$A \stackrel{\text{def}}{=} (y_{1[8]} = y_{5[4]} :: y_{5[4]}) \wedge (y_{1[8]} = y_{2[8]}) \wedge (y_{5[4]} = 1_{[4]})$$
$$B \stackrel{\text{def}}{=} \neg(y_{4[8]} + 1_{[8]} \leq_u y_{2[8]}) \wedge (y_{4[8]} = 1_{[8]}).$$

Encoding A and B into LIA (see Fig. 1) results in the following:

$$\begin{aligned} A_{\text{LIA}} \stackrel{\text{def}}{=} & (x_{y_2} = 16x_{y_5} + x_{y_5}) \wedge (x_{y_1} = x_{y_2}) \wedge (x_{y_5} = 1) \wedge \\ & (x_{y_1} \in [0, 2^8)) \wedge (x_{y_2} \in [0, 2^8)) \wedge (x_{y_5} \in [0, 2^4)) \\ B_{\text{LIA}} \stackrel{\text{def}}{=} & \neg(x_{y_4+1} \leq x_{y_2}) \wedge (x_{y_4+1} = x_{y_4} + 1 - 2^8\sigma) \wedge \\ & (x_{y_4} = 1) \wedge \\ & (x_{y_4+1} \in [0, 2^8)) \wedge (x_{y_4} \in [0, 2^8)) \wedge (0 \leq \sigma \leq 1) \end{aligned}$$

$A_{\text{LIA}} \wedge B_{\text{LIA}}$ is LIA-inconsistent, and an interpolant for $(A_{\text{LIA}}, B_{\text{LIA}})$ is $I_{\text{LIA}} \stackrel{\text{def}}{=} (17 \leq x_{y_2})$. Using β, we obtain the formula $I \stackrel{\text{def}}{=} (17_{[8]} \leq_u y_{2[8]})$, which is a BV-interpolant for (A, B). ◇

Notice that, since we encoded A and B using disjoint sets of auxiliary variables, a formula I generated via β from a LIA-interpolant I_{LIA} is guaranteed to fulfill the third condition of the definition of interpolant. However, I is *not guaranteed* to be an interpolant for (A, B), as shown by the following examples.

Example 4: Consider the BV-interpolation problem:

$$\begin{aligned} A \stackrel{\text{def}}{=} & (y_{2[8]} = 81_{[8]}) \wedge (y_{3[8]} = 0_{[8]}) \wedge (y_{4[8]} = y_{2[8]}) \\ B \stackrel{\text{def}}{=} & (y_{13[16]} = 0_{[8]} :: y_{4[8]}) \wedge \\ & (255_{[16]} \leq_u y_{13[16]} + (0_{[8]} :: y_{3[8]})) \end{aligned}$$

[5] Several optimizations are possible, but they are not discussed here. We refer to [29] for the full details.

[6] Notice that we can always assume w.l.o.g. that I_{LIA} is normalized such that all integer constants occur as argument of binary terms/predicates in which the other argument is not a constant.

[7] Notice that the interpolation procedure of [11] always produces floor terms of this form.

and its LIA-encoding:

$$A_{\text{LIA}} \stackrel{\text{def}}{=} (x_{y_2} = 81) \wedge (x_{y_3} = 0) \wedge (x_{y_4} = x_{y_2}) \wedge$$
$$(x_{y_2} \in [0, 2^8)) \wedge (x_{y_3} \in [0, 2^8)) \wedge (x_{y_4} \in [0, 2^8))$$
$$B_{\text{LIA}} \stackrel{\text{def}}{=} (x_{y_{13}} = 2^8 \cdot 0 + x_{y_4}) \wedge (255 \leq x_{y_{13+(0::y_3)}}) \wedge$$
$$(x_{y_{13+(0::y_3)}} = x_{y_{13}} + 2^8 \cdot 0 + x_{y_3} - 2^{16}\sigma) \wedge$$
$$(x_{y_{13}} \in [0, 2^{16})) \wedge (x_{y_{13+(0::y_3)}} \in [0, 2^{16})) \wedge$$
$$(0 \leq \sigma \leq 1).$$

A LIA-interpolant for $(A_{\text{LIA}}, B_{\text{LIA}})$ is $I_{\text{LIA}} \stackrel{\text{def}}{=} (x_{y_3} + x_{y_4} \leq 81)$. However, the formula $I \stackrel{\text{def}}{=} \beta(I_{\text{LIA}}) \stackrel{\text{def}}{=} (y3_{[8]} + y4_{[8]} \leq_u 81_{[8]})$ *is not* an interpolant for (A, B), because $I \wedge B \not\models_{\text{BV}} \bot$. The problem is that in BV, addition might overflow. In fact, if we make sure this does not happen in I, then we obtain a correct interpolant I' for (A, B):

$$I' \stackrel{\text{def}}{=} ((0_{[1]} :: y3_{[8]}) + (0_{[1]} :: y4_{[8]}) \leq_u 81_{[9]}).$$

◇

The above example shows that, in the translation from I_{LIA} to I, overflows are an issue. However, they are not the only problem that might arise.

Example 5: Consider the BV-interpolation problem:

$$A \stackrel{\text{def}}{=} \neg(y4_{[8]} + 1_{[8]} \leq_u y3_{[8]}) \wedge (y2_{[8]} = y4_{[8]} + 1_{[8]})$$
$$B \stackrel{\text{def}}{=} (y2_{[8]} + 1_{[8]} \leq_u y3_{[8]}) \wedge (y7_{[8]} = 3_{[8]}) \wedge$$
$$(y7_{[8]} = y2_{[8]} + 1_{[8]})$$

and its LIA-encoding:

$$A_{\text{LIA}} \stackrel{\text{def}}{=} \neg(x_{y_4+1} \leq x_{y_3}) \wedge (x_{y_2} = x_{y_4+1}) \wedge$$
$$(x_{y_4+1} = x_{y_4} + 1 - 2^8\sigma_1) \wedge$$
$$(x_{y_2} \in [0, 2^8)) \wedge (x_{y_3} \in [0, 2^8)) \wedge (x_{y_4} \in [0, 2^8)) \wedge$$
$$(x_{y_4+1} \in [0, 2^8)) \wedge (0 \leq \sigma_1 \leq 1)$$
$$B_{\text{LIA}} \stackrel{\text{def}}{=} (x_{y_2+1} = x_{y_3}) \wedge (x_{y_7} = 3) \wedge (x_{y_7} = x_{y_2+1}) \wedge$$
$$(x_{y_2+1} = x_{y_2} + 1 - 2^8\sigma_2) \wedge$$
$$(x_{y_7} \in [0, 2^8)) \wedge (x_{y_2+1} \in [0, 2^8)) \wedge (0 \leq \sigma_2 \leq 1).$$

A LIA-interpolant for $(A_{\text{LIA}}, B_{\text{LIA}})$, computed with the algorithm of [11], is $I_{\text{LIA}} \stackrel{\text{def}}{=} (-255 \leq x_{y_2} - x_{y_3} + 256\lfloor -1\frac{x_{y_2}}{256} \rfloor)$. Applying β to it, we obtain $I \stackrel{\text{def}}{=} \beta(I_{\text{LIA}}) \stackrel{\text{def}}{=} (1_{[8]} \leq_u y2_{[8]} - y3_{[8]} + 0_{[8]} \cdot (255_{[8]} \cdot y2_{[8]} /_u 0_{[8]})$, because of the truncations that occur when converting the LIA-constants -255 and 256 into 8-bit BV-constants. I is not an interpolant for (A, B), since both $A \not\models_{\text{BV}} I$ and $B \wedge I \not\models_{\text{BV}} \bot$. However, in this case avoiding overflows (e.g. by using 16-bit words) is not enough: the formula $I' \stackrel{\text{def}}{=} (65281_{[16]} \leq_u (0_{[8]} :: y2_{[8]}) - (0_{[8]} :: y3_{[8]}) + 256_{[16]} \cdot (65535_{[16]} \cdot (0_{[8]} :: y2_{[8]}) /_u 256_{[16]}))$ is still not an interpolant for (A, B), since $A \not\models_{\text{BV}} I'$. In this case, we could fix the problem by using a signed inequality predicate instead of the unsigned one in I': the formula

$$I'' \stackrel{\text{def}}{=} (65281_{[16]} \leq_s (0_{[8]} :: y2_{[8]}) - (0_{[8]} :: y3_{[8]}) +$$
$$256_{[16]} \cdot (65535_{[16]} \cdot (0_{[8]} :: y2_{[8]}) /_u 256_{[16]}))$$

is a correct BV-interpolant for (A, B). ◇

Using signed inequality, however, does not always work.

Example 6: Consider again the interpolation problem of Example 4 and the interpolant $I' \stackrel{\text{def}}{=} ((0_{[1]} :: y3_{[8]}) + (0_{[1]} :: y4_{[8]}) \leq_u 81_{[9]})$ for (A, B). If we replace $\leq_u$ with $\leq_s$ in I', the resulting formula is not an interpolant for (A, B) anymore. ◇

As mentioned before, currently we address the potential failures in the translation from I_{LIA} to I by explicitly checking whether I is a correct interpolant for (A, B), which amounts to checking whether the BV-formula $(A \wedge \neg I) \vee (B \wedge I)$ is unsatisfiable. If the test fails, we simply discard I and resort to the last layer of our procedure. The investigation of more effective ways of extracting BV-interpolants from LIA-interpolants is part of ongoing and future work.

E. When Everything Else Fails

When none of the above techniques can be successfully applied to the current conjunction of BV-constraints, we resort to the bit-level interpolation procedure described in Sec. III. This makes our algorithm trivially complete. Clearly however, the effectiveness of our procedure crucially depends on how often this last layer is needed in practice. We discuss this topic in the following section.

F. Discussion

In the worst case, our procedure does not behave much differently from the simple bit-level algorithm of Sec. III. Furthermore, our algorithm is also typically more expensive to apply, since it might result in several extra calls to an SMT solver (for both LIA and BV) for each of the (negations of the) theory lemmas occurring in the DPLL(T)-proof of unsatisfiability, whereas the bit-level algorithm requires only one call to an eager proof-producing SMT(BV)-solver. In fact, it is not too difficult to craft some SMT(BV)-formulas for which our technique will always need to resort to the bit-level interpolation layer. On the other hand, for formulas for which this last layer is not needed, our procedure has the clear advantage of producing interpolants which preserve the word-level structure of BV-constraints, rather than flattening everything down to the bit level. Generally, this advantage will show up in all the cases in which the bit-level layer is needed only for a small fraction of the (negations of the) theory lemmas occurring in the proof of unsatisfiability.

We argue that for interpolation problems arising in software verification, the good cases are much more likely to occur than the bad cases, for the following reasons.

First, as already observed by other authors (e.g. [31], [13]), in important domains in which interpolation-based software verification has been successfully applied (e.g. device drivers), programs typically do not contain complex arithmetic expressions. In such cases, our experiments have shown that the LIA-based interpolation procedure of Sec. IV-D typically produces correct BV-interpolants in practice.

The second reason is that in software verification, interpolation is applied to formulas representing unrollings of the

TABLE I
PERFORMANCE RESULTS ON C PROGRAMS REQUIRING BIT-PRECISION

Program	KRATOS BV-1	BV-2	BV-3	BV-4	BV-5	SATABS	WOLVERINE
byte_add_1.c	31.00	T.O.	M.O.	57.30	31.54	T.O.	T.O.
byte_add_2.c	47.98	T.O.	M.O.	72.17	44.42	T.O.	T.O.
num_conversion_1.c	1.85	3.20	3.67	2.67	1.13	23.78	2.16
num_conversion_2.c	48.04	776.53	72.12	763.16	47.73	T.O.	T.O.
gcd_1.c	1.75	20.45	20.56	1.05	1.27	FAIL	515.31
gcd_2.c	29.21	M.O.	M.O.	39.21	28.21	339.86	185.56
gcd_3.c	70.05	T.O.	M.O.	209.34	70.59	T.O.	290.03
gcd_4.c	3.58	M.O.	T.O.	T.O.	4.25	T.O.	1.26
interleave_bits.c	45.90	T.O.	T.O.	T.O.	49.01	836.78	T.O.
modulus.c	4.87	34.00	M.O.	3.30	4.15	T.O.	M.O.
parity.c	387.56	M.O.	M.O.	T.O.	391.84	T.O.	T.O.
soft_float_1.c.cil.c	48.02	T.O.	T.O.	T.O.	T.O.	T.O.	136.88
soft_float_2.c.cil.c	61.34	T.O.	T.O.	70.02	T.O.	1101.54	177.63
soft_float_3.c.cil.c	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.
soft_float_4.c.cil.c	51.67	T.O.	M.O.	247.31	49.88	T.O.	T.O.
soft_float_5.c.cil.c	61.70	T.O.	T.O.	78.54	T.O.	T.O.	193.76
s3_clnt_1.BV.c.cil.c	41.06	50.82	T.O.	48.77	42.32	FAIL	T.O.
s3_clnt_2.BV.c.cil.c	20.96	9.92	116.03	8.59	22.01	T.O.	T.O.
s3_clnt_3.BV.c.cil.c	7.66	T.O.	93.77	T.O.	6.68	T.O.	T.O.
s3_srvr_1.BV.c.cil.c	11.59	35.91	240.77	34.74	11.63	160.74	T.O.
s3_srvr_2.BV.c.cil.c	150.64	62.22	116.54	61.26	152.10	342.11	T.O.
s3_srvr_3.BV.c.cil.c	48.35	124.32	43.63	125.19	48.36	405.48	T.O.
jain_1.c	0.34	0.39	0.30	0.12	0.36	FAIL	T.O.
jain_2.c	0.43	0.48	0.35	0.21	0.44	FAIL	T.O.
jain_4.c	0.55	0.60	0.40	0.33	0.54	FAIL	T.O.
jain_5.c	T.O.	T.O.	T.O.	T.O.	T.O.	FAIL	T.O.
jain_6.c	0.18	0.12	0.09	0.15	0.16	FAIL	T.O.
jain_7.c	0.29	0.23	0.15	0.26	0.27	FAIL	T.O.
TOTAL (solved / time)	**26 / 1176.57**	**14 / 1119.19**	**13 / 708.38**	**21 / 1823.69**	**23 / 1008.89**	**7 / 3210.29**	**8 / 1500.43**

Execution times are in seconds. T.O. indicates timeouts (using a cutoff value of 1200 seconds), M.O. memory outs (3GBytes), FAIL other kinds of errors (e.g. failure in computing interpolants or in refining the abstraction). All the programs are safe.

control-flow graph of programs, represented using a Static Single Assignment (SSA) form. Such formulas make heavy use of "definitional" equalities, i.e. equalities of the form $(x = t)$ in which x does not occur in t. For example, all equalities representing an assignment statement in SSA form are of this kind. Such equalities are exactly the kind of constraints that are exploited by our substitution-based technique of Sec. IV-C.

Finally, as regards performance, we remark that the BV-interpolation layers are invoked only on the (negations of the) BV-lemmas occurring in the final DPLL(T)-proof of unsatisfiability for the input problem $A \wedge B$. At solving time, only the lazy bit-blasting procedure of Sec. IV-A is used, whose efficiency is typically comparable to that of eager encodings into SAT. In general, only a fraction of all the BV-lemmas discovered during search occur in the final proof of unsatisfiability. Moreover, such lemmas typically involve only a subset of the constraints occurring in the formula. In fact, although not guaranteed to be minimal, they contain very often almost no redundant constraints, and are thus usually much easier to solve than the whole input problem.

V. EXPERIMENTAL EVALUATION

We have implemented the procedure described in the previous section within the MATHSAT SMT solver, and we have integrated it within KRATOS [17], a software model checker implementing a CEGAR-based lazy predicate abstraction algorithm with interpolation-based refinement in the style of [2] (but using the "large-block" encoding introduced in [32] for better exploiting the underlying SMT solver). In this section, we experimentally evaluate the effectiveness and efficiency of our technique in the context of software model checking.

All the experiments have been run on a Linux machine with a 2.2GHz Intel Xeon CPU, using a memory limit of 3GB. All the data and executables needed for reproducing the experiments are available at http://es.fbk.eu/people/griggio/papers/fmcad11_bv_interpolation.tar.bz2.

A. Effectiveness

In order to evaluate the effectiveness of our technique, we have collected a set of C programs whose verification requires the use of a bit-precise modeling of operations. These programs can not be proved safe by KRATOS in its default configuration, since by default it models program variables using rational numbers, and program operations using linear rational arithmetic (LRA) and uninterpreted functions. In particular, we use the following benchmark sets:

– ***byte_add*** **and** ***num_conversion*** implement arithmetic operations using shifts and bit-wise operations;
– ***gcd*** check simple assertions on Euclid's algorithm for computing the greatest common divisor;
– ***interleave_bits*****,** ***modulus*** **and** ***parity*** check the correctness of some "bit twiddling hacks" described at http://www-graphics.stanford.edu/~seander/bithacks.html;

– ***soft_float*** check simple assertions on the software floating-point implementation used in the Picosat [33] SAT solver;
– ***s3_clnt* and *s3_srvr*** are modified versions of some SSH programs used in several papers on software model checking, in which some bit-wise and non-linear operations have been introduced;
– ***jain*** are the simple programs used in [8].

We compare KRATOS using BV-interpolation against the only other two software model checkers supporting BV (to the best of our knowledge): SATABS, which implements CEGAR-based predicate abstraction but uses weakest preconditions for refinement [34], and WOLVERINE [35], which implements the interpolation-based lazy abstraction of [3], using an incomplete rewrite-based procedure for BV-interpolation [13]. [8] For KRATOS, we use not only the configuration in which all the layers described in Sec. IV are active (called "BV-1"), but also configurations in which some of the layers have been disabled or rearranged: "BV-2" does not use equality substitution, "BV-3" uses only the bit-level algorithm, "BV-4" tries LIA encoding before equality substitution, and "BV-5" does not use LIA encoding. The results of our experiments are reported in Table I. They show that not only KRATOS outperforms the other systems, but also that all the layers of our procedure contribute to the performance of KRATOS, since the default configuration "BV-1" is the clear winner. In particular, our full procedure can solve twice as many instances as the naïve configuration "BV-3" which uses only the bit-level algorithm of Sec. IV-E. Using "BV-1", the final bit-level layer is needed only for four of the programs, and always for less than 1% of the BV-interpolation problems, and in many cases the equality substitution layer alone is enough.

B. Efficiency

In order to evaluate the efficiency of our technique, we compare KRATOS-BV with the default version of KRATOS using linear rational arithmetic and uninterpreted functions, on programs that can be verified without the need of a bit-precise modeling of program variables and operations. We use common benchmarks for software model checking with predicate abstraction, used e.g. in [18]. The results are reported in Table II.[9] They show that, when the additional precision given by using bit-vectors instead of rationals is not needed, our procedure introduces very little overhead: KRATOS-BV-1 can solve only one instance less than KRATOS-LRA, but in fact there are cases in which KRATOS-BV-1 is one order of magnitude faster than KRATOS-LRA.[10] It is somewhat surprising to observe that, for these programs, even the naïve configuration "BV-3" which uses only the bit-level interpolation layer is not dramatically inferior to KRATOS-LRA.

[8] We used the latest versions of SATABS and WOLVERINE, i.e. version 2.6 and 0.5 respectively. For SATABS, we used Cadence SMV as underlying model checker.

[9] We omit the results for "BV-4" and "BV-5", as they are very similar to those for "BV-2" and "BV-1" respectively.

[10] Such differences are due to the fact that the two versions of KRATOS in general discover different sets of predicates, which lead to the exploration of different abstract search spaces.

TABLE II
PERFORMANCE RESULTS ON C PROGRAMS NOT REQUIRING BIT-PRECISION

	KRATOS configuration			
Program	**LRA**	**BV-1**	**BV-2**	**BV-3**
cdaudio_simpl1.cil.c	37.03	61.79	53.05	59.47
diskperf_simpl1.cil.c	40.14	89.25	52.63	64.55
floppy_simpl3.cil.c	18.37	41.06	28.61	33.55
floppy_simpl4.cil.c	36.75	91.73	47.44	58.97
kbfiltr_simpl1.cil.c	1.37	1.66	1.38	1.90
kbfiltr_simpl2.cil.c	1.68	2.70	2.43	2.94
s3_clnt_1.cil.c	5.59	5.20	65.34	10.62
s3_clnt_2.cil.c	4.71	5.33	20.72	7.33
s3_clnt_3.cil.c	8.52	4.87	14.72	4.86
s3_clnt_4.cil.c	3.20	6.04	29.41	T.O.
s3_srvr_1.cil.c	69.35	7.97	166.88	14.71
s3_srvr_2.cil.c	65.95	224.63	313.30	16.08
s3_srvr_3.cil.c	35.54	8.52	97.51	12.24
s3_srvr_4.cil.c	99.67	185.83	312.21	T.O.
s3_srvr_6.cil.c	90.48	25.60	24.71	163.21
s3_srvr_7.cil.c	218.26	15.10	17.28	40.26
s3_srvr_8.cil.c	72.94	13.83	170.53	23.27
s3_srvr_9.cil.c	5.43	22.92	24.50	53.44
s3_srvr_10.cil.c	9.82	14.68	14.14	255.90
s3_srvr_11.cil.c	36.47	15.49	19.03	156.01
s3_srvr_12.cil.c	19.56	60.78	47.94	328.75
s3_srvr_13.cil.c	289.77	T.O.	82.42	T.O.
s3_srvr_14.cil.c	18.16	22.50	61.08	99.72
s3_srvr_15.cil.c	24.55	27.77	27.28	20.00
s3_srvr_16.cil.c	57.93	12.13	39.05	46.38
TOTAL (solved / time)	**25 / 1271.24**	**24 / 967.38**	**25 / 1733.59**	**22 / 1474.16**

Execution times are in seconds. T.O. indicates timeouts (cutoff time of 600 seconds). All the programs are safe.

VI. CONCLUSIONS AND FUTURE WORK

We have presented an interpolation procedure for bit-vectors based on lazy SMT and on a layer of different techniques optimized for exploiting the structure of typical interpolation problems arising in software verification. We have integrated it in KRATOS, a CEGAR-based software model checker with interpolation-based refinement, and our experiments have shown that the new procedure makes it possible to verify programs requiring a bit-precise modeling of operations which could not be verified by KRATOS before.

For future work, we plan to explore several directions. First, we want to investigate in more depth the problem of computing BV-interpolants by reduction to LIA, in particular to identify smarter ways of generating a correct interpolant in BV from an interpolant for the LIA-encoding of the problem. Second, we plan to experiment with the integration of other layers in our procedure, e.g. based on the application of rewriting rules in the style of [13]. Finally, we would also like to investigate the problem of constructing a word-level interpolant from a bit-level proof of unsatisfiability, by exploring the possibility of extending the work of [12] for equality logic to more general cases.

REFERENCES

[1] K. L. McMillan, "Interpolation and SAT-Based Model Checking," in *Proc. of CAV'03*, ser. LNCS, W. A. Hunt Jr. and F. Somenzi, Eds., vol. 2725. Springer, 2003, pp. 1–13.

[2] T. A. Henzinger, R. Jhala, R. Majumdar, and K. L. McMillan, "Abstractions from proofs," in *Proc. of POPL'04*, N. D. Jones and X. Leroy, Eds. ACM, 2004, pp. 232–244.
[3] K. L. McMillan, "Lazy Abstraction with Interpolants," in *Proc. of CAV'06*, ser. LNCS, T. Ball and R. B. Jones, Eds., vol. 4144. Springer, 2006, pp. 123–136.
[4] ——, "An interpolating theorem prover," *Theor. Comput. Sci.*, vol. 345, no. 1, pp. 101–121, 2005.
[5] V. Sofronie-Stokkermans, "Interpolation in Local Theory Extensions," *Logical Methods in Computer Science (Special issue dedicated to IJCAR 2006)*, vol. 4, no. 4, p. Paper 1, 2008.
[6] D. Kapur, R. Majumdar, and C. G. Zarba, "Interpolation for data structures," in *Proc. of FSE'05*, M. Young and P. T. Devanbu, Eds. ACM, 2006, pp. 105–116.
[7] A. Cimatti, A. Griggio, and R. Sebastiani, "Efficient Generation of Craig Interpolants in Satisfiability Modulo Theories," *ACM Trans. Comput. Log.*, vol. 12, no. 1, p. 7, 2010.
[8] H. Jain, E. M. Clarke, and O. Grumberg, "Efficient Craig Interpolation for Linear Diophantine (Dis)Equations and Linear Modular Equations," in *Proc. of CAV'08*, ser. LNCS, A. Gupta and S. Malik, Eds., vol. 5123. Springer, 2008, pp. 254–267.
[9] A. Fuchs, A. Goel, J. Grundy, S. Krstic, and C. Tinelli, "Ground Interpolation for the Theory of Equality," in *Proc. of TACAS'09*, ser. LNCS, S. Kowalewski and A. Philippou, Eds., vol. 5505. Springer, 2009, pp. 413–427.
[10] A. Brillout, D. Kroening, P. Rümmer, and T. Wahl, "An Interpolating Sequent Calculus for Quantifier-Free Presburger Arithmetic," in *Proc. IJCAR'10*, ser. LNCS, J. Giesl and R. Hähnle, Eds., vol. 6173. Springer, 2010, pp. 384–399.
[11] A. Griggio, T. T. H. Le, and R. Sebastiani, "Efficient Interpolant Generation in Satisfiability Modulo Linear Integer Arithmetic," in *Proc. of TACAS'11*, ser. LNCS, P. A. Abdulla and K. R. M. Leino, Eds., vol. 6605. Springer, 2011, pp. 143–157.
[12] D. Kroening and G. Weissenbacher, "Lifting Propositional Interpolants to the Word-Level," in *Proc. of FMCAD'07*. IEEE Computer Society, 2007, pp. 85–89.
[13] ——, "An Interpolating Decision Procedure for Transitive Relations with Uninterpreted Functions," in *Proc. of HVC'09*, ser. LNCS, K. S. Namjoshi, A. Zeller, and A. Ziv, Eds., vol. 6405. Springer, 2009, pp. 150–168.
[14] R. Bruttomesso, A. Cimatti, A. Franzén, A. Griggio, and R. Sebastiani, "The MathSAT 4 SMT Solver," in *Proc. of CAV'08*, ser. LNCS, A. Gupta and S. Malik, Eds., vol. 5123. Springer, 2008, pp. 299–303.
[15] A. Franzén, "Efficient Solving of the Satisfiability Modulo Bit-Vectors Problem and Some Extensions to SMT," Ph.D. dissertation, DISI - University of Trento, 2010.
[16] R. Jhala and R. Majumdar, "Software model checking," *ACM Comput. Surv.*, vol. 41, pp. 21:1–21:54, October 2009.
[17] A. Cimatti, A. Griggio, A. Micheli, I. Narasamdya, and M. Roveri, "Kratos – a Software Model Checker for SystemC," in *Proc. of CAV'11*, ser. LNCS, G. Gopalakrishnan and S. Qadeer, Eds., vol. 6806. Springer, 2011, pp. 310–316, to appear.
[18] D. Beyer, M. E. Keremoglu, and P. Wendler, "Predicate Abstraction with Adjustable-Block Encoding," in *Proc. of FMCAD'10*, R. Bloem and N. Sharygina, Eds., 2010, pp. 189–187.
[19] K. Dräger, A. Kupriyanov, B. Finkbeiner, and H. Wehrheim, "SLAB: A Certifying Model Checker for Infinite-State Concurrent Systems," in *Proc. of TACAS'10*, ser. LNCS. Springer, 2010.
[20] N. Caniart, "Merit: An Interpolating Model-Checker," in *Proc. of CAV'10*, ser. LNCS, T. Touili, B. Cook, and P. Jackson, Eds., vol. 6174. Springer, 2010, pp. 162–166.
[21] R. Sebastiani, "Lazy Satisfiability Modulo Theories," *Journal on Satisfiability, Boolean Modeling and Computation, JSAT*, vol. 3, no. 3-4, pp. 141–224, 2007.
[22] R. Brummayer and A. Biere, "Boolector: An Efficient SMT Solver for Bit-Vectors and Arrays," in *Proc. of TACAS'09*, ser. LNCS, S. Kowalewski and A. Philippou, Eds., vol. 5505. Springer, 2009, pp. 174–177.
[23] S. Jha, R. Limaye, and S. A. Seshia, "Beaver: Engineering an Efficient SMT Solver for Bit-Vector Arithmetic," in *Proc. of CAV'09*, ser. LNCS, A. Bouajjani and O. Maler, Eds., vol. 5643. Springer, 2009, pp. 668–674.
[24] L. de Moura and N. Bjørner, "Z3: An Efficient SMT Solver," in *Proc. of TACAS'08*, ser. LNCS, C. R. Ramakrishnan and J. Rehof, Eds., vol. 4963. Springer, 2008, pp. 337–340.
[25] V. Ganesh and D. L. Dill, "A Decision Procedure for Bit-Vectors and Arrays," in *Proc. of CAV'07*, ser. LNCS, W. Damm and H. Hermanns, Eds., vol. 4590. Springer, 2007, pp. 519–531.
[26] A. Franzén, A. Cimatti, A. Nadel, R. Sebastiani, and J. Shalev, "Applying SMT in Symbolic Execution of Microcode," in *Proc. of FMCAD'10*, R. Bloem and N. Sharygina, Eds., 2010, pp. 121–128.
[27] N. Eén and N. Sörensson, "Temporal induction by incremental SAT solving," *Electr. Notes Theor. Comput. Sci.*, vol. 89, no. 4, 2003.
[28] R. J. A. Achá, R. Nieuwenhuis, A. Oliveras, and E. Rodríguez-Carbonell, "Practical algorithms for unsatisfiability proof and core generation in SAT solvers," *AI Commun.*, vol. 23, no. 2-3, pp. 145–157, 2010.
[29] M. Bozzano, R. Bruttomesso, A. Cimatti, A. Franzén, Z. Hanna, Z. Khasidashvili, A. Palti, and R. Sebastiani, "Encoding RTL Constructs for MathSAT: a Preliminary Report," *Electr. Notes Theor. Comput. Sci.*, vol. 144, no. 2, pp. 3–14, 2006.
[30] D. Kroening, J. Leroux, and P. Rümmer, "Interpolating Quantifier-Free Presburger Arithmetic," in *Proc. of LPAR'10*, ser. LNCS, C. G. Fermüller and A. Voronkov, Eds., vol. 6397. Springer, 2010, pp. 489–503.
[31] T. Ball, B. Cook, S. K. Lahiri, and L. Zhang, "Zapato: Automatic Theorem Proving for Predicate Abstraction Refinement," in *Proc. of CAV'04*, ser. LNCS, R. Alur and D. Peled, Eds., vol. 3114. Springer, 2004, pp. 457–461.
[32] D. Beyer, A. Cimatti, A. Griggio, M. E. Keremoglu, and R. Sebastiani, "Software model checking via large-block encoding," in *Proc. of FMCAD'09*, 2009, pp. 25–32.
[33] A. Biere, "PicoSAT Essentials," *JSAT*, vol. 4, no. 2-4, pp. 75–97, 2008.
[34] E. M. Clarke, D. Kroening, N. Sharygina, and K. Yorav, "Predicate Abstraction of ANSI-C Programs Using SAT," *Formal Methods in System Design*, vol. 25, no. 2-3, pp. 105–127, 2004.
[35] D. Kroening and G. Weissenbacher, "Interpolation-Based Software Verification with Wolverine," in *Proc. of CAV'11*, ser. LNCS, G. Gopalakrishnan and S. Qadeer, Eds., vol. 6806. Springer, 2011, pp. 573–578.

Accelerating MUS Extraction with Recursive Model Rotation

Anton Belov
Complex and Adaptive Systems Laboratory
University College Dublin
Email: anton.belov@ucd.ie

Joao Marques-Silva
Complex and Adaptive Systems Laboratory
University College Dublin
Email: jpms@ucd.ie

***Abstract*—Minimally Unsatisfiable Subformulas (MUSes) find a wide range of practical applications. A large number of MUS extraction algorithms have been proposed over the last decade, and most of these algorithms are based on iterative calls to a SAT solver. In this paper we introduce a powerful technique for acceleration of MUS extraction algorithms called *recursive model rotation* — a recursive version of the recently proposed model rotation technique. We demonstrate empirically that recursive model rotation leads to multiple orders of magnitude performance improvements on practical instances, and pushes the performance of our MUS extractor MUSer2 ahead of the currently available MUS extraction tools.**

I. Introduction

A minimally unsatisfiable subformula (MUS) of an unsatisfiable CNF formula F is any minimal, with respect to set inclusion, subset of clauses in F that is still unsatisfiable. MUSes find a wide range of practical applications. For example, MUS are used in a number of verification tasks to extract a concise description of inconsistency. As a result, development of effective MUS extraction algorithms is currently a very active area of research — examples of most recent work include [1], [2], [3], [4], [5], [6]. MUS algorithms can be roughly categorized as *constructive* (or, insertion-based), as *destructive* (or, deletion-based), or dichotomic. At the moment, destructive and hybrid MUS extraction algorithms outperform other approaches by a wide margin [1]. However, irrespective of the approach, the main bottleneck of MUS algorithms is the number of repeated calls to a SAT oracle.

Model rotation [1] is a method for detection of clauses that are included in all MUSes of a given formula via the analysis of models returned by a SAT oracle. While theoretically the technique does not guarantee the reduction in the number of SAT calls, in practice it does, and has been reported in [1] to provide for considerable performance gains (up to a factor of 5). In this paper we push the idea further — we propose a modification to the technique that in practice results in a very large reduction in the number of invocations of SAT oracle, significantly boosting the performance of our MUS extractor MUSer2. One of the key aspects of the proposed technique, which we call *recursive model rotation*, is that in principle it could be used with any type of MUS extraction algorithm.

Model rotation and recursive model rotation are described in Section III, following the necessary background in Section II. In Section IV we demonstrate the power of the new technique empirically. The paper is concluded in Section V with the discussion of some of the related work and possible enhancements to the proposed techniques.

II. Background

While we assume the familiarity with classical propositional logic (CPL) and SAT, here we clarify and fix the terminology used in this paper. We focus on formulas in CNF (*formulas*, from hence on), which we treat as (finite) sets of clauses. We assume that clauses do not contain duplicate variables.

Given a formula F we denote the set of variables that occur in F by $Var(F)$, and the set of variables that occur in $C \in F$ by $Var(C)$. An *assignment* h for F is a map $h : Var(F) \to \{0, 1\}$. By $h|_{\neg x}$ be denote the assignment $(h \setminus \{\langle x, h(x)\rangle\}) \cup \{\langle x, 1 - h(x)\rangle\}$. Assignments are extended to formulas according to the semantics of CPL. If $h(F) = 1$, then h is a *model* of F. If a formula F has (resp. does not have) a model, then F is *satisfiable* (resp. *unsatisfiable*) and we write $F \in SAT$ (resp. $F \in UNSAT$).

By $Unsat(F, h)$ we denote the set of clauses falsified by h, i.e. $Unsat(F, h) = \{C \mid C \in F \text{ and } h(C) = 0\}$. If $F \in UNSAT$ and for some clause $C \in F$, we have $F \setminus \{C\} \in SAT$, then C is called a *transition clause* for F.

Formula F is called *minimally unsatisfiable*, in symbols $F \in MU$, if $F \in UNSAT$ and for any $F' \subset F$, $F' \in SAT$. Equivalently, F is minimally unsatisfiable, if every clause in F is a transition clause. Given a formula $F \in UNSAT$, a *minimally unsatisfiable subformula* of F, in symbols $MUS(F)$, is any $F' \subseteq F$ such that $F' \in MU$. In general, a given unsatisfiable formula may have more than one MUS. Clauses that belong to *all* MUSes of F are called *necessary* for F [7] — clearly, every transition clause is necessary and vice versa, as such the terms are often used interchangeably.

Most of the algorithms computing MUSes rely on the identification of the necessary clauses of the input formula F or its subformulas. In the constructive approach the clauses of F are added to an initially empty set F' until F' becomes unsatisfiable. Then, the last clause added to F' is necessary for F'. In the destructive approach the clauses are removed from F until the resulting formula F' becomes satisfiable. Then, the last clause C removed from F is necessary for $F' \cup \{C\}$. We refer the reader to [1] for an overview of the state-of-the-art in MUS computation algorithms.

III. Recursive Model Rotation

Let F be an unsatisfiable formula, let $C \in F$ be a transition clause, and let h be a model of $F \setminus \{C\}$. Note that this implies $Unsat(F, h) = \{C\}$. In fact, we have the following observation [1]:

Proposition 1: Let F be an unsatisfiable formula. Then $C \in F$ is a transition clause if and only if there exists an assignment h such that $Unsat(F, h) = \{C\}$.

Proof: C is a transition clause iff $F \setminus \{C\} \in SAT$ iff there exists an assignment h such that $Unsat(F \setminus \{C\}, h) = \emptyset$ iff $Unsat(F, h) = \{C\}$. ■

Model rotation exploits Proposition 1 to detect additional transition clauses during the computation of MUS. Assuming the destructive approach, let C be a transition clause detected by invoking a SAT solver on the formula $F' = F \setminus \{C\}$ — that is, the SAT solver determined that F' is satisfiable, and returned a model h of F', while F is known to be unsatisfiable. A similar situation occurs in the constructive approach when C is the clause that caused unsatisfiability of the current subset of F. In most MUS extraction algorithms the model h is discarded (a notable exception is MiniUnsat [5] — in Section V we discuss the relationship of our technique with this algorithm). However, consider the assignment $h' = h|_{\neg x}$ for some $x \in Var(C)$. Clearly, $C \notin Unsat(F, h')$, but $Unsat(F, h') \neq \emptyset$. Furthermore, if the set $Unsat(F, h')$ contains *exactly one* clause C', then by Proposition 1, C' is a transition clause. The model h' (of $F \setminus \{C'\}$) is said to be obtained by the *rotation* of the model h with respect to clause C and variable x. Note that now, the model h' can be rotated again — this time, with respect to clause C' and some variable $x' \in C'$ — possibly giving another transition clause. Note that x' should be different from x, as otherwise the rotation will give the initial model h.

Example 1: Let $F = \{C_1, \ldots, C_5\}$, where

$$C_1 = \neg x_1 \vee \neg x_2 \qquad C_3 = x_2 \vee \neg x_3 \qquad C_5 = x_1 \vee x_2$$
$$C_2 = x_1 \vee \neg x_2 \qquad C_4 = x_2 \vee x_3$$

The formula F is unsatisfiable, and the clause C_1 is a transition clause for F, i.e. $F_1 = F \setminus \{C_1\} \in SAT$. Let $h_1 = \{x_1, x_2, x_3\}$ be the model of F_1 returned by a SAT solver. We have $Unsat(F, h_1) = \{C_1\}$. Let $h_2 = h_1|_{\neg x_1}$, that is $h_2 = \{\neg x_1, x_2, x_3\}$. We have $Unsat(F, h_2) = \{C_2\}$, and therefore, C_2 is another transition clause.

In model rotation the process is continued until for some clause C_f and the corresponding model h_f of $F \setminus \{C_f\}$, for every variable $x \in C_f$ the set $Unsat(F, h|_{\neg x})$ is either not a singleton, or contains a clause that is already known to be a transition clause. In the above example, the rotation stops at h_2 as $Unsat(F, h_2|_{\neg x_2}) = \{C_3, C_5\}$.

This stopping criterion guarantees that the total number of model rotations during the execution of an MUS computation algorithm on formula F is at most $|M| \cdot c_{max}$, where M is the computed MUS of F, and c_{max} is the maximum among the lengths of clauses in M. On the other hand, each successful model rotation (i.e. the one that detects a new transition clause) saves a potentially expensive call to a SAT solver. Given that in practical instances the size of MUSes rarely exceeds a few tens of thousands of clauses, it is not surprising that model rotation often provides for significant performance gains — in Section IV we demonstrate these gains empirically.

We now note that in model rotation *at most one variable* from each necessary clause C is used for rotation — the original motivation in [1] was to keep the model rotation a low overhead technique. We observe, however, that the stopping criterion described above still guarantees that number of rotations is linear in the size of the computed MUS, even if *all variables* from C are used for rotation. On the other hand, we have the following result:

Proposition 2: Let F be an unsatisfiable formula, let $C \in F$ be a transition clause, and let h be a model of $F \setminus \{C\}$. Then, the sets $Unsat(F, h|_{\neg x})$ for $x \in Var(C)$ are pairwise disjoint.

Proof: Let x be a variable in C, and let C' be some clause in $Unsat(F, h|_{\neg x})$. Since $C' \notin Unsat(F, h)$, the literal of variable x was *critical* in C' under h (that is, the only literal in C' that evaluates to 1 under h). Since every clause has *at most* one critical literal, the fact follows. ■

Hence, by performing model rotation on different variables of C we are *guaranteed* to obtain disjoint sets of clauses, thus increasing the likelihood of detecting additional transition clauses. Consider again the formula F in Example 1, and assume that the model rotation stopped at assignment $h_2 = \{\neg x_1, x_2, x_3\}$ due to the stopping criterion. We can now "backtrack" to the assignment h_1 and attempt to rotate h_1 with respect to C_1 on variable x_2. The rotation results in the assignment $h_2' = \{x_1, \neg x_2, x_3\}$, and since $Unsat(F, h_2') = \{C_3\}$ we have a new transition clause C_3. Rotation of h_2' on x_3 results in the assignment $h_3 = \{x_1, \neg x_2, \neg x_3\}$, which gives another transition clause C_4. Rotating h_3 on x_2 results in the assignment $h_4 = \{x_1, x_2, \neg x_3\}$ at which point the rotation terminates, because $Unsat(F, h_4) = \{C_1\}$ and C_1 is already known to be a transition clause. In this example, such *recursive* model rotation allows to detect all of the transition clauses of F. Remarkably, as demonstrated in Section IV, the cases when the recursive model rotation finds all, or close to all, of the necessary clauses do occur often on practical instances. The sketch of the algorithm for the recursive model rotation is presented in Algorithm 1. The algorithm is invoked whenever an MUS extractor detects a new transition clause as a result of a call to a SAT solver.

We now note that the "if" direction of Proposition 1 can be generalized as follows:

Proposition 3: Let F be an unsatisfiable formula. Then, for any assignment h the set $Unsat(F, h)$ contains at least one clause from each of the MUSes of F.

Proof: If not, then the set $F \setminus Unsat(F, h)$ includes an MUS of F, and so must be unsatisfiable. ■

Proposition 3 justifies the following heuristic for selection of clauses in deletion-based MUS extraction algorithms: whenever the recursive model rotation arrives at an assignment h with $|Unsat(F, h)| > 1$, try to remove the clauses from $Unsat(F, h)$ next. The idea is that for instances with many

Algorithm 1 RECURSIVE-MODEL-ROTATION(F, M, C, h)

Input: F — an unsatisfiable CNF formula
$M \subseteq F$ — a set of transition clauses of F
$C \in M$ — a transition clause
h — a model of $F \setminus \{C\}$

Effect: M may contain additional transition clauses of F

```
1: for all x ∈ Var(C) do
2:     h' ← h|¬x
3:     if Unsat(F, h') = {C'} and C' ∉ M then
4:         M ← M ∪ {C'}
5:         RECURSIVE-MODEL-ROTATION(F, M, C', h')
6:     end if
7: end for
```

MUSes chances are that the clauses from this set belong to different MUSes, and so among the next few calls to the SAT solver, there will be UNSAT results — these, in turn, are beneficial for the deletion algorithms which use clause set refinement [1] to remove the clauses outside of unsatisfiable core. Our experiments, reported in Section IV, confirm that this rotation-based re-ordering of clauses is indeed beneficial on some classes of problems.

IV. EXPERIMENTAL STUDY

The algorithm for recursive model rotation (RMR) is implemented in our new MUS extractor MUSer2. MUSer2 is a slightly optimized version of MUSer — an MUS extractor from [1]. Both MUSer2 and MUSer implement HYB — a deletion-like MUS computation algorithm that employs clause set refinement, redundancy removal and (non-recursive) model rotation. The results of the experimental evaluation of HYB reported in [1] show clearly that it significantly outperforms all of the publicly available MUS extraction algorithms — we also reproduce some of the data from [1] in this section.

To evaluate the effectiveness of recursive model rotation we performed a comprehensive experimental study on 500 benchmarks submitted to the MUS track of SAT Competition 2011 (http://www.satcompetition.org/2011) — this is also the same set of instances that was used in [1] to evaluate HYB. The benchmark instances were obtained from various industrial applications of SAT, including hardware bounded model checking, FPGA routing, hardware and software verification, equivalence checking, abstraction refinement, design debugging, functional decomposition and bioinformatics. Note that the benchmarks are pre-processed via trimming [1]. The benchmarks are available for download at http://logos.ucd.ie/~jpms/Drops/SAT11. The experiments were performed on an HPC cluster, where each node is dual quad-core Xeon E5450 3 GHz with 32 GB of memory. The timeout was set at 1200 seconds, and memory was limited at 4 GB.

Figure 1 presents the incremental-time plot that provides an overview of the results of our experimental study. The plot contains data from [1] for SAT4J [8] MUS extractor in destructive mode (SAT4J-D), a destructive MUS algorithm Picomus from the Picosat distribution [9], and the algorithm HYB implemented in MUSer. The former two are the top performing publicly available MUS extractors among those evaluated in [1]. In addition, the plot contains data for: MUSer2 with model rotation disabled (MUSer2 no MR), MUSer2 (with model rotation), MUSer2 with recursive model rotation (MUSer2 + RMR), and, finally, MUSer2 + RMR with rotation-based reordering of clauses described in Section III.

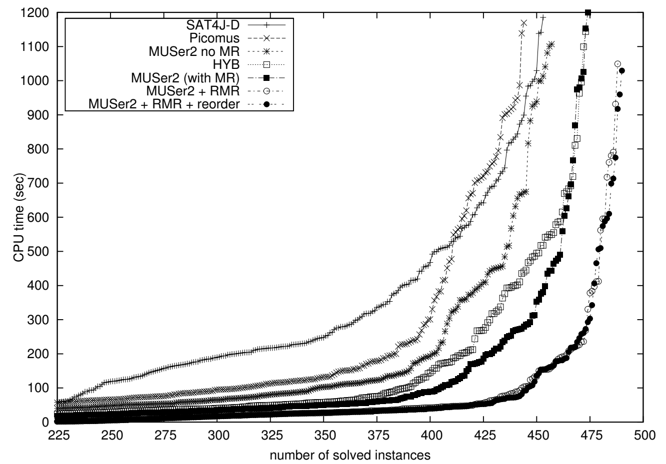


Fig. 1. Incremental-time plot showing data for all MUS extractors. To keep the plot legible, the data for the first 225 instances in not shown.

TABLE I
SUMMARY OF THE PERFORMANCE OF THE VARIANTS OF MUSER2

MUSer2 variant	Solved (from 500)	Total time (sec)	Total SAT calls	Avg. MUS size (% of input size)	Avg. % of tr. clauses by rot
no MR	457	44227	1741954	93.44	n/a
with MR	474	20249	797490	93.45	61.33
+RMR	488	12609	570303	93.46	77.57
+RMR+reorder	**490**	**12153**	570867	93.44	77.44

Columns 3-6 contain data for instances solved by *all* variants of MUSer2. The last column shows the percentage of transition clauses in the computed MUS that were detected by model rotation.

The following conclusions can be drawn. First, we note that MUSer2 has a clear performance edge on the currently publicly available MUS extraction tools. Second, the comparison between MUSer2 and MUSer2 without model rotation demonstrates that model rotation, even in the form introduced in [1], is an extremely powerful acceleration technique for MUS extraction algorithms. Third, we note that recursive model rotation increases the power of model rotation further. Finally, it appears that rotation-based clause re-ordering heuristic might also result in a slight performance edge.

Table I provides additional evidence of the power of model rotation in general, and recursive model rotation in particular. The table presents the number of solved instances for each of the variants of MUSer2 in the second column. The data in the remaining columns is for instances solved by all of the four variants. We note that the addition of recursive model rotation results in significant reduction in overall MUS extraction time, and allows to solve significantly more instances. We also point out that RMR is really a clear-win technique for MUS extraction algorithms as the size of computed MUSes is not affected negatively. Finally, the last column in Table I shows

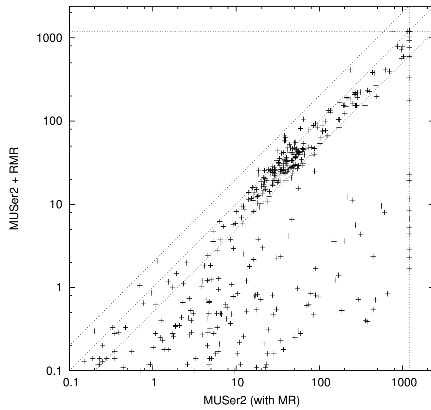


Fig. 2. MUSer2 with model rotation vs. recursive model rotation: CPU time.

the percentage of MUS clauses detected by model rotation — over 77% on average for RMR.

Both Figure 1 and Table I show that the addition of recursion to model rotation results in a significant performance edge. Nevertheless, we present additional scatter plots comparing the two versions of the technique. The plot in Figure 2 shows that on many problems the addition of recursion results in 1, 2 and in some cases 3 orders of magnitude speed-ups. The analysis of our experimental data suggests that recursion allows to detect *significantly* more necessary clauses, in many cases all of them.

The effect of recursive model rotation on some of the specific classes of instances from our benchmark set are demonstrated in Table II — *debug* refers to design debugging instances, *decomp* refers to functional decomposition instances (cf [10]), *ibm* refers to benchmarks from IBM, and *intel* are the abstraction-refinement benchmarks from [6]. While for some of the classes the performance improvements are moderate (although exceeding a factor of 2), on instances from design debugging and abstraction refinement we observe over 2 orders of magnitude speed-ups.

Finally, in Table III we compare the performance of MUSer2 with that of the resolution-based destructive MUS extraction algorithm 1MN [6]. Since the solver that implements the latter is not publicly available, we reproduce the data from [6]. For this experiment we used the original, *untrimmed* instances, and set the timeout to 2 hours, as in [6], and memout to 4 GB. The hardware used in our experiments appears to be similar to the one reported in [6]. We note that with the exception of two 4pipe instances, MUSer2 has a clear performance advantage over 1MN — on some instance (longmult) we observe 2 orders of magnitude speed-ups. On 4pipe MUSer2 is slower by a factor of 1.5, and on 4pipe_k it runs out of memory. The reason for the latter is the SAT solver used in this set of experiments (picosat-935), as our techniques have negligible memory overhead. It should be noted that MUSer2 uses SAT solver as a black-box (as opposed to resolution-based approach of [6] which requires significant modifications to a SAT solver). We suspect that switching to a more efficient SAT solver will resolve this issue.

TABLE II
MUSER2: NO ROTATION VS. RECURSIVE ROTATION, SELECTED CLASSES

Class (total num)	no rotation num solv.	time (sec)	SAT calls	recursive rotation num solv.	time (sec)	SAT calls	% by rot.
debug(120)	103	7041	129651	**120**	**65**	5713	94.82
fdec(143)	143	9738	874946	143	**4679**	307422	64.79
ibm(45)	42	5156	204134	42	**2255**	66563	84.38
intel(49)	41	7441	31640	**48**	**38**	346	97.27

TABLE III
MUSER2 + RMR VS. 1MN [6]: CPU TIME (SEC)

Instance	1MN	MUSer-2 + RMR	MUSer-2 + RMR + reorder
4pipe	**1417**	2101	1776
4pipe_1_ooo	1528	**425**	477
4pipe_2_ooo	2383	**1070**	1227
4pipe_3_ooo	2560	**593**	779
4pipe_4_ooo	2432	**568**	600
3pipe_k	167	104	**90**
4pipe_k	**1426**	MO	MO
barrel5	68	**9**	10
barrel6	348	**95**	150
barrel7	849	115	**103**
barrel8	4115	**1270**	2338
longmult4	14	**0.4**	**0.4**
longmult5	143	2.5	**1.6**
longmult6	968	13	**10**
longmult7	5099	103	**40**

V. CONCLUDING REMARKS

The analysis of models returned by a SAT solver during MUS extraction can be attributed to [5], where it was explored in the context of constructive MUS extractor MiniUnsat. Nevertheless, model rotation differs fundamentally from this work. In our view, recursive model rotation (and clause reordering) is just one example of structure-based techniques for MUS extraction. – we are currently investigating additional techniques of similar nature.

Acknowledgements: We thank the anonymous referees for helpful comments. This work is partially supported by SFI PI grant BEACON (09/IN.1/I2618).

REFERENCES

[1] J. Marques-Silva and I. Lynce, "On improving MUS extraction algorithms," in *Theory and Applications of Satisfiability Testing*, 2011.
[2] C. Desrosiers, P. Galinier, A. Hertz, and S. Paroz, "Using heuristics to find minimal unsatisfiable subformulas in satisfiability problems," *J. of Combinatorial Optimization*, vol. 18, no. 2, 2009.
[3] J. Marques-Silva, "Minimal unsatisfiability: Models, algorithms and applications," in *Int'l Symposium on Multiple-Valued Logic*, 2010.
[4] É. Grégoire, B. Mazure, and C. Piette, "On approaches to explaining infeasibility of sets of Boolean clauses," in *Int'l Conference on Tools with Artificial Intelligence*, 2008.
[5] H. van Maaren and S. Wieringa, "Finding guaranteed MUSes fast," in *Theory and Applications of Satisfiability Testing*, 2008.
[6] A. Nadel, "Boosting minimal unsatisfiable core extraction," in *Formal Methods in Computer-Aided Design*, 2010.
[7] O. Kullmann, I. Lynce, and J. Marques-Silva, "Categorisation of clauses in conjunctive normal forms: Minimally unsatisfiable sub-clause-sets and the lean kernel," in *Int'l Conference on Theory and Applications of Satisfiability Testing*, 2006.
[8] D. Le Berre and A. Parrain, "The Sat4j library, release 2.2," *J. on Satisfiability, Boolean Modeling and Computation*, vol. 7, 2010.
[9] A. Biere, "PicoSAT essentials," *J. on Satisfiability, Boolean Modeling and Computation*, vol. 2, 2008.
[10] R.-R. Lee, J.-H. R. Jiang, and W.-L. Hung, "Bi-decomposing large Boolean functions via interpolation and satisfiability solving," in *Design Automation Conference*, 2008.

Pseudo-Boolean Solving by Incremental Translation to SAT

Panagiotis Manolios
Northeastern University
pete@ccs.neu.edu

Vasilis Papavasileiou
Northeastern University
vpap@ccs.neu.edu

***Abstract*—We revisit pseudo-Boolean Solving via compilation to SAT. We provide an algorithm for solving pseudo-Boolean problems through an incremental translation to SAT that works with any incremental SAT solver as a backend. Experimental evaluation shows that our incremental algorithm solves industrial problems that previous SAT-based approaches do not. We also show that SAT-based algorithms for solving pseudo-Boolean problems should be a part of any portfolio solver.**

I. INTRODUCTION

Boolean Satisfiability (SAT) has been the subject of intensive research over the past decade. Many powerful solvers have been developed, and SAT has been successfully applied to problems in a variety of fields, like electronic design automation, hardware verification, AI planning, and others. In many domains, the need for non-propositional constraints like linear inequalities naturally arises. The *Pseudo-Boolean* (PB) formalism accommodates linear constraints over Boolean variables.

Definition 1. *A* Pseudo-Boolean constraint *is a constraint of the form*

$$c_1 p_1 + c_2 p_2 + \cdots + c_n p_n \,\square\, r \qquad (1)$$

where $\square$ *is one of the relations* $<, \leq, =, <$, *or* $\geq$, *the* variables p_i *(for* $1 \leq i \leq n$*) can take the values* 0 *and* 1 *and the* coefficients c_i *are integers.*[1] *The integer* r *is the* right-hand side. *We call* $c_i p_i$ *a* term. *A PB problem is a conjunction of PB constraints.*

Several kinds of solvers can deal with PB problems. PB satisfiability (or optimization) is a restriction of *Integer Linear Programming* to 0-1 variables. As a result, ILP solvers can be used. PB constraints can also be thought of as a generalization of clauses. Thus, SAT techniques can be applied, *e.g.*, the DPLL procedure can be modified to handle PB constraints [1, 2]. An alternative is to compile PB constraints to CNF and use an off-the-shelf SAT solver [3, 4, 5].

We revisit PB solving via compilation to SAT. An advantage of this approach is that tools based on it automatically become more competitive as the performance of the underlying SAT solvers they depend on improves. Furthermore, SAT solvers provide flexible, robust, mature, and well-engineered interfaces that have found a plethora of interesting applications. Finally, it is desirable to have a portfolio of complementary solvers that in aggregate provide good performance over a large class of PB problems arising in practice. We show that SAT-based approaches should be an integral part of any such portfolio: while they are not competitive on some PB problems (*e.g.*, those that benefit from ILP-based techniques such as cutting planes), they do very well on other types of problems (*e.g.*, those where the ratio of propositional to arithmetic constraints is high).

We propose an algorithm for PB solving that uses a SAT solver for the efficient exploration of the search space, but at the same time exploits the high-level structure of the PB constraints to simplify the problem and direct the search. We are primarily concerned with industrial problems and show that our algorithm can tackle industrial PB instances that were previously beyond the reach of SAT-based solvers.

The rest of the paper is organized as follows. In section II, we present a class of incremental algorithms for solving PB problems. Our algorithms are parameterized by a method for translating from PB constraints to CNF, and in section III, we analyze several different encoding schemes. We experimentally evaluate our solver in section IV. We review related work in section V, and conclude with section VI.

II. INCREMENTAL SOLVING

Our PB solver, PB-SAT, works by translating constraints to CNF *incrementally*, in stages and performs multiple SAT calls. Knowledge we acquire after each call allows us to simplify the remaining untranslated constraints.

Algorithm 1 shows the basic structure of our solver. We assume the existence of a function TRANSLATE that converts a PB constraint C to an equisatisfiable propositional formula TRANSLATE(C). The algorithm is independent of the specifics of TRANSLATE.

In Algorithm 1, Φ corresponds to the set of clauses we have generated so far. We initialize Φ so that it contains *PB-clauses*: PB constraints of the form $l_1 + l_2 + \ldots + l_n \geq 1$, where the l_i's are literals. Optionally, we can conjoin to Φ the translations of cardinality constraints and other constraints that can be efficiently encoded as clauses. Ψ, initialized to contain the non PB-clauses, is used to record the PB constraints that have not yet been translated to CNF.

[1] We convert the constraints to a normal form with only positive coefficients and no relation other than $\geq$. In this normal form, variables can appear negated.

Algorithm 1 PB Solving by incremental translation to SAT

```
1:  procedure PB-SAT(Ψ)
2:      Φ ← {φ ∈ Ψ : φ is a PB-clause}
3:      Ψ ← {ψ ∈ Ψ : ψ is not a PB-clause}
4:      while true do
5:          A, U ← SAT(Φ)
6:          if A = UNSAT then return UNSAT
7:          Ψ ← SIMPLIFY(Ψ, U)
8:          if A satisfies Ψ then return A
9:          Ψ′ ← {ψ ∈ Ψ : ψ falsified by A}
10:         if Ψ′ = ∅ then Ψ′ ← SELECT(Ψ)
11:         Ψ ← Ψ \ Ψ′
12:         for all ψ ∈ Ψ′ do Φ ← Φ ∧ TRANSLATE(ψ)
```

After initialization, our algorithm enters a loop where it calls the SAT solver on the current set of clauses, Φ (line 5). If Φ is unsatisfiable, the SAT solver returns "*UNSAT*" for A and the input to the algorithm is also unsatisfiable. Otherwise, the SAT solver returns a *partial* satisfying assignment A and the set of known unit literals, U. In line 7, we use U to simplify Ψ, the constraints we have yet to translate. How this is done is explained later. Now, if any full assignment that extends A also satisfies Ψ, then A is a partial assignment that satisfies the input to our algorithm, so we return A. Otherwise, any PB constraints that are false under every full assignment extending A are stored in Ψ'. The idea is to only translate these falsified constraints in the next round, but it may turn out that A does not falsify any of the remaining constraints. In that case, we select a non-empty subset of Ψ to translate. (Note that $\Psi \neq \emptyset$ is an invariant holding right after line 8). Next, in line 11, we update Ψ to reestablish the invariant that it contains the constraints left to translate. We end the loop by translating all the PB constraints in (the non-empty) Ψ'.

Note that we translate all constraints falsified by intermediate partial assignments. However, a lazier version of our algorithm could translate only a subset of the falsified constraints. While there are many ways of deciding what to translate during each iteration, the essence of our approach is to incrementally translate constraints in order to obtain useful information that is used to simplify the remaining constraints.

A. Simplification

The SIMPLIFY function in algorithm 1 uses units discovered by the SAT solver during the incremental queries to simplify the remaining constraints. We explain this with an example.

Example 1. *Given the units x_1 and $\neg x_2$, we simplify the constraint $2x_1 + 2x_2 + x_3 + x_4 \geq 4$ to $x_3 + x_4 \geq 2$, which we further simplify to the units x_3, x_4.*

Notice that we propagate knowledge in both directions: (i) we use units from SAT solving to simplify the PB constraints, and (ii) we learn new units from PB constraint propagation. SIMPLIFY propagates the units we know at the PB level, as described above. This process may modify the constraints and return new units. We give these units to the SAT solver, and perform Boolean Constraint Propagation (BCP). If BCP leads to more units, we repeat. We stop when we reach a fixpoint (we no longer learn anything new).

Our incremental strategy works by considering only a subset of the PB constraints: the ones falsified by intermediate assignments. This lazy approach is very useful in applications like synthesis, where we expect the PB constraints to be satisfiable. For such applications, our approach tends to steer the SAT solver towards satisfying assignments. In addition since we return partial assignments, we can return many solutions simultaneously. If the formula is unsatisfiable, by focusing on the PB constraints that are falsified, we may wind up discovering an unsatisfiable core of PB constraints without encoding all the PB constraints.

B. Discovering More Units

SAT solving and propagation at the PB level as per algorithm 1 may not discover *all possible* units. The reason is that a SAT solver does not discover all the units implied by a propositional formula during the search process. Algorithm 2 offers a practical way to discover more units implied by Φ. The basic idea is as follows: we first find A, a satisfying (partial) assignment for Φ. Now, suppose that literal l is true under A, then l may be a unit (certainly $\neg l$ is not), which we check with the SAT query $\Phi \wedge \neg l$. We can control the time this operation takes by imposing a limit on a resource R, for example the decisions or the propagation steps that the SAT solver performs (call to SAT-LIMITED in line 9).

Relying on a single assignment is not a good idea. Instead, we maintain a set α that contains different assignments for Φ. We only perform queries of the form $\Phi \wedge \neg l$ on variables for which every assignment in α assigns a value (recall assignments are partial) and all these values are equal (condition in line 7). We note that this check and the assignment in line 8 can be implemented efficiently using bit-vectors. If a query on a formula $\Phi \wedge \neg l$ returns an assignment, this assignment also satisfies Φ, so we add it to the set α.

Algorithm 2 Extracting units implied by a formula Φ

```
1:  procedure MORE-UNITS(Φ)
2:      A, U ← SAT(Φ)
3:      if A = UNSAT then return UNSAT
4:      α ← {A}
5:      for all l ∈ U do Φ ← Φ ∧ l
6:      for all variables v s.t. v ∉ U ∧ ¬v ∉ U do
7:          if ∀A1, A2 ∈ α : A1(v) = A2(v) then
8:              l ← POLARITY(A′, v) for some A′ ∈ α
9:              B ← SAT-LIMITED(Φ ∧ ¬l, R)
10:             if B = UNSAT then
11:                 U ← U ∪ {l}
12:                 Φ ← Φ ∧ l
13:             else α ← α ∪ {B}
14:     return PICK(α), U
```

In addition to the units implied by Φ, MORE-UNITS returns a satisfying assignment if there is one. The assignment returned can be any of the assignments in α. Notice that we can simply instantiate SAT in algorithm 1 with MORE-UNITS. In our implementation, we use MORE-UNITS only on the initial formula that contains PB-clauses and cardinality constraints.

C. Optimization

Algorithm 1 can be extended to handle optimization problems. Assume that the problem is minimizing the objective function $f(X)$. Whenever we get a satisfying assignment such that $f(X) = V$, we add the constraint $f(X) < V$ in order to obtain solutions that decrement $f(X)$ by at least 1. We also reset the variable phases to random values, so that the next assignment will not be a small variation of the current assignment. When the problem becomes unsatisfiable, we report the last known V and the corresponding assignment.

Solving optimization problems by decrementing by 1 is naive, but straightforward to implement using an incremental SAT solver. We could have used binary search or some related approach. That would require backtracking, which can be implemented using assertion literals. Our preliminary analysis indicated that using binary search would not have helped us solve more optimization problems in the PB Competition [6], hence we did not implement it. However, as we note in Section IV, the community needs more industrial PB benchmarks.

III. TRANSLATION TO CNF

In this section, we explain how different encodings of PB constraints affect the behavior of our incremental solver. Encodings of PB constraints into CNF differ (i) in the size of the resulting formulas, and (ii) with regards to the implications preserved between the variables.

The notion of *arc-consistency* captures the desired property of preserving implications: an encoding (say the one generated by the function TRANSLATE) is *arc-consistent* if an assignment that can be propagated on the original constraints can also be propagated on the translated constraints. For a partial assignment A, a PB constraint C and a literal l, if A can be extended to a model of C but $A \cup \{l\}$ cannot, then unit propagation on TRANSLATE(C) and A will produce $\neg l$. Choosing an encoding is a trade-off between (proximity to) arc-consistency and size.

We implemented translations through adders and BDDs, as described in [3]. The encoding through adders is linear, but it does not maintain arc-consistency. It works by synthesizing a network of adders that adds up the terms in the left-hand side, and a circuit comparing the sum to the right-hand side. The encoding for the sum bit of full adders requires ternary XORs, which are known to be problematic for SAT solvers. However, adder-based encodings have the advantage that they lead to small formulas. The benefit of an incremental approach in this case is that we detect implications of the units we learn by performing PB unit propagation, and simplify the problem accordingly. Some of these implications would be lost if we translated everything at once.

In contrast to adders, translation through BDDs is arc-consistent; however, the size of the resulting BDDs is exponential in the worst case. In our examples, some of the original PB constraints are practically impossible to translate through BDDs. Translation becomes possible after we learn units and simplify the problem. In fact, our lazy algorithm sometimes allows us to solve problems without even constructing BDDs for PB constraints we could not directly translate.

Another reason to prefer BDDs is that they can represent conjunctions of constraints. Frequently there are sets of PB constraints with identical sets of variables. We can hash all constraints with the sorted list of variables as the signature, and conjoin the constraints mapped to the same hash value. It is straightforward to adapt the BDD construction algorithm of [3] to build BDDs for conjunctions of constraints. This can lead to more compact encodings. More importantly, we can achieve arc-consistency for the conjunction of constraints.

In general, we can mix encodings and pick the most suitable for each constraint. We use adders only when the BDD is too big, but we could also use sorters. Incremental translation allows us to use BDDs more frequently.

IV. EXPERIMENTAL EVALUATION

PB-SAT is implemented in Common Lisp and uses PicoSAT [7] as the backend. In principle we can use any SAT solver that provides incremental functionality. The source code is publicly available.[2] We evaluate PB-SAT with instances arising from industrial design problems, and with instances from the 2010 PB Competition [6]. We used three servers equipped with two 4-core Xeon X5677 (3.47GHz) CPUs each, and 32GB or 96GB of RAM. In Section IV-A, we provide evidence of one of our claimed contributions, *viz.*, that we have improved the state of the art in SAT-based approaches to solving PB problems. In Section IV-B, we provide evidence that SAT-based PB solvers should be part of any portfolio of solvers.

A. Industrial Design Problems

We used our solver with a family of 20 industrial PB instances generated by the CoBaSA design tool [8], where 16 are satisfiable, and 4 unsatisfiable. The instances encode *system assembly problems*: an assignment is a way to assemble system components so that various requirements are met. The basic components in these problems are anywhere from 8 to 22 cabinets that provide resources (including CPU time, memory and networking), about 200 applications that consume resources, and up to 300 memory spaces. Applications and memory spaces have to be mapped to cabinets subject to various constraints, which we are going to briefly describe. See [9] for a detailed description of these problems.

The most important variables in these instances are called *map variables*: $M_{c,p}$ is true iff the resource consumer c (*e.g.*, an application or memory space) is mapped to cabinet p. Each

[2] http://www.ccs.neu.edu/home/vpap/pb-sat.html

	# instances	CPLEX	bsolo	wbo	SAT4J	MS+	PB-SAT	VPS1	VPS2
aardal_1	14	14	14	14	14	14	14	14	14
uclid	50	23	45	44	44	48	49	48	49
tsp	100	90	98	100	100	100	100	100	100
wnqueen	100	97	100	100	100	100	100	100	100
dbst	15	13	15	15	15	15	15	15	15
fpga	57	57	57	39	38	39	39	57	57
armies	12	7	6	6	7	6	8	10	10
pigeon	40	39	21	4	3	3	2	39	39
j{30,60,90,120}	81	66	65	68	68	67	67	68	68
rest	17	10	9	7	9	7	8	13	13
all	486	416	430	397	398	399	402	464	465
average time (sec)	-	135.8	38.0	70.3	67.8	83.1	67.7	-	-

(a) Decision Problems

	# instances	CPLEX	bsolo	wbo	SAT4J	PB-SAT	VPS1	VPS2
feature subscription	20	0	19	10	20	19	20	20
caixa	21	21	21	21	21	21	21	21
j{30,60,90,120}	80	47	52	55	55	55	55	55
area	69	69	25	47	11	22	119	120
logic synthesis	74	71	51	27	24	30	71	71
routing	15	15	15	15	15	15	15	15
primes	156	124	105	105	104	114	127	131
factor	192	192	192	190	192	192	192	192
rest	212	137	100	109	100	72	160	167
all	939	676	580	579	542	540	780	792
average time (sec)	-	30.8	50.2	48.8	25.7	81.3	-	-

(b) Optimization Problems

Fig. 1. Experimental Results: Small Integers, Linear Constraints (timeout after 1800 seconds, 2GB RAM limit)

application j has to reside on exactly one cabinet, so we have cardinality constraints of the form

$$\sum_{p \in P} M_{j,p} = 1,$$

where P is the set of cabinets in the system. We also have resource requirements. For a cabinet p that provides r_p units of the resource r, let C_p be the set of consumers that can be potentially be mapped to p. Each $c \in C_p$ needs r_c units of the resource r. We thus have an inequality

$$\sum_{c \in C_p} r_c M_{c,p} \leq r_p.$$

In addition, we have structural requirements, like co-location or separation of components. These requirements are expressible as propositional constraints. For example, if the applications j_1 and j_2 have to be co-located, for every cabinet p there is a constraint $M_{j_1,p} \iff M_{j_2,p}$. Therefore, the instances contain a balanced mix of propositional and arithmetic constraints.

The ILP and native PB solvers we tried worked very well for this class of problems, unlike existing SAT-based approaches. To understand why, we look at a representative instance in more detail. The best result with MiniSat+ [3] is 91 minutes: CNF generation through sorters takes 80 minutes, and PicoSAT can find a satisfying assignment in 11 minutes. All other translation schemes and different SAT solvers give worse results. For example, we ran MiniSat+ using a BDD encoding for 2 hours, at which point it failed to complete and was using over 80GB of RAM. In contrast, PB-SAT can solve the instance in 32 seconds (21 seconds of PicoSAT time) using BDDs for the translation, with the units extraction mechanism of subsection II-B disabled. The reason is that we learn a significant number of units that allow us to simplify the problem. Constraint propagation reveals 7938 units. Before the 7th and last call to PicoSAT, we know 8240 PB units. Algorithm 2 leads to even more learned units, even with a limit of 10 decisions per query: its execution takes 0.5 seconds, and after its execution we know 8506 PB units. These units improve the running time to 19 seconds. SAT solving accounts for 9 seconds.

The instances have between 14000 and 21000 variables and between 68000 and 93000 constraints. PB-SAT solves all instances, taking 62 seconds on average; MiniSat+ timed out (1800 seconds) for all instances and translation schemes.

B. Pseudo-Boolean Competition Instances

For the sake of completeness, we include experimental results for instances from the PB competition (figure 1). We only provide results for instances with small integers

and linear constraints, because a wide range of solvers is available for these. We compare against bsolo [10], wbo [11], SAT4J [12], CPLEX [13] and MiniSat+. We run the best known configuration of each solver: bsolo with cardinality constraint learning, and the resolution version of SAT4J. In the case of MiniSat+, we generate CNF formulas and run PicoSAT, for a direct comparison with our solver. PB-SAT solves 35 decision instances less than the best solver. The difference can be attributed to hand-crafted instances, some of which contain pigeonhole-like problems (*e.g.*, "pigeon" and "fpga").

The results include two *virtual portfolio solvers*. VPS2 stands for a solver that would run all solvers in parallel and report the best result. VPS1 is VPS2 minus PB-SAT. VPS1 "solves" 30 more decision instances than the best solver, and our solver adds an extra instance to the mix. The combination of PB-SAT and CPLEX [13] solves the same instances as VPS1. The combination of CPLEX and any other solver follows closely (1-3 instances less), while any combination of two without CPLEX solves at most 443 instances. PB-SAT contributes 12 extra optimization instances. According to this analysis, the number of solved instances a solver contributes to a portfolio of solvers is valuable information, due to the diversity of techniques. Translation to SAT is a useful addition.

Interestingly, we do not learn *any* units before the last SAT query for 448 out of the 486 decision instances. These instances either consist entirely of clauses and cardinality constraints, in which case we encode everything at once, or the intermediate formulas do not imply any units. For these problems, our incremental approach obviously does not yield any improvements. These benchmarks problems *are not* characteristic of the industrial problems we have seen, and we encourage the community to contribute industrial PB problems to the PB competition benchmark suite.

V. Related Work

Different encodings of PB constraints into SAT have been proposed. Bailleux et al. [4] describe a variant of the BDD encoding. Een and Sorensson implemented the MiniSat+ PB solver [3], which uses adders, sorters, and BDDs. Bailleux et al. [5] present the first polynomial arc-consistent encoding. Abio et al. [14] revisit BDDs, and provide a polynomial, arc-consistent, BDD-based encoding. In addition, encodings for cardinality constraints (an interesting special case) have been explored (*e.g.*, [15]).

Our algorithm can be viewed from the perspective of lazy SMT [16], as we introduce just enough information for the SAT solver to find a consistent assignment, or prove unsatisfiability. SMT has already been used to tackle PB problems: Cimatti et al. [17] extend the SMT framework with the theory of costs C, and use it to express PB constraints. Our approach differs from SMT in that we actually encode the PB constraints, as opposed to learning a clause that precludes a single theory-inconsistent conjunction of literals.

Our technique also bears resemblance to Abstraction-Refinement, *e.g.*, as applied to the theory of arrays [18]. We abstract the problem by omitting information from the encoding, and then refine the abstraction based on assignments that satisfy the partial encoding but not the PB formula.

VI. Conclusions

We presented an algorithm for pseudo-Boolean solving by incremental translation to SAT, and implemented a solver based on this algorithm. Incrementality allows our solver to use unit literals derived from intermediate SAT queries to simplify pseudo-Boolean constraints. In addition, we learn units from constraint propagation at the pseudo-Boolean level. Experimental evaluation on industrial problems shows that our solver improves the state of the art in SAT-based approaches to pseudo-Boolean problems and that any portfolio solver should include a SAT-based solver.

Acknowledgments

This research is funded in part by NASA Cooperative Agreement NNX08AE37A and NSF proposal CCF-1117184. This article reports on work supported by the Defense Advanced Research Projects Agency under Air Force Research Laboratory (AFRL/Rome) Cooperative Agreement No. FA8750-10-2-0233. The views expressed are those of the authors and do not reflect the official policy or position of the Department of Defense or the U.S. Government. We would like to thank some of the anonymous reviewers for making helpful suggestions.

References

[1] D. Chai and A. Kuehlmann, "A Fast Pseudo-Boolean Constraint Solver," in *DAC*, 2003.
[2] H. M. Sheini and K. A. Sakallah, "Pueblo: A hybrid pseudo-boolean SAT solver," *JSAT*, vol. 2, pp. 165–189, 2006.
[3] N. Een and N. Sorensson, "Translating Pseudo-Boolean constraints into SAT," *JSAT*, vol. 2, pp. 1–26, 2006.
[4] O. Bailleux, Y. Boufkhad, and O. Roussel, "A Translation of Pseudo Boolean Constraints to SAT," *JSAT*, vol. 2, pp. 191–200, 2006.
[5] O. Bailleux, Y. Boufkhad, and O. Roussel, "New Encodings of Pseudo-Boolean Constraints into CNF," in *SAT*, 2009.
[6] Vasco Manquinho and Olivier Roussel, "Pseudo-Boolean Competition 2010." See `http://www.cril.univ-artois.fr/PB10/`.
[7] A. Biere, "PicoSAT Essentials," *JSAT*, vol. 4, pp. 75–97, 2008.
[8] P. Manolios, D. Vroon, and G. Subramanian, "Automating component-based system assembly," in *ISSTA*, 2007.
[9] C. Hang, P. Manolios, and V. Papavasileiou, "Synthesizing Cyber-Physical Architectural Models with Real-Time Constraints," in *CAV*, 2011.
[10] V. M. Manquinho and J. Marques-Silva, "On Using Cutting Planes in Pseudo-Boolean Optimization," *JSAT*, vol. 2, pp. 209–219, 2006.
[11] V. Manquinho, J. Marques-Silva, and J. Planes, "Algorithms for Weighted Boolean Optimization," in *SAT*, 2009.
[12] "SAT4J." See `http://www.sat4j.org/`.
[13] "CPLEX." See `http://www-01.ibm.com/software/integration/optimization/cplex-optimizer/`.
[14] I. Abio, R. Nieuwenhuis, A. Oliveras, and E. Rodriguez-Carbonell, "BDDs for Pseudo-Boolean Constraints - Revisited," in *SAT*, 2011.
[15] J. Marques-Silva and I. Lynce, "Towards Robust CNF Encodings of Cardinality Constraints," in *CP*, 2007.
[16] R. Nieuwenhuis, A. Oliveras, and C. Tinelli, "Solving SAT and SAT Modulo Theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T)," *JACM*, vol. 53, no. 6, pp. 937–977, 2006.
[17] A. Cimatti, A. Franzen, A. Griggio, R. Sebastiani, and C. Stenico, "Satisfiability Modulo the Theory of Costs: Foundations and Applications," in *TACAS*, 2010.
[18] V. Ganesh and D. L. Dill, "A Decision Procedure for Bit-Vectors and Arrays," in *CAV*, 2007.

Automated Specification Analysis Using an Interactive Theorem Prover

Harsh Raju Chamarthi
Northeastern University
Email: harshrc@ccs.neu.edu

Panagiotis Manolios
Northeastern University
Email: pete@ccs.neu.edu

***Abstract*—A method for analyzing designs and their specifications is presented. The method makes essential use of an interactive theorem prover, but is fully automatic. Given a design and a specification, the method returns one of three possible answers. It can report that the design does not satisfy the specification, in which case a concrete counterexample is provided. It can report that the design does satisfy the specification, in which case a formal proof to that effect is provided. If neither of these cases hold, then a summary of the analysis is reported. We have implemented and experimentally validated the method in ACL2s, the ACL2 Sedan.**

I. Introduction

Many formal methods techniques have been developed that help designers build complex, dependable systems. At one extreme we have interactive theorem proving, which places few restrictions on the kinds of systems and properties that can be verified, but which requires well trained professionals with a deep understanding of logic and proof. At the other extreme, we have methods that find certain classes of errors in a fully automated way, but which place severe restrictions on the kinds of systems and properties they can analyze.

Is it possible to have the best of both worlds? Is it possible to have a powerful, expressive modeling language with a powerful deductive engine that can be used to interactively prove theorems *and* that can be used to automatically generate counterexamples? In this paper, we show how to do just that. We present an algorithm that makes essential use of interactive theorem proving technology but analyzes specifications in a fully automated way.

Our algorithm allows us to turn an interactive theorem prover into an *extensible*, *automatic*, analysis tool that can be used by regular engineers to provide increased assurance in the correctness of their designs. The user is responsible only for modeling and specifying the properties of their design; they are not responsible for providing proofs. It is in this regard that our approach is *automatic*. Our approach is *extensible* because it can exploit any existing or newly developed libraries of definitions, theorems and proof techniques. For example, the use of libraries for reasoning about non-linear arithmetic, set theory, the theory of lists, etc, can lead to significant improvements in the ability to prove theorems and to generate counterexamples.

The main idea of our algorithm is to use the deductive verification engine of an interactive theorem prover to semantically decompose properties into subgoals that are either shown to be true or that can be tested to find counterexamples. Deduction and testing proceed in an interleaved, synergistic fashion. When the deductive engine generates a subgoal that it cannot further simplify, we test it by selecting a variable in the subgoal and assigning it a value. We then use the deductive engine to propagate the consequences of that assignment, which may lead to further deductive simplifications or to backtracking if propagation reveals a conflict. At this level of abstraction, the process is similar to the DPLL select, assign, propagate loop. There are significant differences with DPLL, however. Variables can be over infinite domains, so selecting variables and assigning them reasonable values requires a careful analysis. Propagation in our context can involve arbitrary deductive reasoning, *e.g.*, it can prune away infinite subspaces. Backtracking also requires care because it is very difficult to analyze conflicts when variables range over infinite domains.

We present an abstract algorithm that makes minimal assumptions about the underlying interactive theorem prover. The assumptions are outlined in Section II and the abstract algorithm is presented in Section III. We have implemented our algorithm in the ACL2 Sedan (ACL2s), a freely available, open-source, well-supported theorem prover that uses ACL2 as its core reasoning engine. ACL2s is an Eclipse plug-in that provides a modern integrated development environment designed to bring computer-aided reasoning to the masses. ACL2s has been used in several sections of a required freshman course at Northeastern University to teach several hundred undergraduate students how to reason about programs. We evaluate our algorithm in Section IV. We present a case study on hardware verification and we also compare our algorithm with Alloy on a collection of examples from the literature. In addition, our algorithm was used by freshmen students to debug their programs and specifications. For this purpose, the algorithm was very successful, as in almost all cases, it was able to automatically to generate counterexamples when students made mistakes. Related work appears in Section V and conclusions in Section VI.

II. Preliminaries

In this section, we outline the assumptions our algorithm depends on. We assume that the specification language L is a multi-sorted first-order logic which can be extended by

introducing new function and predicate symbols using well-founded recursive definitions, and that L executable.

We further assume that properties (also interchangeably referred to as formulas, conjectures, or specifications\) have no nested quantifiers and are of the form $hyp_1 \wedge \cdots \wedge hyp_n \Rightarrow concl$. Properties are implicitly universally quantified.

We assume the existence of an Interactive Theorem Prover (ITP) than can reason about specifications written in L. We will treat the ITP as a blackbox and all that we require from the ITP are two procedures: SMASH and SIMPLIFY.

SMASH takes as input a *goal*, a well-formed formula written in L, and returns a list of *subgoals*. We require that SMASH preserves validity, *i.e.*, the conjunction of the subgoals returned is valid iff the original goal is valid. Modern interactive theorem provers use various techniques for this, including decision procedures for Boolean logic, case analysis, evaluation, linear arithmetic, congruence closure, and rewriting.

SIMPLIFY takes as input an L-formula, c, and a list of assumptions, H. SIMPLIFY *simplifies* c assuming H is true, and returns a single formula that is equivalent to c under assumptions H.

An *assignment* of a formula is a mapping from the free variables in the formula to values in the domain of L. An assignment may fail to satisfy all hypotheses, $hyp_1, \cdots hyp_n$ of a formula P. In such a case, we say that the assignment is *vacuous*. Vacuous assignments are not helpful. For example, suppose that we are analyzing a compiler, whose specification says that the compiler transforms well-formed programs into semantically equivalent well-formed programs. That this property holds for ill-formed programs is trivial, and not interesting. Therefore, we classify assignments as either: (1) *vacuous*, assignments that do not satisfy all of the hypotheses, (2) *counterexamples*, assignments that satisfy all the hypotheses, but not the conclusion or (3) *witnesses* assignments that satisfy all the hypotheses and also the conclusion. We note that in order to simplify the presentation, in this paper we use assumptions that are stronger than they really need to be. For example, in ACL2s, we do not require that all functions are executable.

Algorithm 1 Analyze

Input: Property P

1: $n := 0$
2: **while** $\neg SCond \wedge n \leq$ SLIMIT **do**
3: $\quad A, n :=$ Search$(P), n+1$
4: $\quad$ update *summary* (record counterexample)
5: **if** $SCond$ **then**
6: $\quad$ print *summary* and **exit**
7: $S :=$ SMASH(P)
8: **if** $S \neq \{P\} \wedge S \neq \{\}$ **then**
9: $\quad$ **for all** $p \in S$ **do**
10: $\quad\quad$ Analyze(p)
11: **if** P is "goal" **then**
12: $\quad$ print *summary* and **exit**
13: **return**

III. THE ABSTRACT ANALYZE ALGORITHM

Analyze (algorithm 1) takes as input a property P and analyzes P by recursively decomposing P into simpler properties and searching for counterexamples to them.

Analyze first (lines 2-4) tries to repeatedly search for counterexamples until either a user-defined stopping condition is satisfied or limit on the number of search attempts is reached. The limit is a user-defined parameter stored in SLIMIT. The procedure Search (described next) uses a DPLL-like algorithm to incrementally search for falsifying assignments to P. Assignments obtained are checked (if they are indeed complete falsifying assignments) and recorded as counterexamples.

Useful information is tracked in a global data structure *summary*. It is used to record counterexamples, successful proofs (if P was proved by the theorem prover), subgoals that failed to provide either proofs or counterexamples (these subgoals, which correspond to a particular case of the original property can be examined more closely by the user) and other statistics like, the number of unsuccessful search attempts, the number of counterexamples and witnesses found,[1] and the number of subgoals analyzed.

The user-specified stopping condition is a predicate on *summary*, for example a typical stopping condition would be: *number of counterexamples found should be greater than 3*; a more intricate stopping condition would involve some notion of coverage. If the user-specified stopping condition is satisfied a summary of the analysis is printed and the procedure exits. Otherwise the property is semantically decomposed (line 7) using the SMASH procedure of the theorem prover into simpler properties. Each such simpler property is recursively analyzed (lines 9-10). In case the theorem prover is unable to simplify the input property, or it successfully proves the validity of the input property, the appropriate information is recorded and the procedure simply returns, unless the input property is the top-level *goal*, in which case (lines 11-12), we print the summary and exit.

Searching for counterexamples

Search (Algorithm 2) takes as input a property P and searches for a counterexample by incrementally constructing a complete (falsifying) assignment to P. The algorithm proceeds by selecting a free variable, assigning it a value and propagating this new information to obtain a partially instantiated property P'. If P' is clearly inconsistent, then we backtrack, otherwise we continue till we obtain a complete assignment.

The partial assignment is stored in the local stack A. Stacks S and B record information necessary to backtrack to an earlier state (iteration) of the search process. S stores the sequence of partially instantiated properties. B stores the sequence of variables in the order in which they were selected. B also associates two values with each variable, i) number of assigns made to the variable and ii) a string recording the type

[1] To simplify the exposition we only show how counterexamples are found, but witnesses can also be found in a similar manner.

of assignment to the variable, if it was decided by Assign, string "decision" is stored, else string "implied" is stored.

Algorithm 2 Search

Input: Property P,
1: **local** Assignment A
2: **local** Stack B (of (var, # assigns, type of assign))
3: **local** Stack S (of Property)
4: A, S :=[ ], push(P, S)
5: x_0 := Select(P)
6: push((x_0, 0, "na"), B)
7: **while** A is not complete **do**
8: P := head(S)
9: $(x, i, _)$:= head(B)
10: **if** P has a constraint of form $x = c$ **then**
11: $v := [|c|]$; $t :=$ "*implied*"
12: **else**
13: v, t := Assign(x, P)
14: /*Update # and type of assign of x*/
15: pop(B); push(($x, i+1, t$), B)
16: **if** $x = x_0$ and $i >$ BLIMIT **then**
17: **return** *fail*
18: P' := Propagate(x, v, P)
19: $hyps$:= hyps(P'); $concl$:= conclusion(P')
20: **if** $false \notin hyps \wedge concl \neq true$ **then**
21: A, S := push((x, v), A), push(P', S)
22: B := push((Select(P'), 0, "na"), B)
23: **else** /*Inconsistent assignment */
24: **repeat**
25: S, B, A := pop(S), pop(B), pop(A)
26: (x', i', t') := head(B)
27: **until** (t' = "*decision*" $\wedge$ $i' <$ BLIMIT) $\vee$ size(B) ≤ 1
28: **if** $x' = x_0 \wedge (t'$ = "*implied*" $\vee$ i' = BLIMIT) **then**
29: **return** *fail*
30: **return** A

Procedure Search first initializes A to be empty and pushes P onto stack S. It calls the procedure Select (described next) to choose the first variable x_0 to be assigned. x_0 is pushed onto stack B, its assign counter(number of times the variable is assigned) is initialized to 0 and the string specifying the type of assignment is set to "na" (denoting *not assigned*).

The main search loop (lines 7-29) implements the iterative construction of A. The selected variable x and property P in the current iteration of the search loop are obtained by reading the top of the stacks B and S. If x is constrained by an equality($x = c$ where c is a constant expression), then we simply assign x the value v (obtained by evaluating c), otherwise, the instantiation is performed by the procedure Assign which returns the value v to be assigned to x and also the type of assignment t. The assign counter for x is incremented and the type of assignment is recorded in B. We will defer discussing the details of Assign to the next section, for now think of it as an oracle that finds a value v that satisfies simple local constraints involving only x. $\triangleright$ [2]

The procedure Propagate (described later) is used to simplify P using the theorem prover in light of the new assignment to x, deducing as much new information as possible, resulting in either a partially concretized property(P') or an inconsistency. Inconsistency is (syntactically) recognized if either $false$ is found in the hypotheses(of P') or the conclusion(of P') is equal to $true$.

If no inconsistency was found (checked in line 20), the assignment A is extended, the partially concretized property P' is pushed onto S and a free variable from P' is selected and pushed onto B to be instantiated in the next iteration of the main search loop (lines 21-22).

If an inconsistency is found, we backtrack to the last decision (by popping the stacks and undoing the assigns in A) that has not exhausted its limit, SLIMIT (lines 24-27). While backtracking to the last decision, we never pop the first variable selection stored in B at the start of the search loop. If assigns to x_0 are exhausted, then Search fails (lines 16-17, 28-29), moreover if we backtracked to x_0 and it's type of assign is "implied" then too we return *fail*. The search is repeated until all free variables have been assigned values (line 7) returning a complete assignment (line 30).

Algorithm 3 Select

Input: P is a property
1: Do congruence closure on P
2: G := buildVariableDependencyGraph(P)
3: dag_G := ComputeSCCs(G);
4: $sortedList_{dag_G}$:= TopologicalSort(dag_G)
5: X := pickLast($sortedList_{dag_G}$)
6: **if** X is set (of mutually-dependent variables) **then**
7: **return** some vertex in X
8: **else**
9: **return** X

Selecting variables to assign

Select (Algorithm 3) procedure describes the mechanism to choose a variable in a property. It takes as input a property P, performs static analysis to determine a certain type of dependency relationship among the variables (described below) of P, and selects the variable with the *least* dependency. We will motivate this notion of *dependency* in the context of the Search algorithm with an example. In the following x, y, z, w are constrained to be integers and $hash$ is a standard hash function.

$$P: \quad z = y^2 \wedge y = hash(x) \wedge w = hash(y) \Rightarrow z > w^2$$

Since we are interested in finding counterexamples, we have four constraints to satisfy, the first three are the hypotheses, and the final constraint is the negated conclusion. Lets assume

[2] In the concrete algorithm (next section) we randomly sample the variable's type domain, but in general one could use more heavyweight methods such as constraint-solving.

there is some procedure available for assigning a value to a variable without falsifying any constraint. Which variable should we (select and) assign first? Notice that equality constraint fixes the value of y as soon as x is assigned, and the value of z and w as soon as y is assigned a value that does not falsify other constraints. Clearly choosing x before choosing y is beneficial from the point of view of computation *i.e.*, we just evaluate $hash(x)$ to obtain the value of y. Selecting y before x, causes difficulty in satisfying the constraint $y = hash(x)$, since computing the inverse hash function might be non-trivial. Moreover, any constraint solver used in Assign might not be powerful enough to handle non-linear arithmetic of $hash$. Treating *equality* in a special manner we can see that there is a certain relation among the variables of the constraints that is similar to the notion of data dependency in compiler literature. We shall call such a relation a *v-dependency* which we define more precisely below. The idea behind the algorithm is to select the variable with the *least dependency*, breaking down the task of simultaneously solving the constraints, into a more local directed approach of solving the constraints one by one; we want to finally select variables in an order such that we can reduce the chances of running into an inconsistency and backtracking. We construct a directed graph with variables as nodes and the directed edges in the graph denote the *depends on* binary relation. The edges are also annotated with the logical relation that caused the edge to be drawn in the first place. We call an edge annotated with relation R an R-edge. The *variable dependency graph* for P initially consists of only nodes, one for each variable and no edges. The graph is constructed by iterating over the constraints of P using the following rules, which form the core of the procedure buildVariableDependencyGraph. We assume x and y are (distinct) free variables of P and *term* is inductively defined to be either a variable, a constant expression, or a function application with arguments that are *terms*.

1) If P has a constraint of the form $x = c$, where c is a constant expression, we force x to be a leaf node (no outgoing edges). Once a node is marked leaf, it overrides the other rules.
2) If P has a constraint of the form $x =$ *fterm* such that $y \in$ freeVars(*fterm*) and $x \notin$ freeVars(*fterm*), we add an $=$-edge from node x to node y. *fterm* is a function application as defined above.
3) If P has a constraint of the form $x \bowtie$ *fterm* such that $\bowtie$ is a binary relation, $y \in$ freeVars(*fterm*) and $x \notin$ freeVars(*fterm*), we add an $\bowtie$-edge from node x to node y.
4) If P has a constraint of the form $x \bowtie y$ where $\bowtie \in \{<, \leq, >, \geq\}$ we don't draw an edge.
5) If P has a constraint of the form *R(term*$_1$, *term*$_2$, ..., *term*$_n$*)*, such that $x \in$ freeVars(*term*$_i$), $y \in$ freeVars(*term*$_j$), $i \neq j, n \geq 2$ and R is an arbitrary n-ary relation, then we perform the following. Let $n_=$ and $n_\bowtie$ be the number of incoming edges to a node labeled with $=$ and $\bowtie$ respectively. If x and y have no incoming or outgoing edges, we add a bidirectional R-edge between x and y, else we add a R-edge between x and y pointing to the node (variable) that has a greater $(n_=, n_\bowtie)$ value lexicographically, else we don't add an edge.

Using the above definition of *v-dependency*, procedure Select constructs the *variable dependency graph* for P after applying congruence closure (replace equivalent variables by their representative chosen lexicographically) to P (lines 1-2). Congruence closure helps simplify the graph since constraints such as $x = y$ are quite common. After the graph is constructed, using the forementioned rules, its strongly-connected components are computed (lines 2-3). The resultant directed acyclic graph (dag) obtained is topologically sorted. The algorithm then picks the component (a set of variables) which has no outgoing edges (*i.e.*, has no *dependency* on other components). If the component has just one variable, then usually it is the node which is a leaf (*i.e.*, no dependency), in which case we return it. If there are more than one variables to choose from (in case of multiple variables in the connected component), the procedure returns the variable with the lexicographically smallest name (lines 4-6). Note that Select tries to ensure the following rule of thumb: select a variable only when every variable it *depends* on has already been assigned a value; this is not always the case.

Algorithm 4 Propagate

Input: Var x, Value v, Property P
1: $hyps :=$ hyps(P)
2: $hyps$.add$(x = v)$
3: $shyps :=$ simplifyAssumingRest$(hyps)$
4: $concl :=$ conclusion(P)
5: $sconcl :=$ SIMPLIFY$(concl, shyps)$
6: $P' := \bigwedge shyps \Rightarrow sconcl$; **return** P'

Propagating new assignments

Example

We illustrate the working of Search on a simple example involving numbers and some arithmetic functions. Consider the following property P defined on integers x, y, z, w; $hash$ and min are textbook *hash* and *minimum* functions.

$$x = hash(y) \wedge y = hash(z) \wedge z > 0 \wedge w < min(x, y) \Rightarrow w < z$$

Before the main search loop begins, a variable is selected to be instantiated. The variable dependency graph for P (constructed following the forementioned rules) is shown in Figure 1.

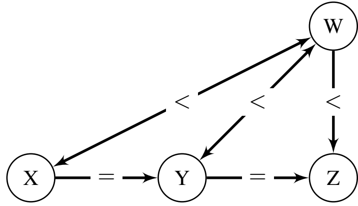


Fig. 1. Variable dependency graph for P

The graph has two strongly-connected components, one containing just z, the other containing x, y, w. The topological sort returns the vertices in decreasing finish times of the depth-first search on the *dag*. We pick the last component (*i.e.*, the one with the least dependency). Since this component z is not a set we simply return z. After having selected the variable to instantiate (z), we use Assign to pick a value for it, satisfying the local constraint on it, $z > 0$, along with the implicit constraint that z is an integer. Lets say the oracle procedure Assign picked 1. Then we propagate this assignment by adding the constraint $z = 1$ in P and using the ITP to simplify the hypotheses and conclusion in light of this new information. Propagate returns the following simplified property:

$$P' : \quad x = hash(y) \wedge y = 5184444 \wedge w < min(x, y) \Rightarrow w < 1$$

whose dependency graph is shown in Figure 2.

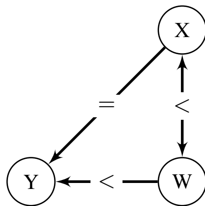


Fig. 2. Dependency graph for P'

Since $false$ does not appear in the hypotheses(and neither does $true$ in the conclusion), P' is not inconsistent and we add $z = 1$ to the partial assignment A and the search for the rest of the assignment is continued. Note that Propagate helps eliminate some edges in the variable dependency graph of P, breaking cycles(mutual dependency) in the connected component, invariably helping the Select algorithm in the next iteration of the main search loop.

The motivation for Propagate is that one assignment to a variable, should result in assignment of the maximum number of remaining variables. In this case, the assignment to z, results in y being selected (because it is a leaf node) and being directly assigned a value by virtue of the equality constraint $y = 5184444$. Notice that this is an assignment of type "implied" and was propagated due to the decision assignment ($z = 1$) by the oracle procedure Assign in the previous iteration. This information is again propagated resulting in the furthur grounded property:

$$P'' : \quad x = 5562452 \wedge w < min(x, 5184444) \Rightarrow w < 1$$

whose dependency graph is shown in Figure 3

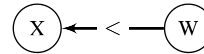


Fig. 3. Dependency graph for P''

Notice that since x is constrained to be equal to the constant expression 5562452, it is a leaf node and this eliminates the edge that existed in Fig 2 from x to w. The last node in the topological sort of the *dag* of Fig 3, x, is returned by Select, thereby forcing a value satisfying the equality constraint $x = 5562452$. This assignment is further propagated, resulting in the almost grounded property having just one free variable:

$$P''' : \quad w < 5184444 \Rightarrow w < 1$$

Assigning w (using implicit constraint that w is an integer and the local constraints $w < 5184444$ and $w \geq 1$) a value 0 or value 5184445, will lead to inconsistency (after the propagation), in which case we need to throw away the current assign. If in the process we exhaust the limit on number of assigns (BLIMIT) for w we backtrack all the way to the decision variable z, by undoing the assignment for x and y, in A, popping P'' and P' from S and continuing (the main search loop). If an assign to w, say $w := 2$, did not lead to an inconsistency, then we have a complete assignment A, we quit the loop and return A, which is a counterexample of P.

We have implemented the proposed method in ACL2 Sedan [10]. We employ the ACL2 interactive theorem proving system [15] to provide the interface methods SIMPLIFY and SMASH. The engineering of the interface with the ACL2 theorem prover and the extension to ACL2, in support of this interface, is described in [5]. We will briefly describe how we implemented the Assign method that was left unspecified. In view of delegating most of the heavy work to the theorem prover we incorporated the lightweight method of random testing inspired by the success of Quickcheck-like tools [6]. ACL2 formulas tend to be executable, hence testing in ACL2 simply involves executing a formula under an instantiation of its free variables. To assign a value to a variable, we need to know its domain, which in a given formula is decided by the "type-like" hypotheses constraining the variable. The domain can be characterized by an *enumerator* which is a surjective function from natural numbers to elements of the

domain. In our implementation we enable automatic testdata generation by supporting a notion of an enumerable type in the otherwise untyped language of ACL2. Separation of concerns between enumerators and random number generators also gives us the flexibility to choose between pseudo-geometric, pseudo-uniform random testing and bounded exhaustive testing. Assign does static analysis to infer the (enumerable) type of a variable from the type hypotheses of P, if the domain of the type is greater than one, we *decide* a value to return (using the enumerator and the chosen sampling distribution), otherwise, we simply return the *implied* singleton value.

IV. Experimental Evaluation and Discussion

We present two experiments[3] to evaluate our method. In Section IV-A, we present an in-depth hardware case-study, analyzing the design of a simple, yet non-trivial, pipelined machine, demonstrating the effectiveness of our method in uncovering subtle design errors. In Section IV-B we compare our method with the popular Alloy method (Alloy modeling language and Alloy Analyzer). We modeled various Alloy examples in ACL2 and analyzed them with our method. We find counterexamples to all failed properties (falsified by Alloy), but more importantly we prove all the properties that Alloy posits are theorems (based on the absence of small counterexamples). Surprisingly, in addition to the counterexamples, we also found all the proofs, automatically.

A. Hardware: Finding hazards in a Pipeline Machine

Pipelining is a key optimization technique used to increase performance in modern microprocessors. The *instruction-set architecture* (ISA) model is a natural functional specification for any pipelined design. The correctness of the implementation *i.e.*, *machine architecture* (MA) can be established by showing that all behaviors (execution traces) of MA are observationally equivalent to behaviors of its specification (ISA).

We analyze a three stage pipeline, consisting of fetch, read, and execute/write-back stages. The machine fetches an instruction pointed to by the program counter in the fetch stage, reads the source register from the register file in the read stage, and updates the destination register with the result of the operation it performs (execution) in the write-back stage. The primary challenge in designing a correct pipeline implementation is respecting program dependency and avoiding resource conflicts among instructions that are in different stages of the pipeline. Consider the following sequence of ADD instructions:

$$I_1 : r_3 = r_2 + r_1$$

$$I_2 : r_4 = r_3 + r_2$$

Instruction I_2 will read stale data for register r_3, if read phase of I_2 overlaps with the execution phase (write-back) of instruction I_1. In such a scenario (called Read-after-Write data hazard), to correctly handle the data dependency, the pipeline must be *stalled* to allow the older instruction (I_1) to execute and update the destination register (r_3) before the younger dependent instruction (I_2) reads it. In our pipeline machine model, we will on purpose introduce a design error by failing to stall the read for I_2 in the above scenario. Another scenario that we consider is related to handling of branch/jump instructions. By the time, the program counter is updated to fetch from the target of a BEZ/JMP instruction, subsequent instructions from the sequential program code have already been fetched. To prevent the wrongly fetched instruction from polluting the architectural state (control hazard), it is required to invalidate the latches holding information related to instructions from the wrong execution path. A common error occurring in initial phases of the design of a pipeline machine, is to forget invalidating latch 2, in the scenario that latch 1 is invalid (explain a little more).

The objective of the experiment was to evaluate the effectiveness of our method to find these important and subtle design errors (data and control hazards). How do we find these bugs using our method? Given that the designer has written both the ISA and MA models of the pipeline machine, one just needs to formalize the aforementioned correctness definition and analyze it. We will use a notion of refinement, where the main idea is to show that infinite behavior of MA and ISA are observationally equivalent under an appropriate refinement map. By using the theory of Well-founded equivalence bisimulation (WEB) refinement, we can establish this by proving a local property that only requires reasoning about MA states, their successors, and ISA state and their successors [17]. The refinement map is straightforward, except for the matter of relating the program counters of MA and ISA states. Since the observable effect of any instruction only appears in the write-back stage, the observable program counter is simply the PC value of the oldest instruction in the pipeline. Let M' denote the state of the machine after it has taken one step *i.e.*, it has been run for one hardware clock cycle. Then the safety part of our WEB refinement proof obligation is that if ISA state S and MA state M are observationally equivalent, and both take a step to S' and M' respectively, then either S is observationally equivalent to M', or S' is observationally equivalent to M' (stepping MA for one cycle resulted in an observable architecturalfallback change) *i.e.*, `(obs= S M)` $\Rightarrow$ `(obs= S M')` $\vee$ `(obs= S' M')`

Analyzing this high-level property, our method is able to uncover both the design errors in our MA machine which manifested as hazards. The counterexamples (instances of MA that falsified the safety property) were illuminating; they pointed out the kind of hazards and the scenarios in which they occurred. We recommend the reader to play around with the model provided to see if the tool can uncover other scenarios he/she has seen before.

A few observations are in line. No assertions were provided. No lemmas were written down. No manual tests (microprograms) were provided as inputs. No test driver needed to be given. The only effort on part of the designer was in writing

[3] We recommend the reader download the experiments from http://ccs.neu.edu/home/harshrc/fmcad11

the ISA and MA models in ACL2, defining the datatypes (used for automatic test data generation), specifying the abstraction function (for observational equivalence) and formulating the high-level correctness property.

B. Software: Comparison with Alloy

Alloy [13] is a declarative modelling language based on sets and relations, primarily used for describing high-level specifications and designs. Alloy Analyzer [14] is a tool that supports automatic analysis of models written in Alloy. Given a bound on the number of model elements, called *scope*, the Alloy Analyzer (AA) translates Alloy models (and its specifications\) into Boolean formulas, uses off-the-shelf SAT solvers to generate satisfying instances and translates them back to corresponding set and relation instances of the objects in the model. Alloy is a first-order relational logic with transitive closure, which allows expressing rich structural properties using succinct expressions. However to enable feasible automatic analysis, it has poor support for two features that we feel naturally apply in many types of modelling/design examples: recursive definitions and arithmetic. The ACL2 language, on the other hand, has excellent support for recursive definitions (in fact, most interesting properties are expressed using recursive definitions\) and arithmetic [19]. In view of this (and our limited Alloy expertise), we avoid doing a comparison on problems that we perform well (*e.g.*, the property involving `hash` function in Section III is inexpressible in Alloy due to absence of multiplication), and restrict ourselves to examples (from the Alloy distribution) that we think Alloy performs well on.

	Alloy Analyzer			**Our method**	
Property	Scope	Time	Result	Time	Result
delUndoesAdd	31	80.91	–	0.07	QED
addIdempotent	31	112.66	–	0.19	QED
addLocal	3	0.05	CE	12.63	CE
lookupYields	3	0.05	CE	0.83	CE
writeRead	44	179.89	–	0.02	QED
writeIdempotent	29	98.03	–	0.01	QED
hidePreservesInv	87	86.03	–	0.26	QED
cutPaste	3	0.19	CE	0.49	CE
pasteAffectsHidden	29	138.34	–	0.42	QED
markSweepSound	8	29.03	–	0.28	QED
markSweepComplete	7	46.51	–	0.34	QED

TABLE I
COMPARISON WITH ALLOY ANALYZER (AA)

We analyzed 11 properties from 4 Alloy problems (specifications\), except the markSweep problem, all the others are from the Alloy book [13] and can alternatively be downloaded from the Alloy distribution.[4] Table 1 shows results, comparing the performance of our method implemented in ACL2s, with the performance of the Alloy Analyzer (AA). The time (in seconds) is measured on an Intel Core i3, 2.8GHz, 4GB memory machine. The Alloy analysis time is the total of the time spent on generating CNF and solving it using the SAT4J solver. The time taken by our method is what the ACL2 macro `time$` reports and includes the time taken by the ACL2 theorem prover. The Scope column for AA either denotes the minimum scope that finds a counterexample, or the maximum scope for which AA can check the property before reaching the timeout fixed at 180 seconds. The Result column shows either 'CE','QED' or '–', that stand for Counterexample found, Proof found, Neither Counterexample nor Proof found, respectively.

The first 4 properties are from the model of an email client's address book supporting aliases and groups, the *writeRead* and *writeIdempotent* properties are from the abstract memory problem, the next 3 properties are from an Alloy model describing the design of a media file management software. The last 2 rows are the Soundness and Completeness properties of the mark-and-sweep model, where live (reachable from root) nodes of the heap are *marked* and garbage (unreachable from root) nodes are *sweeped* into a freelist. The mark-and-sweep Alloy model was taken from an experiment in [12] where Alloy specifications are automatically translated to SMT2 language supported by the Z3 SMT solver [9].

We took the above examples and modelled them in the ACL2 language; mimicking the original formulation in Alloy as much as possible. In particular we used *set* types and *map* types *i.e.*, binary relations, which are part of the rich datatype support provided by ACL2s [10]. These respectively make use of the ordered sets library [8] and the records library [16]) in the ACL2 standard library distribution. These libraries provide a generic collection of reasoning rules (used in rewriting) about sets and records. In fact they are powerful enough to prove all the properties that Alloy exhaustively checked within the scope. No intermediate lemmas were provided, no hint or guidance was offered to the theorem prover, the proof of *pasteAffectsHidden* by ACL2s was as unassisted as the counterexample generated by Alloy for *cutPaste*. The counterexamples generated by our method, in few cases, required a change in the ACL2s settings when random testing (default) was not good enough to catch the counterexample, we had to revert to bounded exhaustive testing, which is also as automatic as Alloy, but not as efficient, as we observe in Table 1 in the entry of *addLocal*.

In experiments shown in [12], it is found that the correctness of the translated (from Alloy into Z3) mark-and-sweep model could not be proven by Z3; the authors mention that this problem is particularly difficult due to the fact that the simulation of recursion involved in mark-and-sweep by transitive closure results in deeply-nested quantifiers that Z3 cannot handle. We modelled the problem in ACL2, used sets and maps as mentioned before, the mark procedure (involving transitive closure) is modelled using a simple recursive definition. We then formalize the following properties that imply correctness:
Soundness: *No live node appears in the freelist*
Completeness: *All garbage nodes are eventually collected*
We were able to prove the above properties automatically. Again, no domain-specific lemmas were used, no hints were given to the theorem prover, no expert knowledge of theorem prover was required. This might seem surprising, and we

[4] Alloy Analyzer 4: http://alloy.mit.edu/alloy4

must deflate some optimism here, by pointing out that this automation will not scale for non-trivial models, but surely we must not overlook the effectiveness of powerful libraries (*e.g.*, set reasoning) by the tool-writer put to use by the choice of right abstractions (*e.g.*, using set datatypes) by the designer.

V. Related Work

Counterexample Generation in Interactive Theorem Provers

Random Testing is a well-studied, scalable, lightweight technique for finding counterexamples to executable formulas. Many Interactive Theorem Provers motivated by the success of QuickCheck and related random testing tools [6] have implemented random testing libraries *e.g.*, Isabelle/HOL [1], Agda [11] and PVS [18]. The other standard technique for generating counterexamples for a conjecture is to use a SAT or SMT solver. This requires translating from a rich, expressive logic to a restricted logic with limited expressiveness. The major constraint on such approaches is that a counterexample to the translated formula should also be a counterexample to the original formula. However, the absence of a counterexample does not imply that the conjecture is true. Some tools making use of the above technique are Pythia [20], SAT Checking [21], Refute [22] and Nitpick [2]. The work mentioned above has the same goal as our work: automatically exhibit counterexamples to false properties. However, unlike our work, none of the above mentioned approaches *use* the interactive theorem prover to generate counterexamples for arbitrary properties.

Combining Testing and Interactive Theorem Proving

Ideas for combining formal specifications and testing date back to at least 1981 [4]. One of the first examples of combining testing and interactive theorem proving was carried using Agda [11]. Random testing was used to check for counterexamples, and the point was made that the user could apply random testing also to subgoals. Another instance of leveraging a theorem prover to improve testing is the HOL-Testgen tool [3] which was designed for specification-based testcase generation. Compared to the above approaches, our method has a more fine-grained and tighter integration with the interactive theorem prover.

A. Automatic Analysis tools

Alloy is a declarative specification language based on relations and sets. The Alloy Analyzer can automatically find small counterexamples to Alloy specifications. This is done by translating the Alloy specification into a boolean satisfiability formula and using an off-shelf SAT Solver to find a solution. Model checking [7].

VI. Conclusions

We presented an algorithm that uses an interactive theorem prover to automatically analyze models and specifications. Our approach has several advantages over related work. It allows designers to use expressive languages to model systems at various levels of abstraction, with support for data structures, arithmetic, and recursive procedures. It is fully automated and compares favorably to existing methods for analyzing high-level models. Our algorithm is implemented and freely availabe in ACL2s, the ACL2 Sedan.

VII. Acknowledgments

We would like to thank Peter Dillinger, Mitesh Jain, and Matt Kaufmann.

References

[1] S. Berghofer and T. Nipkow. Random testing in Isabelle/HOL. In *SEFM*, pages 230–239. IEEE Computer Society, 2004.

[2] J. C. Blanchette and T. Nipkow. Nitpick: A counterexample generator for higher-order logic based on a relational model finder. In M. Kaufmann and L. C. Paulson, editors, *ITP*, volume 6172 of *Lecture Notes in Computer Science*, pages 131–146. Springer, 2010.

[3] A. D. Brucker and B. Wolff. Symbolic test case generation for primitive recursive functions. In J. Grabowski and B. Nielsen, editors, *FATES*, volume 3395 of *LNCS*, pages 16–32. Springer, 2004.

[4] R. Cartwright. Formal program testing. In *Proceedings of the 8th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '81, pages 125–132, New York, NY, USA, 1981. ACM.

[5] H. R. Chamarthi, P. Dillinger, M. Kaufmann, and P. Manolios. Integrating testing and interactive theorem proving. See URL `http://www.ccs.neu.edu/home/harshrc/ITaITP.pdf`.

[6] K. Claessen and J. Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. In *ICFP*, pages 268–279, 2000.

[7] E. Clarke. Model checking. In *Foundations of Software Technology and Theoretical Computer Science*, pages 54–56. Springer, 1997.

[8] J. Davis. Finite set theory based on fully ordered lists. In *Fifth International Workshop on the ACL2 Theorem Prover and Its Applications (ACL2 '04)*, November 2004.

[9] L. M. de Moura and N. Bjørner. Z3: An efficient smt solver. In C. R. Ramakrishnan and J. Rehof, editors, *TACAS*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.

[10] P. Dillinger and P. Manolios. ACL2 Sedan homepage. See URL `http://www.acl2s.ccs.neu.edu/doc`.

[11] P. Dybjer, Q. Haiyan, and M. Takeyama. Combining testing and proving in dependent type theory. In D. A. Basin and B. Wolff, editors, *TPHOLs*, volume 2758 of *LNCS*, pages 188–203. Springer, 2003.

[12] A. A. E. Ghazi and M. Taghdiri. Relational reasoning via smt solving. In *FM*, 2011.

[13] D. Jackson. *Software Abstractions - Logic, Language, and Analysis*. MIT Press, 2006.

[14] D. Jackson, I. Schechter, and I. Shlyakhter. Alcoa: the alloy constraint analyzer. In *ICSE*, pages 730–733, 2000.

[15] M. Kaufmann and J. S. Moore. ACL2 homepage. See URL `http://-www.cs.utexas.edu/users/moore/acl2`.

[16] M. Kaufmann and R. Sumners. Efficient rewriting of operations on finite structures in ACL2. In *Third International Workshop on the ACL2 Theorem Prover and its Applications (ACL2 '02)*, April 2002.

[17] P. Manolios. *Mechanical Verification of Reactive Systems*. PhD thesis, University of Texas at Austin, August 2001.

[18] S. Owre. Random testing in PVS. In *Workshop on Automated Formal Methods(AFM)*, volume 10, Seattle, WA, USA, 2006.

[19] D. Russinoff. A mechanically checked proof of ieee compliance of the floating point multiplication, division and square root algorithms of the amd-k7¡ sup¿TM¡/sup¿ processor. *LMS Journal of Computation and Mathematics*, 1\(-1\):148–200, 1998.

[20] A. Spiridonov and S. Khurshid. Automatic generation of counterexamples for ACL2 using Alloy. In *Seventh International Workshop on the ACL2 Theorem prover and its Applications (ACL2 '07)*, 2007.

[21] R. Sumners. Checking ACL2 theorems via SAT checking. In *Third International Workshop on the ACL2 Theorem Prover and its Applications (ACL2 '02)*, April 2002.

[22] T. Weber. Sat-based finite model generation for higher-order logic. Ph.D. thesis, Dept. of Informatics, T.U.München, 2008.

Proving and Explaining the Unfeasibility of Message Sequence Charts for Hybrid Systems

Alessandro Cimatti Sergio Mover Stefano Tonetta
FBK-irst, I38050, Trento, Italy
{cimatti,mover,tonettas}@fbk.eu

***Abstract*—Networks of Hybrid Automata are a clean modelling framework for complex systems with discrete and continuous dynamics. Message Sequence Charts (MSCs) are a consolidated language to describe desired behaviors of a network of interacting components. Techniques to analyze the feasibility of an MSC over a given HA network are based on specialized bounded model checking techniques, and focus on efficiently constructing traces of the network that witness the MSC behavior. Unfortunately, these techniques are unable to deal with the "unfeasibility" of the MSC, i.e. that no trace of the network satisfies the MSC.**

In this paper, we tackle the problem of MSC unfeasibility: first, we propose specialized techniques to *prove* that an MSC can not be satisfied by any trace of a given HA network; second, we show how to *explain* why an MSC is unfeasible.

The approach is cast in an SMT-based verification framework, using a local time semantics, where the timescales of the automata in the network are synchronized upon shared events. In order to prove unfeasibility, we generalize k-induction to deal with the structure of the MSC, so that the simple path condition is localized to each fragment of the MSC. The explanations are provided as formulas in the variables representing the time points of the events of the MSCs, and are generated using unsatisfiable core extraction and interpolation. An experimental evaluation demonstrates the effectiveness of the approach in proving unfeasibility, and the adequacy of the automatically generated explanations.

I. INTRODUCTION

Complex embedded systems (e.g. control systems for railways, avionics, and space) are made of several interacting components, and feature both discrete and continuous variables. Networks of communicating hybrid automata [18] (HAs) are increasingly used as a formal framework to model and analyze the behavior of such systems: local activities of each component amount to transitions local to each HA; communications and other events that are shared between/visible for various components are modelled as synchronizing transitions of the automata in the network; time elapse is modelled as implicit shared timed transitions.

A fundamental step in the design of these networks is the validation of the models performed by checking if they accept some desired interactions among the components. The language of Message Sequence Charts (MSCs) and its extensions are often used to express scenarios of such interactions. MSCs are especially useful for the end users because of their clarity and graphical content.

The ability to check whether a network of HAs may exhibit behaviors that satisfy a given MSC is an important feature to support user validation. Efficient techniques to analyze the feasibility of an MSC over a given HA network are based on specialized bounded model checking techniques, and focus on efficiently constructing traces of the network that witness the MSC behavior. Unfortunately, these techniques are unable to deal with the unfeasibility of the MSC, i.e. the case where no trace of the network satisfies the MSC.

In this paper, we tackle the problem of MSC unfeasibility, along two main directions: first, we propose specialized techniques to prove that an MSC cannot be satisfied by any trace of a given HA network; second, we show how to explain why an MSC is unfeasible.

In order to *prove unfeasibility*, we propose a specialized algorithm, which generalizes k-induction to deal directly with the structure of the MSC. The search is structured around the events in the MSC, which are used as intermediate "islands". In addition to pre-simplifying the encoding of the fragments of the MSC between events, we apply the simple path condition to each fragment, so that the encoding length of each fragment is no longer increased as soon as we detect that no new states can be reached. The MSC is deemed unfeasible for the network when no fragment can be further extended.

In order to *explain why* an MSC is unfeasible, our approach can generate various information. One is a subset of the MSC that is itself unfeasible for the network, which helps to focus on a subset of the messages, and on the HAs in the network that are involved. Another one is a set of timing conditions over the events in the MSC, which are themselves sufficient to conclude unfeasibility. The explanations are provided as formulas in linear arithmetic, constraining the assignments to the variables representing (some of) the time points of the events of the MSCs. To the best of our knowledge, this is the first work explaining MSC unfeasibility. We remark that here we are trying to provide diagnostic information in case of a *false existential property*, and thus the traditional diagnostics used in model checking for universal properties (e.g. simulation traces) provides no help.

The technical underpinning of this work is the "local time" semantics [6] for HAs, which exploits the fact that automata can be "shallowly synchronized". The intuition is that each automaton can proceed based on its individual "local time scale", unless they perform a synchronizing transition, in which case they must realign their absolute time. The framework allows to reason locally about the simple path conditions for each process, and also to extract more structured explanations, possibly not involving all the processes in the network and

the MSC events.

We implemented the approach and carried out an extensive evaluation, over a wide set of networks and benchmark MSCs. The new approach is able to effectively refute MSCs, significantly outperforming the corresponding approaches based on automata construction, and to provide interesting explanations.

The paper is structured as follows. In Section II, we present some background on networks of HAs, on SMT, and on k-induction. In Section III-A, we describe the language we use to describe the scenarios and the SMT encoding based on their structure. In Section IV, we discuss MSC-direct induction. In Section V, we discuss method to find explanations of unfeasibility. In Section VI, we discuss related work. In Section VII, we experimentally evaluate our approach. In Section VIII, we draw some conclusions.

II. Background

A. Networks of hybrid automata

A *Labelled Transition System* (LTS) is a tuple $\langle Q, A, Q_0, R\rangle$ where Q is the set of states, A is the set of actions/events (also called alphabet), $Q_0 \subseteq Q$ is the set of initial states, $R \subseteq Q \times A \times Q$ is the set of labeled transitions.

A *trace* is a sequence of events $w = a_1, \dots, a_k \in A^*$. Given $A' \subseteq A$, the projection $w_{|A'}$ of w on A' is the subtrace of w obtained by removing all events in w that are not in A'. A *path* π of S over the trace $w = a_1, \dots, a_k \in A^*$ is a sequence $q_0 \overset{a_1}{\to} q_1 \overset{a_2}{\to} \dots \overset{a_k}{\to} q_k$ such that $q_0 \in Q_0$ and, $\langle q_{i-1}, a_i, q_i\rangle \in R$ for all i such that $1 \le i \le k$. We say that π accepts w.

The *parallel composition* $S_1||S_2$ of two LTSs $S_1 = \langle Q_1, A_1, Q_{01}, R_1\rangle$ and $S_2 = \langle Q_2, A_2, Q_{02}, R_2\rangle$ is the LTS $\langle Q_1 \times Q_2, A_1 \cup A_2, Q_{01} \times Q_{02}, R\rangle$ where:
$R := \{\langle\langle q_1, q_2\rangle, a, \langle q_1', q_2'\rangle\rangle | \langle q_1, a, q_1'\rangle \in R_1, \langle q_2, a, q_2'\rangle \in R_2\}$
$\cup\{\langle\langle q_1, q_2\rangle, a, \langle q_1', q_2\rangle\rangle \mid \langle q_1, a, q_1'\rangle \in R_1, a \notin A_2\}$
$\cup\{\langle\langle q_1, q_2\rangle, a, \langle q_1, q_2'\rangle\rangle \mid \langle q_2, a, q_2'\rangle \in R_2, a \notin A_1\}$.

The parallel composition of two or more LTSs $S_1|| \dots ||S_n$ is also called a *network*. If an event is shared by two or more components, we say that the event is a synchronization event; otherwise, we say that the event is local. We denote with τ_i the set of local events of the i-th component.

Given a network $\mathcal{N}$ and a state $q \in Q_1 \times \dots \times Q_n$, the *reachability problem* is the problem of checking if there is a path $q_0 \overset{a_1}{\to} q_1 \overset{a_2}{\to} \dots \overset{a_k}{\to} q_k$ of S with $q_k = q$.

Hybrid Automata (HAs) [18] enrich the discrete states and transitions of LTSs with continuous variables and further conditions that constrain how these variables continuously evolve within a discrete state. In particular, a HA is a tuple $\langle Q, A, Q_0, R, X, \mu, \iota, \xi, \theta\rangle$ where:

- Q is the set of states,
- A is the set of events,
- $Q_0 \subseteq Q$ is the set of initial states,
- $R \subseteq Q \times A \times Q$ is the set of discrete transitions,
- X is the set of continuous variables,
- $\mu : Q \to P(X, \dot{X})$ is the flow condition,
- $\iota : Q \to P(X)$ is the initial condition,
- $\xi : Q \to P(X)$ is the invariant condition,
- $\theta : R \to P(X, X')$ is the jump condition,

where X' represent the value of variables X after a discrete transition, $\dot{X}$ represent the derivative of variables X during a continuous evolution, and P represents the set of predicates over the specified variables.

A Linear HA (LHA) is an HA where all the conditions are Boolean combinations of linear inequalities and the flow conditions contain variables in $\dot{X}$ only. We assume also that the invariant conditions of a LHA are conjunctions of inequalities.

A *network* $\mathcal{H}$ of HAs is the parallel composition of two or more HAs. We consider the local-time semantics, which is equivalent to the standard global-time semantics of [18], but instead of synchronizing the components on a shared timed event, it enriches all shared events with time-stamps, introduces local timed events, and synchronizes the components on shared events forcing the time-stamps to be equal [6], [10].

In the following, we consider a network $\mathcal{H} = H_1|| \dots ||H_n$ of HAs with $H_i = \langle Q_i, A_i, Q_{0i}, R_i, X_i, \mu_i, \iota_i, \xi_i, \theta_i\rangle$ such that, for all $1 \le i < j \le n$, $X_i \cap X_j = \emptyset$ (i.e. the set of continuous variables of the hybrid automata are disjoint).

The *local-time semantics* (or time-stamps semantics) of $\mathcal{H}$ is the network of LTSs $\mathcal{N}_{\text{LocTime}}(\mathcal{H}) = S_1|| \dots ||S_n$ with $S_i = \langle Q_i', A_i', Q_{0i}', R_i'\rangle$ where:

- $Q_i' = \{\langle q, \overline{x}, t\rangle \mid q \in Q_i, \overline{x} \in \mathbb{R}^{|X_i|}, t \in \mathbb{R}_{\ge 0}\}$,
- $A_i' = \{\langle a, t\rangle \mid a \in A_i, t \in \mathbb{R}_{\ge 0}\} \cup \{\text{Time}_i\}$,
- $Q_{0i}' = \{\langle q, \overline{x}, 0\rangle \mid q \in Q_{0i}, \overline{x} \in \iota_i(q)\}$,
- $R_i' = \{\langle\langle q, \overline{x}, t\rangle, \langle a, t\rangle, \langle q', \overline{x}', t\rangle\rangle \mid \langle q, a, q'\rangle \in R_i, \langle \overline{x}, \overline{x}'\rangle \in \theta_i(q, a, q'), \overline{x} \in \xi_i(q), \overline{x}' \in \xi_i(q')\} \cup \{\langle\langle q, \overline{x}, t\rangle, \text{Time}_i, \langle q, \overline{x}', t'\rangle\rangle \mid$ there exists f satisfying $\mu_i(q)$ s.t. $f(t) = \overline{x}, f(t') = \overline{x}', f(\epsilon) \in \xi_i(q), \epsilon \in [t, t'], t \le t'\}$.

The definition of the local-time semantics is such that the set of actions of each LTS contains a local timed event Time_i and couples containing a discrete action and a time-stamp (i.e. the amount of time elapsed in the automaton). Thus, each automaton performs the time transition locally, changing its local time-stamp. When two automata synchronize on $\langle a, t\rangle$ they agree on the action a and on the time-stamp t. Instead, in the global-time semantics, all the automata are forced to synchronize on the time transition $\langle \text{Time}, \delta\rangle$, agreeing on the time elapsed during the transition (δ variable).

B. SMT encoding of hybrid automata

As described in [18], LHAs can be analyzed with symbolic techniques. Let us consider a network $\mathcal{H} = H_1|| \dots ||H_n$ of LHAs whose semantics is given by the network of LTSs $S_1|| \dots ||S_n$ where $S_i = \langle Q_i, A_i, Q_{i0}, R_i\rangle$. The states Q_i can be represented by a set V_i of symbolic variables. The events of A_i can be represented by a set of symbolic variables W_i. Sets of states are represented with formulas over V_i, while sets of transitions are represented with formulas over V_i, W_i, and V_i', which are the next values of V_i. In particular, it is possible to define a formula $I_i(V_i)$ that represents the initial states and a formula T_i that represents the transitions of H_i.

The details of the encoding we use can be found in [8]. Here, we just notice that we use a scalar input variable ε to represent the events of H_i adding two distinguished values, namely T and S, to represent a timed transition and stuttering, respectively. When stuttering, the system does not change any variable. Moreover, when using the local-time semantics, the variable t_i represents the local time of H_i and is also used as time-stamp of the events (thus, to ensure that shared events happen at the same time).

As standard in Bounded Model Checking, given an integer k, we can build a formula whose models correspond to all paths of length k of the represented LTS S. The formula introduces $k+1$ copies of every variable in the encoding of the automata. Given a formula ϕ, we denote with ϕ^i the result of substituting the current and next variables of ϕ with their i-th and $(i+1)$-th copy, respectively. The paths of S of length k can be encoded into the formula $\mathit{path}(k) := I^0 \wedge \bigwedge_{0 \le i < k} T^i$.

A typical optimization used in BMC for timed and hybrid systems is to force the alternation of timed and discrete transitions [1], [4].

Most of modern solvers, both for SAT and SMT, have an *incremental* interface such that, if a problem is fed to the solver incrementally, the solver can first tackle smaller parts of the problem and then pass to large parts managing to reuse the lemmas discovered during the previous searches.

C. K-induction

K-induction [32] is a technique that proves that if a set of states is not reachable in k steps, then it is not reachable at all. On the lines of the induction principle, it consists of a base step, which solves the bounded reachability problem with a given bound k of steps, and an inductive step, which concludes that k is sufficient to solve the (unbounded) reachability problem. The idea of the inductive step is to check either if the initial states cannot reach new (non-visited) states in $k+1$ steps, or if the target set of states cannot be reached in $k+1$ steps (hereafter, we will consider only the first condition). These checks can be solved by means of satisfiability.

The formula $\mathit{simple}(k) := \bigwedge_{0 \le i < j \le k} \neg \bigwedge_{v \in V} v^i = v^j$ can be used to strengthen the path encoding to represent only simple (loop-free) paths. If the formula $\mathit{kind}(k) := I(V^0) \wedge \pi(k+1) \wedge \mathit{simple}(k+1)$ is unsatisfiable, then there is no initial simple path with more than k states. Thus, if, for all $i \le k$, $\mathit{path}(k) \wedge \mathit{target}^k$ is unsatisfiable and $\mathit{kind}(k)$ is unsatisfiable as well, then *target* is not reachable.

If the target is not reachable in a finite-state LTS, there is a k for which the above conditions are unsatisfiable. In hybrid systems, it is very common that the LTSs contain infinite paths, typically with monotonically increasing variables (such as the local time) and, therefore, it is difficult to apply k-induction.

In [33], k-induction has been integrated with predicate abstraction [16] to deal with infinite-state systems. Typically, an abstraction defines an equivalence relation EQ_α among the the concrete states that are not distinguished by the abstraction. As for predicate abstraction, given a certain set $\mathbb{P}$ of predicates over the variables V, the equivalence relation is defined as $EQ_{\mathbb{P}}(V, \overline{V}) := \bigwedge_{P \in \mathbb{P}} P(V) \leftrightarrow P(\overline{V})$.

Abstract k-induction embeds the definition of the predicate abstraction in the encoding of the path. In particular, the formula $\mathit{path}_\alpha(k) := \bigwedge_{1 \le h < k} (T(V^{h-1}, \overline{V^h}) \wedge EQ_\alpha(\overline{V^h}, V^h)) \wedge T(V^{k-1}, V^k)$ is satisfiable iff there exist a path of k steps in the abstract state space. The formula $\mathit{simple}_\alpha(k)$ is defined as $\mathit{simple}_\alpha(k) := \bigwedge_{0 \le i < j \le k} \neg EQ_\alpha(V^i, V^j)$. The formula $\mathit{path}_\alpha(k) \wedge \mathit{simple}_\alpha(k)$ is satisfiable iff there exists a simple path of length k in the abstract state space. Finally, the formula kind_α, defined as $\mathit{kind}_\alpha(k) := I(\overline{V^0}) \wedge EQ_\alpha(\overline{V^0}, V^0) \wedge \mathit{path}_\alpha(k) \wedge \mathit{simple}_\alpha(k)$, is satisfiable iff there exists an initial simple path of length k.

Similarly to the concrete case, if, for all $i \le k$, $\mathit{path}_\alpha(k) \wedge EQ_\alpha(V^k, \overline{V^k}) \wedge \mathit{target}^k$ is unsatisfiable and $\mathit{kind}_\alpha(k)$ is unsatisfiable as well, then *target* is not reachable in the abstraction (and therefore also in the concrete state space).

III. MSC Feasibility

A. Constrained Message Sequence Charts

A Message Sequence Chart (MSC) [20] defines a single (partial-order) interaction of the components of a network $\mathcal{N}$. MSCs have been extended in several ways. We consider here a particular variant, enriched with additional constraints, which turns out to be very useful and easy to handle with the SMT-based approach.

An MSC m is associated with a set of events $A_m \subseteq A_{\mathcal{N}}$, subset of the events of the network. We assume that A_m contains all and only the shared events of the network ($A_m = \bigcup_{1 \le i < j \le n} A_i \cap A_j$). In particular, in the case of hybrid automata the timed events are not part of A_m.

The MSC defines a sequence of events for every component S of the network, called instance of S. An instance σ for the LTS S is a sequence $a_1; \ldots; a_l \in (A_m \cap A_S)^*$ of events of S. S accepts the instance ($S \models \sigma$) iff there exists a trace w accepted by S such that the sub-sequence of events in A_m is equal to σ ($w_{|A_m} = \sigma$). In other words, S accepts the instance iff there exists a path π of S over a trace compatible with the instance σ. In such cases, we say that $\pi \models \sigma$.

We denote the j-th event a_j of the instance σ_i with $\sigma_i[j]$, the number l of events in σ_i with $|\sigma_i|$, the local segment between the event $\sigma_i[j]$ and $\sigma_i[j+1]$ of σ_i with $\mathit{lsg}(\sigma_i[j])$, where the first local segment before a_1 is $\mathit{lsg}(\sigma_i[0])$ and the final local segment after $a_{|\sigma_i|}$ is $\mathit{lsg}(\sigma_i[|\sigma_i|])$.

If $\pi \models \sigma$, π must be in the form $q_0 \xrightarrow{\tau} \ldots \xrightarrow{\tau} q_{h_1} \xrightarrow{\sigma[1]} q_{h_1+1} \xrightarrow{\tau} \ldots \xrightarrow{\tau} q_{h_{|\sigma|}} \xrightarrow{\sigma[|\sigma|]} q_{h_{|\sigma|+1}} \xrightarrow{\tau} \ldots \xrightarrow{\tau} q_{h_{|\sigma+1|}}$, where $q_h \in Q$ and τ are local events of S. We denote the subsequences of the path π in which it is split by σ as follows:

- $pre_j(\pi) = q_{h_j}$, it is the source state of the transition labeled with $\sigma[j]$ in π.
- $post_j(\pi) = q_{h_j+1}$, it is the destination state of the transition labeled with $\sigma[j]$ in π.
- $loc_j(\pi) = q_{h_j+1}; \ldots; q_{h_{j+1}}$, it is the sequence of states between the j-th and the $j+1$-th shared events, where we denoted 0 with h_0.

An MSC is the parallel composition $\sigma_1||\dots||\sigma_n$ where σ_i is an instance of S_i. The network $\mathcal{N}$ of LTSs accepts the MSC m ($\mathcal{N} \models m$) iff there exists a trace w accepted by $\mathcal{N}$ such that, for every S_i, the sub-sequence of events in $A_m \cap A_{S_i}$ is equal to σ_i ($w_{|(A_m\cap A_{S_i})} = \sigma_i$). In other words, $\mathcal{N}$ accepts the instance iff there exists a path of $\mathcal{N}$ over a trace compatible with every instance of the MSC. If $\mathcal{H}$ is a network of HAs, then we say that $\mathcal{H} \models m$ iff $\mathcal{N}_{\text{LocTime}}(\mathcal{H}) \models m$.

We define a Constrained MSC (CMSC) as a pair $\langle m, \phi\rangle$ where m is an MSC $\sigma_1||\dots||\sigma_n$ and ϕ is a formula over the variables $v_i[j]$ with $1 \le i \le n$ and $1 \le j \le |\sigma_i|$, where $v_i[j]$ represents the value of the variable v of the i-th component at the time of the j-th event $\sigma_i[j]$ of σ_i. $\mathcal{N} \models \langle m, \phi\rangle$ iff there exists a path $\pi = \pi_1||\dots||\pi_n$ such that $\pi_i \models \sigma_i$ and the assignments of $pre_j(\pi_i)$ to $v_i[j]$ satisfy ϕ.

The model checking problem for a CMSC $\langle m, \phi\rangle$ is the problem of checking if a network satisfies a CMSC. The classical approach is based on the construction of a monitor (or a network of monitors) that, composed with $\mathcal{N}$, forces $\mathcal{N}$ to follow only paths that satisfy the MSC.

An MSC $\sigma_1||\dots||\sigma_n$ is consistent iff for every pair of instances σ_i and σ_j the projection on the common alphabet is the same, i.e., if $A = A_i \cap A_j$, $\sigma_{i|A} = \sigma_{j|A}$. Henceforth, we assume that the MSCs are consistent. The check of consistency is trivial and can be done syntactically with a simple traversal of the MSC's structure.

B. Scenario-driven encoding

The drawbacks of the traditional SMT-based encoding is that it cannot exploit the sequence of messages prescribed by the MSC in order to simplify the search because of the uncertainty on the number of local steps between two events. We encode the path of each automaton independently, exploiting the local time semantics, and then we add constraints that force shared events to happen at the same time, as in *shallow synchronization* [8]. Moreover, we fix the steps corresponding to the shared events and we parametrize the encoding of the local steps with a maximum number of transitions.

We extend the encoding presented in [10] with different numbers of steps for different local segments of the MSC.

Let us consider a network $\mathcal{H} = H_1||\dots||H_n$ of LHAs and the encoding $\langle V_i, W_i, I_i, T_i\rangle$ representing the LHA H_i, for $1 \le i \le n$, in the local-time semantics. We denote with $T_{i|\phi}$ the transition condition restricted to the condition ϕ, i.e., $T_{i|\phi} = T_i \wedge \phi$. We abbreviate $T_{i|\varepsilon=a}$ with $T_{i|a}$ and $T_{i|\varepsilon\in\tau_i\cup\{\text{s}\}}$ with $T_{i|\tau}$ (notice that τ_i, the set of local actions, contains also the timed event T).

We associate a bound $k_i[j]$ to the j-th segment $lsg(\sigma_i[j])$ of the i-th instance. $k_i[j]$ is used to limit the number of transitions in the local path $loc_j(\pi)$ of a path π satisfying the instance σ_i. We use $\overline{k}_i$ to denote $\langle k_i[0], \dots, k_i[h_i]\rangle$ and $\overline{k}$ to denote $\langle \overline{k}_1, \dots, \overline{k}_n\rangle$.

Note that the event $\sigma_i[j]$ is preceded by $\sum_{v=0}^{j-1} k_i[v] + j - 1$ transitions consisting of local transitions ($\sum_{v=0}^{j-1} k_i[v]$) and shared events $(j-1)$. $idx_i[j]$ defines the index used to encode the event $\sigma_i[j]$ as $idx_i[j] := \sum_{v=0}^{j-1} k_i[v] + j - 1$.

The following encoding represents all paths of the network compatible with the MSC where the local transitions of the j-th segment of the i-th instance have been unrolled up to $k_i[j]$ times (note that the "up to" is due to the ability of stuttering):

$$
\begin{aligned}
enc(m, \overline{k}) \;&:=\; \bigwedge_{1\le i\le n} enc(\sigma_i, \overline{k}_i) \wedge \\
&\bigwedge_{1\le j<i\le n} sync(\sigma_j, \sigma_i) \wedge (t_j^{\sum_{v=0}^{|\sigma_j|} k_j[v]} = t_i^{\sum_{v=0}^{|\sigma_i|} k_i[v]}) \\
enc(\sigma_i, \overline{k}_i)) \;&:=\; I_i^0 \wedge \bigwedge_{1\le h\le k_i^0} T_{i|\tau}^{h-1} \wedge \\
&\bigwedge_{1\le j\le|\sigma_i|} (T_{i|a_j}^{idx_i[j]} \wedge \bigwedge_{1\le h\le k_i[j]} T_{i|\tau}^{idx_i[j]+h}) \\
sync(\sigma_j, \sigma_i) \;&:=\; \bigwedge_{1\le z\le|\sigma_{j|A}|=|\sigma_{i|A}|} t_i^{idx_i[f_i^{ij}(z)]} = t_j^{idx_j[f_j^{ij}(z)]}
\end{aligned}
$$

where $A = A_i \cap A_j$ and the function f_i^{ij} maps the z-th event a_z shared between σ_i and σ_j to the index of a_z in σ_i. More, specifically, if $\sigma_{j|A} = \sigma_{i|A} = a_1; \dots a_l$, then $f_i^{ij}, f_j^{ij} : \mathbb{N} \to \mathbb{N}$ are such that $a_z = \sigma_i(f_i^{ij}(z)) = \sigma_j(f_j^{ij}(z))$, for $1 \le z \le l$.

Intuitively, $enc(m, \overline{k})$ encodes the unrolling of each component according to its instance and guarantees that the different unrollings have the same time for every occurrence of a shared event and the same final time.

In order to encode the paths that satisfy a CMSC we have just to conjoin the additional constraints:

$$enc(\langle m, \phi\rangle, \overline{k}) \;:=\; enc(m, \overline{k}) \wedge \phi[v_i^{idx_i[j]}/v_i[j]]$$

where for all the instances i, $1 \le i \le n$, and all events j, $1 \le j \le |\sigma_i|$, we substitute $v_i[j]$ in ϕ with the timed variable $v_i^{idx_i[j]}$.

Theorem 1: If $enc(\langle m, \phi\rangle, \overline{k})$ is satisfiable then $\mathcal{H} \models \langle m, \phi\rangle$. Vice versa, if $\mathcal{H} \models \langle m, \phi\rangle$, then there exists integers $\overline{k}$ such that $enc(\langle m, \phi\rangle, \overline{k})$ is satisfiable.

IV. Scenario-driven induction

In this section, we describe how the structure of the MSC can be exploited to tailor k-induction to prove the unfeasibility of the scenario. For the base case, we use the encoding of [10]. For the inductive step, we apply the simple path condition to each segment of the scenario and prove that such partitioned simple-path condition is equivalent to the path condition applied to composition of the network and the scenario monitor. The use of different local bounds as presented in Section III-B allows k-induction to stop the unrolling of the local path at different depths according to the local structure of the component at the considered segment.

A. Partitioned simple-path condition

Our goal is to find an inductive condition $kind(\langle m, \phi\rangle, \overline{k})$ such that, in the finite-state case, $\mathcal{N} \not\models \langle m, \phi\rangle$ if and only if there exist $\overline{k}$ such that $enc(\langle m, \phi\rangle, \overline{k})$ and $kind(\langle m, \phi\rangle, \overline{k})$ are unsatisfiable. In the hybrid case, we would like that the "if" condition still holds, while the "only if" condition

should hold when the corresponding inductive condition for the composition of the network with the MSC monitor holds (relatively complete). The difficulties are that:

- the projection of a simple path on a component may be not a simple path;
- if a simple path is the concatenation or the parallel composition of two paths, these may be not the longest simple paths of their segments.

The CMSC $\langle m, \phi \rangle$ defines a partial order $<_m$ among the segments of m defined as the reflexive and transitive closure of the smallest relation such that:

- $lsg(\sigma_i[j]) <_m lsg(\sigma_i[j'])$ if $0 \leq j < j' \leq h_i$;
- $lsg(\sigma_i[j]) <_m lsg(\sigma_{i'}[j'])$ if there exists $lsg(\sigma_{i''}[j''])$ such that there is a synchronization between $\sigma_i[j]$ and $\sigma_{i''}[j'']$ and $lsg(\sigma_{i''}[j'']) <_m lsg(\sigma_{i'}[j'])$.

Given a CMSC $\langle m, \phi \rangle$ and the local path $lsg(\sigma_i[j])$ we define the partial CMSC $\langle \overline{m}_i[j], \overline{\phi}_i[j] \rangle$ where:

- $\overline{m}_i[j] = \overline{\sigma}_1 || \ldots || \overline{\sigma}_n$ such that for all $1 \leq v \leq n$, $|\overline{\sigma}_v| \leq |\sigma_v|$ and for all $1 \leq z \leq |\overline{\sigma}_v|$ $\overline{\sigma}_v[z] = \sigma_v[z]$ and $lsg(\overline{\sigma}_v[z]) <_m lsg(\sigma_i[j])$ or $lsg(\overline{\sigma}_v[z]) = lsg(\sigma_i[j])$, while for all $|\overline{\sigma}_v| < z \leq |\sigma_v|$ $lsg(\sigma_v[z]) \not<_m lsg(\sigma_i[j])$.
- $\overline{\phi}_i[j]$ contains only the constraints of ϕ which are over variables in $\overline{m}_i[j]$.

We define the local simple path condition as follows:

$$kind_i[j] := enc(\langle \overline{m}_i[j], \overline{\phi}_i[j] \rangle, \overline{k}) \wedge simple_i[j]$$
$$simple_i[j] := \bigwedge_{1 \leq h, z \leq k_i[j]} s_i^{idx_i[j]+h} \neq s_i^{idx_i[j]+z}$$

Theorem 2: If there exist $\overline{k}$ s.t. $enc(\langle m, \phi \rangle, \overline{k})$ is unsatisfiable and, for all i, j, $kind_i[j]$ is unsatisfiable, then $\mathcal{N} \not\models m$.

In order to check if k-induction holds incrementally, we visit the MSC m according to the partial order $<_m$. We incrementally apply the partitioned simple path condition to the local segments of m. The incremental checks exploit the standard Push/Pop/Assert incremental interface of the solver.

B. K-induction for hybrid systems

1) Alternation of timed and discrete transitions: The alternation of timed and discrete transitions has been proposed in different works to optimize the search of BMC for timed and hybrid systems [1], [4]. With k-induction, the alternation is fundamental to allow a concrete search to close. In fact, without forcing the alternation, the system will likely have infinite loop-free paths where timed transitions change some continuous variables infinitely often.

In order to enhance k-induction with alternation, the following points must be taken into account:

- since consecutive discrete transitions are possible, the timed transition must permit the elapsed time to be zero; therefore, the loop-free condition of k-induction must be relaxed in order to allow self loops with a timed transition with no elapsed time;
- the scenario-based encoding of the bounded model checking problem exploits stutter transitions in order to encode paths with up to k steps (instead of exactly k steps); the stuttering makes the alternation ineffective because it allows infinite loop-free paths alternating timed and stutter transitions; therefore, it is fundamental to avoid stuttering when considering the simple path condition.

2) Enabling a partitioned abstraction: The structure of local transitions between two shared events is often simple and without loops. In these cases, the alternation without stuttering allows k-induction to prove the unfeasibility of scenarios. If instead there are loops in the local structure, they may correspond to infinite loop-free paths. In order to prove the unfeasibility of scenarios also in these cases, we combine k-induction with predicate abstraction as in [33].

We can associate to different segments of the MSC different abstractions of the local transition relation. This way, we can obtain a fined-grained abstraction which abstract away the continuous components only where necessary.

V. UNFEASIBILITY EXPLANATION

We identify the following types of explanations to understand the reasons of the unfeasibility of the CMSC $\langle m, \phi \rangle$:

1) which parts of the CMSC cannot be executed by the network;
2) why the paths of the network consistent with m cannot satisfy ϕ;
3) why the paths of a component consistent with the corresponding instance of m are inconsistent with the rest of the CMSC.

We answer these questions by exploiting both unsat cores and interpolation. The unsatisfiable core for an unsatisfiable formula ϕ is a formula ψ iff ψ is unsatisfiable and $\phi = \psi \wedge \psi'$, for a (possibly empty) formula ψ'. Given two formulas A and B, with $A \wedge B \models \bot$, the Craig interpolant of $A \wedge B$ is a formula I such that $\models A \rightarrow I$, $B \wedge I \models \bot$, and which contains only variables common to A and B. Intuitively, the interpolant is an over-approximation of A "guided" by B.

In particular, after reaching the maximum bound in every local segment of the CMSC, we can build the proof of unsatisfiability of the BMC problem with such bounds. The unsat core extracted from the proof contains a subset of the unrolling of the components along the MSC and a (possibly empty) subset of the CMSC constraints which are incompatible. Since the local paths, events, and constraints are asserted in different conjuncts of the encoding, the unsat core is fine-grained enough to distinguish them.

By partitioning the encoding into the constraints obtained by unrolling the network (A) and the constraints of the CMSC (B), we can compute an interpolant of their unsatisfiability. This way, we obtain a formula over the variables at the time of the events implied by the network executing the MSC and inconsistent with the constraints of the CMSC. Note that if the interpolant is false, we can deduce that the constraints are not responsible of the unfeasibility and that the unrolling of the network is inconsistent by itself.

Finally, by partitioning the encoding into the unrolling of one component along its instance (A) and the rest (B, i.e.,

other components and constraints), the interpolation produces a formula over the variables at the time of the events implied by the component executing the instance and inconsistent with the other components or with the constraints of the CMSC. Note that if the interpolant is true, it means that the component does not play a role in the unfeasibility. On the contrary, if the interpolant is false, the component does not have a path compatible with the instance.

Note that, when the abstraction is used to prove the unfeasibility of the scenario, the explanations based on unsat core and interpolation are still valid.

VI. Related Work

MSCs [20] are a basic building block to describe the interactions among components. Several works, such as High-Level Message Sequence Charts [26] and Live Sequence Charts (LSC) [12], extend the language of the MSCs increasing their expressive power. We consider a basic version of MSCs which describes a single (partial-order) composition of sequences of events, augmented with additional constraints [2], [5]. We consider a trace-based semantics for the MSC, where the MSC predicates over the observable events of a system [23], [24]. While several works use MSCs to describe the entire system [3], [28], we instead use the MSC as a specification language.

A common approach to deal with the verification of MSC specifications consists in translating the scenario into automata or temporal logic formulas. LSCs are translated into timed automata in the Uppaal model checker [25], while in [22] the authors propose a translation from charts with timing constraints and synchronous events to Timed Büchi Automata. These works deal with expressive specification languages but they do not exploit the structure of the scenario. Moreover, in case of unfeasibility, these techniques do not provide explanations that narrow the events of the scenario or that gives meaningful information about a specific component.

The approach which translates the MSC into an automaton reduces the feasibility problem of the MSC to a reachability problem. Thus, the works on Bounded Model Checking (BMC) for hybrid systems [1], [4], [8], [14], [15], [34] can be used to solve the feasibility problem. The BMC encodings for hybrid automata can be further optimized exploiting the step semantics [17], [21], which allows independent transitions of different automata to be executed in parallel. However, BMC is unable to prove the unfeasibility of the MSC. When we encode the MSC into an automaton the unfeasibility problem can be solved using unbounded model checking techniques, such as k-induction [32]. K-induction is complete for finite state systems, but it was applied also to infinite state systems in [13], [29], [33]. In [13] the authors use k-induction to verify timed and hybrid automata and they generalize the simple path condition to simulation relations. K-induction is combined with predicate abstraction [16] in [33]. These works are not tailored to the problem of deciding the unfeasibility of a scenario and do not provide explanations in the case of unsatisfiability.

In [10] we propose a Bounded Model Checking encoding tailored to check the feasibility of a scenario in a network of hybrid automata. This approach turns out to be very efficient in dealing with complex scenarios, since it exploit the local-time semantics [6] in order to partition the encoding with respect to the MSC structure. However, the approach is unable to prove the unfeasibility of the scenario. We extend that work in order to prove the unfeasibility of a scenario and to provide meaningful explanations of unfeasibility.

Unsat cores and interpolation are often used to explain and generalize the source of unsatisfiability. Unsat cores are typically subsets of the conjuncts forming the unsatisfiable formula. However, other forms are possible, especially in the context of temporal unsatisfiability [31]. Interpolation for temporal properties is proposed in [30] as a theoretical framework for analyzing vacuity for discrete systems; the practical implications are not addressed in depth. In [31], it is suggested that k-induction can be used to find a k for which the BMC encoding of a temporal formula yields its unsatisfiability and that the unsat core contains the relevant parts of the formula that cause the unsatisfiability. However, mapping the BMC unsat core back to the original problem is not always easy. We achieve this by exploiting the scenario-based encoding that respects the structure of the scenario.

VII. Experimental evaluation

The techniques discussed in the previous sections were implemented in an extension of the NuSMV model checker [9], which is able to deal with networks of HAs, formalized in the HyDI language [11]. The NuSMV extension features an SMT-based approach to the verification of hybrid systems, and is tightly integrated with MathSAT [7], a state-of-the-art, full-fledged Satisfiability-Modulo-Theory solver (SMT). MathSAT provides the functionalities of incremental reasoning, unsatisfiable core extraction, and interpolation, which are used for bounded model checking, inductive reasoning, and explanation extraction.

In the experimental evaluation, we used the following benchmarks: the *Distributed Controller* [19], the *Audio Protocol* proposed in [19], the *Nuclear Reactor* [35], a hybrid version of the Fischer mutual exclusion protocol, and the *Electronic Height Control System (EHC)* described in [27]. All the test cases, the executable and the results of the evaluation are available at http://es.fbk.eu/people/mover/tests/FMCAD11/.

A. Scenario-driven Induction vs K-Induction

First, we compared the scenario-based induction with k-induction applied to the monolithic encoding of the network of HAs and the automata translated from the MSC.

The monolithic encoding is obtained composing the network with the automata obtained from the MSC. The construction of the monitor automata is described in details in [10]. In particular, we rely on the "distributed" monitor automata, where we build a monitor for each instance of the MSC, and the step semantics, which enables multiple transition to be executed in parallel. The combination of both approaches demonstrated to be the most efficient among the different automata construction and encoding presented in [10].

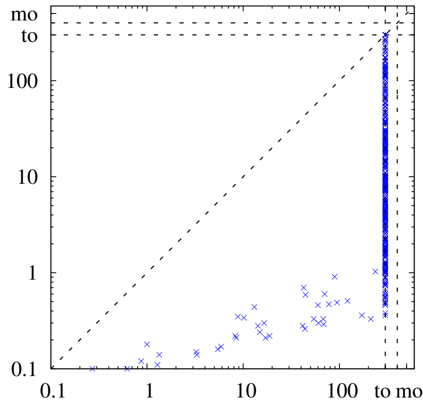


Fig. 1. Run times (sec.): monolithic induction (x axes) vs. scenario-induction (y axes)

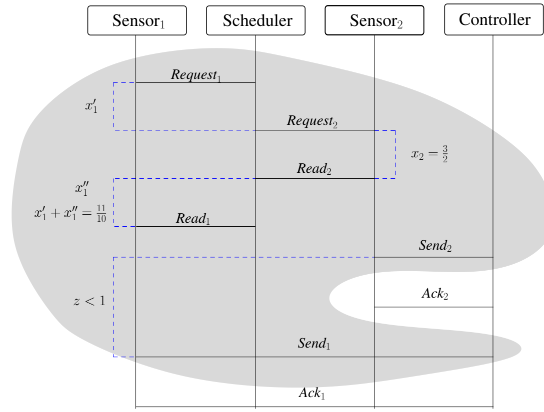


Fig. 2. The MSC for the distributed controller

In order to test the scalability of both approaches, we considered a set of unfeasible MSCs of different lengths, and parameterized the number of HAs in the network. We set a time out of 300 seconds and a memory out of 2 GB. The scatter plot in Figure 1 shows the execution time for both methods on all the instances. The Scenario-based induction is clearly superior to monolithic k-induction. This is due to the exploitation of the structure of the scenario: this results in localized simple path conditions, that are both simpler, and more effective, so that unsatisfiability is detected with a much shorter unrolling.

B. Unfeasibility Explanation

Then, we analyzed the unfeasibility explanations on the three benchmarks with non-trivial scenarios, showing their usefulness in identifying the causes of unfeasibility.

1) Distributed Controller [19]: the benchmark models the interactions of two sensors (sensor$_1$ and sensor$_2$) with a controller of a robot. The two sensors interact with a scheduler to access a shared processor. The time needed for computation by the two sensors is bounded but it is non-deterministic, and is tracked in the scheduler with two stopwatches (x_1 and x_2). Also the controller sets a time-out (variable $z = 0$) after the receipt of the first message. If the time-out expires ($z = 10$) the controller discards all the received data.

The MSC shown in Figure 2 models the interaction where sensor$_1$ requests the processor; the scheduler grants it for a total duration of x_2 time; sensor$_2$, which has a higher priority, requests and receives grant to the processor; when sensor$_2$ finishes its computation (event *read*$_2$), sensor$_1$ finishes to read data while, in parallel, sensor$_2$ sends its data to the controller; finally, the sensor$_1$ and the controller synchronize on *send*$_1$ and *ack*$_1$. The time spent to process the data of sensor$_1$ is given by the stopwatch x_1. In Figure 2 x_1 is the sum of the intervals x'_1 and x''_1. Moreover, we add two additional conditions on the duration of x_1 and x_2 in the scheduler ($x_2 = 1.5$ and $x_1 = 1.1$), and we fix the maximum time spent by the controller before receiving the data from sensor$_1$ ($z < 1$). The MSC augmented with these constraints is unfeasible.

We prove the unfeasibility of the scenario directly on the concrete system, since all the automata cannot loop performing only local transitions. The analysis takes 3 seconds and the longest simple path is 2 in the controller automaton, and 1 in the other automata. In the Figure 2 we outline in gray the elements of the scenario, events and constraints, which contribute to the unfeasibility. In particular, we find that the unfeasibility depends on all the events of the MSC apart from the events *Ack*$_1$ and *Ack*$_2$. Moreover, we discover that all the additional constraints of the scenario, $x_2 = 1.5$, $x_1 = 1.1$ and $z < 1$, contribute to the unfeasibility.

We exploit the interpolation techniques to get the constraints $z >= x_1$. In fact, z counts the time elapsed in the controller between the send$_1$ event and the send$_2$ event. This means that the controller cannot receive the send$_1$ message before x_1 seconds, which is the time spent to process data from sensor$_1$. If we fix $z >= 1.1$ then the scenario is feasible. We find a similar result if we look at the interpolant obtained partitioning the encoding in the constraints from sensor$_1$ (the A formula) and the rest of the network and the scenario (the B formula). We denote with $time^{event}_{\text{component}}$ the time variable of $component$ when performing $event$. The interpolant is $6 <= time^{request_1}_{\text{sensor}_1} - time^{read_1}_{\text{sensor}_1} + time^{send_1}_{\text{sensor}_1}$. Since $time^{request_1}_{\text{sensor}_1}$ is 6, from the initial condition and invariants of sensor$_1$, we can infer that the scenario and the other processes in the network do not allow $time^{read_1}_{\text{sensor}_1} <= time^{send_1}_{\text{sensor}_1}$, which is a necessary condition for sensor$_1$.

2) Audio Control Protocol [19]: this protocol transmits an arbitrary-length bit sequence from a sender to a receiver based on the timing-based Manchester encoding. The protocol relies on division of the elapsed time in slots. Every slot corresponds to a bit. The sender transmits a signal up in the slots corresponding to bits with value 1 (thus, a slot without signals correspond to bit 0). The protocol is robust to bounded errors in the timers used by the sender and receiver.

The considered scenarios consist of unfeasible timed sequences of up. For example, the sequence $\langle up, 4\rangle$, $\langle up, 8\rangle$,

Fig. 3. The automaton structure of the EHC controller with a line that traces the scenario sequence of events.

$\langle up, 12\rangle, \langle up, 16\rangle, \langle up, 19\rangle, \langle up, 23\rangle$ does not respect the protocol, since the 4-th and 5-th events must be separated by 3 seconds.

Scenario-based induction proves that the scenario is unfeasible in 41 seconds. The explanation extracted from the unsat core identifies the 4-th and 5-th events as the cause of unfeasibility. Interpolation "explains" that the inconsistency arises because the sender requires the 5-th event to happen after at least 3.8 seconds; it also shows that the receiver does not play any role in the inconsistency.

3) Electronic Height Control System [27]: this benchmark presents a case where the concrete k-induction is not able to prove the unfeasibility. We therefore rely on abstraction and we show that, despite the over-approximation, the explanation is effective in pinpointing the cause of unfeasibility.

This industrial case study models a system that controls the height of a car's chassis. A timer tells the controller when to read the height from a filter, while disturbances which changes the height of the vehicle are modelled by the environment. The structure of the controller is depicted in Figure 3. The MSC describes a scenario where the height of the chassis falls outside the allowed thresholds, first below and then above the permitted height intervals. The sequence of events in the scenario is highlighted by the dashed line.

The scenario is not feasible due to the timing constraints imposed by the timer on each event and to the dynamics of the environment which requires an incompatible time to pass from the initial level of the chassis to a value read outside the allowed threshold. More precisely, the timer forces every event to happen every second, while the filter chassis level f read by the sensors evolve according to the differential equation $\dot{f} = \frac{h-f}{T}$, where h represents the current level. This is approximated by the linear-phase portrait partitioning which linearizes the differential equation into flow conditions of the form $\dot{f} \in [a, b]$. The constants fixed by the authors of [27] are sufficient to prove the inconsistency.

K-induction proves that the controller and the timer do not have a simple path longer than 1 alternating timed and discrete transitions (since there is no local transition). While, on the concrete state space of the environment, the portrait partitioning creates discrete loops that correspond to infinite simple paths. Therefore we rely on abstraction. We use a set of predicates in the form $t \in [i, i+1]$, $h \in [at, bt]$ and $f \in [at, bt]$ where i is an integer while a and b are the constants used in the partitioning. We localize the abstraction by using $t \in [i, i+1]$ only in the i-th event and considering the partition consistent with the initial values.

With this setting, the tool proves the unfeasibility of the scenario in 4.4 seconds reaching a depth of the longest abstract simple path equal to 6 for the local path before the first event and 9 as for the local path before the second event. The tool correctly reports an unsat core which identifies the first two events as the cause of unfeasibility. The interpolation with regards to components reports that while the timer requires that the second event must happen in no more then 3 seconds, the environment requires the same event to happen at least after 3.3 seconds.

VIII. Conclusions and Future Work

In this paper we have proposed a new approach to proving that a network of hybrid automata has no trace that satisfies a given MSC. We have also proposed the first algorithm to explain the unfeasibility of a scenario. The approach is made practical by the use of segments of the MSC to guide the search, and on the localization of simple paths. The experiments show that the proposed method significantly outperforms techniques based on the reduction to reachability, and is able to construct interesting explanations.

In the future, we will address the issue of non-linear hybrid systems, the use of hierarchical information that is often available in the network, and an automation of the abstraction-refinement loop.

Acknowledgments

This work is sponsored by the European Space Agency under contract number A0/1-6226/10/D/HK Innovative Rover OperatioNs Concepts - Autonomous Planning (IRONCAP).

References

[1] E. Ábrahám, B. Becker, F. Klaedtke, and M. Steffen. Optimizing bounded model checking for linear hybrid systems. In *VMCAI*, pages 396–412, 2005.

[2] S. Akshay, B. Bollig, and P. Gastin. Automata and logics for timed message sequence charts. In *FSTTCS*, pages 290–302, 2007.

[3] R. Alur and M. Yannakakis. Model checking of message sequence charts. In *CONCUR*, pages 114–129, 1999.

[4] G. Audemard, M. Bozzano, A. Cimatti, and R. Sebastiani. Verifying Industrial Hybrid Systems with MathSAT. *ENTCS*, 119(2):17–32, 2005.

[5] H. Ben-Abdallah and S. Leue. Timing constraints in message sequence chart specifications. In *FORTE*, pages 91–106, 1997.

[6] J. Bengtsson, B. Jonsson, J. Lilius, and W. Yi. Partial Order Reductions for Timed Systems. In *CONCUR*, pages 485–500, 1998.

[7] R. Bruttomesso, A. Cimatti, A. Franzén, A. Griggio, and R. Sebastiani. The MathSAT 4 SMT Solver. In *CAV*, pages 299–303. Springer, 2008.

[8] L. Bu, A. Cimatti, X. Li, S. Mover, and S. Tonetta. Model Checking of Hybrid Systems using Shallow Synchronization. In *FORTE*, pages 155–169, 2010.

[9] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV 2: An OpenSource Tool for Symbolic Model Checking. In *CAV*, pages 359–364, 2002.

[10] A. Cimatti, S. Mover, and S. Tonetta. Efficent Scenario Verification for Hybrid Automata. In *CAV*, 2011.
[11] A. Cimatti, S. Mover, and S. Tonetta. Hydi: a language for symbolic hybrid systems with discrete interaction. In *EUROMICRO-SEAA*, 2011.
[12] W. Damm and D. Harel. LSCs: Breathing Life into Message Sequence Charts. *Formal Methods in System Design*, 19(1):45–80, 2001.
[13] L. de Moura, H. Rueß, and M. Sorea. Bounded Model Checking and Induction: From Refutation to Verification. In *CAV*, pages 14–26, 2003.
[14] M. Fränzle and C. Herde. Efficient Proof Engines for Bounded Model Checking of Hybrid Systems. *ENTCS*, 133:119–137, 2005.
[15] M. Fränzle and C. Herde. HySAT: An efficient proof engine for bounded model checking of hybrid systems. *Formal Methods in System Design*, 30(3):179–198, 2007.
[16] S. Graf and H. Saïdi. Construction of Abstract State Graphs with PVS. In *CAV*, pages 72–83, 1997.
[17] K. Heljanko and I. Niemelä. Bounded LTL model checking with stable models. *Theory and Practice of Logic Programming*, 3(4-5):519–550, 2003.
[18] T. A. Henzinger. The Theory of Hybrid Automata. In *LICS*, pages 278–292. IEEE CS, 1996.
[19] T. A. Henzinger and P. Ho. Hytech: The cornell hybrid technology tool. In *Hybrid Systems II, LNCS 999*, pages 265–293, 1995.
[20] ITU-T. Recommendation Z.120 - Message Sequence Charts. 1996.
[21] Dubrovin J., T. Junttila, and K. Heljanko. Exploiting step semantics for efficient bounded model checking of asynchronous systems. *Sci. Comput. Program.*, 2011.
[22] J. Klose and H. Wittke. An automata based interpretation of live sequence charts. In *TACAS*, pages 512–527, 2001.
[23] P. Ladkin and S. Leue. On the semantics of message sequence charts. In *FBT*, pages 88–104, 1992.
[24] Peter B. Ladkin and Stefan Leue. Interpreting message flow graphs. *Formal Asp. Comput.*, 7(5):473–509, 1995.
[25] S. Li, S. Balaguer, A. David, K. G. Larsen, B. Nielsen, and S. Pusinskas. Scenario-based verification of real-time systems using uppaal. *Formal Methods in System Design*, pages 200–264, 2010.
[26] S. Mauw and M. A. Reniers. High-level message sequence charts. In *SDL Forum*, pages 291–306, 1997.
[27] O. Müller and T. Stauner. Modelling and verification using linear hybrid automata - a case study. *Mathematical and Computer Modelling of Dynamical Systems*, 71, 2000.
[28] M. Pan, L. Bu, and X. Li. Tass: Timing analyzer of scenario-based specifications. In *CAV*, pages 689–695, 2009.
[29] L. Pike. Real-time system verification by k-induction. Technical Report NASA/TM-2005-213751, NASA, 2005.
[30] M. Samer and H. Veith. On the Notion of Vacuous Truth. In *LPAR*, pages 2–14, 2007.
[31] V. Schuppan. Towards a notion of unsatisfiable and unrealizable cores for LTL. *Science of Computer Programming*, In press, 2010. DOI: 10.1016/j.scico.2010.11.004.
[32] M. Sheeran, S. Singh, and G. Stålmarck. Checking Safety Properties Using Induction and a SAT-Solver. In *FMCAD*, pages 108–125, 2000.
[33] S. Tonetta. Abstract Model Checking without Computing the Abstraction. In *FM*, pages 89–105, 2009.
[34] David Walter, Scott Little, Chris J. Myers, Nicholas Seegmiller, and Tomohiro Yoneda. Verification of analog/mixed-signal circuits using symbolic methods. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 27(12):2223–2235, 2008.
[35] F. Wang. Symbolic parametric safety analysis of linear hybrid systems with BDD-like data structures. *IEEE TSE*, 31(1):38–51, 2005.

Post-Silicon Fault Localisation Using Maximum Satisfiability and Backbones

Charlie Shucheng Zhu Georg Weissenbacher Sharad Malik
Department of Electrical Engineering, Princeton University

***Abstract*—The localisation of faults in integrated circuits is a challenging problem and a dominating factor in the overall verification effort. Electrical bugs, in particular, surface only in the fabricated prototypes, leading to behaviour deviating from the *golden model*. Limited observability complicates their localisation: Logging mechanisms such as trace buffers allow us to retain only a limited execution history.**

A symbolic analysis of the RTL design can find discrepancies between the values recorded in the trace buffer and the intended behaviour. Contemporary MAX-SAT solvers are then able to identify a maximal subset of the RTL design that is consistent with the observed behaviour. The elements in the *complement* of this subset represent potential locations of the fault.

The scalability of contemporary decision procedures dictates the size of a *window* of execution cycles which we can analyse using symbolic techniques. Current MAX-SAT-based fault localisation techniques require this window to span the fault as well as the error it causes. To address the scalability issues resulting from large window sizes, we propose to *slide* a smaller window along the temporal axis, constraining it with the information recorded in the trace buffer for the respective execution cycles.

In this scenario, the localisation attempt may fail: The limited information provided by the trace buffer may be insufficient to pin down the exact temporal and spatial location of the fault. We propose to use *backbones* to identify information that can be propagated across *sliding windows*. The backbone of a symbolic representation of a circuit is the set of signals that are immutable under the given constraints (e.g., the output and trace buffer values). This additional information has several benefits: Firstly, it may be instrumental in locating the fault. Secondly, it may enable a reduction of the size of the of trace buffers and the sliding window. Our preliminary experimental results demonstrate that the use of backbones allows us to reduce the size of the sliding windows or the trace buffer.

I. INTRODUCTION

The localisation of faults in fabricated prototypes, referred to as *silicon debug* or *post-silicon validation*, is a challenging and time-consuming problem. According to [1], the temporal and spatial isolation of a fault "typically dominate[s] the effort expended during the debug process for a bug." One of the aspects that distinguishes post-silicon validation from simulation or formal verification of the RTL design is the ability to execute long test scenarios. This comes at the cost of *limited observability* of signals in integrated circuits. Logging techniques such as trace buffers enable us to track a relatively small number of signals over a limited amount of time (e.g., a few thousand execution cycles). If a test case reveals an error, this limited execution history has to suffice to locate the fault causing the erroneous behaviour of the chip.

To locate the fault, we require sufficient information about the execution history to reconstruct the scenario that led to the error. Such information can be obtained by augmenting the processor with hardware recorders (referred to as *trace buffers*) that keep track of limited information to enable the reconstruction of the instruction sequence leading up to the error. An architecture-specific analysis such as IFRA [2] can then be used to narrow down the location of the fault. The advantage of this approach is that the reproduction of the failure is not required for localisation purposes. The design of such an analysis, however, requires considerable insight and needs to be adapted for individual processor architectures.

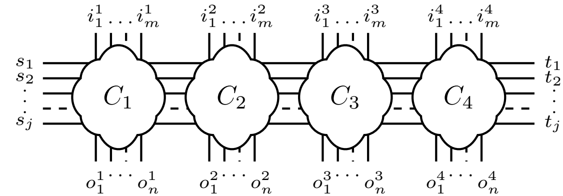


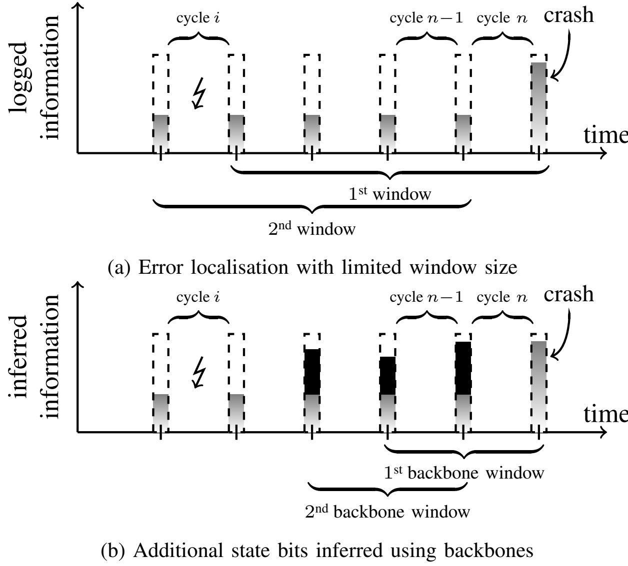
Fig. 1: Unfolded circuit encoding four execution cycles

We propose an adaptation of a fault-localisation technique which is *architecture independent* and has been successfully applied for fault diagnosis [3] as well as design debugging [4], [5], [6], [7]. Given the RTL design in a language such as Verilog, it is possible to construct a symbolic representation of k execution steps by unfolding the combinational logic C of the sequential circuit. The unfolding yields an *iterative logic array* [8] as illustrated in Figure 1. The resulting formula encodes *all correct* executions within a *window* of length k.

To identify the spatial and temporal location at which the behaviour of the device under test deviates from that of the *golden model*, we constrain the symbolic representation with the values recorded in the trace buffers. If the resulting formula is unsatisfiable, the fault must have occurred within the given window. Using the techniques presented in [9], [10], one can then compute the maximal subsets of the circuit which are *consistent* with the observed behaviour of the integrated circuit. The *complements* of these subsets (known as *minimal correction sets*) identify potential locations of the fault.

The success of this technique hinges on the size of the window and the information recorded in the trace buffers. The former is dictated by the scalability of the underlying decision procedure. The latter is determined by practical issues such as cost and required performance of the integrated circuit.

Figure 2a illustrates the situation in which a transient electrical fault in execution cycle i causes an error after cycle



(a) Error localisation with limited window size

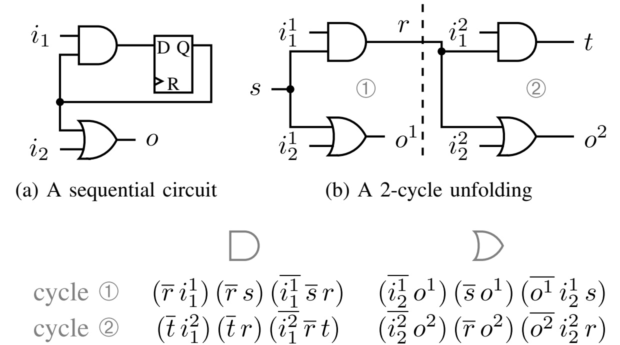
(b) Additional state bits inferred using backbones

Fig. 2: Error localisation with and without backbones

n. The dashed bars represent the state bits of the circuit for each execution step, and the shaded area indicates the fraction of those bits recorded in the trace buffer. The diagram suggests that we have near-complete information about the state bits in the *crash state* (obtained by means of scan chains [11]), but very limited information about what happened before the error occurred. The curly brackets at the bottom of Figure 2a illustrate two attempts to use a window of size four to locate the fault. The first attempt fails because the fault does not lie within the window. The second attempt fails because the information recorded in the trace buffer is insufficient to derive a contradiction: By shifting the window to the left, the analysis drops crucial information about the crash state.

Figure 2b illustrates the use of *backbones* to infer additional information that can be propagated across the sliding windows. The backbone of an unfolding of a circuit is the set of signals that are immutable under the respective constraints (i.e., the bits of the crash state and the trace buffer values). For instance, the backbone of the formula $(x \oplus y) \cdot (x + z)$ under the constraint $x \mapsto \mathbf{1}$ is $\{x \mapsto \mathbf{1}, y \mapsto \mathbf{0}\}$. The backbone of an unfolding (possibly smaller than the window used for localisation) that is constrained by the recorded state bits potentially provides additional state bits (indicated by the black bars in Figure 2b) which can be subsequently used in overlapping windows. Thus, backbones (which can be computed using the algorithms in [12]) can be used to propagate information from the crash state backwards in time to earlier cycles. This additional information can be crucial to fault localisation. We emphasise that our analysis is static and therefore not restricted to reproducible permanent faults.

Contributions: We address the scalability limits of symbolic reasoning by *sliding* an analysis window of fixed size along the temporal axis. We present a novel application of *backbones* allowing us to reduce the loss of information resulting from *limited observability* in post-silicon validation.



(a) A sequential circuit

(b) A 2-cycle unfolding

cycle ①	$(\overline{r}\, i_1^1)\,(\overline{r}\, s)\,(\overline{i_1^1}\,\overline{s}\, r)$	$(\overline{i_2^1}\, o^1)\,(\overline{s}\, o^1)\,(\overline{o^1}\, i_2^1\, s)$
cycle ②	$(\overline{t}\, i_1^2)\,(\overline{t}\, r)\,(\overline{i_1^2}\,\overline{r}\, t)$	$(\overline{i_2^2}\, o^2)\,(\overline{r}\, o^2)\,(\overline{o^2}\, i_2^2\, r)$

(c) CNF encoding of unfolded circuit. As usual, the operators $+$ and $\cdot$ are dropped for compactness. The clauses are grouped with respect to the gates and cycles by which they are contributed.

Fig. 3: A simple example

II. Fault Localisation Using Max-SAT

In the following we use propositional operators $(\cdot, +, \oplus)$ and atoms $(i, o, s, \ldots)$ to represent gates and signals, respectively. As usual, a literal is either an atom or its negation, a clause is a sum of literals, and a formula in conjunctive normal form (CNF) is a product of clauses. Every unfolded combinational circuit has an equi-satisfiable representation in CNF which can be obtained in polynomial time [13]. Each instance of a gate in the unfolded circuit corresponds to a group of clauses in the corresponding CNF representation. We use C_i to denote the CNF instantiation of the combinational logic of the RTL design representing the ith execution cycle (c.f. Figure 1). Similarly, we use T_i, a conjunction of literals, to represent the state bits recorded by the trace buffer after the ith execution step. Note that T_i is simply $\mathbf{1}$ if no state bits were recorded at that point. Moreover, we assume that T_0 and T_n represent the information available about the initial and the crash state, respectively. Finally, I_i and O_i represent the input and output constraints of cycle i. Accordingly, the CNF formula

$$W_m^k \stackrel{\text{def}}{=} \bigwedge_{i=m}^{m+k-1} \underbrace{T_{(i-1)} \cdot I_i}_{\text{input constraints}} \cdot \underbrace{C_i}_{\text{circuit}} \cdot \underbrace{O_i \cdot T_i}_{\text{output constraints}} \tag{1}$$

represents a *window* of k consecutive execution cycles starting at cycle m constrained with the corresponding inputs, outputs, and state bits recorded by the trace buffer.

Given an *unsatisfiable* instance of (1), the goal is to find the clauses (gates, respectively) that are most likely responsible for the fault. This can be achieved by identifying a *minimal correction set* (MCS), i.e., a *minimal* set of gates that need to be dropped from (1) such that the formula becomes satisfiable. This corresponds to finding the maximum set of clauses that are consistent, an NP-hard optimisation problem commonly known as Max-SAT. More specifically, we are only interested in dropping clauses that correspond to gates; the constraints introduced by trace buffer values and inputs or outputs are *hard constraints*. This specialisation of Max-SAT is known as *partial* Max-SAT. Algorithms to solve partial Max-SAT instances are discussed in [9] and [10], for instance.

Consider the sequential circuit in Figure 3a. After a reset of the flip-flop, we expect the output o to remain $\mathbf{0}$ as long as the input signal i_2 is constantly $\mathbf{0}$. Assume, however, that we observe an output value of $\mathbf{1}$ after two cycles when executing the described scenario on the chip. Figure 3b depicts a two-cycle unfolding of the circuit. Figure 3c shows the corresponding CNF encoding. Assume that we observe and record the values $o^1 \mapsto \mathbf{0}$ and $o^2 \mapsto \mathbf{1}$ during a test-run with the initial state $s \mapsto \mathbf{0}$ and the stimuli $i_2^1 \mapsto \mathbf{0}$, and $i_2^2 \mapsto \mathbf{0}$. Note that we have no information about the signal r. These observations contribute the hard constraint $\overline{o^1} \cdot o^2 \cdot \overline{s} \cdot \overline{i_2^1} \cdot \overline{i_2^2}$, which is not satisfiable in conjunction with the formula in 3c. Using a MAX-SAT solver, we can derive that the conjunction becomes satisfiable if we drop either $(\overline{r}\, s)$ or $(\overline{o^2}\, i_2^2\, r)$ (both of which are an MCS) from 3c. Accordingly, either the AND-gate in cycle one or the OR-gate in cycle two must have defaulted.

This approach also addresses multiple faults by using fault cardinality constraints [3]. Existing MCS-based fault localisation techniques require the window W_m^k (Formula 1) to span the fault as well as the crash state, which may result in a large formula exceeding the capabilities of the decision procedure.

In contrast, we *restrict the analysis to a window of fixed size* which we *slide* along the temporal axis. In the following section, we discuss a novel application of *backbones* allowing us to reduce the loss of information resulting from the limited window size.

III. Propagating Information Using Backbones

We will now consider the case in which the scalability of the underlying decision procedure dictates a window size smaller than the number of cycles of the entire test run. For the purpose of an example, we revisit the scenario in Figure 3 and assume that the window size is limited to a single cycle. Observe that neither the first line of Figure 3c in conjunction with $\overline{o^1} \cdot \overline{s} \cdot \overline{i_2^1}$ nor the second line in conjunction with $o^2 \cdot \overline{i_2^2}$ yields a contradiction; the technique introduced in §II fails. The reason is that we lack crucial information, namely the value of the signal r. We propose the use of *backbones* to aid the reconstruction of this information.

Formally, the *backbone* of a satisfiable propositional formula F comprises the values of all atoms p in F for which either $(\overline{F} + p)$ or $(\overline{F} + \overline{p})$ holds, i.e., p takes the *same* value in *all* satisfying assignments of F. We use a SAT solver to compute an initial satisfying assignment and subsequently try to flip the value of each literal, thus changing the assignment. Literals whose values differ in subsequent assignments are not part of the backbone. This algorithm performs one call to the SAT solver per literal *in the worst case*. Only the first call has to solve the instance from scratch; the subsequent calls are incremental, making the algorithm practical on large scale circuits [12]. The backbone of the formula from our example

$$\underbrace{(\overline{t}\, i_1^2)\,(\overline{t}\, r)\,(\overline{i_1^2}\,\overline{r}\, t)\,(\overline{i_2^2}\, o^2)\,(\overline{r}\, o^2)\,(\overline{o^2}\, i_2^2\, r)}_{\text{cycle ②}}\quad \underbrace{(o^2)\,(\overline{i_2^2})}_{\text{hard constraints}}\ ,$$

is $o^2 \mapsto \mathbf{1}\ i_2^2 \mapsto \mathbf{0}$, and $r \mapsto \mathbf{1}$. This backbone provides a value for the previously unknown signal r. In general, backbones *under-approximate* the information available to the SAT solver in the analysed time-frame.

Effectively, this computation propagates the error encoded in the constraint $(o^2)\,(\overline{i_2^2})$ backwards. By propagating the newly learnt information to cycle ① we obtain the desired contradiction:

$$\underbrace{(\overline{r}\, i_1^1)\,(\overline{r}\, s)\,(\overline{i_1^1}\,\overline{s}\, r)\,(\overline{i_2^1}\, o^1)\,(\overline{s}\, o^1)\,(\overline{o^1}\, i_2^1\, s)}_{\text{cycle ①}}\ \underbrace{(\overline{o^1})\,(\overline{s})\,(\overline{i_2^1})}_{\text{constraint}}\ \ \underbrace{(r)}_{\text{backbone}}$$

As expected, a MAX-SAT solver is able to determine that $(\overline{r}s)$ must be dropped, and that the AND-gate in cycle ① is a potential culprit. Notably, we missed the second fault candidate due to the limited window size and the resulting lack of information.[1] We emphasise, however, that in the absence of the information provided by the backbone, the approach described in §II is unable to diagnose the fault altogether.

In general, we use the information provided by the backbone of an instance of Formula (1) for a given m and k to augment the trace buffer (c.f. Figure 2b) of windows overlapping the interval of cycles $[m-1, m+k)$. To this end, we compute the *largest* conjunctions $B_{(m-1)}$, …, $B_{(m+k-1)}$ of literals over the signals in the respective cycles which are implied by Formula (1). Accordingly, $B_i \Rightarrow T_i$ for $i \in [m-1, m+k)$. We use $B_m^k \stackrel{\text{def}}{=} \bigwedge_{i=m}^{m+k} B_{(i-1)}$ to denote the backbone of W_m^k. Let W_m^k and W_n^l be two *satisfiable* windows with overlap o (i.e., $n = m + k - o$ and $0 \le o \le \min(k, l)$). Then, $W_m^k \cdot W_n^l = W_m^{(k+l-o)}$, according to Formula (1). Now assume that $W_m^{(k+l-o)}$ is unsatisfiable but too large for a MAX-SAT solver. Using backbones, we *approximate* W_n^l using B_n^l, i.e., $W_n^l \Rightarrow B_n^l$. The information in B_n^l can enable the localisation of faults that W_m^k failed to reveal. We construct $\widehat{W}_m^k = W_m^k \cdot B_n^l$. If $\widehat{W}_m^k$ is unsatisfiable, we use the approach described in §II to find the gates that need to be dropped from W_m^k to make $\widehat{W}_m^k$ satisfiable. Since $W_m^k \cdot W_n^l \Rightarrow \widehat{W}_m^k$, these gates are necessarily a subset of the gates that need to be dropped to make $W_m^k \cdot W_n^l$ (i.e., the larger window $W_m^{(k+l-o)}$) satisfiable.

IV. Experimental Results

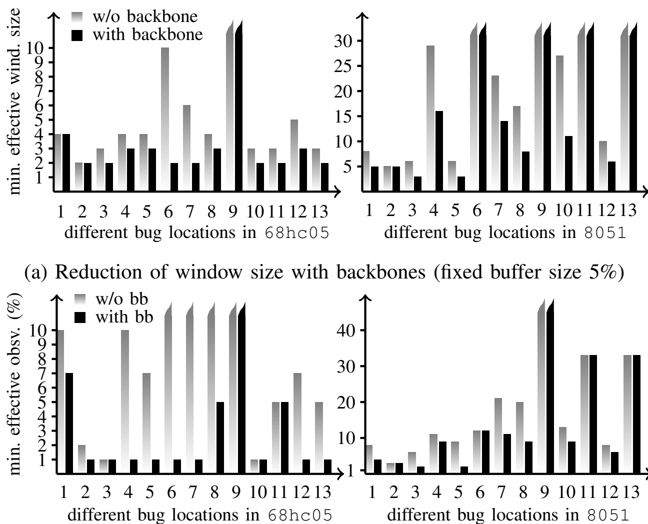
We evaluated our approach using the `68hc05` and `8051` processor designs used as case study in [14] (obtained from opencores.org). We converted the Verilog RTL designs into the DIMACS CNF format using the following translation steps:

$$\text{Verilog} \xrightarrow{\text{Altera Quartus}^2} \texttt{blif} \xrightarrow{\text{ABC}^3} \texttt{aig} \xrightarrow{\text{AIGER}^4} \texttt{cnf}$$

We randomly injected permanent *stuck-at-constant* faults (though we emphasise that our approach supports arbitrary fault models) into the RTL designs and obtained 26 failing test scenarios (13 for each design) of 2000 cycles length by means of SAT-based symbolic simulation. In each test scenario, we injected one fault at a time for the smaller `68hc05` design

[1] Note though, that a forward analysis yields the second fault candidate in our example: The backbone of cycle ① under the constraint $\overline{o^1} \cdot \overline{s} \cdot \overline{i_2^1}$ yields $r \mapsto \mathbf{0}$, which is inconsistent with the observations in cycle ②.

[2] altera.com [3] www.eecs.berkeley.edu/~alanmi/abc/ [4] fmv.jku.at/aiger/



(a) Reduction of window size with backbones (fixed buffer size 5%)

(b) Reduction of trace buffer size with backbones (using a fixed window size of 3 for the 68hc05 and 5 for the 8051 design)

Fig. 4: Experimental evaluation of backbones

and five faults at a time for the 8051 logic. All faults surface up to 1000 cycles before the end of the trace.

We recorded different percentages of the latches (chosen at random) in the trace buffer. We used the tool CAMUS [10], which implements the fault localisation algorithm discussed in §II, and the *iterative SAT-testing* algorithm described in [12] to compute backbones. The window size k is the same for computing backbones and localisation. Our implementation uses a fixed overlap of $o = k - 1$. We slid the window backwards in time along the temporal axis of the test scenarios, propagating the backbones from previously analysed windows.

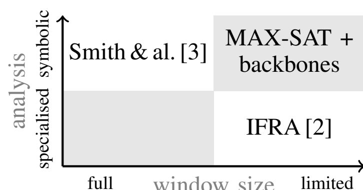
Using this setup, we ran two experiments: We *(a)* determined the minimum window size required to detect a fault for a fixed trace buffer size of 5% (Figure 4a) and *(b)* fixed the window size and increased the size of the trace buffer until the fault could be detected (Figure 4b). (Spiked bars indicate values that are off the scale.) We found that backbones enable a significant reduction of *(a)* the window size as well as *(b)* the size of the trace buffer required to locate faults.

V. Related Work

Instruction footprints [2] (c.f. §I) enable the localisation of faults by providing sufficient information about the execution. This approach requires a design dependent localisation analysis which needs to be adapted for individual architectures.

Smith et al. [3] describes a technique similar to the one in §II. The approach covers multiple faults and different fault models, but requires the *window* to span all cycles of the test run. The work does not address limited observability.

The figure to the right relates our approach and the work presented in [2] and [3]. Our approach addresses limited observability and window-size using a generic SAT-based analysis. Yang et al. [15] proposes a SAT-based technique that, given a test scenario that results in a failure, identifies signals that are relevant to the analysis of the failure and should therefore be recorded in (configurable) trace buffers. This technique could potentially be helpful to increase the size of the backbones.



Paula et al. [14] proposes to compute *signatures* of states to narrow down the set of predecessor states of the crash state, effectively enabling backwards stepping. This allows to identify the error in an earlier cycle in a subsequent test run. The approach requires the repeated reproduction of the failure, which renders the approach infeasible for the localisation of transient electrical faults (which our approach makes possible).

There is a number of papers based on the the approach described in §II that address *pre-silicon* debugging (with full observability) by constraining a *faulty* RTL model with *correct* input/output pairs (given as a specification) [4], [5], [6], [7].

VI. Conclusion

We presented a novel fault localisation technique which addresses the limited observability in post-silicon validation and demonstrated its applicability using small case studies.

References

[1] D. Josephson, “The good, the bad, and the ugly of silicon debug,” in *Design Automation Conference*. ACM, 2006, pp. 3–6.
[2] S.-B. Park and S. Mitra, “Post-silicon bug localization for processors using IFRA,” *Communications of the ACM*, vol. 53, February 2010.
[3] A. Smith, A. G. Veneris, M. F. Ali, and A. Viglas, “Fault diagnosis and logic debugging using Boolean satisfiability,” *Transactions on CAD of Integrated Circuits and Systems*, vol. 24, no. 10, pp. 1606–1621, 2005.
[4] S. Safarpour, H. Mangassarian, A. G. Veneris, M. H. Liffiton, and K. A. Sakallah, “Improved design debugging using maximum satisfiability,” in *Formal Methods in Computer-Aided Design*. IEEE, 2007, pp. 13–19.
[5] A. Sülflow, G. Fey, R. Bloem, and R. Drechsler, “Using unsatisfiable cores to debug multiple design errors,” in *Great Lakes Symposium on VLSI*. ACM, 2008, pp. 77–82.
[6] Y. Chen, S. Safarpour, A. Veneris, and J. Marques-Silva, “Spatial and temporal design debug using partial MaxSAT,” in *Great Lakes Symposium on VLSI*. ACM, 2009, pp. 345–350.
[7] Y. Chen, S. Safarpour, J. Marques-Silva, and A. Veneris, “Automated design debugging with maximum satisfiability,” *Transactions on CAD of Integrated Circuits and Systems*, vol. 29, pp. 1804–1817, 2010.
[8] M. Abramovici, M. A. Breuer, and A. D. Friedman, *Digital systems testing and testable design*. Computer Science Press, 1990.
[9] Z. Fu and S. Malik, “On solving the partial MAX-SAT problem,” in *Theory and Applications of Satisfiability Testing*, ser. LNCS, vol. 4121. Springer, 2006, pp. 252–265.
[10] M. H. Liffiton and K. A. Sakallah, “Algorithms for computing minimal unsatisfiable subsets of constraints,” *Journal of Automated Reasoning*, vol. 40, no. 1, pp. 1–33, 2008.
[11] M. J. Y. Williams and J. B. Angell, “Enhancing testability of large-scale integrated circuits via test points and additional logic,” *IEEE Transactions on Computers*, vol. C-22, pp. 46–60, 1973.
[12] J. Marques-Silva, M. Janota, and I. Lynce, “On computing backbones of propositional theories,” in *European Conference on Artificial Intelligence (ECAI)*. IOS Press, 2010, pp. 15–20.
[13] G. Tseitin, “On the complexity of proofs in poropositional logics,” in *Automation of Reasoning: Classical Papers in Computational Logic 1967–1970*, vol. 2. Springer, 1983.
[14] F. M. De Paula, M. Gort, A. J. Hu, S. J. E. Wilton, and J. Yang, “BackSpace: formal analysis for post-silicon debug,” in *Formal Methods in Computer-Aided Design*. IEEE, 2008, pp. 5:1–5:10.
[15] Y.-S. Yang, B. Keng, N. Nicolici, A. G. Veneris, and S. Safarpour, “Automated silicon debug data analysis techniques for a hardware data acquisition environment,” in *International Symposium on Quality of Electronic Design*. IEEE, 2010.

ALGEBRAIC APPROACH TO ARITHMETIC DESIGN VERIFICATION

Mohamed Abdul Basith*, Tariq Ahmad*, André Rossi †, and Maciej Ciesielski*
*ECE Department, University of Massachusetts Amherst
email: {basith,tbashir,ciesiel}@ecs.umass.edu
†Lab-STICC, Université de Bretagne Sud, Lorient, France
email: andre.rossi@univ-ubs.fr

Keywords: Formal verification, Equivalence checking, Arithmetic bit level, SMT

***Abstract*—The paper describes an algebraic approach to functional verification of arithmetic circuits specified at bit level. The circuit is represented as a network of half adders, full adders, and inverters, and modeled as a system of linear equations. The proof of functional correctness of the design is obtained by computing its algebraic signature using standard LP solver and comparing it with the reference signature provided by the designer. Initial experimental results and comparison with SMT solvers show that the method is efficient, scalable and applicable to large arithmetic designs, such as multipliers.**

I. INTRODUCTION

With the increased size and complexity of integrated circuits (IC) and systems on chip (SoC), design verification becomes a dominating factor of the overall design flow. Of particular importance (and difficulty) is verification of arithmetic datapaths and their components, such as multipliers. Unlike gate-level logic designs, which can be handled using Boolean methods, arithmetic designs require treatment on higher abstraction levels. Techniques based on decision diagrams or SAT solvers that work at the bit level are not scalable for complex arithmetic systems as they require "bit-blasting", flattening of the entire design into bit-level netlists. Modern verification methods use SMT solvers and symbolic algebra techniques, but they suffer from lack of adequate models that can harness the inherent bit-level nature of arithmetic circuits.

The work described in this paper aims at overcoming some of these limitations. It presents a novel approach to functional verification of bit-level arithmetic circuits using linear algebra techniques. The proof of correctness is obtained by modeling the arithmetic circuit as a network of half/full adders and computing its algebraic signature using a standard LP solver. The computed signature is then compared to the reference signature provided by the designer.

II. PREVIOUS WORK

Several approaches have been proposed to check an arithmetic circuit against its specification at a higher level of abstraction. Different variants of decision diagrams and canonical graph-based representations have been proposed for this purpose, including BDDs [1], BMDs [2], TEDs [3] and others. BDDs have been used extensively in logic synthesis, symbolic simulation and SAT but their application to verification of arithmetic circuits is limited due to high memory requirements. BMDs and TEDs provide more efficient representation of arithmetic circuits but require word-level information about the design, which is often not available or is hard to extract from bit-level netlists.

Computer symbolic algebra methods have been applied to model arithmetic designs as polynomials over finite rings [4]. Their applicability to verification of arithmetic circuits is also limited as it relies on a word-level representation of the datapaths. An approach to verification of bit-level implementations using theory of Grobner basis over fields has been proposed by [5] and adopted by others. A technique based on term rewriting was proposed [6] for RTL equivalence checking, using a database of rewrite rules for typical multiplier implementation schemes. However, the method cannot be automated for non-standard implementations.

In [7] a gate level network of an addition circuit (a basic component of the multiplier) is modeled as a network of half adders, called *arithmetic bit-level* (ABL) network. ABL components are modeled by polynomials over unique ring, and the normal forms are computed w.r.t. the Grobner basis over rings $Z/2^n$ using modern computer algebra algorithms. In our view this model is unnecessarily complicated and does not scalable to practical designs. A simplified version of this technique has been recently proposed whereby the expensive Grobner base computation is replaced by direct generation of polynomials representing individual outputs in terms of the primary inputs [8]. However, no general method for deriving such (potentially very large) polynomials and comparing them in a systematic way against the specification has been proposed. Our paper addresses this issue using efficient linear algebra techniques.

Another approach to solving arithmetic verification problems is based on SMT (Satisfiability Modulo Theories). SMT techniques combine SAT with specialized solvers for some well-defined theories, such as Boolean logic, linear integer arithmetic, theory of equality of uninterrupted functions, and others [9] [10]. While the application of SMT solvers to property and model checking is unquestionable, their use in functional verification of custom arithmetic circuits has not been yet addressed. This paper proposes a new theory that can enhance capabilities of SMT solvers.

III. ALGEBRAIC MODEL

It can be shown that any (logic or arithmetic) circuit can be expressed as a network of half-adders (HA), full-adders (FA) and inverters. Each arithmetic or logic operator is then modeled with a set of linear equations that relate the input and output signals. This section describes modeling of

the arithmetic network and its components using algebraic equations.

A half-adder (HA) with binary inputs $a,\ b$ and outputs S (sum) and C (carry out) is represented as

$$a + b = 2C + S \tag{1}$$

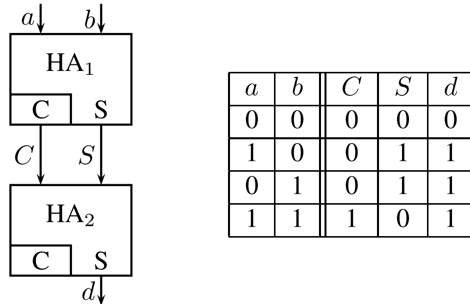
Similarly, a full adder (FA) with inputs $a,\ b,\ c_{in}$ and outputs S and C is represented as

$$a + b + c_{in} = 2C + S \tag{2}$$

Logic gates can be similarly represented by algebraic equations by deriving their functions from a half adder. Specifically, $\text{XOR}(a, b)$ is simply a sum output, S, of the half adder $\text{HA}(a, b)$, and the $\text{AND}(a, b)$ is the carry-out output, C, of $\text{HA}(a, b)$. Equations for an OR gate, $d = \text{OR}(a, b)$, can be similarly derived from the carry out (AND) output of the HA by inverting its inputs and outputs, $(1-a)+(1-b) = 2(1-d)+S$, resulting in $a+b = 2d-S$. Combining this equation with the equation (1) for HA gives $C+S = d$. As a result, an $\text{OR}(a, b)$ gate can be modeled with the following equations involving two half adders:

$$\begin{cases} a + b = 2C + S \\ C + S = d \end{cases} \tag{3}$$

Figure 1 shows the HA model for basic logic gates (AND, OR, XOR). The correctness of the equations can be verified with the attached truth table. Finally, the inverter gate $y = \text{INV}(x)$ can be trivially modeled by the following equation: $x + y = 1$.



a	b	C	S	d
0	0	0	0	0
1	0	0	1	1
0	1	0	1	1
1	1	1	0	1

Fig. 1. Modeling of logic gates using HA operators: $a+b = 2C+S; C+S = d$, where $S = \text{XOR}(a, b)$, $C = \text{AND}(a, b)$ and $d = \text{OR}(a, b)$.

Using these models, an arithmetic circuit can be represented by a system of linear equations, with variables x representing inputs (x_I), outputs (x_O) and internal signals (x_S). There is one equation for each HA, FA, XOR gate or AND gate, and a pair of equations (3) for an OR gate (c.f. Figure 1).

Algebraic equations representing the network are then combined in order to eliminate the internal variables from the equations and to represent the outputs of the circuit solely in terms of the primary inputs. The resulting expression is called *Algebraic Signature* of the design, denoted $Sig(N)$. Formally, algebraic signature is obtained by finding a linear combination of the network equations that results in an expression that relates the input and output variables.

The algebraic signature is then compared to the *Reference Signature* of the network, $Ref(N)$, which provides the expected relationship between primary inputs and outputs of the network (the golden model). The reference signature is basically the difference between the n-bit encoding of the output word (output signature) and a linear combination of input signals (input signature).

Reference signature is provided by the designer and can be obtained directly from the specification of the design. For example:

7-3 counter: The input signature of the 7-3 counter is simply the sum of the input bits, $x_1, \ldots, x_7$. With the output encoded in three bits, x_8, x_9, x_{10} the reference signature is

$$Ref(N) = (4x_8+2x_9+x_{10})-(x_1+x_2+x_3+x_4+x_5+x_6+x_7) \tag{4}$$

n-bit adder: For an n-bit binary adder, N_A, with inputs $\{a_0, \ldots, a_{n-1}, b_0, \ldots, b_{n-1}\}$ and outputs $\{S_0, \ldots, S_{n-1}, C_n\}$, the reference signature is given by:

$$Ref(N_A) = 2^n C_n + \sum_{i=0}^{n-1} 2^i S_i - (\sum_{i=0}^{n-1} 2^i a_i + \sum_{i=0}^{n-1} 2^i b_i) \tag{5}$$

2×2-bit unsigned multiplier: Since the multiplier is a non-linear circuit, we first need to convert its primary inputs $\{a_0, a_1, b_0, b_1\}$ into new variables (partial product terms), pp_I, as follows:

$$\begin{aligned} A \cdot B &= (2a_1 + a_0).(2b_1 + b_0) \\ &= 4a_1b_1 + 2a_1b_0 + 2a_0b_1 + a_0b_0 \\ &= 4pp_3 + 2pp_2 + 2pp_1 + pp_0 \end{aligned} \tag{6}$$

The variables pp_i are primary inputs to the multiplier. Assuming that the multiplier's result is encoded in 4 bits, $\{z_0, z_1, z_2, z_3\}$, the reference signature is given by:

$$Ref(N_{M2}) = (8z_3+4z_2+2z_1+z_0)-(4pp_3+2pp_2+2pp_1+pp_0) \tag{7}$$

The reference equation for signed multiplier can be derived similarly.

We shall now illustrate the idea of computing the algebraic signature using the following example.

Example 1. Figure 2 represents a 7-3 counter, a circuit that counts the number of 1s at the inputs $\{x_1, \ldots, x_7\}$ and encodes the result in a 3-bit word $S_2, S_1, S_0 = \{x_8, x_9, x_{10}\}$.

The following equations can be derived for this network using the FA model described above.

$$\begin{cases} x_1 + x_2 + x_3 - 2x_{11} - x_{12} = 0 \\ x_4 + x_5 + x_6 - 2x_{13} - x_{14} = 0 \\ x_{12} + x_{14} + x_7 - 2x_{15} - x_{10} = 0 \\ x_{11} + x_{13} + x_{15} - 2x_8 - x_9 = 0 \end{cases} \tag{8}$$

The algebraic signature of the 7-3 counter is obtained by multiplying the individual rows of equation 8 by coefficients α = {-1, -1, -1, -2}, respectively, and adding them to produce the following expression:

$$Sig(N) = (4x_8+2x_9+x_{10})-(x_1+x_2+x_3+x_4+x_5+x_6+x_7) \tag{9}$$

As we can see, the computed algebraic signature is identical to its reference signature (4) proving that the design is correct, i.e., it performs the expected function.

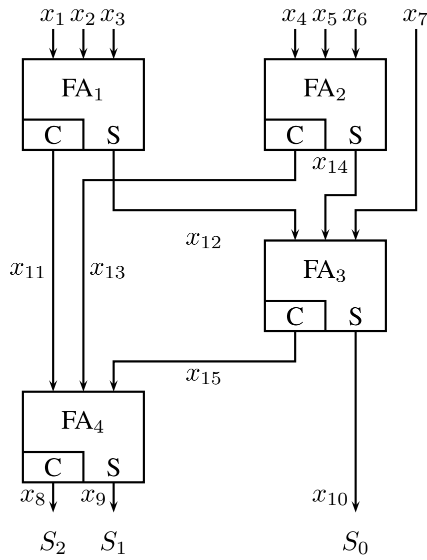


Fig. 2. Arithmetic network of a 7-3 counter.

IV. Mathematical Formulation

Let n be the total number of signals in the network, each represented by a variable, and m be the number of linear equations in the system. The network can be represented in matrix form as

$$Ax = b \tag{10}$$

where A is an $m \times n$ matrix, x is an n-vector representing the signals, and b is a constant vector of size m. Vector x of signal variables is further partitioned into the set of input signals x_I, output signals x_O, and internal signals x_S so the above system of equations can be written as: $A_I x_I + A_O x_O + A_S x_S = b$. A_I, A_O, A_S are sub-matrices of A restricted to the columns associated with input, output and internal signals, respectively. For the 7-bit counter of Fig. 2 we have $x_I = [x_1, \cdots, x_7]^T$, $x_O = [x_8, x_9, x_{10}]^T$, $x_S = [x_{11}, x_{12}, x_{13}, x_{14}, x_{15}]^T$, and b=0. Matrix A is given as follows:

$$A = \left[\begin{array}{ccccccc|ccc|ccccc} 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -2 & -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & -2 & -1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & -1 & 0 & 1 & 0 & 1 & -2 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & -2 & -1 & 0 & 1 & 0 & 1 & 0 & 1 \end{array}\right] \tag{11}$$

Similarly, the reference signature can be represented in this system as

$$Ref(N) = [r_O, -r_I]^T \cdot [x_O, x_I] \tag{12}$$

where x_O and x_I are the sets of variables representing output and input signals, and r_O, r_I are integer signature vectors associated with these variables. For the 7-3 counter example, with $x_O = [x_8, x_9, x_{10}]^T$ and $x_I = [x_1, \cdots, x_7]^T$, we have

$$Ref(N) = [4\ 2\ 1\ , -1\ -1\ -1\ -1\ -1\ -1\ -1]\cdot[x_O,\ x_I] \tag{13}$$

Given the reference signature $Ref(N)$, provided by the user, and its corresponding reference vector $[r_O, -r_I]$, the system computes the algebraic signature vector $r = [r_O, -r_I, r_S]$ of the network. The goal is to determine if the computed algebraic signature $Sig(N) = r^T x$ matches the given reference signature $Ref(N) = [r_O, -r_I]^T \cdot [x_O, x_I]$

As explained in Section III, signature $Sig(N) = r^T x$ is obtained as a linear combination α of the rows of Ax. Our goal is to compute vector α such that

$$[A_O, A_I, A_S]^T \alpha = [r_O, -r_I, r_S] \tag{14}$$

This is done by first solving the following linear system for α using standard LP solver:

$$\begin{cases} A_O^T \alpha = r_O \\ A_I^T \alpha = -r_I \end{cases} \tag{15}$$

Here r_S is relaxed, i.e., the internal variables are not taken into account. If this system has no solution, i.e., there is no linear combination of rows of Ax that will produce an algebraic signature whose inputs and outputs match those of the reference signature $Ref(N)$, the circuit is *incorrect* (w.r.t. that signature). If the system has a solution, the signature vector r_S associated with internal variables is computed as follows:

$$r_S = A_S^T \alpha \tag{16}$$

Ideally we are interested in having the internal variables eliminated ($r_S = 0$) as a condition for satisfying the reference signature. Applying this approach to the 7-3 counter circuit, we obtain α^T = [-1 -1 -1 -2], from which the signature vector can be calculated as $r = A^T \alpha$. The computed $r = r_O$ and r_I match those of the reference equation and $r_s = A_S^T \alpha = 0$; that is, all the internal signals have been eliminated from the signature.

But what if the computed signature $Sig(N)$ contains internal signals, i.e., if $r_S \neq 0$? We refer to such an expression as a *residual expression*, $RE(N) = Sig(N) - Ref(N)$. Does the existence of $RE(N)$ mean that the system does not satisfy the reference signature and the design is incorrect? It can be shown that this is not necessarily the case and that $r_S = 0$ is a sufficient but not a necessary condition for the design to be correct. In fact, a sufficient and necessary condition for circuit correctness is that RE reduces to zero for all the variable valuations that are produced by the network. In this case the network signature matches exactly the reference signature and the design is correct. This is illustrated with the following example.

Example 2. Consider a 2×2 signed multiplier network, shown in Figure 3. The combination of HA$_3$ and HA$_4$ models an OR gate. Inputs to the network are partial product terms pp_i, generated from the actual inputs of the multiplier, a_1, a_0, b_1, b_0, by a standard partial product generator. Hence, the expected input signature for the network is:

$$\begin{aligned} Sig_I(N) &= (-2a_1 + a_0)(-2b_1 + b_0) \\ &= 4a_1b_1 - 2a_1b_0 - 2a_0b_1 + a_0b_0 \\ &= 4pp_3 - 2pp_2 - 2pp_1 + pp_0 \end{aligned} \tag{17}$$

Hence the reference signature for this design is: $Ref(N) = -8z_3 + 4z_2 + 2z_1 + z_0 - 4pp_3 + 2pp_2 + 2pp_1 - pp_0$, where the first four terms are the output signature, obtained directly from the encoding of the output bits.

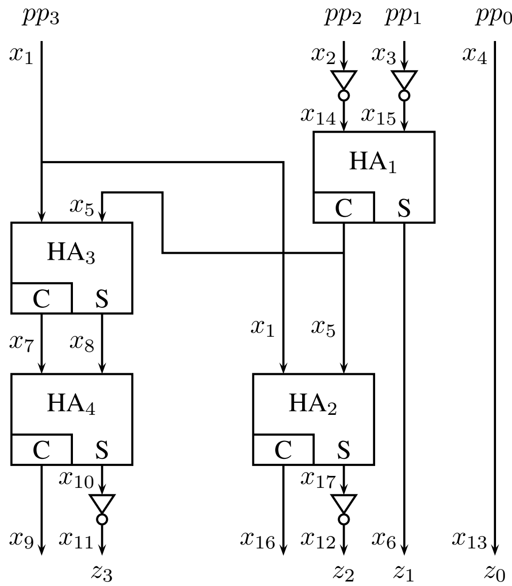


Fig. 3. Signed 2×2 multiplier network.

The algebraic signature $Sig(N)$ computed by the system is:

$$\begin{aligned} &Sig(N) = \\ &-8z_3 + 4z_2 + 2z_1 + z_0 - 4pp_3 + 2pp_2 + 2pp_1 - pp_0 \\ &+16x_9 - 4x_8 + 4x_{17} \end{aligned} \tag{18}$$

We note that the signature contains a residual expression $RE = -16x_9 + 4x_8 - 4x_{17}$. However, it can be shown that this expression always evaluates to zero. Namely, x_9=0 since it is the carry-out C output of HA$_4$ modeling the OR gates, which is always zero (refer to the truth table in Figure 1). The remaining variables, $x_8,\ x_{17}$, are two equivalent outputs S of HA$_2$ and HA$_3$ that share the same inputs. Hence x_8=x_{17}, which reduces RE to zero. Such an analysis of internal equivalences allows one to determine whether the residual expression evaluates to zero. If it does, the network performs the desired function expressed by the reference signature and the circuit is considered correct. Otherwise the circuit is incorrect, i.e., it does not perform the function described by the reference signature.

V. Experimental Results

The arithmetic verification technique described in the paper has been implemented as a prototype program written in C. The program uses GLPK package [11] to solve the linear system needed to compute an algebraic signature of the network.

A detailed flow of the verification procedure based on algebraic signature computation is shown in Fig. 4. The input to the system is the description of the arithmetic network N, composed of arbitrary logic gates, HA and FA operators, along with the reference signature provided by the designer. The system computes a complete signature of the network and reports if there is a non-empty residual expression $RE(N)$. If $RE \neq 0$, additional constraints need to be extracted from the network and imposed on RE in an attempt to prove that it is zero. These constraints come in two flavors: 1) signal *equalities*, caused by fanout of internal signals, e.g., $x_8 = x_{17}$ in Example 2; and 2) Boolean *constants*, such as x_9=0 in Example 2.

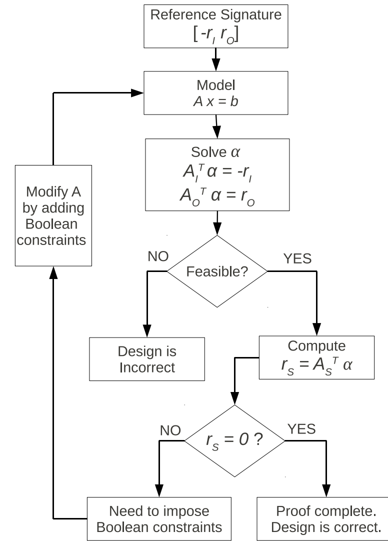


Fig. 4. Flowchart of the functional verification system.

Note that by construction (equation 15) the signature vector of a correctly designed circuit will always match its reference signature, otherwise the system has no solution and the circuit is declared incorrect.

We conducted a set of experiments on a number of arithmetic circuits, including large integer multipliers. First, a bit-level structural verilog code was generated for each multiplier using a generic multiplier generator software. (courtesy of the University of Kaiserslauten). The verilog code was parsed to transform the multiplier circuit to a network of HA, FA and basic logic gates from which a system of linear equations was generated, as described in Section IV. Finally, our program with link to GLPK was used to compute the algebraic signature for the network, given the expected reference signature.

Since multipliers are non-linear networks, we concentrated on the part of the designs which uses partial products as its inputs. Equation 17 illustrates the generation of partial product, a_ib_j, for a 2-bit area multiplier. Similar expressions can be readily obtained for Booth-recoded products. Such recoded product generator can be easily proved using Boolean methods.

Table I shows our results for a set of signed integer multipliers up to 256 × 256 bits. The experiment was conducted on a 2 GHz machine running Linux, with Intel(R) Dual Core(TM) T3200 processor and 3GB RAM. Since most of the research in this field has been done in the context of property checking rather than strictly functional verification, we could only compare our results to those in [12], for arithmetic proof (AP) of integer multipliers. The table gives the size of the multiplier (in the number of bits n of each operand); the number of linear equations ($constr$); the CPU time to compute

the signature and the CPU time for arithmetic proof (AP) of integer multipliers, reported in [12]. The AP results were computed on a comparable 64-bit 2 GHz Power5 machine, and reported only for 24, 53 and 64 bit integer multipliers.

The computed signatures were free of residual expressions after imposing simple Boolean constraints (constants 0) related to the OR gate configuration discussed earlier.

Size (n)	This work mult $n \times n$		AP [12]	Z3	Yices
	Constr.	CPU (sec)	sec	sec	sec
3	21	0.00	-	0.23	0.02
4	44	0.00	-	466.36	0.05
8	216	0.00	-	MO	TO
16	944	0.02	-	MO	TO
24	2184	0.04	7	MO	TO
30	3450	0.07		MO	TO
32	3936	0.09	-	MO	TO
53	8268	0.77	480	MO	TO
64	12096	1.14	840	MO	TO
128	48768	17.09	-	MO	TO
192	110016	45.23	-	MO	TO
256	195840	151.95	-	MO	TO

TABLE I
CPU TIME FOR COMPUTING ALGEBRAIC SIGNATURE OF n-BIT INTEGER SIGNED MULTIPLIERS. (MO = OUT OF MEMORY 3 GB; TO = TIMEOUT AFTER 1800 SEC)

The runtime complexity of the procedure to compute algebraic signature of the network is less then $O(n^2)$ in terms of the number of gates in a gate-level implementation of the design, c.f. Figure 5. In principle, given a network N described

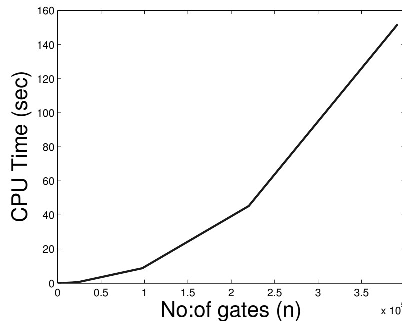


Fig. 5. Runtime complexity of the computation of algebraic signature.

by a linear system $Ax = b$, checking if the network satisfies the reference signature $Ref(N)$ can be cast as a SAT problem. Specifically, we need to show that $(Ax = b) \wedge (Ref(N) \neq 0)$ is unsatisfiable (unSAT). We performed this test for the multiplier circuits using two SMT solvers, Yices and Z3, that support Linear Integer Arithmetic as one of their theories. The results are shown in the last two columns of Table I. The SMT solvers were not able to solve this problem for multipliers with more than 8 bits. Z3 runs out of memory (3 GB) while Yices is unable to complete the computation in 30 minutes.

VI. Conclusions and Future Work

The purpose of this work was to show a potential of the proposed algebraic technique to verify functionality of arithmetic circuits. The method is based on computing algebraic signature and comparing it with the reference signature that uniquely defines behavior of the design. If the computed signature contains non-zero residual expression RE, the signature computation must be followed by a proof that RE reduces to zero. This requires extracting constraints that are not properly captured by the linear model. Alternatively, such constraints can be imposed on the linear system directly. In this case the correct design should have no residual expression. In fact this was the case with the multipliers presented in Section V. We believe that such constraints are not hard to extract and are related to only a few types of configurations, such as constant 0 and equivalence of signals derived from a fanout, as discussed earlier. This issue is currently under investigation.

The described technique is also applicable to property checking, by representing the property by its algebraic signature and checking if it is consistent with the signature of the network. The feasibility of the resulting linear system will indicate whether such a consistency is maintained or not.

Finally, the method is limited to designs with known reference signature and such a signature must be a linear expression. This is certainly the case for portions of the designs composed of half adder networks (such as Wallace trees) often encountered in complex arithmetic designs. Application to other types of circuits needs to be examined.

Acknowledgments

This work has been supported by a grant from the National Science Foundation under award No. CCF-0702506.

References

[1] R. Bryant, "Graph-Based Algorithms for Boolean Function Manipulation," in *IEEE Trans. on Computers*, August 1986, vol. 35, pp. 677–691.
[2] R. Bryant and Y. Chen, "Verification of Arithmetic Functions with Binary Moment Diagrams," in *Proc. Design Automation Conference*, 1995, pp. 535–541.
[3] M. Ciesielski, P. Kalla, and S. Askar, "Taylor Expansion Diagrams: A Canonical Representation for Verification of Data Flow Designs," *IEEE Trans. on Computers*, vol. 55, no. 9, pp. 1188–1201, Sept. 2006.
[4] N. Shekhar, P. Kalla, and F. Enescu, "Equivalence Verification of Polynomial Data-Paths Using Ideal Membership Testing," in *IEEE Trans. on Computer-Aided Design*, July 2007, vol. 26, pp. 1320–1330.
[5] Y. Watanabe, N. Homma, T. Aoki, and T. Higuchi, "Application of Symbolic Computer Algebra to Arithmetic Circuit Verification," in *Proc. Intl. Conf. on Computer Design*, 2007, pp. 25–32.
[6] S. Vasudevan, V. Viswanath, R. W. Sumners, and J. A. Abraham, "Automatic Verification of Arithmetic Circuits in RTL using Stepwise Refinement of Term Rewriting Systems," in *IEEE Trans. on Computers*, 2007, vol. 56, pp. 1401–1414.
[7] O. Wienand, M. Wedler, D. Stoffel, W. Kunz, and G. Greuel, "An Algebraic Approach for Proving Data Correctness in Arithmetic Data Paths," in *Proc. Intl. Conf. on Computer-Aided Verification*. July 2008, pp. 473–486, Springer-Verlag Berlin Heidelberg 2008.
[8] E. Pavlenko, M. Wedler, D. Stoffel, and W. Kunz, "STABLE: A new QF-BV SMT Solver for hard Verification Problems combining Boolean Reasoning with Computer Algebra," in *Proc. Design Automation and Test in Europe*, 2011.
[9] A. Biere, M. Heule, H. V. Maaren, and T. Walsch, *Satisfiability Modulo Theories in Handbook of Satisfiability*, IOS Press, 2008, Chapter 12.
[10] D. Kroening and O. Strichman, *Decision Procedures, An Algorithmic Point of View*, Springer, 2008.
[11] "GNU, GLPK Linear Programming Kit," http://www.gnu.org/software/glpk/, 2009.
[12] U. Krautz, M. Wedler, W. Kunz, K. Weber, C. Jacobi, and M. Pflanz, "Verifying Full-Custom Multipliers by Boolean Equivalence Checking and an Arithmetic BitLevel Proof," in *Proc. Asia and South Pacific Design Automation Conference*, 2008, pp. 398–403.

Time-Bounded Analysis of Real-Time Systems

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

Arie Gurfinkel
SEI/CMU
arie@cmu.edu

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

***Abstract*—Real-Time Embedded Software (RTES) constitutes an important sub-class of concurrent safety-critical programs. We consider the problem of verifying functional correctness of periodic RTES, a popular variant of RTES that execute *periodic tasks* in an order determined by *Rate Monotonic Scheduling* (RMS). A computational model of a periodic RTES is a finite collection of terminating tasks that arrive periodically and must complete before their next arrival.**

We present an approach for *time-bounded* verification of safety properties in periodic RTES. Our approach is based on sequentialization. Given an RTES $\mathcal{C}$ and a time-bound $\mathcal{W}$, we construct (and verify) a sequential program $\mathcal{S}$ that over-approximates all executions of $\mathcal{C}$ up to time $\mathcal{W}$, while respecting priorities and bounds on the number of preemptions implied by RMS. Our algorithm supports partial-order reduction, preemption locks, and priority locks. We implemented our approach for C programs, with properties specified via user-provided assertions. We evaluated our tool on several realistic examples, and were able to detect a subtle concurrency issue in a robot controller.

I. Introduction

Real-Time Embedded Software (RTES) is an important sub-class of concurrent safety-critical programs. They play a crucial role in controlling systems ranging from airplanes and cars, to infusion pumps and microwaves. We are increasingly reliant on these *cyber-physical* systems to maintain our modern technology-driven way of life. As such, verifying the correct operation of RTES is an important and open challenge. Addressing this challenge is the subject of our paper.

Specifically, we focus on systems that receive as input a collection of *periodic tasks*, where each task τ_i has, among other things, a terminating task body T_i and a period P_i. The tasks are *prioritized* in a Rate-Monotonic [1] fashion, which means that tasks with shorter periods have higher priorities. We call such an input pattern a *periodic program*.

A periodic program $\mathcal{C}$ is executed by running its tasks periodically and concurrently with asynchronous *priority-sensitive* interleaving. Thus, at each scheduling point, the active task with the highest priority is selected for execution. A task τ_i becomes inactive at the end of its body T_i, and is reactivated after P_i time has passed since its last activation. A single execution of a task body is called a *job*. It is convenient to view an execution of $\mathcal{C}$ as an asynchronous priority-sensitive interleaving of the jobs statements, where the statements arise from the infinite job streams corresponding to the periodic execution of the task bodies.

Periodic programs constitute an important fragment of RTES that interact with the physical world. In particular, the task periods are dictated by the physical environment and the underlying control algorithms. Consider, for example, the nxt/OSEK-based [2] LEGO MINDSTORM robot controller. It has three periodic tasks: a *balancer*, with a 4 millisecond (ms) period, maintains the balance of the robot; *obstacle*, with a 50 ms period, monitors a sonar sensor to detect obstacles; and *bluetooth*, with a 100 ms period, monitors a bluetooth link for remote commands from the user. Another example is a generic avionic mission system that was described in [3]. It includes 10 periodic tasks, including weapon release (10 ms), radar tracking (40 ms), target tracking (40 ms), aircraft flight data (50 ms), display (50 ms) and steering (80 ms). Other examples of periodic programs include phase-array radars and aircraft collision-avoidance systems.

These examples demonstrate the fact that periodic programs are used for developing a wide range of RTES that interact with the physical world, and play an important role in the correct operation of safety-critical systems. Statically predicting behavior – by verifying logical and timing properties of periodic programs – is a problem of great practical relevance.

Despite a wide body of work, the state-of-the-art in verification of real-time and concurrent programs does not address logical properties of periodic programs (with the exception of the recent work of Kidd et al. [4], which we discuss in Sec. VIII). On one hand, techniques for verifying properties of timed systems [5], [6] are based on Timed Automata [7]. They abstract away significantly the behavior (i.e., control- and data-flow) of target systems, and, therefore, are unsuitable for analyzing logical properties. On the other hand, approaches for concurrent software verification (e.g., [8]) employ a non-deterministic scheduler model (i.e., tasks do not have priorities or periods), and thus cannot handle the execution semantics of periodic programs. Against this backdrop, our main contribution is the development and evaluation of an approach to verify logical properties of periodic programs.

Specifically, we present an approach for *time-bounded* verification of safety properties of periodic programs. The inputs are: (i) a periodic program $\mathcal{C}$; (ii) a safety property expressed via an assertion A embedded in $\mathcal{C}$, (iii) an initial condition $Init$ of $\mathcal{C}$, and (iv) a time bound $\mathcal{W}$. Time-bounded verification can be seen as an analogue of Bounded Model Checking (BMC) for RTES, since time is a natural way to bound an execution of a periodic program for the purpose of verification.

Our solution for the time-bounded verification problem is based on *sequentialization* – reducing verification of a concurrent program to verification of a sequential program. It is inspired by work on sequentialization for *Context-Bounded*

Analysis [8], [9], [10], [11], [12] (CBA) and Bounded Model Checking [13] (BMC). A key distinguishing aspect of our work is that instead of bounding the number of context switches (as in CBA), or the number of execution steps (as in BMC), the input time bound $\mathcal{W}$ translates in our model to a bound on the number of jobs. This is a natural consequence of the fact that tasks are periodic and, therefore, are activated a finite number of times within $\mathcal{W}$. Our solution also handles two types of locks used commonly in periodic programs, and incorporates two forms of partial-order reduction aimed at reducing analysis time.

We have implemented our solution in a tool, called REK. Our tool is able to verify periodic programs implemented in C where tasks communicate via shared variables and synchronize via locks. We have used it to verify several variants of a nxt/OSEK-based [2] LEGO MINDSTORM robot controller. In some instances, we found subtle concurrency issues in the controller using our tool. We have also evaluated our tool on several custom versions of the reader-writer protocol.

The rest of the paper is structured as follows: in the next section we formally define a periodic program and its semantics. In Sec. III, we describe time-bounded and job-bounded abstractions. In Sec. IV, we describe our encoding method. In Sec. V, we extend our model with locks, and in Sec. VI describe partial-order reduction. In Sec. VII, we describe our case study and experimental results. Finally, we discuss related work in Sec. VIII, and conclude in Sec. IX.

II. PRELIMINARIES

A *task* τ is a tuple $\langle I, T, P, C, A\rangle$, where I is a task identifier, T – a bounded procedure (i.e., no unbounded loops or recursion) called the task body, P – a period, C – the worst case execution time of T, and A, called the release time, is the time at which the task is first enabled[1]. An N-task *periodic program* $\mathcal{C}$ is a set $\{\tau_0, \ldots, \tau_{N-1}\}$ of N tasks. For simplicity, we assume that the id of task τ_i is i.

In this paper, we restrict the priorities of the tasks to be *rate-monotonic* – tasks with smaller period have higher base priority. For simplicity, we assume that the index of a task represents its base priority. Thus, a task with a lower id has a lower base priority (and higher period).

A periodic program is executed by running each task periodically, starting at the release time. For $k \geq 0$ the k-th job of τ_i becomes enabled at time $A_i^k = A_i + k \times P_i$. The execution is asynchronous and priority-sensitive – at each point the CPU is given to an enabled task with the highest priority. Priorities can change dynamically, but must avoid *priority inversion* – when a low base priority task preempting a higher base priority task. This is known, somewhat misleadingly, as a *fixed-priority preemptive scheduling*.

Formally, the semantics of an N-task periodic program $\mathcal{C} = \{\tau_0, \ldots, \tau_{N-1}\}$ is the asynchronous concurrent program:

$$\|_{i=0}^{N-1}\ k_i := 0\,;\, \textbf{while}(\textsc{Wait}(\tau_i, k_i))\ (T_i\,;\, k_i := k_i + 1) \quad (1)$$

[1]We assume that time is given in some fixed time unit (e.g., milliseconds).

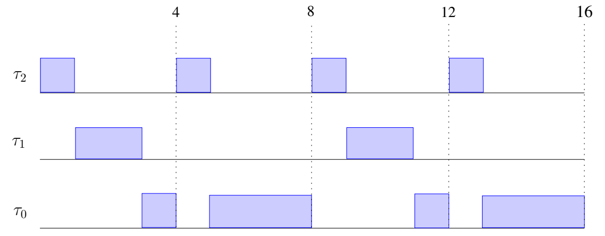


Fig. 1. A schedule of three tasks from Example 1.

where k_i is a numeric variable and $\textsc{Wait}(\tau_i, k_i)$ returns FALSE if the current time is greater than $A_i^{k_i}$, and otherwise it disables τ_i until the time is $A_i^{k_i}$ and then returns TRUE.

An execution of each task body T_i in (1) is called a *job*. A job's *arrival* is the time when it becomes enabled (i.e., $\textsc{Wait}(\tau_i, k)$ in (1) returns TRUE); *start* and *finish* are the times when its first and last instructions are executed, respectively; *response time* is the difference between its finish and arrival times. The *response time* of a task is the maximum of response times of all of its jobs in all possible executions.

Note that WAIT in (1) returns TRUE if a job has finished before its next period. If this is always the case, i.e., WAIT never returns FALSE, then the program is called *schedulable*.

Formally, a periodic program $\mathcal{C}$ is schedulable if for each task τ_i, the response time RT_i is less than the period P_i. Response times are computed using Rate Monotonic Analysis (RMA) [14]. For a periodic program $\mathcal{C} = \{\tau_0, \ldots, \tau_{N-1}\}$, the response time RT_i of task τ_i is the smallest solution to the following equation

$$RT_i = C_i + \sum_{i<k<N} \lceil \frac{RT_i}{P_k} \rceil \cdot C_k\ . \quad (2)$$

Intuitively, the response time of a task is equal to its worst-case execution time plus the time taken by all higher-priority tasks that preempted it. RT_i is computed by solving (2) iteratively starting with $RT_i = C_i$ [14]. Note that for the highest-priority task the response time RT_{N-1} is its execution time C_{N-1}.

Example 1 *Consider the task set:*

Task	C_i	P_i
τ_2	*1*	*4*
τ_1	*2*	*8*
τ_0	*8*	*16*

Solving (2) gives us $RT_2 = 1$, $RT_1 = 3$ and $RT_0 = 16$. A schedule demonstrating these values is shown in Fig. 1.

In this paper, we are interested in logical properties of periodic programs. We assume that any program we analyze meets its basic timing constraints. For that reason, we only deal with schedulable periodic programs.

III. TIME-BOUNDED PERIODIC PROGRAMS

In this section, we present time-bounded semantics of periodic programs and define a job-bounded abstraction that is the formal foundations of our verification technique.

Given a periodic program $\mathcal{C} = \{\tau_0, \ldots, \tau_{N-1}\}$ and a time-bound $\mathcal{W}$, the time-bounded program $\mathcal{C}_\mathcal{W}$ executes like $\mathcal{C}$ for time $\mathcal{W}$ and then terminates. We assume that $\mathcal{W}$ is divisible by the period of each task. Furthermore, we assume that the first job of each task finishes before its period, i.e., $A_i \le P_i - RT_i$. Under these assumptions, the time bound imposes a natural limit on the number of jobs J_i of each task:

$$J_i = \frac{\mathcal{W}}{P_i} \ . \tag{3}$$

Therefore, the semantics of $\mathcal{C}_\mathcal{W}$ is equivalent to the asynchronous concurrent program:

$$\|_{i=0}^{N-1}\ k_i := 0; \textbf{while}(k_i < J_i \wedge \text{WAIT}(\tau_i, k_i))\ (T_i; k_i := k_i + 1) \ . \tag{4}$$

This is analogous to the semantics of $\mathcal{C}$ in (1) except that each task τ_i executes J_i jobs.

Job-bounded Abstraction. Since we are interested in logical properties, we need to abstract the absolute time in (4) with relative order of execution. A simple abstraction is to interpret WAIT as a non-deterministic delay, effectively replacing a time-bound with a job-bound. We further refine this abstraction using the following observation about RMA (2). Let τ_i and τ_j be two tasks of a schedulable periodic program $\mathcal{C}$ such that $i < j$. Then RMA defines the *preemption bound* of i by j, written PB_i^j, as follows

$$PB_i^j = \lceil \frac{RT_i}{P_j} \rceil \ . \tag{5}$$

That is, PB_i^j is an upper bound on the number of times τ_j can preempt τ_i. Thus, in addition to treating WAIT as a non-deterministic delay, we only allow for a job of task j to preempt a job of task i if $j > i$ and it does not violate the preemption bound PB_i^j. In other words, we only schedule at most PB_i^j jobs of τ_j while a job of τ_i is active. We call this the job-bounded abstraction of $\mathcal{C}_\mathcal{W}$ and denote it by $\mathcal{C}_{J(\mathcal{W})}$.

Theorem 1 (Soundness of Job-bounded Abstraction) *For a periodic program $\mathcal{C} = \{\tau_0, \ldots, \tau_{N-1}\}$ and a time-bound $\mathcal{W}$ s.t. $\forall i \cdot (A_i \le P_i - RT_i) \wedge (\mathcal{W} | P_i)$, every execution of $\mathcal{C}_\mathcal{W}$ is also an execution of $\mathcal{C}_{J(\mathcal{W})}$.*

Obviously, job-bounded abstraction is incomplete, i.e., $\mathcal{C}_{J(\mathcal{W})}$ may have more executions than $C_\mathcal{W}$. The primary reason being that the preemption bounds are only upper bounds since they are computed from the worst-case execution times, and rounded upwards in (5). The incompleteness due to rounding up is shown by the following example.

Example 2 *Let $\mathcal{C} = \{\tau_0, \tau_1\}$ such that $P_0 = 25$, $RT_0 = 22$ and $P_1 = 10$, $RT_1 = 10$, and $\mathcal{W} = 100$. Then, $PB_0^1 = 3$ and, therefore, in $\mathcal{C}_{J(\mathcal{W})}$ two consecutive jobs of τ_0 can be preempted three times, each, by jobs of τ_1. However, in $\mathcal{C}_\mathcal{W}$ two consecutive jobs of τ_0, taken together, can be preempted at most five times by jobs of τ_1.*

In the next section, we give an algorithm for constructing and verifying the job-bounded abstraction $\mathcal{C}_{J(\mathcal{W})}$ from a periodic program $\mathcal{C}$.

IV. Sequentialization of a Periodic Program

We use a two-step approach to verify an N-task periodic program $\mathcal{C} = \{\tau_0, \ldots, \tau_{N-1}\}$ under a time bound $\mathcal{W}$. The first step, sequentialization, outputs a non-deterministic sequential program with assume statements $\mathcal{S}$, as shown in Alg. 1. The program $\mathcal{S}$ is semantics preserving w.r.t. $\mathcal{C}_{J(\mathcal{W})}$ (see Theorem 2). The second step is the verification of $\mathcal{S}$ with an off-the-shelf program verifier. In the rest of this section, we focus on sequentialization.

A. Sequentialization: Intuition

Our key insight is that any execution π of $\mathcal{C}$ can be partitioned into scheduling *rounds* in the following way: (a) π begins in round 0, and (b) a round ends and a new one begins every time a job ends (i.e., the last instruction of some task body is executed). For example, the bounded execution shown in Fig. 1 is partitioned into 7 rounds as follows: round 0 is the time interval $[0, 1]$ – the end of the first job of τ_2, round 1 is $[1, 3]$ – the end of the first job of τ_1, round 2 is $[3, 5]$ – the end of the second job of τ_2 (note that there is only one job of τ_0 and it ends at time 16), round 3 is $[5, 9]$, etc.

Observe that in each round, the tasks are executed sequentially in the order of their priority starting with the task of the lowest priority. Furthermore, a bounded execution in which exactly k jobs start and end has exactly k rounds.

The basic idea of our sequentialization is to reduce a bounded concurrent execution with k jobs into a sequential execution with k rounds. Initially, jobs are allocated (or scheduled) to rounds. Then, each round is executed independently (and sequentially) starting from a guessed initial state. Finally, we check for each $0 \le i < (k-1)$, that the guessed initial state at round $i+1$ is the final state of round i.

Our sequentialization is inspired by the work of Lal and Reps [9], but differs from it in several significant ways. First, we deal with periodic programs and not non-deterministically scheduled concurrent programs. Second, since we are not exploring non-deterministic schedules, we use a more refined scheduling scheme than the non-deterministic round-robin used in [9]. Third, we partition each task (or, correspondingly, a thread in [9]) into jobs and preserve *all* executions in which *all* jobs terminate. In contrast, [9] only preserves executions with a bounded number of thread-preemptions. Finally, we take into account that preemption between tasks must respect priorities and preemption bounds.

B. Sequentialization: Details

The sequential program $\mathcal{S}$ starts in MAIN (see Alg. 1). It first allocates jobs to rounds (line 5), initializes global variables (lines 6–8), executes all jobs of all tasks sequentially starting with the first job of the lowest priority task (lines 9–13), and finally checks correctness of guessed variables and assertions (lines 14–15).

Job scheduling. A job schedule is a pair of mappings $start$ and end that map each job of each task to a starting and ending round, respectively. We say that a job schedule is legal iff it satisfies the following three properties: (a) jobs are sequential

– for a given task, for any pair $i < j$, job i starts and finishes (i.e., precedes) job j, and (b) jobs are well-nested – if a higher and a lower priority jobs overlap, then the higher priority job must start and end within the rounds of the lower priority one, and (c) jobs respect preemption bounds – no more than $PB_{t_i}^{t_j}$ jobs of task t_j are scheduled inside any job of task t_i.

SCHEDULEJOBS fills arrays $start$ and end with a non-deterministic *legal* job schedule. Line 24 ensures that the jobs are sequential, while line 25 ensures that they are well-nested, and line 26 ensures that they respect preemption bounds. To understand line 26, suppose that $PB_{t_1}^{t_2} = 3$, $j_2 = 4$ and that j_2 is nested in j_1. Then, to satisfy $PB_{t_1}^{t_2}$, we require that job 1 of t_2 has ended before j_1 started. Otherwise, because the jobs are sequential and well-nested, j_1 contains jobs 1, 2, 3, and 4 of t_2 – that is, it is preempted by more than three t_2 jobs.

Global variables. For every global variable g, we create two arrays $g[\,]$ and $v_g[\,]$ such that $g[i]$ is the value of g in round i, and $v_g[i]$ is the guess of the initial value of g in round i. The element $g[0]$ is initialized in line 7 to the user-specified initial value i_g, and each other elements $g[r]$ is assigned the corresponding initial guess $v_g[r]$ in line 8.

In addition, a variable rnd tracks the current round, job tracks the current job, and $endRnd$ is the scheduled end round of the current job.

Tasks. For each task t, MAIN uses a modified version $\hat{T}_t$ obtained from the original task body T_t by preceding each statement st with a call to CS to emulate a preemption, and replacing every global variable g in st with $g[rnd]$. This is based on an assumption, without loss of generality, that each global variable g is explicitly loaded and stored, i.e., g only appears in statements of the form $g := l$ or $l := g$, where l is a local variable.

Preemption. CS models a preemption (a context switch to a higher priority task) by increasing non-deterministically the value of rnd to the round in which this task resumes execution. However, not every round between the start and end of the current job is legitimate. Line 21 ensures that the execution is not resumed in a round used by a higher-priority task. CS returns TRUE iff a preemption has occurred. This value is used later in Sec. VI.

Prophecy-Check. Line 14 ensures that the value of each global variable at the end of a round is equal to its guessed value at the beginning of the next round.

Assertions. Assertions need special handling. They can only be checked after the guesses have been validated via Prophecy-Check. Without loss of generality, we assume a single call to assert in the body of each task. We use an array *localAssert* that maps a task and a job to the value of the assertion in it. The element *localAssert*$[t][j]$ is initialized to TRUE (line 6), set to the value of the asserted expression (line 29), and asserted to be TRUE (line 15) after the Prophecy-Check.

Our sequentialization procedure is semantic preserving as expressed in the following theorem.

Theorem 2 *Let $\mathcal{C}$ be a periodic program and $\mathcal{W}$ a time-bound satisfying the conditions in Theorem 1. Then, for every execution π of $\mathcal{C}_{J(W)}$ that violates an assertion in job j of task t, there is a corresponding execution π' of the sequential program $\mathcal{S}$ that violates localAssert$[t][j]$, and vice versa.*

V. LOCKS

We support two types of locks: preemption locks and priority ceiling locks. These locks are common in periodic programs because they are non-blocking (acquiring a lock always succeeds) and avoid common pitfalls such as priority inversion and deadlocks.

A periodic program has a single preemption lock pl. Acquiring pl disables the scheduler, preventing all priority-based preemptions. Releasing pl re-enables the scheduler. An example of such a mechanism is the `taskLock` / `taskUnlock` routines in VxWorks [15].

Priority ceiling locks, or *priority locks* for short, are based on dynamically raising the priority of the current job. Each priority lock lck is associated with a fixed priority LOCKPRIORITY(lck), which is given to a task if it acquires lck. It is illegal for a task with current priority p to acquire a priority lock lck such that LOCKPRIORITY(lck) is less than p. Releasing a lock restores the priority. Priority locks are used in the Highest Locker-Priority (HLP) protocol [16], where the priority of a resource r is as high as the priority of any task accessing r. This guarantees mutual exclusion (assuming there is only one CPU) while avoiding blocking, deadlocks, and priority inversion. Multiple priority locks must be acquired in increasing order of priority, but can be released in an arbitrary order. We now show our encoding of these locks.

Preemption locks. The preemption lock pl is modeled by introducing a Boolean variable $lock$ into $\hat{T}_t$. Specifically, $lock =$ TRUE iff the current task has acquired pl. The variable is FALSE initially and is reset to FALSE at the end of a job. Finally, calls to CS are conditioned by $lock =$ FALSE.

Priority locks. To model priority locks, we need to model the dynamic priority of each task. Let BASEPRIORITY(t) be a function that returns the base priority of task t. We introduce an array $priority$ such that for every round r, $priority[r]$ is the priority of the task currently executing in round r. The array is maintained by the function $\hat{T}_t$-WRAPPER (shown in Alg. 2) that wraps the body $\hat{T}_t$ of each task t. The wrapper function saves the current priority (line 2), ensures that it is below the base priority of the current task (line 3), raises the priority to the priority of the current task (line 4), executes the task body (line 5), and finally resets the priority (line 6). Note that the task body is only executed if the task has higher priority than the current dynamic one, and that the priority can be raised further inside the task body itself.

We model priority locks with two functions GETLOCK and RELEASELOCK shown in Alg. 2. The set of all locks held by a task is maintained in a set $lockSet$ that is local to each task. Information about locks of other tasks is propagated through priorities. Acquiring a lock (GETLOCK) raises the dynamic priority to the one of the lock, releasing a lock (RELEASELOCK) resets the priority to the base priority of the task or to the priority of the highest lock in the $lockSet$ in

Algorithm 1 A sequential program $\mathcal{S}$ for a periodic program $\mathcal{C}$ bounded by time $\mathcal{W}$. Notation: T is the set of tasks of $\mathcal{C}$; G is the set of global variables of $\mathcal{C}$; $\mathsf{J}(t)$ is the set of jobs of task t; $R = \sum_{t \in \mathsf{T}, j \in Job(t)} |\mathsf{J}(t)|$ is the number of rounds, last(t, j) is true iff j is the last job of $t \in \mathsf{T}$; for $t_i, t_j \in \mathsf{T}$, $t_i < t_j$ is true iff t_i is of lower priority than t_j; '$*$' is a non-deterministic value.

1: **var** $rnd, job, endRnd, start[\,][\,], end[\,][\,]$
2: $\forall g \in \mathsf{G} \cdot$ **var** $g[\,], v_g[\,]$
3: **var** *localAssert*$[\,][\,]$
4: **function** MAIN( )
5:     SCHEDULEJOBS()
6:     $\forall t \in \mathsf{T}, j \in \mathsf{J}(t) \cdot localAssert[t][j] :=$ TRUE
7:     $\forall g \in \mathsf{G} \cdot g[0] := i_g$
8:     $\forall g \in \mathsf{G} \forall r \in [1, R) \cdot g[r] := v_g[r]$
9:     **for** $t \in \mathsf{T}, job \in \mathsf{J}(t)$ **do**
10:         $rnd := start[t][job]$
11:         $endRnd := end[t][job]$
12:         $\hat{T}_t()$
13:         assume$(rnd = endRnd)$
14:     $\forall g \in \mathsf{G}, r \in [0, R-1) \cdot$ assume$(g[r] = v_g[r+1])$
15:     $\forall t \in \mathsf{T}, j \in \mathsf{J}(t) \cdot$ assert$(localAssert[t][j])$

16: **function** CS(Task t)
17:     **if** (*) **then return** FALSE
18:     $o := rnd$
19:     $rnd := *$
20:     assume$(o < rnd \leq endRnd)$
21:     $\forall t' \in \mathsf{T}, j' \in \mathsf{J}(t') \cdot$ assume($t < t' \Rightarrow (rnd \leq start[t'][j'] \vee rnd > end[t'][j'])$)
22:     **return** TRUE

23: **function** SCHEDULEJOBS( )

// Jobs are sequential

24: $\forall t \in \mathsf{T}, j \in \mathsf{J}(t) \cdot$ assume($0 \leq start[t][j] \leq end[t][j] \leq R \wedge (\neg \text{last}(t, j) \Rightarrow end[t][j] \leq start[t][j+1])$)

// Jobs are well-nested

25: $\forall t_1 \in \mathsf{T}, t_2 \in \mathsf{T}, j_1 \in \mathsf{J}(t_1), j_2 \in \mathsf{J}(t_2) \cdot$ assume($\big(t_1 < t_2 \wedge start[t_1][j_1] \leq end[t_2][j_2] \wedge start[t_2][j_2] \leq end[t_1][j_1]\big) \Rightarrow (start[t_1][j_1] \leq start[t_2][j_2] \leq end[t_2][j_2] < end[t_1][j_1])$)

// Jobs respect preemption bounds

26: $\forall t_1 \in \mathsf{T}, t_2 \in \mathsf{T}, j_1 \in \mathsf{J}(t_1), j_2 \in \mathsf{J}(t_2) \cdot$ assume($\big(t_1 < t_2 \wedge j_2 \geq PB_{t_1}^{t_2} \wedge start[t_1][j_1] \leq start[t_2][j_2] \leq end[t_2][j_2] < end[t_1][j_1]\big) \Rightarrow end[t_2][j_2 - PB_{t_1}^{t_2}] < start[t_1][j_1]$)

27: **function** $\hat{T}_t$( )
Same as T_t*, but*
  each statement 'st' is replaced with:
28:     CS(t) ; $st[g \leftarrow g[rnd]]$,
  and each 'assert(e)' is replaced with:
29:     *localAssert*$[t][job] := e$

case this set is not empty. Note that this allows for acquiring multiple locks and releasing them in an arbitrary order. We check that the priority of the locks is assigned correctly (i.e., acquiring a lock must never lower the dynamic priority) by adding an assertion that checks this in line 8 of GETLOCK.

Correctness. We need to show that CS and SCHEDULEJOBS in Alg. 1, which are based on the base priorities, are still correct when priorities can change due to priority locks.

First, we need to show that the schedules permitted by SCHEDULEJOBS cover the legitimate schedules in the presence of priority locks. This is indeed the case because priority-locks cannot lead to *priority inversion* – a situation in which a job with a low base priority preempts a job with a high base priority. We will demonstrate this using jobs j_1 and j_2 with base priorities 1 and 2, respectively. j_1 cannot preempt j_2 even if it raises its own priority to 3, simply because j_2 is not running when j_1 acquires the lock. It can, therefore, only delay the start time of j_2, not preempt it. Thus, it is not necessary to explore schedules in which a low-base-priority job preempts a high-base-priority job.

Next, consider CS. Line 21 in CS guarantees that j_1 does not resume control in a round in which j_2 is still active. If j_1 raises its priority then j_2 cannot preempt it, and therefore this constraint blocks a computations in which j_1 resumes when j_2 was initially scheduled to run. Such a computation is illegal, however, because it corresponds to a preemption of a high-priority job j_1 by a lower-priority job j_2, it is accordingly blocked by the assume statement on line 3 of $\hat{T}_t$-WRAPPER.

Finally, since j_1 can raise its priority, it seems that we also need the opposite constraint (i.e., that j_2 does not resume control in a round in which j_1 is still active). However, there is no need for this constraint because SCHEDULEJOBS does not allow a schedule in which j_1 preempts j_2.

VI. PARTIAL-ORDER REDUCTION

Computations of a concurrent system have a natural partitioning: two computations are in the same class iff they reach the same observable states. Thus, for verification, it suffices to examine only one representative from each class. This is known as *Partial-Order Reduction* (POR) [17], [18].

POR is used widely and effectively in explicit-state Model Checking (e.g., [19]). Recently, it has been shown to be

Algorithm 2 Priority locks.

```
1: function T̂_t-WRAPPER( )
2:     sp := priority[rnd]
3:     assume(sp ≤ BASEPRIORITY(t)))
4:     priority[rnd] := BASEPRIORITY(t)
5:     T̂_t( )
6:     priority[rnd] := sp
7: function GETLOCK(Lock lck)
8:     assert(LOCKPRIORITY(lck) ≥ priority[rnd])
9:     lockSet := lockSet ∪ {lck}
10:    priority[rnd] := LOCKPRIORITY(lck)
11: function RELEASELOCK(Task t, Lock lck)
12:    lockSet := lockSet \ {lck}
13:    if lockSet = ∅ then
14:        priority[rnd] := BASEPRIORITY(t)
15:    else
16:        priority[rnd]:=priority of highest lock in lockSet
```

effective for symbolic methods as well [20]. In symbolic verification, POR translates into additional constraints to the underlying verification engine (in our case, CBMC and SAT). This does not always makes the solver faster. We report on our experience with this reduction in Sec. VII.

We propose two approaches for POR: syntactic and semantic. In syntactic POR, we first partition global variables into two groups – *task local* (accessed by a single task) and *shared* (accessed by multiple tasks). Intuitively, *task local* variables are local to a task, but preserved across jobs (e.g., *static* variables in C). Second, we allow preemptions only before statements that access a shared variable. Note that this also reduces the number of variables in the sequentialization since global variables that are not shared do not need to be guessed across rounds. We do not describe this reduction in more details since it is well-known and used by many other sequentialization approaches (e.g. [9]).

The idea behind semantic POR is to allow, for each shared variable g, a task to be preempted: (i) before a store $g := l$ only by a computation that loads or stores g, and (ii) before a load $l := g$ only by a computation that stores g. Intuitively, if a preempting computation does not affect the access to g then it is scheduled after the access, while preserving behavior.

The semantic POR is implemented by adding Boolean global read/write flags W_g and R_g for each shared g to indicate whether g was stored or loaded, respectively. These flags are treated as regular shared variables (i.e., guessed at the beginning of each round and checked at the end of the program). Each task body is changed as shown in Alg. 3. Only the case for a store $g := l$ is shown; the load $l := g$ is similar and is illustrated later with an example.

When a preemption happens before $g := l$, the read/write flags of g are reset (line 4) in the round in which the preemption happens. Then, at least one of the flags is assumed to become true in the round in which the task resumes. Thus, any computation in which the current task is preempted but g is not

Algorithm 3 A fragment of $\hat{T}_t$ from Alg. 1 with POR. Only the case of a store to shared variable g is shown.

```
1: function T̂_t( )
   Same as T_t, but
     each statement 'g := l' is replaced with:
2:     oldRnd := rnd
3:     if CS(t) then
4:         W_g[oldRnd] := R_g[oldRnd] := FALSE
5:         assume(W_g[rnd] ∨ R_g[rnd])
6:     W_g[rnd] := TRUE
7:     g[rnd] := l
```

accessed, is blocked. Note that since in the sequential program we can access any round at any time, the resetting of the read/write flags (line 4) follows the preemption sequentially, but precedes it in the execution order. Finally, line 6 sets W_g to true to indicate that g was stored.

Example 3 *Under semantic POR, the two assignments*

1: $x := g \,;\, g := y$

in T_t*, where* x, y *are local and* g *is shared, become the sequence in* $\hat{T}_t$ *that appears in Fig. 2.*

```
1: oldRnd := rnd
2: if CS(t) then
3:     W_g[oldRnd] := R_g[oldRnd] := FALSE
4:     assume(W_g[rnd])
5: R_g[rnd] := TRUE
6: x := g[rnd]
7: oldRnd := rnd
8: if CS(t) then
9:     W_g[oldRnd] := R_g[oldRnd] := FALSE
10:    assume(W_g[rnd] ∨ R_g[rnd])
11: W_g[rnd] := TRUE
12: g[rnd] := y
```

Fig. 2. An encoding for Example 3.

Let $\mathcal{S}^{po}$ denote the sequentialization with POR. The following theorem shows that it is semantics preserving.

Theorem 3 *Let* $\mathcal{C}$*,* $\mathcal{W}$ *and* $\mathcal{S}$ *be as in Theorem 2 and* $\mathcal{S}^{po}$ *the corresponding POR. Then, for every execution* π *of* $\mathcal{S}$ *that violates a local assertion localAssert*$[t][j]$ *of task* t *and job* j *there is a corresponding execution* π' *of* $\mathcal{S}^{po}$ *that violates localAssert*$[t][j]$*, and vice versa.*

VII. CASE STUDIES

We implemented our approach in a tool called REK. REK is built on top of CIL [21]. It takes as input C programs annotated with entry points of each task, their periods, worst case execution times, and the time bound $\mathcal{W}$. The output is a sequential C program $\mathcal{S}$ that is then verified by CBMC [22].

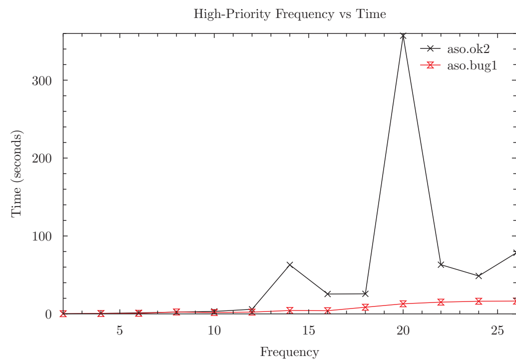


Fig. 3. Analysis time versus frequency of the highest-period time in 'aso'.

To evaluate our approach, we have used REK to verify several periodic programs. In the rest of this section, we report on this experience. The tool and the case studies are available at: http://www.andrew.cmu.edu/user/arieg/Rek.

Robot controller. The NXTway-GS controller, nxt for short, runs on nxtOSEK [2] – a real-time operating system ported to the LEGO MINDSTORM platform. nxtOSEK supports programs written in C with periodic tasks and priority ceiling locks. It is the target for Embedded Coder Robot NXT – a Model-Based Design environment for using Simulink models with LEGO robots.

The basic version of the controller has 3 periodic tasks: a *balancer*, with period of 4ms, that keeps the robot upright and monitors the bluetooth link for user commands, an *obstacle*, with period 50ms, that monitors a sonar sensor for obstacles, and a 100ms background task that prints debug information on an LCD screen.

We verified several versions of this controller. All of the properties verified involved the high-frequency balancer task. The balancer goes through 3 modes of execution: INIT, CALIBRATE, and CONTROL. In INIT mode all variables are initialized, and in CALIBRATE a gyroscope is calibrated. After that, balancer goes to CONTROL mode in which it iteratively reads the bluetooth link, reads the gyroscope, and sends commands to the two motors on the robot's wheels.

The results are shown in the top part of Table I. We have used $\mathcal{W} = 100$ms, which is the minimum time needed for all tasks to execute at least once. We did not enable semantic POR since it was irrelevant in this case (all shared variables were accessed by all tasks in all paths).

Experiments nxt.ok1 (nxt.bug1) check that the balancer is in a correct (respectively, incorrect) mode at the end of the time bound. Experiment nxt.ok2 checks that the balancer is always in one of its defined modes. Experiment nxt.bug3 checks that whenever *balancer* detects an obstacle, the *balancer* responds by moving the robot. We found that since the shared variables are not protected by a lock there is a race condition that causes the *balancer* to miss a change in the state of *obstacle* for one period. Experiment nxt.ok3 is the version of the controller where the race condition has been resolved using locks.

For the second part of the robot case study, we modified the original design to separate handling of each sensor by a separate task. Our design, called 'aso' has 3 tasks: *balancer*, *observer*, and *bluetooth*. The first two are the same as before, and the *bluetooth* is responsible for the bluetooth communication. We wanted a design in which the *balancer* task is lock-free, which was challenging. During our design, we unintentionally introduced subtle concurrency errors which were detected by REK.

TABLE I
Experimental results. OL and SL = # lines of code in the original C program and the generated sequentialization $\mathcal{S}$, respectively; GL = size of the GOTO program produced by CBMC; Var and Clause = # variables and clauses in the SAT instance, respectively; S = verification result – 'Y' for SAFE and 'N' for UNSAFE; Time = verification time in sec.

Name	Program Size		SAT Size			S	Time (s)
	OL	SL	GL	Var	Clause		
nxt.ok1	377	2,265	7,848	136K	426K	Y	22.16
nxt.bug1	378	2,265	7,848	136K	426K	N	9.95
nxt.ok2	368	2,322	8,572	141K	439K	Y	13.92
nxt.bug2	385	2,497	10,921	144K	451K	N	17.48
nxt.ok3	385	2,497	10,905	144K	449K	Y	18.32
aso.bug1	401	2680	13106	178K	572K	N	16.32
aso.bug2	400	2,682	13060	176K	566K	N	15.01
aso.ok1	398	2,684	13,026	175K	560K	Y	66.43
aso.bug3	426	3,263	19,211	373K	1,187K	N	59.66
aso.bug4	424	3,250	18,503	347K	1,099K	N	31.51
aso.ok2	421	3,251	18,589	348K	1,101K	Y	328.32
RW1	190	3,428	5,860	42K	125K	Y	20.74
RW1-PO	190	5,021	7,626	45K	134K	Y	14.71
RW2	239	4,814	8,121	52K	152K	Y	165.89
RW2-PO	239	7,356	10,388	56K	164K	Y	162.20
RW3	285	7,338	21,163	139K	419K	Y	436.86
RW3-PO	285	12,002	26,283	153K	467K	Y	199.13
RW4	244	7,255	19,745	117K	350K	Y	321.25
RW4-PO	244	12,272	24,261	130K	392K	Y	59.66
RW5	188	3,198	5,208	41K	119K	Y	47.83
RW5-PO	188	4,791	7,138	45K	131K	Y	20.35
RW6	257	5,231	7,634	54K	157K	Y	165.33
RW6-PO	257	8,235	10,119	59K	173K	Y	157.43

The results of these experiments are shown in the second part of Table I. We checked consistency of communication between the tasks. The experiments are: aso.bug1 and aso.bug2 – initial versions with inadequate locking leading to race conditions. aso.ok1 is a correct design with preemption locks. aso.bug3 is our first attempt at a lock-free implementation that was fundamentally flawed and had to be abandoned. aso.bug4 and aso.ok2 are a buggy and a correct version of the final design in which *obstacle* and *bluetooth* synchronize via priority locks and *balancer* is lock-free.

During the case study, we found it very convenient to increase the period of the highest priority task (thus decreasing its frequency). In many cases, this dramatically reduced verification time, while allowing us to draw meaningful conclusions from the counterexamples. Of course, this approach is not sound in general. Fig. 3 shows the relationship between the analysis time and the frequency of *balancer* for a correct and an incorrect version of the controller. In case the design is buggy, the time increased monotonically with the frequency. However, for a safe design, the time behaves erratically: e.g., increasing the frequency from 20 to 26 made it easier to verify. Such erratic behavior is common with SAT.

Reader-Writer. Reader-Writer (RW) is a common communication pattern in concurrent programs. We implemented three

lock-free flavors of a RW protocol (RW1, RW3, and RW5), and their counterparts with locks (RW2, RW4, and RW6). We checked consistency of communication between the tasks. Each protocol was analyzed with 3 to 6 tasks (depending on the protocol), with $\mathcal{W}$ such that every task executes once, and with an increasing number of shared variables. The results are shown in Table I. For each protocol, we only report on the hardest instance solved in under 10 mins. In these examples, semantic POR yields significant reduction in verification time. The results with POR are shown in Table I in rows named "PO". Note that in all cases, the number of variables and clauses with POR is larger, yet the verification time is smaller.

VIII. Related work

There is a large body of work in verification of logical properties of both sequential and concurrent programs (see [23] for a recent survey). However, these techniques abstract away time completely, by assuming a non-deterministic scheduler model. In contrast, we use a priority-sensitive scheduler model, and abstract time partially via our job-bounded abstraction.

A number of projects [5], [6] verify timed properties of systems using discrete-time [24] or real-time [7] semantics. They abstract away data- and control-flow, and verify models only. We focus on the verification of implementations of periodic programs, and do not abstract data- and control-flow.

Recently, Kidd et al. [4] showed a number of decidability results for reachability in finite-state periodic programs with recursion and locks. They apply sequentialization as well. The key idea is to share a single stack between all tasks and model preemptions by function calls. However, they not report on an implementation. In contrast, we focus on a practical solution to a bounded version of this problem.

In the context of concurrent software verification, several flavors of sequentialization have been proposed and evaluated (e.g., [9], [10], [11], [12]). Our procedure is closest to the LR [9] style. However, it differs from LR significantly, as discussed earlier (see Sec. IV-A), and provides a crucial advantage over LR for periodic programs, as discussed next.

In both cases, ours and LR, the number of variables is proportional to the number of rounds, R. However, LR allows more computations since it does not enforce priority constraints. In addition to yielding fewer false warnings, our approach guarantees better coverage for the same number of variables, as shown below:

Take two programs: (i) an N-task periodic program $\mathcal{C}$ with a time bound that permits exactly one job per task; (ii) the analogue of $\mathcal{C}$, called $\mathcal{C}'$, in a non-real-time setting, i.e., N threads scheduled non-deterministically, each executing one task of $\mathcal{C}$. Consider the value of R required to cover all reachable states, in our encoding $\mathcal{S}$ vs. the LR encoding of $\mathcal{C}'$. For LR, R is the number of possible context switches, which by itself is proportional to the number of statements over shared variables in $\mathcal{C}'$. This value of R is needed to explore a pathological path in $\mathcal{C}$ where, in each round, a single thread is executed and the threads are picked in reverse-round-robin order. In contrast, our approach only needs $R = N$, a much smaller value in most practical cases. In fact, the pathological path above is illegal, since scheduling tasks in reverse-round-robin order violates priority constraints.

IX. Conclusion

Periodic programs, i.e., periodic RTES with rate-monotonic scheduling, are an important sub-class of embedded real-time software. In this paper, we address the time-bounded verification of safety properties of periodic program implementations. We present a solution involving two steps – convert the target periodic program to a non-deterministic sequential program, and then verify it with an off-the-shelf verification tool. Our approach is sound, preserves both data- and control-flow, and abstracts the effect of time via preemption-bounds. Some of our techniques are applicable to other types of systems. Specifically, note that we only used the assumption that the verified system follows RMS in order to compute preemption bounds. Other periodic RTES can be modeled with our method by either supplying these bounds as part of the input, or by removing line 26 in Alg. 1 (this may hinder completeness, however). In addition, our partial order reduction is not restricted to periodic RTES, and is applicable when analyzing general multi-threaded programs.

We have implemented our approach in a tool, and used it to identify subtle concurrency errors in a robot controller. We believe that our work opens up several avenues for future work in real-time software verification, notably unbounded verification of periodic programs and the use of automated abstraction refinement techniques.

References

[1] C. L. Liu and J. W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment," *Journal of the ACM (JACM)*, vol. 20, no. 1, pp. 46–61, January 1973.

[2] "nxtOSEK/JSP Open Source Platform for LEGO MINDSTORMS NXT," http://lejos-osek.sf.net.

[3] D. C. Locke, D. R. Vogel, L. Lucas, and J. B. Goodenough, "Generic Avionics Software Specification," Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, Technical report CMU/SEI-90-TR-8-ESD-TR-90-209, December 1990.

[4] N. Kidd, S. Jagannathan, and J. Vitek, "One Stack to Run Them All - Reducing Concurrent Analysis to Sequential Analysis under Priority Scheduling," in *Proceedings of the 17th International SPIN Workshop on Model Checking of Software (SPIN '10)*, Enschede, The Netherlands, September 2010, pp. 245–261.

[5] K. G. Larsen, P. Pettersson, and W. Yi, "UPPAAL in a Nutshell," *International Journal on Software Tools for Technology Transfer (STTT)*, vol. 1, no. 1-2, pp. 134–152, December 1997.

[6] V. A. Braberman and M. Felder, "Verification of Real-Time Designs: Combining Scheduling Theory with Automatic Formal Verification," in *Proceedings of the 7th ACM SIGSOFT Symposium on Foundations of Software Engineering (FSE '99)*, ser. Lecture Notes in Computer Science, vol. 1687. Toulouse, France, September, 1999. New York, NY: Springer-Verlag, September 1999, pp. 494–510.

[7] R. Alur and D. L. Dill, "A Theory of Timed Automata," *Theoretical Computer Science (TCS)*, vol. 126, no. 2, pp. 183–235, April 1994.

[8] S. Qadeer and D. Wu, "KISS: Keep It Simple and Sequential," in *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation (PLDI '04)*. Washington, DC, USA, June 9-11, 2004. New York, NY: Association for Computing Machinery, June 2004, pp. 14–24.

[9] A. Lal and T. W. Reps, "Reducing Concurrent Analysis Under a Context Bound to Sequential Analysis," in *Proceedings of the 20th International Conference on Computer Aided Verification (CAV '08)*, ser. Lecture Notes in Computer Science, A. Gupta and S. Malik, Eds., vol. 5123. Princeton, NJ, USA, July 7-14, 2008. New York, NY: Springer-Verlag, July 2008, pp. 37–51.

[10] S. L. Torre, P. Madhusudan, and G. Parlato, "Reducing Context-Bounded Concurrent Reachability to Sequential Reachability," in *Proceedings of the 21st International Conference on Computer Aided Verification (CAV '09)*, ser. Lecture Notes in Computer Science, A. Bouajjani and O. Maler, Eds., vol. 5643. Grenoble, France, June 26 - July 2, 2009. New York, NY: Springer-Verlag, July 2009, pp. 477–492.

[11] N. Ghafari, A. J. Hu, and Z. Rakamaric, "Context-Bounded Translations for Concurrent Software: An Empirical Evaluation," in *Proceedings of the 17th International SPIN Workshop on Model Checking of Software (SPIN '10)*, Enschede, The Netherlands, September 2010, pp. 227–244.

[12] M. Emmi, S. Qadeer, and Z. Rakamaric, "Delay-Bounded Scheduling," in *Popl11*, T. Ball and M. Sagiv, Eds. Austin, TX, USA, January 26-28, 2011. New York, NY: Association for Computing Machinery, January 2011, pp. 411–422.

[13] A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, and Y. Zue, *Bounded Model Checking*, ser. Advances in computers. Academic Press, 2003, vol. 58.

[14] N. Audsley, A. Burns, K. Tindell, and A. Wellings, "Applying New Scheduling Theory to Static Priority Preemptive Scheduling," *Software Engineering Journal (SEJ)*, vol. 8, no. 5, pp. 284–292, May 1993.

[15] "VxWorks Programmer's Guide."

[16] R. Mall, *Real-Time Systems: Theory and Practice*. Prentice Hall, 2009.

[17] P. Godefroid, *Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State-Explosion Problem*, ser. Lecture Notes in Computer Science. Springer-Verlag, 1996, vol. 1032.

[18] D. Peled, "All from One, One for All: on Model Checking Using Representatives," in *Proceedings of the 5th International Conference on Computer Aided Verification (CAV '93)*, ser. Lecture Notes in Computer Science, C. Courcoubetis, Ed., vol. 697. Elounda, Greece, June 28 - July 1, 1993. New York, NY: Springer-Verlag, June 1993, pp. 409–423.

[19] D. Bosnacki and G. J. Holzmann, "Improving Spin's Partial-Order Reduction for Breadth-First Search," in *Proceedings of the 12th International SPIN Workshop on Model Checking of Software (SPIN '05)*, ser. Lecture Notes in Computer Science, P. Godefroid, Ed., vol. 3639. San Francisco, CA, August 22–24, 2005. New York, NY: Springer-Verlag, August 2005, pp. 91–105.

[20] V. Kahlon, C. Wang, and A. Gupta, "Monotonic Partial Order Reduction: An Optimal Symbolic Partial Order Reduction Technique," in *Proceedings of the 21st International Conference on Computer Aided Verification (CAV '09)*, ser. Lecture Notes in Computer Science, A. Bouajjani and O. Maler, Eds., vol. 5643. Grenoble, France, June 26 - July 2, 2009. New York, NY: Springer-Verlag, July 2009, pp. 398–413.

[21] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer, "CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs," in *Proceedings of the 11th International Conference on Compiler Construction (CC '02)*, ser. Lecture Notes in Computer Science, R. N. Horspool, Ed., vol. 2304. Grenoble, France, April 8–12, 2002. New York, NY: Springer-Verlag, April 2002, pp. 213–228.

[22] E. Clarke, D. Kroening, and F. Lerda, "A Tool for Checking ANSI-C Programs," in *Proceedings of the 10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '04)*, ser. Lecture Notes in Computer Science, K. Jensen and A. Podelski, Eds., vol. 2988. Barcelona, Spain, March 29–April 2, 2004. New York, NY: Springer-Verlag, March–April 2004, pp. 168–176.

[23] V. D'Silva, D. Kroening, and G. Weissenbacher, "A Survey of Automated Techniques for Formal Software Verification," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 27, no. 7, pp. 1165–1178, July 2008.

[24] F. Laroussinie, N. Markey, and P. Schnoebelen, "Efficient Timed Model Checking for Discrete-Time Systems," *Theoretical Computer Science (TCS)*, vol. 353, no. 1-3, pp. 249–271, March 2006.

Timing Analysis of Interrupt-Driven Programs under Context Bounds

Jonathan Kotker Dorsa Sadigh Sanjit A. Seshia
EECS Department, UC Berkeley
jamhoot@eecs.berkeley.edu dsadigh@berkeley.edu sseshia@eecs.berkeley.edu

Abstract— **Timing analysis is a key step in the design of dependable real-time embedded systems, yet existing analysis tools do not work well for interrupt-driven code, which is commonly used in embedded systems. In this paper, we present a technique for timing analysis of interrupt-driven software. We show that for systems that use priority pre-emptive scheduling, if there is a finite arrival time between interrupts, one can use bounds on the number of context switches to perform timing analysis. Our work builds upon prior work on timing analysis for sequential programs. We present empirical evidence to show that we can accurately predict the execution time along any path of an interrupt-driven program on a standard micro-controller.**

I. Introduction

Timing is central to the correctness of real-time embedded systems. Timing properties are determined by the behavior of both the control software and the platform the software executes on. The verification of such properties is made difficult by their heavy dependence on characteristics of the platform, including details of the processor and memory hierarchy. Even so, over the past two decades, there has been steady progress in the field of timing analysis for *purely sequential software* (see [1], [2]). Most of the progress has been on the classic problem of estimating the worst-case execution time (WCET) of a terminating software task. Such an estimate can be used as conservative checks on real-time constraints as well as for use in scheduling algorithms. While determining a bound on the WCET has many uses, it is not the only problem of interest. As tools typically overestimate the WCET, when the WCET exceeds the timing bound, one cannot be sure whether the program can really miss its deadline. One would also like to find a test case demonstrating that the program can miss its deadline. Recent methods [2] have sought to address this problem for sequential programs.

In practice, though, embedded software is not purely sequential. In many real-world applications, the control software comprises several tasks that execute concurrently. Programming with interrupts is an extremely common form of concurrency that the control software uses to obtain sensor data from its physical environment. Apart from a main function, the control software has one or more interrupt-service routines (ISRs). An ISR is invoked when its corresponding interrupt is raised, e.g., when a new sensor sample is available. For such an interrupt-driven program, there is a need to ensure that the task meets its deadline *even* in the presence of interrupts. However, the state-of-the-art of timing analysis for interrupt-driven software is extremely poor. For instance, in NASA's recent report on "unintended acceleration" in certain Toyota automobiles [3], several limitations of state-of-the-art timing analysis tools are noted, including the lack of support for interrupts.

The reason for this lack of progress on timing analysis of interrupt-driven software is not hard to guess. It is the exponential explosion in the number of interleavings of various software tasks (such as the main function and the ISRs for various interrupts). This path explosion especially impacts timing analysis, since timing is a *highly path-sensitive property* — the execution time of a basic block of a program can depend a great deal on the path it lies on. This is in contrast with verifying invariants (such as assertion violations), where one is concerned with checking if a particular "error" location is reachable without regard to how it is reached. Moreover, interrupts also impact processor operation, e.g., by flushing the CPU pipeline. Most current state-of-the-art WCET analysis techniques are based on using abstract interpretation to create an abstract timing model of the processor [1]. Even for sequential programs, the creation of an abstract timing model is an extremely tedious manual process. With interrupt-driven programs, the process is even harder due to the need to model the impact of interrupts on hardware and also due to the severe imprecision abstract interpretation suffers due to the large number of joins required on reconvergent interleaved paths.

Even with these challenges, good embedded software design often follows rules that can ease the problem. First, in many systems, there is a strict priority assignment between various tasks in the system, and the task scheduler follows *priority pre-emptive scheduling* — a task runs to completion unless a higher-priority task preempts it. Second, there is usually a *finite lower bound on inter-arrival time* between interrupts, dictated, for example, by the rate at which a sensor generates samples. This inter-arrival time bound imposes a restriction on how frequently a task can be interrupted. Finally, careful coding practices involve the use of "atomic sections" by disabling interrupts in selected parts of the program.

In this paper, we present a novel approach to the problem of timing analysis of interrupt-driven software that takes advantage of the above design rules. In particular, we make the following contributions:

- We show how a lower bound on inter-arrival time of

interrupts in turn imposes an upper bound on the number of "context switches" between the interrupted task and the ISR. This enables the use of *context-bounded* analysis, similar to the work pioneered by Qadeer et al. [4], [5]. The use of atomic sections and priority pre-emptive scheduling further reduces the number of interleavings that need to be considered.

- Even with these reductions, the number of interleaved paths can still be exponential in the context bound, and very large in practice. Obtaining measurements for a large number of paths can be very tedious and expensive. We show that we can leverage work for sequential program timing analysis to mitigate this problem. In particular, we adopt the idea of using the execution time of *basis paths* to predict the times of other program paths [2], [6]. The number of basis paths is guaranteed to be polynomial in the size of the program.
- We demonstrate our approach with experiments on a real embedded platform, the Luminary Micro LM3S8962 board with an ARM Cortex M-3 processor [7], interfaced to sensors on the iRobot Create mobile robot [8]. We show that we can accurately predict not only the WCET of various programs, but also the execution times of arbitrary program paths. When a particular deadline is violated, our approach can generate a test case exhibiting how this occurs.

To our knowledge, our approach is the first timing analysis technique for interrupt-driven software that can not only generate worst-case execution time estimates, but also can generate accurate predictions for the actual timing (not just bounds) along arbitrary program paths. Importantly, our approach is extremely *portable*: in contrast with traditional WCET techniques that rely on tedious manual modeling of the platform, our approach only requires automated systematic generation of measurements on the target platform, from which we make accurate predictions of program timing on paths that have not been tested.

The rest of the paper is organized as follows. We introduce the problem, along with basic terminology, definitions, and related work in Section II. The core of our approach is presented in Section III. Section IV presents an experimental evaluation. We conclude in Section V with directions for future work.

II. Background and Related Work

We define terminology and the problems considered in this paper in Section II-A, and compare with related work in Section II-B.

A. Problem Definition

Real-time embedded programs are reactive programs that execute repeatedly within a top-level "$\mathbf{while}(1)$ loop". We are concerned with the tasks invoked within this loop, which are required to be terminating programs. For this paper, we are concerned with programs structured as a single "main" task along with one or more interrupt-service routines (ISRs) which are written typically as other tasks (think of C functions). Typically, the boot-up sequence of the system involves registering the ISRs as handlers for the various interrupts that the system must respond to.

We present a simple imperative language to model these interrupt-driven programs. Figure 1 shows the program syntax. An interrupt-driven program P is composed of N tasks, each of which is a sequential program. Each task T has an associated *priority level* p, which is a positive integer. We will assume that each task has a unique priority level, and a larger priority level indicates higher priority. A task of priority p_i can interrupt a task with priority p_j if $p_i > p_j$. Once a higher-priority task has interrupted a lower-priority task, it runs to completion unless it is interrupted by a task with still higher priority. This scheduling scheme is known as *priority pre-emptive scheduling*, and is widely implemented in embedded platforms.

$$\begin{array}{rl} S ::= & v := e \mid \mathbf{skip} \mid \mathbf{if}\, e\, \mathbf{then}\, S_1\, \mathbf{else}\, S_2 \\ & \mid S_1; S_2 \mid \mathbf{while}\, e\, \mathbf{do}\, \langle B \rangle\, S \\ & \mid \mathbf{atomic}\, \{\, S\, \} \mid \mathbf{timed_while}\, \tau\, \mathbf{do}\, S \\ T ::= & \langle S, p \rangle \\ P ::= & T_1 \parallel T_2 \parallel \ldots \parallel T_N \end{array}$$

Fig. 1: **Syntax for Interrupt-Driven Programs.** v and e denote an l-value and an expression in any standard imperative programming language such as C. The $\mathbf{skip}$ statement is a no-op. Every $\mathbf{while}$ loop has an associated loop bound B. T denotes a sequential task with an associated priority p, and P denotes a program composed of n tasks.

The code for a task T follows standard syntax of an imperative language such as C, with a few small exceptions. Assignments have the form $v := e$ where v is an l-value and e is any expression in C including procedure calls. For simplicity, we disallow recursive procedure calls; in any case, it is highly desirable in real-time embedded software to impose finite bounds on recursion depth. The syntax of Fig. 1 includes $\mathbf{if}$ statements as a way of modeling all conditional constructs, including $\mathbf{switch}$ statements. We will use $\mathbf{switch}$ statements where required for brevity. The main exceptions to standard program notation are with regard to $\mathbf{while}$ loops and the presence of a special $\mathbf{atomic}$ program construct, as described below:

1) Each $\mathbf{while}$ loop must have a statically-known upper bound B on the number of loop iterations. We assume each loop is annotated with such a bound. We will use the standard $\mathbf{for}$-loop notation where it is more convenient to do so.
2) There is a special timed-while loop construct $\mathbf{timed_while}$ which has an associated deadline τ. This loop runs for exactly τ cycles and terminates thereafter. This construct models timed loops common in embedded code that waits for an event for a specific amount of time, with termination guaranteed by the expiration of a hardware timer.

3) We include a special **atomic** construct which models a piece of code S that runs uninterrupted. This construct is typically implemented by disabling interrupts before running S and re-enabling interrupts after S completes execution. Using such atomic code sections within sequential code is considered good programming practice to ensure that certain operations are completed atomically irrespective of the presence of interrupts.

We assume that interrupts cannot occur infinitely often during the execution of P and that there is a finite lower bound on the inter-arrival times of interrupts. We believe this is a reasonable assumption that holds in practice for real-time embedded systems.

Given the above model, we are concerned with answering the following three types of timing analysis questions. For each question, the inputs include an interrupt-driven program P and the platform it executes on. The platform is the complete hardware and software environment of P, including the compiler, processor, and memory architecture.

- **P1: Threshold Property Checking.**
  Does P always complete within τ cycles? If not, provide a test case (counterexample).
- **P2: Worst-Case Execution Time Prediction.**
  Predict the worst-case execution time of P and generate corresponding test case.
- **P3: Predicting Timing along All Paths.**
  Predict the execution time (not a bound) of program P along *all* paths, where a path involves following a specific interleaving of tasks and particular paths within tasks.

One can observe that problem **P3** is more general than **P1** and **P2** in that if one can solve **P3**, one can answer questions **P1** and **P2** as well. Therefore, in Section III, we focus on addressing problem **P3**. We demonstrate our results for all three problems in Section IV.

Our technique relies on the notion of context-bounded analysis [4], [5]. Following the definition introduced by Qadeer and Rehof [5], a *context* is an uninterrupted sequence of actions by a single task. A bound of K on the number of contexts implies a bound of $K - 1$ on the total number of context switches between tasks.

B. Related Work

As noted above, Qadeer et al. introduced the idea of verifying multithreaded software by using context bounds [4], [5]. However, their work focuses on traditional propositional temporal properties. Our paper is the first to apply the idea of context-bounded analysis to the problem of timing analysis.

Brylow and Palsberg [9] consider the topic of deadline analysis in interrupt-driven programs — checking whether every interrupt is serviced before its deadline. They assume that worst-case execution times are already determined for certain program fragments and use this in their analysis. In contrast, we are concerned with predicting execution time properties of the entire interrupt-driven program, and can generate the WCET estimates required in their analysis.

The WCET analysis community has mainly focused on analyzing sequential programs without interrupts. A recent industrial experience report [10] states the difficulty of estimating the WCET of an interrupt service routine in welding control software, writing: "It was difficult to detect if other interrupts had disturbed the measurement of the current interrupt." While there has been work on testing non-timing "functional" properties of interrupt-driven software (e.g., [11]), there is no systematic work for verifying timing properties of such programs. The work on *schedulability analysis* — in which one analyzes if a task can meet its deadline in spite of preemption by other tasks — is related; however, that work treats tasks as atomic objects (see, e.g., [12]), whereas we perform a detailed program analysis of tasks, considering interleaved program paths and interaction of tasks through shared variables. To the best of our knowledge, our technique is the first systematic approach for performing WCET analysis (and other timing analysis) on interrupt-driven programs.

Kidd et al. [13] present an approach to transform a concurrent real-time program with priority pre-emptive scheduling to a sequential program so that any state reachable in the original concurrent program can be reached by performing reachability analysis of the sequential program. This is close to our work in that we could conceivably use their reduction; however, additional assumptions will be needed on inter-arrival time of interrupts, as in our paper. Other methods for more compactly transforming context-bounded concurrent programs to sequential programs are also available [14], [15]; however, with priority pre-emptive scheduling the benefit of these transformations is somewhat limited. Our contribution is to show how the ideas of context-bounding and basis paths can be combined to perform accurate timing analysis of interrupt-driven software.

III. Approach

Consider an interrupt-driven program $P = T_0 \| T_1 \| \ldots \| T_N$, where i denotes the priority level of T_i. We will consider T_0 to be the main function, and all other tasks to be ISRs. Thus, there are n interrupts, which we denote by $\iota_1, \iota_2, \ldots, \iota_n$. As part of the problem description, we are also supplied a lower bound α on the time between interrupts – the "inter-arrival" time of interrupts. Finally, the platform of interest is also specified.

The high-level idea of our approach is to reduce the problem of timing analysis of interrupt-driven programs to timing analysis of sequential programs, by deriving a context bound that is adequate to explore all interleaved paths of P. The approach operates in the following five steps.

1) Use the finite inter-arrival times of interrupts to derive a context bound CB for P that is adequate to explore all interleaved paths of P.
2) Use CB to generate a single sequential program P_{seq} that is path-equivalent to P for the context bound CB.

3) Invoke GameTime [2], a timing analysis technique for sequential programs, on P_{seq}. The key idea in GameTime is to extract a subset of program paths that forms a basis (in the standard linear algebra sense) for the set of all program paths. We term these paths as *basis paths*. GameTime also uses an SMT solver [16] to generate test cases that drive program execution down these basis paths. The key difference with previous applications of GameTime is that the generated basis paths are *interleaved* paths in P, involving context switches between the main function and ISRs.
4) Execute test cases for basis paths on the platform with an interrupt-generation test harness, and measure the execution time of basis paths.
5) Use measured times to infer a platform model, using the GameTime learning algorithm. The inferred model is used to predict execution times of other paths, and answer Problems **P1**, **P2**, and **P3**.

For ease of presentation, we will describe the process somewhat out of order. We will start first with the third item, our technique for timing analysis of sequential programs, then describe the remaining steps.

A. Timing Analysis of Sequential Programs using Basis Paths

While there are several tools for estimating worst-case execution time of sequential programs [1], the only tool we are aware of which can address Problems **P1** and **P3** is GameTime [2], [6]. Our approach therefore builds upon GameTime.

In this section, we give a brief overview of the relevant aspects of GameTime. Most important is the notion of basis paths which helps us deal with the large number of interleaved program paths.

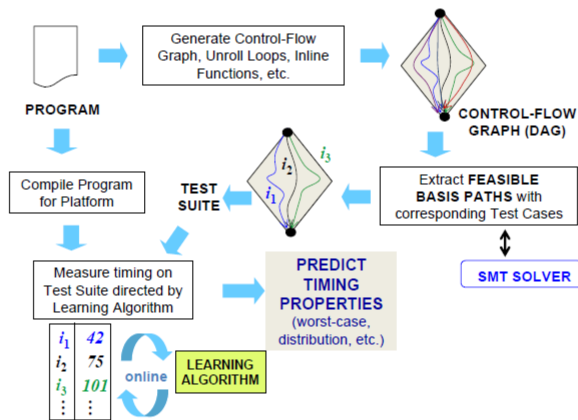


Fig. 2: GameTime overview [17]

Figure 2 depicts the operation of GameTime. The process begins (see top-left corner) with the generation of the control-flow graph (CFG) of the program, in which all loops have been unrolled to the maximum loop bound, and all function calls have been inlined into the top-level function. The CFG is assumed to have a single source node (entry point) and a single sink node (exit point); if not, dummy source and sink nodes are added without loss of generality. The next step is a critical one, where a subset of program paths, called *basis paths* are extracted. These basis paths are those that form a basis for the set of all paths, in the standard linear algebra sense of a basis. Symbolic execution is used to generate an satisfiability modulo theories (SMT) formula for each candidate basis path. An SMT solver is invoked to ensure that the basis paths are feasible; it generates test cases to drive execution down those paths.

The original program (*not* the unrolled, inlined version) is compiled for the target platform, and executed on these test cases. In the basic GameTime algorithm (described in [2], [6]), the sequence of tests is randomized, with basis paths being chosen uniformly at random to be executed. The overall execution time of the program is recorded for each test case. From these end-to-end execution time measurements, GameTime's learning algorithm generates a weighted graph model that is used to make predictions about timing properties of interest. The predictions hold with high probability; details of theoretical results can be found in the previous papers on GameTime [2], [6]. We provide here a less formal and more intuitive description of the theoretical guarantees for the problems of interest **P1** - **P3**:

P3: Given any δ, GameTime can predict the execution time of any program path to within a tolerance of ϵ with probability $1-\delta$ by running a number of tests that is polynomial in the program size, in $\ln(\frac{1}{\delta})$, and a parameter $\mu_{\max}$ (described below).
The tolerance ϵ is $O(b\mu_{\max})$, where b is the number of basis paths, and $\mu_{\max}$ is an upper bound on the mean perturbation to program path timing due to path-specific variations to basic block time. Essentially, ϵ depends on how much the time of a basic block can vary based on the path it lies on: the greater the mean variation, the larger the value of ϵ.

P2: For WCET estimation, GameTime provides a similar high-probability guarantee on finding the *path* along which the WCET is exhibited. Once this path is identified, one can simply execute this on the target platform and measure the corresponding execution time. Thus, if GameTime correctly finds the worst-case path, it accurately computes the WCET.
More specifically, given the mean perturbation bound $\mu_{\max}$, if the worst-case path timing is larger than the timing of any other path by a margin ρ (which is also $O(b\mu_{\max})$), then GameTime is guaranteed to find the worst-case path with probability $1-\delta$ by running a number of tests that is polynomial in the program size, in $\ln(\frac{1}{\delta})$, and $\mu_{\max}$.

It is easy to see how Problem **P1** also receives a similar high-probability theoretical guarantee. However, if the underlying assumption on the margin ρ does not hold, GameTime might

not correctly predict the WCET. In general, it is possible for GAMETIME to generate an estimate that under- or over-approximates the timing of a program path. In practice, though, we have found the estimates to be accurate (within a few percent relative error) and the worst-case path has been always correctly predicted, even on architectures that include caches, complex pipelines, and branch prediction [6].

We explain the basis path generation process using a simple sequential program that performs modular exponentiation, given in Figure 3(a). Modular exponentiation is a necessary primitive for implementing public-key encryption and decryption. In this operation, a base b is raised to an exponent e modulo a large prime number. In this particular benchmark, we use the *square-and-multiply* method to perform the modular exponentiation, based on the observation that

$$b^e = \begin{cases} (b^2)^{e/2} = (b^{e/2})^2, & e \text{ is even,} \\ (b^2)^{(e-1)/2} \cdot b = (b^{(e-1)/2})^2 \cdot b, & e \text{ is odd.} \end{cases} \quad (1)$$

The unrolled version of the code of Figure 3(a) for a 2-bit exponent is given in Figure 3(b).

In the CFG extracted from a program, nodes correspond to program counter locations, and edges correspond to basic blocks or branches.

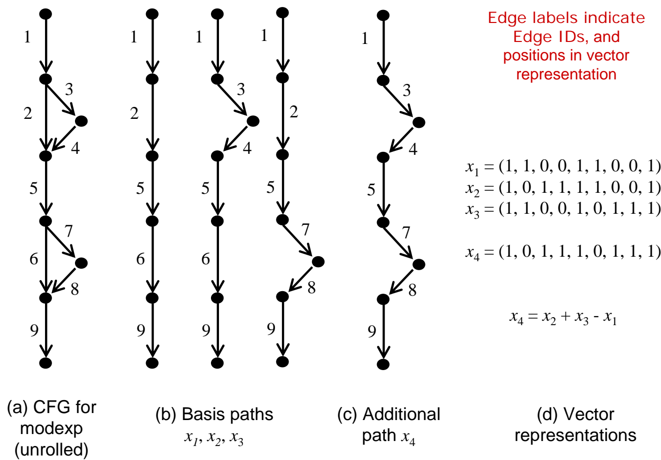


Fig. 4: **CFG and Basis Paths for Code in Fig. 3(b)**

Figure 4(a) denotes the control-flow graph for the code in Figure 3(b). Each source-sink path in the CFG can be represented as a 0-1 vector with m elements, where m is the number of edges. The interpretation is that the ith entry of a path vector is 1 iff the ith edge is on the path (and 0 otherwise). For example, in the graph of Fig. 4(a), each edge is labeled with its index in the vector representation of the path. Edges 2 and 3 respectively correspond to the else (0th bit of `exponent` = 0) and then branches of the condition statements at lines 3 and 9 respectively in the code, while edge 5 corresponds to the basic block comprising lines 6 and 7. We denote by $\mathcal{P}$ the subset of $\{0,1\}^m$ corresponding to valid program paths. Note that this set can be exponentially large in m.

A key feature of GAMETIME is the ability to exploit correlations between paths so as to be able to estimate program timing along any path by testing a relatively small subset of paths. This subset is a basis of the path-space $\mathcal{P}$, with two valuable properties: any path in the graph can be written as a linear combination of the paths in the basis, and the coefficients in this linear combination are bounded in absolute value. The first requirement says that the basis is a good representation for the exponentially-large set of possible paths; the second says that timings of some of the basis paths will be of the same order of magnitude as that of the longest path. These properties enable us to repeatedly sample timings of the basis paths to reconstruct the timings of all paths. As GAMETIME constructs each *basis path*, it ensures that it is feasible by formulating and checking an SMT formula that encodes the semantics of that path; a satisfying assignment yields a test case that drives execution down that path.

Fig. 4(b) shows the basis paths for the graph of Fig. 4(a). Here x_1, x_2, and x_3 are the paths corresponding to `exponent` taking values 00, 10, and 01 respectively. Fig. 4(c) shows the fourth path x_4, expressible as the linear combination $x_2 + x_3 - x_1$ (see Fig. 4(d)).

The number of feasible basis paths b is bounded by $m - n + 2$ (where n is the number of CFG nodes). Note that our example graph has a "2-diamond" structure, with 4 feasible paths, any 3 of which make up a basis. In general, an "N-diamond" graph with 2^N feasible paths has at most $N + 1$ basis paths.

B. Using Context Bounds to Generate a Sequential Program

Let us assume for this section that we are given a fixed context bound CB. We will explain in Section III-C how a finite inter-arrival time of interrupts can be used to generate a finite context bound.

Given a context bound CB and an interrupt-driven program $P = T_0 \| T_1 \| \ldots \| T_N$, we generate a sequential program P_{seq} that is path-equivalent to P up to context bound CB. Recall that T_j has higher priority than T_i if $j > i$, and that the main function is T_0. The procedure iteratively replaces each T_j, starting with $j = N$, with a replacement sequential program T'_j, such that every interleaved path starting in T_j and possibly involving higher-priority tasks is a program path in T'_j. Thus, T'_0 is the desired sequential program P_{seq}.

The sequential programs T'_j update a set of dummy shared variables that track the number of context switches and the program locations at which context switches occur. We describe below how we obtain T'_j from T_j.

Without loss of generality, suppose that T_j is a sequence of k atomic statements:

$$T_j \triangleq S_1; S_2; S_3; \ldots S_k$$

Thus, for each higher-priority task T_i, $i > j$, there are $k + 1$ possible locations where it may be invoked, plus the possibility that it may not interrupt T_j at all. We encode the possible switching points as well as the choice of tasks at

```
modexp(base, exponent) {
 result = 1;
 for(i=EXP_BITS; i>0; i--) {
 // EXP_BITS = 2
  if ((exponent & 1) == 1) {
   result = (result * base) % p;
  }
  exponent >>= 1;
  base = (base * base) % p;
 }
 return result;
}
```

(a) Original code P

```
modexp_unrolled(base, exponent) {
 result = 1;
 if ((exponent & 1) == 1) {
  result = (result * base) % p;
 }
 exponent >>= 1;
 base = (base * base) % p;
 // unrolling below
 if ((exponent & 1) == 1) {
  result = (result * base) % p;
 }
 exponent >>= 1;
 base = (base * base) % p;
 return result;
}
```

(b) Unrolled code Q

Fig. 3: **Modular exponentation.** Both programs compute the value of $\texttt{base}^{\texttt{exponent}}$ modulo p.

those switching points using a nondeterministic choice symbol "$*$", which is replaced by a fresh Boolean variable when generating an SMT formula by symbolic path execution. Also, each invocation of a higher-priority task increments a global variable C that tracks the number of context switches. C is initialized to 0 when P begins execution, and a higher-priority task can interrupt a lower-priority task only if $C < CB$.

The sequential program T'_j has the format

$$R_1; S_1; R_2; S_2; R_2; \ldots R_k; S_k; R_{k+1}$$

where each R_l, $l = 1, 2, \ldots, k+1$, is the following piece of code:

$$
\begin{array}{l}
\textbf{for } i = 1..CB \textbf{ do} \\
\quad \textbf{if } C < CB \textbf{ then} \\
\quad\quad \textbf{switch}(*) \, \{ \\
\quad\quad\; \textbf{case } j+1: \; C := C+1; T'_{j+1}; \textbf{break} \\
\quad\quad\; \textbf{case } j+2: \; C := C+1; T'_{j+2}; \textbf{break} \\
\quad\quad\quad \ldots \\
\quad\quad\; \textbf{case } N: \; C := C+1; T'_N; \textbf{break} \\
\quad\quad\; \textbf{default}: \; \textbf{skip} \\
\quad\quad \}
\end{array}
$$

In the above code snippet, the outer **for** loop encodes the fact that there can be at most CB invocations of a higher-priority task between atomic statements. The nondeterministic choice "$*$" encodes the choice of an arbitrary higher-priority task or no ISR invocation (in the event the "**default**" case is chosen).

It is easy to see that each intermediate statement R_l in T'_N reduces to **skip** and hence T'_N is path-equivalent to T_N. Building on this base case, we can easily obtain the following theorem by induction on N.

Theorem 1: For all $j = 0, 1, 2, \ldots, N$, the set of program paths of T'_j equals the set of all interleaved paths of $T_j \| T_{j+1} \| T_{j+2} \| \ldots \| T_N$ with at most $CB-1$ context switches.

In particular, the set of program paths of $T'_0 = P_{\text{seq}}$ is equal to the set of all interleaved paths of P with at most $CB - 1$ context switches (i.e., a context bound of CB).

C. From Inter-Arrival Times to Context Bounds

Let α be the lower bound on the inter-arrival time of interrupts on the platform of interest. We argue how α can be used to generate a context bound CB that is sufficient to include all executions of the interrupt-driven program P.

We start by hypothesizing that $CB = 1$. With this context bound, we generate a sequential program as described in Sec. III-B and compute the WCET T_W. If T_W is less than α, we know that P will complete execution before a second interrupt is raised. Thus, we can terminate with $CB = 1$.

However, if $T_W \geq \alpha$, it is possible that the main function of P is interrupted twice before terminating. Thus, we set $CB = 2$, regenerate the corresponding sequential program, and re-compute the WCET T_W. This time, we compare T_W with 2α. If $T_W < 2\alpha$, we can terminate with $CB = 2$. Otherwise, we increase CB by one and repeat the procedure. In general, when $CB = k$, we compare T_W with $k\alpha$, terminating when $T_W < k\alpha$, and otherwise incrementing CB to $k + 1$ and iterating.

If the time taken by an ISR (in the presence of higher-priority interrupts) is less than the minimum inter-arrival time of interrupts, this procedure is guaranteed to terminate with a finite context bound. To see this, note that on each iteration, T_W will grow by a smaller factor than α. This is typically the case for real-time embedded software: the rule of thumb is that ISRs must terminate very quickly in order to guarantee that every interrupt is serviced. The execution time of ISRs are typically a small fraction of the minimum inter-arrival time α.

D. Generating Measurements for Basis Paths and Predictions

The sequential program P_{seq} is fed as input to GAMETIME, which generates basis paths for this program along with the corresponding test cases. Each test case includes an assignment to program variables as well as to the nondeterministic choice variables that indicate where tasks are interrupted and by which higher-priority tasks.

We then execute the test cases within a harness that triggers interrupts at the right locations as indicated by the test case.

This harness is specific to each platform, involving the use of a few inline assembly instructions at each interrupt point (location). While this involves a slight modification to the original code, given the small number of inline assembly instructions, we believe any skew to program timing is miniscule.

Measurements can be obtained using one of a range of execution time measurement techniques – again these are platform-specific. Perhaps the most non-intrusive (but rather expensive) method is the use of a logic analyzer. Somewhat simpler is the use of on-chip cycle counters or on-board timers. These are applicable provided the code fragment is small enough that the timer register does not overflow. We use the latter approach as it is applicable for our benchmarks. Any alternative accurate measurement technique can be used instead. It is important to note that getting accurate measurements on the embedded platform can be a time-consuming process, involving repeated re-compilation and logging of measurements — therefore, it is desirable to limit the number of measurements taken. (We will see in the next section how the notion of basis paths helps us to limit the number of measurements taken, while retaining prediction accuracy.) Further platform-specific details about measurement are given in Section IV-B.

Once the measurements are obtained for the basis paths, we invoke GAMETIME's learning algorithm (as described in Section III-A) to provide answers to the problem of interest (**P1**, **P2**, or **P3**).

E. Efficiency Analysis

In this section, we calculate the number of basis paths that GAMETIME requires to perform its timing analysis and compare it to the total number of paths that are possible through a sequential program, to demonstrate the efficiency of the GAMETIME approach.

We assume that the control-flow graph of a task T_i $(1 \leq i \leq j)$ has m_i edges, n_i nodes, and p_i possible interrupt points, with a context bound of CB. Let $m = \max_i m_i$ and $p = \max_i p_i$. Since, in the worst case, the number of interrupt points can exceed the number of basic blocks, $m_i = O(p_i)$ and $m = O(p)$. For ease of analysis, we first consider a specific task T_i that can only be interrupted by exactly one higher priority task $T_j, j > i$. To generate the sequential task T_i' corresponding to task T_i, we make copies of the control-flow graph of T_j and attach a copy to each interrupt point in task T_i. The control-flow graph of T_i' thus has $O(m_i + p_i \cdot CB \cdot m_j)$ edges, which is $O(p^2 \cdot CB)$. As described earlier and in [2], the number of basis paths is linear in the number of edges, and so GAMETIME will infer $O(p^2 \cdot CB)$ basis paths. In contrast, since there are p_i possible interrupt points, and each interrupt point can be taken at least once and at most CB times, we have at least one unique program path through T_i for every choice of CB out of $p_i \cdot CB$ interrupt points. Thus, there are $O((pCB)^{CB})$ total paths through the control-flow graph of T_i. This simple case of two tasks is representative of the difference between the total number of paths through the control-flow graph of a sequentialized task and the number of basis paths that GAMETIME requires.

We can generalize this to the case of multiple tasks: consider a specific task T_i that can be interrupted by any higher priority task $T_r, (i < r \leq j)$. We can generate the sequential task T_i' corresponding to task T_i as follows: We do not need to sequentialize T_j since it is the highest priority thread and thus cannot be pre-empted by any other thread. Thus, the sequential task T_j' is the same as T_j. We sequentialize the task T_{j-1} as described in the case of two tasks to create a control-flow graph with $O(m_{j-1} + p_{j-1} \cdot CB \cdot m_j)$ edges. We can then sequentialize the task T_{j-2} by noticing that either T_{j-1}' or T_j' can interrupt at each interrupt point of T_{j-2}. The task T_{j-2}' thus has $O(p_{j-2} \cdot CB \cdot (m_{j-1} + p_{j-1} \cdot CB \cdot m_j)) = O(p_{j-2} \cdot CB \cdot m_{j-1} + p_{j-2} \cdot p_{j-1} \cdot CB^2 \cdot m_j)$ edges. Proceeding inductively, we see that the size of the control-flow graph of the sequential task T_i' is $O(\sum_{r=i}^{j-1} (\prod_{\ell=i}^{r} p_\ell CB) m_{r+1}))$ edges. However, not all the paths through this CFG are feasible, due to the context bound. In fact, to determine the number of basis paths, notice that with a context bound of CB, the effective product $\prod_{\ell=i}^{r} p_\ell CB$, after eliminating paths with more than CB context switches, has at most CB terms. Thus, the number of basis paths grows as $O((pCB)^{CB} m)$. Note that this is polynomial in the size of the tasks and is independent of the number of tasks. Using more compact transformations to a sequential program (e.g., [14], [15]) it might be possible to further reduce this bound.

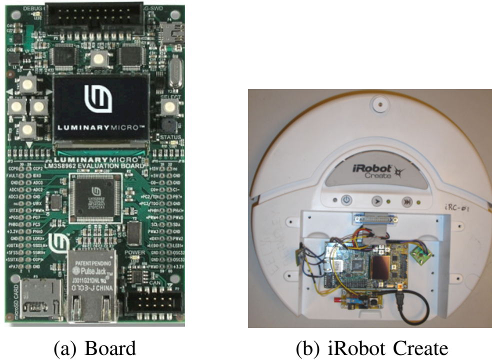
To determine the *total* number of paths through the sequential task T_i', we recognize that any of the $p_i \cdot CB$ interrupt points can be the location of one of the (at most) CB context switches. An interleaving through k tasks is a combination of CB out of $(p \cdot CB)^k$ choices of combinations of interrupt points. Thus, the total number of paths grows as $O((p \cdot CB)^{(k \cdot CB)})$. Note that this grows exponentially in the number of tasks.

IV. EXPERIMENTAL RESULTS

The goal of the experiments reported here is to demonstrate that our approach can, by measuring only a small linear subset of interleaved paths, accurately predict (i) the worst-case execution time for interrupt-driven programs (which we check by exhaustively enumerating all program paths), and (ii) the execution time along any arbitrary program path.

A. Physical Apparatus and Benchmarks

We used the Luminary Micro LM3S8962 board [7], interfaced to the iRobot Create autonomous robot platform [8] for our experiments. This microcontroller is shown in Figure 5(a) and the iRobot Create in Figure 5(b). The Luminary Micro board contains a 32-bit ARM Cortex M3 microcontroller, running at 50 MHhz. This microcontroller is interfaced to a range of peripherals: of special interest for our experiments is the UART interface to built-in iRobot sensors and the analog-to-digital interface to an ADXL-322 accelerometer. The built-in



(a) Board (b) iRobot Create

Fig. 5: Luminary Micro LM3S8962 and iRobot Create

sensors include buttons that a human can press, cliff sensors, and a bump sensor. Since the robot moves slowly, and humans cannot press buttons very quickly twice in a row, the minimum inter-arrival time of interrupts α was estimated at about 1ms for our experiments.

Our benchmarks included a toy example based on the modular exponentiation program introduced earlier, plus several real iRobot control programs. A summary of the benchmarks used is presented in Table I. The benchmarks are described in more detail below, and are also available online at `http://uclid.eecs.berkeley.edu/gametime/fmcad11/`.

Name	LOC	Size of CFG		Total Num. of paths	b	Context bound
		n	m			
modexp	60	60	70	500	12	1
iRobot-1	210	55	60	33	5	1
iRobot-2	230	141	160	3362	17	1
iRobot-3	230	97	108	1281	10	2
iRobot-4	280	213	244	33728	30	1
iRobot-5	250	179	206	65088	27	1

TABLE I: **Characteristics of Benchmarks.** "LOC" indicates number of lines of C code for the task. The Control-Flow Graph (CFG) size refers to that of the sequential program P_{seq} fed as input to GameTime: n is the number of nodes, m is the number of edges. The column b refers to the number of basis paths in the graph, as deduced by GameTime. The total number of paths indicates the total number of interleaved execution paths, not accounting for path feasibility.

B. Generating Interrupts and Measurements

To measure the timing of each basis path, we need to force an interrupt at one or more program points, depending on the context bound. There are two types of interrupts that can be forced: software and hardware interrupts. For this platform, the overhead of invoking an ISR through hardware interrupts is similar to that using software interrupts; therefore, for convenience, we decided to force software interrupts.

Software interrupts can be modeled by embedding the ARM assembly instruction `SVC` into the C code under analysis. This instruction is a supervisor call that forces a software interrupt. To use this instruction, we modify the interrupt vector table to include a custom interrupt handler. In the code under analysis, we then insert an `SVC` assembly instruction whose argument is the position of the interrupt handler in the vector table, so that on execution, the instruction goes directly to the vector table and invokes the interrupt handler for the interrupt we wish to trigger.

Obtaining Timing Measurements: The execution time of the program was measured using an on-board timer called `SysTickTimer`. This timer can be set to periodically generate an interrupt by counting down from a large starting value. The period of the timer is user-specified and is large enough that it will not finish until long after the program under analysis finishes. To get the execution time for the program under analysis, we start the timer off before the program runs and read off its value when the program finishes. We assume that the program would finish within 16,777,261 cycles (the highest possible value for the `SysTickTimer` period), which is a realistic assumption for our set of benchmarks.

C. Modular Exponentiation

Our first example is a version of the modular exponentiation example introduced in Sec. III-A. An arbitrary prime number was used for our benchmarks. For our experiments, we used a base of two, with four-bit exponents. Two of the four conditionals were moved into a mock 'interrupt handler'; the program comprising the remaining two conditions formed the "main" task. Thus, the program comprises two tasks: each with two of the conditionals. Each code fragment of the form below is considered an atomic statement.

```
if ((exponent & 1) == 1)
  result = (result * base) % p;
exponent >>= 1;

base = (base * base) % p;
```

We used GameTime to determine the values that would sensitize the basis paths in the control flow graph of the "main" task, and it provided 12 test cases. With a context bound of 1, there are three program points where the ISR can be invoked. Since there are 16 values of `exponent`, and three possible interrupt points, the total number of test cases is $16 * 3 = 48$.

With the measurements for the 12 test cases corresponding to the basis paths, GameTime was employed to infer a timing model using which it predicted the runtimes of each of the 48 different test cases.

In figure 6, we plot the predicted values and the measured values for each of the 48 test cases. The predicted values match the measured values with an error of less than 5%. In particular, our approach accurately predicts the worst-case execution time and produces the corresponding test case. We observe that the WCET estimate, about 290 cycles or 5.8μs, is much less than the 1ms inter-arrival time of interrupts, implying that the context bound of 1 is sufficient.

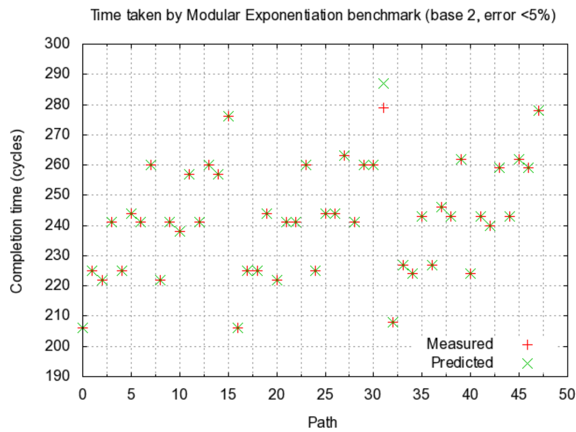


Fig. 6: **Time taken by the Modular Exponentiation benchmark**

D. iRobot Driving Code

The iRobot Create control programs we consider here involve a sample operation where the robot attempts to keep moving forward until it senses an obstacle, in which case it will try to back up, turn, and move around the obstacle. The robot can be stopped by pushing a button on its console. The accelerometer detects changes in the speed of the robot, such as when it accelerates on level ground or when it climbs a hill.

All iRobot sensors (with the exception of the accelerometer) trigger the same UART interrupt that is serviced with a single ISR. This ISR reads the values of the sensors or the status of the buttons from a UART queue, and updates local variables accordingly. The accelerometer triggers a different interrupt that is serviced by a different routine that also updates local variables with the accelerometer readings.

The code that produces the iRobot behavior described above is an infinite loop encoding a state machine. The body of this loop involves the next-state update operation based on sensor data, and this is what we used the five iRobot benchmarks shown in Table I. The "main task" in all benchmarks has a similar structure: it updates the state of the robot based on sensor readings, button presses, or accelerometer readings, if any, and the new state, if changed, modifies the velocity of the iRobot. All benchmarks also have at least two interrupt points: each conditional in the state update is considered atomic, and the velocity modification is also considered atomic. The first four benchmarks used only the sensor interrupt handler; the last used only the accelerometer interrupt handler.

The first iRobot benchmark, iRobot-1, is a highly simplified version of the behavior described above. A context bound of 1 was sufficient. The simplified state machine allows us to manually enumerate and time all of the feasible paths, and also to use the basis paths to predict the time for all paths. The measured and predicted timings for the ten feasible paths are shown in figure 7: the timings agree within one percent, and the path that was predicted to take the longest time is also the path that was measured to take the longest time. The WCET is less than 2500 cycles, which is less than 50μs, much smaller than the 1ms inter-arrival time, ensuring that the context bound of 1 is sufficient.

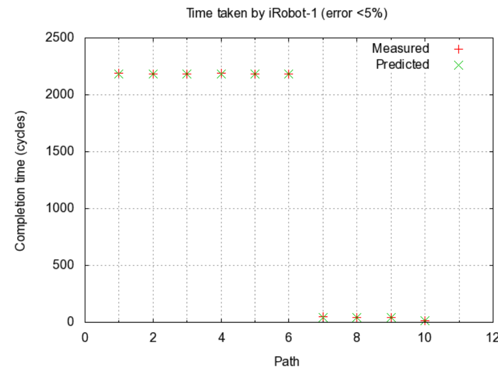


Fig. 7: **Time taken by iRobot-1 benchmark**

The second benchmark adds one more state to the state machine. For this and the remaining benchmarks, the number of interleaved program paths (as seen from Table I) is over 1000 — thus, it is not possible to time all the possible paths. Therefore, for these benchmarks, we arbitrarily selected 16 paths to be measured. The true (measured) execution times of these paths are compared with the runtimes predicted from measuring just the basis paths and running GAMETIME. The resulting plot for the iRobot-2 is shown in Figure 8. Again, a context bound of 1 suffices.

To experiment with a larger context bound, we assumed the minimum inter-arrival time to be $\alpha = 50\mu$s, and analyzed the third benchmark iRobot-3. With a context bound of 1, the WCET exceeded this value. However, with a context bound of 2, the WCET is 4357 cycles, or about 87μs, which is less than $2\alpha = 100\mu$s. As shown in Figure 9, the error margin between predicted and true (measured) values is almost zero.

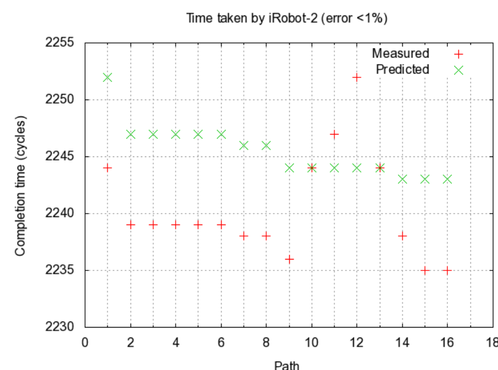


Fig. 8: **Time taken by iRobot-2 benchmark**

The fourth benchmark iRobot-4 adds more states to the state machine, while the fifth benchmark iRobot-5 keeps only those states that use the accelerometer. Nonetheless, the error margin

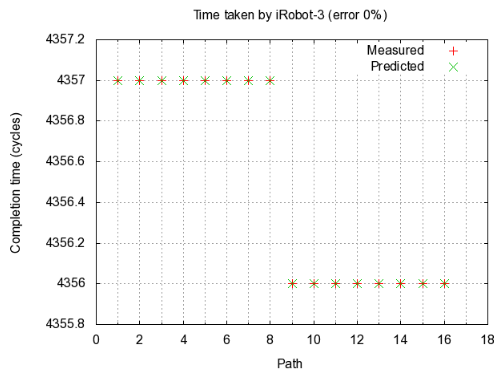


Fig. 9: **Time taken by iRobot-3 benchmark.** Note that there are only two execution times exhibited by these paths: 4357 and 4356 cycles.

in both benchmarks, for the 16 chosen paths, is less than 2 percent. In both cases, a context bound of 1 sufficed.

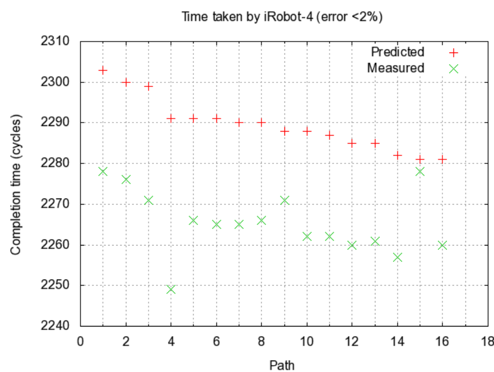


Fig. 10: **Time taken by iRobot-4 benchmark**

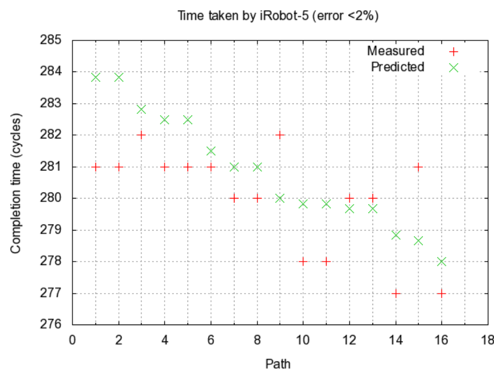


Fig. 11: **Time taken by iRobot-5 benchmark**

V. Conclusion and Future Work

In this paper, we present a new approach for timing analysis of interrupt-driven programs. The key ideas in our approach are to bound the exploration of the path space using the twin notions of context bounds and basis paths. We have demonstrated for a real embedded platform and control software that our approach can accurately predict not only the worst-case execution time, but also the execution time of arbitrary interleaved program paths without needing to exhaustively enumerate and test them. For future work, we plan to expand our experimental evaluation to include larger benchmarks with several interrupt service routines (tasks).

Acknowledgments. This work was supported in part by NSF grants CNS-0644436 CNS-0627734, and CNS-1035672, an Alfred P. Sloan Research Fellowship, the Toyota Motor Corporation, and the Hellman Family Faculty Fund. We thank the anonymous referees for their comments.

References

[1] Reinhard Wilhelm et al., "The Determination of Worst-Case Execution Times—Overview of the Methods and Survey of Tools," *ACM Transactions on Embedded Computing Systems (TECS)*, 2007.

[2] S. A. Seshia and A. Rakhlin, "Game-theoretic timing analysis," in *Proc. IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2008, pp. 575–582.

[3] NASA Engineering and Safety Center, "NASA report on Toyota unintended acceleration investigation, appendix a: Software," http://www.nhtsa.gov/staticfiles/nvs/pdf/NASA_FR_Appendix_A_Software.pdf.

[4] S. Qadeer and D. Wu, "KISS: keep it simple and seqeuential," in *In PLDI 04: Programming Language Design and Implementation*, 2004, pp. 14–24.

[5] S. Qadeer and J. Rehof, "Context-bounded model checking of concurrent systems," in *Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2005.

[6] S. A. Seshia and A. Rakhlin, "Quantitative analysis of systems using game-theoretic learning," *ACM Transactions on Embedded Computing Systems (TECS)*, 2011, to appear.

[7] Luminary Micro, Inc., "Luminary Micro microcontroller datasheet," http://chess.eecs.berkeley.edu/eecs149/sp09/docs/Datasheet_LM3S8962.pdf.

[8] iRobot Corporation, "iRobot Create User's Manual," http://www.irobot.com/filelibrary/pdfs/hrd/create/Create\%20Manual_Final.pdf.

[9] D. Brylow and J. Palsberg, "Deadline analysis of interrupt-driven software," *IEEE Transactions on Software Engineering*, 2004.

[10] J. Gustafsson and A. Ermedahl, "Experiences from applying WCET analysis in industrial settings," in *Proc. IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC)*, 2007, pp. 382–392.

[11] J. Regehr, "Random testing of interrupt-driven software," in *Proc. 5th ACM International Conference on Embedded Software (EMSOFT)*, 2005, pp. 290–298.

[12] M. G. Harbour, M. H. Klein, and J. P. Lehoczky, "Timing analysis for fixed-priority scheduling of hard real-time systems," *IEEE Trans. Software Engineering*, vol. 20, no. 1, pp. 13–28, 1994.

[13] N. Kidd, S. Jagannathan, and J. Vitek, "One stack to run them all - reducing concurrent analysis to sequential analysis under priority scheduling," in *17th International SPIN Workshop on Model Checking Software (SPIN)*, 2010, pp. 245–261.

[14] A. Lal and T. W. Reps, "Reducing concurrent analysis under a context bound to sequential analysis," in *CAV'08*, ser. LNCS, vol. 5123, 2008, pp. 37–51.

[15] S. K. Lahiri, S. Qadeer, and Z. Rakamarić, "Static and precise detection of concurrency errors in systems code using SMT solvers," in *CAV'09*, ser. LNCS, vol. 5643, 2009, pp. 509–524.

[16] C. Barrett, R. Sebastiani, S. A. Seshia, and C. Tinelli, "Satisfiability modulo theories," in *Handbook of Satisfiability*, A. Biere, H. van Maaren, and T. Walsh, Eds. IOS Press, 2009, vol. 4, ch. 8.

[17] S. A. Seshia and J. Kotker, "GameTime: A toolkit for timing analysis of software," in *Proceedings of the 17th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2011.

Automated Error Localization and Correction for Imperative Programs

Robert Könighofer and Roderick Bloem
Institute for Applied Information Processing and Communications (IAIK),
Graz University of Technology

Abstract—**In this paper, we present a novel debugging method for imperative software, featuring both automatic error localization and correction. The input of our method is an incorrect program and a corresponding specification, which can be given in form of assertions or as a reference implementation. We use symbolic execution for program analysis. This allows for a wide range of different trade-offs between resource requirements and accuracy of results. Our error localization method rests upon model-based diagnosis and SMT-solving. Error correction is done using a template-based approach which ensures that the computed repairs are readable. Our method can handle all sorts of incorrect expressions, not only under a single-fault assumption but also for multiple faults. Finally, we present experimental results, where an implementation for C programs is used to debug mutants of the TCAS case study of the Siemens suite.**

I. Introduction

A lot of research has been done in the past decades to automate detection of errors in programs, be it software or hardware. But once an error is detected, the hard work only begins: the error has to be located and corrected. This is usually done manually, which is time-consuming, costly, frustrating, and increases time-to-market. More and better automation in these steps is definitely needed.

Many existing approaches, especially for automatic error correction, are based on fully formal methods with limited scalability when it comes to larger state spaces. On the other hand, simulation-based methods suffer from limited accuracy. Trade-offs are usually not possible. Another often insufficiently addressed issue is that synthesized corrections have to be readable. A method which produces repairs as Boolean functions that cannot be understood is of limited use, because the repaired program cannot be maintained. Furthermore, repairs should affect only small parts of the program. This lowers the chances that unspecified properties of the program get lost.

We propose a novel method for automatic error localization and correction, explicitly addressing all these challenges. It is outlined in Fig. 1. An incorrect program and its specification are the inputs. The specification may be given via assertions in the code or as a reference implementation. First, we preprocess the program to express that components may be faulty. Our fault model can handle all kinds of incorrect expressions

This work was supported in part by the European Commission through project DIAMOND (FP7-2009-IST-4-248613).

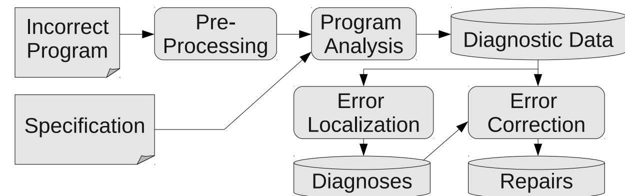


Fig. 1. The flow of our debugging method.

and is not restricted to single-faults. Next, symbolic execution is applied to transform the debugging problem into the domain of logic. Error localization rests upon model-based diagnosis [27], [24]. It computes sets of components of the program which can be replaced in such a way that the program becomes correct. In the correction step, such replacements are finally computed. This is done using a template-based approach, where repairs are iteratively refined. This refinement is guided by counterexamples. All reasoning is done with a Satisfiability Modulo Theories (SMT) solver.

Symbolic execution is a semi-formal technique to analyze program behavior path-by-path. This makes our method degrade nicely: considering more paths means better accuracy but also higher resource requirements. Error localization and correction provide additional parameters for trade-offs. Fine-grained error localization leads to repairs that affect only small program parts. Readable repairs are obtained by restricting the search to templates for expressions. Multiple templates can be used successively: starting with a simple template, one can switch to a more expressive one if no repair is found. Within a certain template, we use heuristics to find simple instantiations first. Our debugging method is especially suited for debugging simple pieces of software, e.g., C programs modeling hardware designs at a high abstraction-level.

Model-based diagnosis [27], [24] has already been applied to locate errors in logic programs [7], functional programs [32], VHDL designs [14], Java programs [26], knowledge bases [11], ontologies [13], and temporal logic specifications [25]. We apply the technique to an abstraction of a program found by symbolic execution and combine it with error correction. In [18], errors are located using a model-checker. Our diagnosis approach is similar, but we require only one symbolic execution pass rather than several model-checker calls, and we compute diagnoses differently. Another related

diagnosis method is presented in [21]. It computes a maximal set of statements that can remain unchanged for the program to become correct for a given input. The complement forms the diagnosis. This is encoded as a Maximum Satisfiability (MAX-SAT) problem. In contrast, our method allows to use several failure-inducing inputs simultaneously; a diagnosis must allow to fix the program for *all* these inputs.

One approach to program repair is to transform a finite-state program into a finite-state game and to compute a repair as a strategy in this game [19], [20]. This has also been extended to programs with virtually infinite state space (i.e., software) using predicate abstraction [17]. In contrast, we use symbolic execution, a technique especially suitable for software.

Our repair method is very related to program sketching [30], [29], a paradigm where the user provides a program with unknown parts ("holes") and a specification. A tool synthesizes the holes automatically. For complex unknowns, the user has to provide so-called "generators", which are functions containing only unknown integer values. These generators serve the same purpose as our repair templates: reducing the synthesis of components to the search for integer constants. The main difference is that our method works in a push-button manner, i.e., templates do not have to be (but can be) provided by the user. Moreover, in the repair setting, holes have to be computed first, they should be small, and their implementation readable. Templates for expressions are also used to synthesize loop invariants for program verification [6]. The differences to our approach for computing repairs lie in the constraints that have to be fulfilled (program correctness rather than inductiveness) and how instances are computed (iterative refinement rather than quantifier elimination). The idea of synthesizing parts of a program by iterative, counterexample-guided refinements has already been used in [3] and [30]. We extend the basic idea with a heuristic to speed up convergence. Program Synthesis is also addressed in [31], where imperative programs are synthesized from a given specification and flowgraph structure. This work uses program a verification tool performing a fixed-point computation to synthesize a solution.

A quite different repair approach is to repeatedly mutate an incorrect program and check if it becomes correct [8]. The problem is the huge search space for mutants. Our repair method is much more systematic. Finally, there are genetic programming methods, combining mutation with crossing and selection according to some notion of fitness [1], [12].

In summary, the contributions of this paper are as following.

- We present a new debugging approach which produces readable repairs at the source level and degrades nicely.
- We combine many existing techniques in a novel way: symbolic execution, model-based diagnosis, templates for unknown expressions, and iterative repair refinement.
- We show how model-based diagnosis can be applied to a program abstraction found by symbolic execution.
- We present a heuristic to speed up repair refinement.
- Finally, we present experimental results using an implementation of our debugging approach for C programs.

This paper is organized as follows. Section II explains techniques underlying our approach and establishes notation. Section III presents our debugging method as outlined in Fig. 1. Section IV discusses alternatives and trade-offs. Section V presents first experimental results, and Section VI concludes.

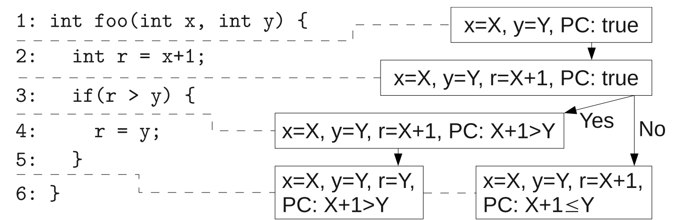


Fig. 2. Example: Symbolic execution.

II. Preliminaries

A. Symbolic Execution

Symbolic execution [5], [23] is a program analysis technique. It executes a program with symbols as inputs. Symbols are placeholders that can take on any value in some domain. Symbolic execution keeps track of the symbolic values (expressions involving symbols and constants) of all program variables. Whenever a branching point is reached, the execution forks. For every branch, a condition expressing when it is taken is computed. Along an execution path, the branch conditions are accumulated to a path condition. Thus, a path condition states when a certain execution path is activated. In practice, the maximum path length and the number of paths to analyze are limited to ensure termination.

Example 1. *Fig. 2 illustrates symbolic execution on an example. Two symbols X and Y are used for the unknown values of x and y. Boxes contain execution states, dashed lines link them to the program, and arrows indicate the execution flow. In Line 3, the execution forks since both branches are feasible. The condition which has to be fulfilled for the program to reach a certain state is denoted as PC. The path conditions can be read from the PC-fields in the leaves of the tree. This program has two paths with conditions $X + 1 > Y$ and $X + 1 \leq Y$.*

Concolic execution [15], [28] is a variant of symbolic execution where the program is executed using concrete and symbolic inputs simultaneously. The execution path is determined by the concrete values. Along this path, symbolic variable values are tracked and a path condition is computed. After one execution, the conjuncts of the path condition are used to compute concrete input values that trigger a different path. For our purposes, concolic execution produces the same outcomes as symbolic execution, namely path conditions.

B. Model-Based Diagnosis

Model-based diagnosis [24], [27] (MBD) is a method to locate errors in a system by explaining conflicts between a model of the system and an observation of some incorrect behavior. We follow the notation of [27] in this work.

Let SD be a model of a system, and let OBS be an observation of an erroneous behavior, both given as sets of logical

sentences. The system consists of a set of components CMP. A component $c \in \mathsf{CMP}$ can behave abnormally (denoted $\mathsf{AB}(c)$) or normally (written $\neg\,\mathsf{AB}(c)$). Every component c is described with a logical sentence of the form $\neg\,\mathsf{AB}(c) \Rightarrow N_c$, with N_c defining the normal behavior of c. That is, abnormal components can behave arbitrarily. The system description SD is composed of component descriptions and a set of logical sentences defining the interplay of components. The observation OBS contradicts SD in the sense that, if all components behaved normally, it would be impossible to observe OBS. That is, the set $\mathsf{SD} \cup \mathsf{OBS} \cup \{\neg\,\mathsf{AB}(c) \mid c \in \mathsf{CMP}\}$ of logical sentences is inconsistent, i.e., contains a logical contradiction.

MBD computes diagnoses, which are sets of components that may be responsible for observation OBS. Formally, a set $\Delta \subseteq \mathsf{CMP}$ is a *diagnosis* iff $\mathsf{SD} \cup \mathsf{OBS} \cup \{\neg\,\mathsf{AB}(c) \mid c \in \mathsf{CMP} \setminus \Delta\}$ is consistent. The components in Δ may be responsible for OBS because assuming that these components behave abnormally renders OBS possible. A diagnosis Δ is *minimal* if no subset $\Delta' \subset \Delta$ is a diagnosis. If Δ is a diagnosis, then clearly every $\Delta' \supseteq \Delta$ is a diagnosis as well. Hence, we are only interested in minimal diagnoses.

Diagnoses can be computed via conflicts. A *conflict* is a set $C \subseteq \mathsf{CMP}$ of components such that $\mathsf{SD} \cup \mathsf{OBS} \cup \{\neg\,\mathsf{AB}(c) \mid c \in C\}$ is inconsistent. I.e., a conflict is a set of components that cannot all have behaved normally. A conflict C is *minimal* if no subset $C' \subset C$ is a conflict. A *hitting set* for a collection $\mathcal{K}$ of sets is a set H such that $\forall K \in \mathcal{K} \,.\, H \cap K \neq \emptyset$ holds. A hitting set H is *minimal* if no subset $H' \subset H$ is a hitting set. A set $\Delta \subseteq \mathsf{CMP}$ is now a minimal diagnosis iff Δ is a minimal hitting set for all conflicts. The intuitive reason is that a diagnosis must explain all conflicts, so it must share at least one element with every conflict. Reiter [27] presents a hitting set computation algorithm which computes conflicts on-the-fly and produces diagnoses in order of increasing cardinality.

C. Notation of Vectors and Domains

We write $\mathcal{S}$ for the set of finite strings. Overlines are used to indicate vectors. For two vectors $\overline{a} = (a_1, \ldots, a_m)$ and $\overline{b} = (b_1, \ldots, b_n)$, we write $\overline{a}||\overline{b}$ for the concatenation $(a_1, \ldots, b_n)$. For symbolic execution, we assume that all symbols are taken from a sufficiently large set $\mathbb{S}$. To simplify notation, we also assume that all symbols range over some domain $\mathbb{D}$ of values. For instance, $\mathbb{D}$ may be $\mathbb{Z}$ or $\mathbb{B}^m$. An extension to different domains for different symbols is straightforward.

We denote with D_{ex} a domain of symbolic expressions and with D_{co} a domain of symbolic conditions. Let $e_1, e_2 \in D_{ex}$ and $c_1, c_2 \in D_{co}$. We assume that $e_1 = e_2$, $e_1 \leq e_2$, $e_1 \geq e_2$, $c_1 \vee c_2$, $c_1 \wedge c_2$, and $\neg c_1$ are in D_{co} as well, with the expected semantics ("=" means equality here, not an assignment). For $c \in D_{co}$ we write $c[\overline{a}]$ with $\overline{a} = (a_1, \ldots, a_m) \in \mathbb{S}^m$ to indicate that c may only depend on the symbols $a_1, \ldots, a_m$. Let $\overline{b} = (b_1, \ldots, b_m) \in (\mathbb{S} \cup \mathbb{D})^m$ be a vector of symbols and constants. Then $c[\overline{b}]$ denotes condition $c[\overline{a}]$ where all symbols a_i, with $1 \leq i \leq m$, have been replaced by b_i. Analogously for expressions. Finally, we assume that a sound and complete decision procedure (e.g., an SMT-solver) for the satisfiability of conditions $c \in D_{co}$ is available.

III. Debugging Method

This section introduces our debugging approach, as outlined in Fig. 1, in more detail. A discussion of practical aspects and some alternatives will be given in Section IV. The input of our debugging method is a program P and a specification S. The program may contain calls to a special function `input`, returning an unknown input value $v \in \mathbb{D}$. As a specification, assertions in the code are supported natively. This also allows using reference implementations: the reference implementation is executed with the same inputs and results are compared with assertions. The user can define the desired notion of equivalence with suitable assertions. The program is assumed to violate the specification for some input.

A. Pre-Processing

Our method needs to report components of the program P as possibly faulty, and to suggest replacements. This requires a notion of a component and a corresponding fault model.

The ideal fault model can explain all errors, is fine-grained, and enables efficient algorithms. Clearly, these properties cannot all be maximized at the same time. As a trade-off, we assume that only the right-hand side (RHS) of assignments may be erroneous. Alternatives will be discussed in Section IV. The reasons for our decision are as follows. Assignments are the fundamental operation in any imperative program. The fault model allows for efficient program analysis, since an unknown RHS can be handled symbolically, just like an input. (See Section III-B.) Moreover, it is fine-grained and can easily be extended to incorrect expressions by assigning every expression to a temporary variable. Consequently, we consider the RHS of every assignment as a replaceable component of the program. The rest of the program is deemed unmodifiable.

Example 2. *Consider the following program (in C syntax).*

```
1 int max(int x, int y) {
2   int r = x;
3   if(y > x)
4     r = x;
5   assert(r >= x && r >= y);
6   return r;
7 }
```

It will serve as a running example. It should compute the maximum of two numbers but contains an error in Line 4, which should read "`r = y`*". With* $\mathsf{CMP} = \{c_1, c_2\}$ *we identify two components,* c_1 *being the* `x` *in Line 2 and* c_2 *being the* `x` *in Line 4. By re-writing Line 3 to "*`int tmp = y > x; if(tmp)`*", the condition* `y > x` *can be handled as a third component* c_3*. However, for simplicity of explanations,* c_3 *is not considered as a component in the running example.*

Next, we express that components may be faulty. We create a modified program $\tilde{P}$, which is identical to P except that each assignment `LHS = RHS;` is textually replaced by

```
LHS = cmp(c, RHS, v1, v2, ..., vn);
```

Here, cmp is a special function indicating components, c is (a unique identifier of) the component $c \in \mathsf{CMP}$, and $\mathtt{v1}, \dots, \mathtt{vn}$ are the variables which are in scope when component c is executed. We can think of cmp as a shorthand for
`assumeCorrect(c) ? RHS : rep_c(v1,…,vn)`.
If component c is assumed to be correct, there is no need to modify it. Otherwise it can be replaced by a new expression, which is embodied by the (yet unknown) function rep_c. The error localization step will find out which components are assumed to be incorrect. The error correction step will compute implementations of the functions rep_c for all incorrect components.

We define a function $\mathrm{Vars} : \mathsf{CMP} \to \mathcal{S}^*$ mapping each component $c \in \mathsf{CMP}$ to a vector of variable names $\mathtt{v1}, \dots, \mathtt{vn}$. These are the variables in scope when c is executed. The mapping can be computed easily during pre-processing and will be required for applying and outputting repairs.

Example 3. *The program P from Example 2 gives $\tilde{P} =$*

```
1 int max(int x, int y) {
2   int r = cmp(1,x,x,y);
3   if(y > x)
4     r = cmp(2,x,x,y,r);
5   assert(r >= x && r >= y);
6   return r;
7 }
```

where the calls to cmp *intuitively mean*
`assumeCorrect(c1) ? x : rep_c1(x,y)` *and*
`assumeCorrect(c2) ? x : rep_c2(x,y,r)`.

B. Program Analysis

We execute $\tilde{P}$ symbolically or concolically to get diagnostic information. Unlike standard symbolic execution, we maintain two sets of symbols: input symbols and repair symbols.

Input symbols represent unknown input values. Whenever the function input is called, a fresh input symbol is created. We denote the vector of created input symbols (in arbitrary order) as $\bar{i} = (i_1, \dots, i_A) \in \mathbb{S}^A$. Similarly, repair symbols represent the unknown values returned by the function cmp. These symbols will be denoted as $\bar{r} = (r_1, \dots, r_B) \in \mathbb{S}^B$.

With the following functions, every repair symbol r_b is associated with additional information. $\mathrm{CmpOf} : \{r_1, \dots, r_B\} \to \mathsf{CMP}$ maps r_b to the component which produced it. $\mathrm{Org} : \{r_1, \dots, r_B\} \to D_{ex}$ maps r_b to the value that would be produced by the unmodified component $\mathrm{CmpOf}(r_b)$. $\mathrm{Vals} : \{r_1, \dots, r_B\} \to D_{ex}^*$ maps r_b to the values of all variables in scope when $\mathrm{CmpOf}(r_b)$ produced r_b. These functions are built up from the symbolic values of the first parameter, the second parameter, and the subsequent parameters of cmp, respectively.

We say that an execution path is a sequence of statements which starts with the initial statement and ends with the termination of the program. For every execution path p in $\tilde{P}$, a path condition $\mathrm{PC}^p[\bar{i}||\bar{r}] \in D_{co}$ is created during the symbolic execution. The paths are divided into the two sets PASS and FAIL. A path p is in FAIL if it violates the specification, i.e., ends in an abnormal program termination after an assertion violation. It is in PASS otherwise. We define a condition $\pi[\bar{i}||\bar{r}] \in D_{co}$ as

$$\pi[\bar{i}||\bar{r}] = \bigvee_{p \in \mathrm{PASS}} \mathrm{PC}^p[\bar{i}||\bar{r}]. \tag{1}$$

Lemma 1. *Let $\overline{v_i} \in \mathbb{D}^A$ and $\overline{v_r} \in \mathbb{D}^B$ be two vectors of concrete values. Assuming that all paths of $\tilde{P}$ have been analyzed, the condition $\pi[\overline{v_i}||\overline{v_r}]$ is* true *iff $\tilde{P}$ fulfills the specification S, given that $\overline{v_i}$ is used as input vector and $\overline{v_r}$ is the vector of values returned in calls to the function* cmp.

Proof: Let p_e be the path activated by $\overline{v_i}$ and $\overline{v_r}$ in $\tilde{P}$. Clearly, $\mathrm{PC}^p[\overline{v_i}||\overline{v_r}]$ is true iff p=p_e. Lemma 1 holds since $\mathrm{PC}^{p_e}[\overline{v_i}||\overline{v_r}]$ is a disjunct of $\pi[\overline{v_i}||\overline{v_r}]$ iff p_e satisfies S. ■

Proposition 1. *Let P be an incorrect program and let $\overline{v_i} \in \mathbb{D}^A$ be an input vector. Assuming that all paths of $\tilde{P}$ have been analyzed, the condition*

$$\exists \bar{r} \,.\, \pi[\overline{v_i}||\bar{r}] \wedge \bigwedge_{r_b \text{ in } \bar{r}} r_b = \mathrm{Org}(r_b)[\overline{v_i}||\bar{r}]$$

is true *iff P fulfills the specification S when executed with input $\overline{v_i}$.*

Proof: Let $\overline{v_r} \in \mathbb{D}^B$ be the concrete values returned by calls to cmp in $\tilde{P}$. According to Lemma 1, $\pi[\overline{v_i}||\overline{v_r}]$ evaluates to true iff $\tilde{P}$ fulfills S for input $\overline{v_i}$. The additional conjuncts in the formula require that all components in $\tilde{P}$ return the same value as the respective components in P. Hence, the formula valuates to true iff P fulfills S for input $\overline{v_i}$. ■

Lemma 1 states how the condition $\pi[\bar{i}||\bar{r}]$ can be used to make statements about the correctness of the instrumented program $\tilde{P}$, depending on the inputs and the components. Proposition 1 establishes the link to the correctness of the original program P, using the information in Org.

Definition 1. *The tuple $\Gamma = (\mathsf{CMP}, \mathrm{Vars}, \bar{i}, \bar{r}, \mathrm{CmpOf}, \mathrm{Org}, \mathrm{Vals}, \pi[\bar{i}||\bar{r}])$ is called* diagnostic data*.*

Example 4. *For P from Example 2 we have CMP=$\{c_1, c_2\}$, $\mathrm{Vars}(c_1)$=(x, y), $\mathrm{Vars}(c_2)$=(x, y, r), $\bar{i}$=(X, Y), $\bar{r}$=(R_1, R_2), $\mathrm{CmpOf}(R_1)$=c_1, $\mathrm{CmpOf}(R_2)$=c_2, $\mathrm{Org}(R_1)$=X, $\mathrm{Org}(R_2)$=X, $\mathrm{Vals}(R_1)$=(X, Y), $\mathrm{Vals}(R_2)$=(X, Y, R_1), and $\pi[\bar{i}||\bar{r}] = (Y > X \wedge R_2 \geq X \wedge R_2 \geq Y) \vee (Y \leq X \wedge R_1 \geq X \wedge R_1 \geq Y)$.*

The diagnostic data Γ is the output of the program analysis step and will serve as input for error localization and error correction (recall again Fig. 1).

C. Error Localization

Our method for error localization rests upon MBD as introduced in Section II-B. This section explains how MBD can be applied in our setting. The next section will then discuss how diagnoses can actually be computed.

Standard MBD takes as input a model of a system together with a contradicting observation. The contradiction manifests itself in conflicts, which need to be explained. In our setting, a program conflicts with its specification, so we need a different

notion of a conflict. Deriving diagnoses from conflicts works in the standard way.

We define a function $\mathsf{repairable} : 2^{\mathsf{CMP}} \to \{\mathsf{true}, \mathsf{false}\}$. Intuitively, $\mathsf{repairable}(Q)$ maps a set $Q \subseteq \mathsf{CMP}$ to true iff program $\tilde{P}$ can be repaired for all inputs, assuming that all components $c \in Q$ are correct and need not be modified. Formally, we define

$$\mathsf{repairable}(Q) \Leftrightarrow \forall \bar{i} \,.\, \exists \bar{r} \,.\, \pi[\bar{i}||\bar{r}] \wedge \bigwedge_{r \in R} r = \mathrm{Org}(r)[\bar{i}||\bar{r}], \quad (2)$$

where R stands for $\{r \mid \mathrm{CmpOf}(r) \in Q\}$ and $\forall \bar{i}$ is a shorthand for $\forall i_1 \ldots \forall i_A$. Likewise for $\exists \bar{r}$. The definition says that a program is repairable iff for all inputs, there exist values that can be returned by the components (the function `cmp` in $\tilde{P}$) such that the specification is fulfilled. Components which are assumed to be correct can only return the value that would be returned by the original version of that component.

Lemma 2. *The function* repairable *is monotonic in that, for all* $Q' \subseteq Q \subseteq \mathsf{CMP}$, $\mathsf{repairable}(Q)$ *implies* $\mathsf{repairable}(Q')$.

Monotonicity is obvious since removing elements from Q only removes conjuncts in the definition of repairable.

Definition 2. *A set* $\Delta \subseteq \mathsf{CMP}$ *is a diagnosis for program* P *iff* $\mathsf{repairable}(\mathsf{CMP} \setminus \Delta) = \mathsf{true}$. *A set* $C \subseteq \mathsf{CMP}$ *is a conflict iff* $\mathsf{repairable}(C) = \mathsf{false}$.

A diagnosis is a set of components that can be modified such that P becomes correct. The reason is that, for every input, it is possible to find some value that can be returned by the components $c \in \Delta$ such that the specification is fulfilled. Hence, diagnoses represent fault candidates. A conflict is a set of components from which at least one component has to be modified in order to obtain a correct program.

Example 5. *For the program* P *in Example 2 we have:*

Case	*Set* Q	$\mathsf{repairable}(Q)$	*Diagnosis*	*Conflict*
1	$\emptyset$	true	$\{c_1, c_2\}$	
2	$\{c_1\}$	true	$\{c_2\}$	
3	$\{c_2\}$	false		$\{c_2\}$
4	$\{c_1, c_2\}$	false		$\{c_1, c_2\}$

We have that $\mathsf{repairable}(\{c_1\})$=true *because* c_2 *can be modified to render* P *correct. For every input* X,Y, *there is a value (namely* Y*) to return by* c_2 *such that the assertion holds. Hence,* c_2 *may be responsible for the incorrectness of* P *— it is a diagnosis. On the other hand,* $\mathsf{repairable}(\{c_2\})$=false *because for* X=0, Y=1 *the specification is violated no matter what is returned by* c_1*, simply because the value is overwritten by* c_2*. Hence,* c_1 *cannot be responsible for the incorrectness, i.e.,* $\{c_1\}$ *is not a diagnosis. The other two cases are trivial.*

D. Computation of Diagnoses

The following theorem, which is a slight adaptation of Theorem 4.4 from [27], states that minimal diagnoses can be computed as minimal hitting sets for the collection of conflicts.

Theorem 1. *A set* $\Delta \subseteq \mathsf{CMP}$ *of components is a minimal diagnosis for program* P *iff it is a minimal hitting set for the collection* $\mathcal{K}$ *of conflicts for* P.

Proof: Using Lemma 2, the proof in [27] applies. ■

We use the hitting set tree algorithm of Reiter [27] (with the fix of [16]) to compute diagnoses. It requires a procedure to compute a conflict not containing a certain set N of elements, if such a conflict exists. Such a procedure can be implemented by returning $\mathsf{CMP} \setminus N$ if $\mathsf{repairable}(\mathsf{CMP} \setminus N) = \mathsf{false}$ and `None` otherwise. Deciding repairability according to Eq. 2 is computationally hard or, depending on $\mathbb{D}$, D_{ex} and D_{co}, even undecidable. The reason is the quantifier alternation. Therefore, we check repairability only for a given set $J \subseteq 2^{(\mathbb{D}^A)}$ of input vectors. That is, instead of $\mathsf{repairable}(Q)$ we compute

$$\mathsf{repairable}'(Q) \Leftrightarrow \bigwedge_{\overline{v_i} \in J} \exists \bar{r} \,.\, \pi[\overline{v_i}||\bar{r}] \wedge \bigwedge_{r \in R} r = \mathrm{Org}(r)[\overline{v_i}||\bar{r}]$$

with $R = \{r \mid \mathrm{CmpOf}(r) \in Q\}$. We use only inputs that make P violate S because for all other inputs P is trivially repairable. When applying concolic execution for program analysis, such concrete input values are computed anyway. Using symbolic execution, path conditions can be solved to obtain values for J.

The quantifier-free part of $\mathsf{repairable}'$ is in D_{co}. Therefore, a query $\mathsf{repairable}'(Q)$ can be solved using one satisfiability check per input vector. An alternative is to swap the conjunction over the inputs with the quantification, rename all repair symbols to fresh ones for every conjunct, and use only one satisfiability check. In more detail, this works as follows. Let $\overline{r_i}$ be the vector of fresh symbols corresponding to $\bar{r}$ for input $\overline{v_i}$, and let r_i be the fresh symbol corresponding to symbol r in $\bar{r}$. We now have that $\mathsf{repairable}'(Q)$ is true iff

$$\bigwedge_{\overline{v_i} \in J} \pi[\overline{v_i}||\overline{r_i}] \wedge \bigwedge_{r \in R} r_i = \mathrm{Org}(r)[\overline{v_i}||\overline{r_i}] \quad (3)$$

is satisfiable.

The performance of Reiter's algorithm increases if the computed conflicts are minimal. A minimal conflict not containing a certain set N of elements can be computed in different ways. One way is to use a failure-preserving minimization algorithm like Delta Debugging [33] or QuickExplain [22] to repeatedly invoke $\mathsf{repairable}'$ with different subsets of $\mathsf{CMP} \setminus N$ until a minimal subset for which $\mathsf{repairable}'$ evaluates to false is found. Another option is based on the observation that every minimal conflict corresponds to an unsatisfiable core in Eq. 3. By rearranging the conjuncts, Eq. 3 can be rewritten to

$$\left(\bigwedge_{\overline{v_i} \in J} \pi[\overline{v_i}||\overline{r_i}]\right) \wedge \bigwedge_{c \in Q} \bigwedge_{\{r \mid \mathrm{CmpOf}(r) = c\}} \bigwedge_{\overline{v_i} \in J} r_i = \mathrm{Org}(r)[\overline{v_i}||\overline{r_i}].$$

This illustrates that every component $c \in Q$ corresponds to a certain conjunct in the definition of $\mathsf{repairable}'$. A minimal conflict not containing a certain set N of components can therefore be computed as a minimal unsatisfiable core of a

constraint system with $\bigwedge_{\overline{v_i} \in J} \pi[\overline{v_i}||\overline{r_i}]$ as a fixed part and

$$\bigcup_{c \in (\mathsf{CMP} \setminus N)} \left(\bigwedge_{\{r | \mathrm{CmpOf}(r)=c\}} \bigwedge_{\overline{v_i} \in J} r_i = \mathrm{Org}(r)[\overline{v_i}||\overline{r_i}] \right)$$

as a set of retractable constraints. Hence, if the solver is able to compute unsatisfiable cores, this feature can be exploited to compute minimal conflicts more efficiently.

Theorem 2. *Every diagnosis Δ with respect to the definition of* $\mathsf{repairable}$ *is also a diagnosis with respect to* $\mathsf{repairable}'$.

Proof: Clearly, we have that $\mathsf{repairable}(Q)$ implies $\mathsf{repairable}'(Q)$ for all $Q \subseteq \mathsf{CMP}$. ■

Theorem 2 states that using $\mathsf{repairable}'$ instead of $\mathsf{repairable}$ can only lead to false positives but not to missing diagnoses.

E. Error Correction

Our method for error correction takes as input an incorrect program P, the diagnostic data $\Gamma = (\mathsf{CMP}, \mathrm{Vars}, \overline{i}, \overline{r}, \mathrm{CmpOf}, \mathrm{Org}, \mathrm{Vals}, \pi[\overline{i}||\overline{r}])$, and a diagnosis $\Delta \subseteq \mathsf{CMP}$. If successful, it produces a repaired program P' which differs from P only in the components Δ. Assuming that program analysis was perfectly accurate, P' cannot violate its specification for any input. The focus of our algorithm is on efficiency and readability of repairs rather than completeness.

New expressions have to be synthesized for all components $c \in \Delta$. We reduce the search for expressions to the search for constants by creating templates for unknown expressions. Templates consists of program variables and template parameters. Concrete parameter values define a concrete expression.

Example 6. *The template $k_0{+}k_1{\cdot}\mathtt{v1}{+}k_2{\cdot}\mathtt{v2}$, where k_0, k_1, k_2 are parameters and* `v1`, `v2` *are program variables, can express any linear expression over the variables. The values $k_0{=}{-}2$, $k_1{=}1$, and $k_2{=}0$ represent expression* `v1-2`.

Templates also provide control over the expressions subjected to search. To get simple repairs, it makes sense to start with simple templates and switch to more expressive templates if no repair is found with the simple ones.

Formally, for every component $c \in \Delta$, we create a template $T_c[\overline{k_c}||\overline{\mathtt{pv}_c}] \in D_{ex}$ as an expression over two vectors of fresh symbols $\overline{k_c} \in \mathbb{S}^*$ and $\overline{\mathtt{pv}_c} \in \mathbb{S}^{|\mathrm{Vars}(c)|}$. The symbols $\overline{\mathtt{pv}_c}$ represent the values of the program variables in scope when component c is executed. Symbols in $\overline{k_c}$ represent unknown parameter values. We write $\overline{k}$ for the concatenation of all $\overline{k_c}$ with $c \in \Delta$. Moreover, we define $K_c = |\overline{k_c}|$ and $K = |\overline{k}|$.

For all components $c \in \Delta$, let $\overline{v_{k,c}} \in \mathbb{D}^{K_c}$ be concrete values for the template parameters $\overline{k_c}$, and let $\overline{v_k}$ be the concatenation of all $\overline{v_{k,c}}$. We write $P' = \mathsf{apply}(\overline{v_k}, P)$ to denote that program P is transformed to program P' by replacing all components $c \in \Delta$ with expression $T_c[\overline{v_{k,c}}||\mathrm{Vars}(c)]$. That is, in all templates, parameters are replaced by the values defined in $\overline{v_k}$, program variable symbols are replaced by the respective variable names, and components $c \in \Delta$ of P are replaced by the so instantiated templates.

In order to check if a certain template instantiation yields a correctly repaired program, we define a function $\mathsf{correct} : \mathbb{D}^A \times \mathbb{D}^K \to \{\mathsf{true}, \mathsf{false}\}$ such that $\mathsf{correct}(\overline{i}, \overline{k})$ is true iff

$$\exists \overline{r} \,.\, \pi[\overline{i}||\overline{r}] \wedge \bigwedge_{r \notin R} r = \mathrm{Org}(r)[\overline{i}||\overline{r}] \wedge$$
$$\bigwedge_{r \in R} \exists \overline{\mathtt{pv}_c} \,.\, r = T_c[\overline{k_c}||\overline{\mathtt{pv}_c}] \wedge \overline{\mathtt{pv}_c} = \mathrm{Vals}(r), \quad (4)$$

where c is short for $\mathrm{CmpOf}(r)$ and $R = \{r \mid \mathrm{CmpOf}(r) \in \Delta\}$. The intuition behind Eq. 4 is as follows. $\pi[\overline{i}||\overline{r}]$ expresses when $\tilde{P}$ behaves correctly, depending on the unknown inputs $\overline{i}$ and the unknown values $\overline{r}$ returned by the components. Every symbol r that has been produced by an incorrect component $c \in \Delta$ is bound to the value that would be produced by the corresponding template T_c. This value is obtained by binding the symbols $\overline{\mathtt{pv}_c}$ to the values $\mathrm{Vals}(r)$ the program variables had when c was executed to produce r (the equality is meant element-wise). Every symbol r produced by a correct component $c \notin \Delta$ is bound to the value $\mathrm{Org}(r)$ that would have been produced by the unmodified component c.

Lemma 3. *Let P be an incorrect program, $\overline{v_i} \in \mathbb{D}^A$ be an input vector, and $\overline{v_k} \in \mathbb{D}^K$ be template parameter values. Then,* $\mathsf{correct}(\overline{v_i}, \overline{v_k})$ *maps to* true *iff the program $P' = \mathsf{apply}(\overline{v_k}, P)$ fulfills the specification S when executed with input $\overline{v_i}$.*

Proof: Let $\overline{v_r} \in \mathbb{D}^B$ be the concrete values returned by calls to `cmp` in $\tilde{P}$. According to Lemma 1, $\pi[\overline{v_i}||\overline{v_r}]$ evaluates to true iff $\tilde{P}$ fulfills S for input $\overline{v_i}$. The additional conjuncts in Eq. 4 make $\mathsf{correct}$ map to true iff a special version $\tilde{P}'$ of $\tilde{P}$ satisfies S for input $\overline{v_i}$. In $\tilde{P}'$, all components $c \notin \Delta$ return the same value as the original implementation of c in P. All components $c \in \Delta$ return the values that would have been returned by template T_c, instantiated with parameters defined in $\overline{v_k}$. This program $\tilde{P}'$ is exactly $P' = \mathsf{apply}(\overline{v_k}, P)$. ■

Theorem 3. *Let $\overline{v_k}$ be a vector of concrete templates values such that* $\mathsf{correct}(\overline{v_i}, \overline{v_k})$ *holds for all input vectors $\overline{v_i}$. Then, $P' = \mathsf{apply}(\overline{v_k}, P)$ is a correct program.*

Proof: Lemma 3 implies that P' cannot violate its specification S for any input. Hence, P' is correct. ■

F. Computation of Repairs

This section explains how repairs can be computed following Theorem 3. Observe that all quantified variables are bound to a value in Eq. 4. Therefore, an equivalent condition $\mathsf{correct}'[\overline{i}||\overline{k}] \in D_{co}$ can be defined by replacing all quantified variables by their value. What remains is the implicit quantifier alternation $\left(\exists \overline{k} \,.\, \forall \overline{i} \,.\, \mathsf{correct}'[\overline{i}||\overline{k}]\right)$ in Theorem 3, which renders the problem intractable or even undecidable. For error localization, we handled this issue by requiring correctness for some inputs only. Here, we avoid false positives. We follow the idea of [30] and [3] to compute repairs through iterative refinements that are guided by counterexamples.

The process is illustrated in Fig. 3. There is a database I of input vectors $\overline{v_i} \in \mathbb{D}^A$, which is initially empty. In every iteration, a repair candidate is computed in form of template

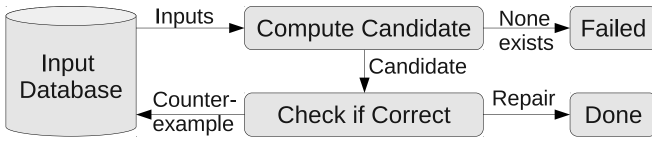


Fig. 3. Counterexample-guided repair refinement.

parameter values $\overline{v_k} \in \mathbb{D}^K$ such that $P' = \mathsf{apply}(\overline{v_k}, P)$ is correct for all inputs $\overline{v_i}$ in I. This is done by computing a satisfying assignment $\overline{v_k}$ for the symbols $\overline{k}$ in condition

$$\bigwedge_{\overline{v_i} \in I} \mathsf{correct}'[\overline{v_i}||\overline{k}]. \quad (5)$$

If Eq. 5 is unsatisfiable, the program cannot be repaired with the given templates and the procedure aborts. Otherwise, it is checked if $\overline{v_k}$ repairs the program for *all* inputs, i.e., if

$$\neg\, \mathsf{correct}'[\overline{i}||\overline{v_k}] \quad (6)$$

is unsatisfiable. If so, then $P' = \mathsf{apply}(\overline{v_k}, P)$ is a correct program and we are done. Otherwise, a satisfying assignment $\overline{v_i}$ for $\overline{i}$ in Eq. 6 is extracted. This $\overline{v_i}$ is a counterexample for the correctness of P'. It is added to I and another iteration is started, which produces a better candidate. This is repeated. We limit the number of iterations to ensure termination. If further repairs should be computed, we add conjuncts to Eq. 5 requiring that $\overline{k}$ is different to all previously computed repairs.

Example 7. *Let $\Delta = \{c_2\}$ be the diagnosis for the program P from Example 2, and let $k_0+k_1\cdot x+k_2\cdot y+k_3\cdot r$ be the template for c_2. We have that $correct((X,Y),(k_0,k_1,k_2,k_3)) =$*

$$\exists R_1, R_2\,.((Y > X \wedge R_2 \geq X \wedge R_2 \geq Y) \vee (Y \leq X \wedge R_1 \geq X \wedge R_1 \geq Y)) \wedge R_1 = X \wedge R_2 = k_0 + k_1 \cdot X + k_2 \cdot Y + k_3 \cdot X,$$

which can be simplified to $correct'[(X,Y,k_0,k_1,k_2,k_3)] =$

$$(Y \leq X) \vee (k_0 + k_1 \cdot X + k_2 \cdot Y + k_3 \cdot X \geq Y).$$

The computation of a repair could proceed as following. First, a satisfying assignment for $\overline{k} = (k_0,k_1,k_2,k_3)$ in Eq. 5 is computed with $I = \emptyset$. A possible solution is $\overline{k} = (0,0,0,0)$, which corresponds to the expression "`0`". Next, it is checked if replacing c_2 by `0` renders P correct. This is done by checking Eq. 6 for satisfiability, i.e., by searching for a counterexample. Eq. 6 is equal to $\neg(Y \leq X \vee 0 \geq Y)$, a satisfying assignment is $X = 2$, $Y = 4$. Hence, replacing c_2 by `0` does not repair P for all inputs. The database of inputs is extended to $I = \{(2,4)\}$, and an improved candidate expression is computed by solving Eq. 5, which is now equal to $(4 \leq 2) \vee (k_0 + k_1 \cdot 2 + k_2 \cdot 4 + k_3 \cdot 2 \geq 2)$. A solution is $\overline{k} = (1,0,1,0)$, which corresponds to the expression "`y+1`". Again, we verify if replacing c_2 by `y+1` renders P correct. Now, Eq. 6 is equal to $\neg(Y \leq X \vee 1 + Y \geq Y)$ and hence unsatisfiable. This means that no more counterexample exists. Replacing c_2 by `y+1` is a valid repair, the algorithm terminates.

G. Heuristics to Speed Up Convergence

Repair refinement can be seen as a game with two players. Player 1 comes up with candidates, Player 2 attempts to disprove them. In our experiments, we discovered two problems of this procedure. First, even if simple repairs exist, the play may end up computing and excluding more and more complex candidates. E.g., for one program, the sequence of candidates

Iteration	Candidate for a certain component
1	`0`
2	`-v0`
3	`250*v1 + 248*v2 - 2*v3 - v4`
4	and so on, becoming more and more complex

was observed, although the constant `500` was a repair for that component. Second, if both players do the least to fulfill their duty, progress may be insufficient. E.g., for the program in Example 2 with $\Delta = \{c_2\}$, the following may happen:

Repair candidate for c_2	Counterexample
`0`	`x`=0, `y`=1
`1`	`x`=1, `y`=2
`2`	and so on

We solve these two issues heuristically by improving the two players. Intuitively, we want "simple" candidates and "nasty" counterexamples. We say that a candidate is "simple" if many template parameters k_i are small or, even better, equal to some special value s_i, which makes terms in the template disappear (e.g., zero in case of a template for linear expressions). To implement this, we define a set $\rho_1 \subseteq 2^{D_{co}}$ of constraints as

$$\rho_1 = \bigcup_{k_i \in \overline{k}} \big(k_i = s_i\big) \cup \bigcup_{k_i \in \overline{k}} \big(k_i \leq M \wedge k_i \geq -M\big),$$

where M is a constant defining what "small" means. We compute template parameters by solving a Maximum Satisfiability (MAX-SAT) problem with Eq. 5 as fixed part and ρ_1 as the set of retractable constraints from which as many as possible should be fulfilled. Likewise, we say that a counterexample is "nasty" if it contains large, uncorrelated values. Again, we formulate a MAX-SAT problem with Eq. 6 as fixed part and

$$\rho_2 = \bigcup_{i_a \in \overline{i}} \big(i_a \geq N \vee i_a \leq -N\big)$$

as the set of retractable constraints, where N is a constant which is much larger than M. In order to break correlations between values in the counterexample we additionally randomize it: values are changed to large random values as long as the modified input vector is still a counterexample.

IV. Discussion and Alternatives

Our debugging method offers a lot of configuration parameters. This includes the number of execution paths to analyze, the number $|J|$ of inputs for diagnosis, the maximum number of repair refinements, and the templates to use. Moreover, the domains D_{ex} and D_{co} (i.e., the SMT-theories) determine which language constructs can be handled exactly, and which ones have to be approximated. As an advantage, our method

can by tailored to a broad range of programs. On the other hand, it may take some attempts to find a good configuration.

Our debugging method acts conservatively in that it targets a known good program termination for every input. The reason is the way $\pi[\bar{i}||\bar{r}]$ is defined in Eq. 1. An alternative is to use

$$\pi[\bar{i}||\bar{r}] = \neg \bigvee_{p \in \mathrm{FAIL}} \mathrm{PC}^p[\bar{i}||\bar{r}], \quad (7)$$

to avoid known specification violations. Using Eq. 1, diagnoses and repairs may be missed. Using Eq. 7, we may find false positives and allow endless loops. Both have their merits.

The more calls to `cmp` are introduced during pre-processing, the more execution paths become feasible. This observation can be exploited to refine the diagnostic data for error correction: A separate symbolic execution pass can be triggered for every diagnosis Δ before doing correction. In this pass, only the components in Δ are instrumented with calls to `cmp`. This gives higher path coverage for repair at the costs of having an additional program analysis step per diagnosis.

In principle, the fault model can be extended to include also faults in the LHS of assignments and even to missing or additional statements. A naive way is to apply case-splitting, but this is computationally expensive. More clever methods are subject to future work.

In first experiments, we observed that the quality of the produced repairs heavily depends on the quality of the given specification. This is neither surprising, nor is this problem specific to our method. It can happen that the computed correction simply prevents executions from ever reaching specific assertions. We plan to address this issue in the future by incorporating additional requirements such as the avoidance of unreachable code.

V. Experimental Results

In this section, we present first experimental results to demonstrate the feasibility of our approach. We implemented our debugging method for C programs. For program analysis we extended CREST [2], a concolic testing tool. Yices version 1.0.28 [10] is utilized with linear integer arithmetic as SMT-solver. Supporting other solvers and theories, especially bit-vectors and arrays, is planned. Thus, arrays and pointers are only handled approximatively at the moment. Currently, we use only templates for linear expressions. For expressions which occur as a condition in the program, we use templates of the form $k_0 + k_1 \cdot \mathtt{v1} + k_2 \cdot \mathtt{v2} + \ldots\ \mathtt{OP}\ 0$, where $\mathtt{v1}, \mathtt{v2}, \ldots$ are program variables, $k_0, k_1, \ldots$ are template parameters, and $\mathtt{OP} \in \{=, <, >, \leq, \geq\}$. The unknown comparison operator is encoded symbolically so that it can be handled like any other template parameter. Our implementation is part of a larger tool named FoREnSiC, which is under development and will feature also other formal, semi-formal, and dynamic debugging methods.

In our experiments, we set $|J| = 2$ for error localization, we limited the number of repair refinements to 10, the number of repairs to compute per diagnosis to 5, and set a time-out to all SMT-solver calls to 60 seconds. The experiments were performed on an Intel P7350 processor with 2×2.0 GHz and 3 GB RAM, running a 32-bit Linux. The implementation and scripts to reproduce the results are available for download[1].

TABLE I
Performance Results.

Column	1	2	3	4	5	6	7	8	9
	Diagnosis		Repair		No Heuristic		Sketch (8 bit)		
	Time	$\|\{\Delta\}\|$	Time	Found	Time	Found	Template Variables	Time	Memory
	[sec]	[-]	[sec]	[-]	[sec]	[-]	[-]	[s]	[MB]
tcas2	70	2	271	2	355	1	1	29	297
tcas7	123	2	42	5	444	0	4	20	1459
tcas8	122	2	37	5	465	0	1	6.3	42
tcas16	123	2	43	5	1027	0	2	22	764
tcas17	125	2	41	5	234	0	2	11	666
tcas18	124	2	38	5	35	0	2	9.3	744
tcas19	123	2	40	5	691	0	1	6.3	42
tcas36	3.0	0	-	-	-	-	4	7.0	796
total	813	14	512	32	3250	1		111	4810

Performance results

Table I summarizes performance results for some mutants of the TCAS program from the Siemens suite [9]. The program implements a traffic collision avoidance system for aircrafts in about 180 lines of code. We use the reference implementation as specification, which effectively doubles the size. For the TCAS examples, we do not consider conditions as components because they appear to challenge the solver, resulting in time-outs for many cases. We will try to overcome this issue by improving the encoding of the symbolic search for comparison operators, and by switching to a more recent solver. Consequently, Table I contains only mutants where the error is on the RHS of an assignment. We limited the number of paths to analyze with concolic execution to 400.

In every mutant of Table I, 44 components were identified. Program analysis took about 5 seconds. The time for diagnosis is listed in Column 1. Column 2 gives the number of diagnoses found. The Columns 3 and 4 show the error correction time and the number of found repairs with the heuristic of Section III-G enabled. The Columns 5 and 6 contain the same information for the heuristic being disabled. The Columns 7 to 9 summarize a comparison of our repair method with the program sketching tool Sketch [30][2]. We re-implemented the TCAS mutants in the input language of Sketch. Then, we manually replaced the faulty components with repair templates, using holes for the unknown template parameters. In this setting, Sketch ran out of memory for all cases. In order to have Sketch find a repair, we had to reduce the bit-width of an integer to 8 (for which we had to lower

[1] See http://www.iaik.tugraz.at/content/research/design_verification/others/. An official release of FoREnSiC will follow.

[2] We used Sketch version 1.3.0 with the solver ABC. When using MiniSat as a solver, the tool run out of memory.

constants in the program). Moreover, we had to reduce the number of program variables in the templates. Column 7 gives the maximum number of template variables so that Sketch can still find a repair. The last two columns list the time and memory requirements of Sketch, respectively.

In our experiments, we observed that a low number of inputs J (we use only two) is sufficient for our method to yield precise diagnoses. (See Column 2). Only for `tcas36`, no diagnosis could be found. The reason is that (by far) not all execution paths through the pre-processed program were analyzed. However, with other parameter configurations (e.g., using Eq. 7 instead of Eq. 1 and an extra program analysis pass per diagnosis; cf. Section IV) our method finds 5 repairs also for this mutant. The time for error localization is rather high compared to error correction (Column 1 vs. Column 3). This may be due to an inefficient implementation: we do not yet utilize unsatisfiable core functionality of the solver to compute minimal conflicts, as described in Section III-D. The program has many global variables, so each repair may depend on many variables. Nevertheless, error correction is surprisingly fast in our experiments. Furthermore, our heuristic to improve convergence in repair computation works well. It leads to more repairs being found in less time. Our tool is able to check a repaired program for correctness using the model checker CBMC [4]. This was successful in all cases.

At least for the analyzed TCAS examples, our repair method seems to perform better than Sketch. The repair templates used by our tool contain all variables which are in scope at the respective location in the program. This means at least 10 program variables and 11 template parameters for each template. For Sketch, we had to drastically reduce the number of program variables in repair templates in order to obtain a repair. (See Column 7.) Moreover, the memory requirements of our implementation are insignificant (below 80 MB in all cases). A plausible explanation is that Sketch breaks the synthesis problem down to Boolean satisfiability problems, while we use an SMT-solver. Furthermore, our tool did not analyze all execution paths of the TCAS mutants. Note, however, that the comparison with Sketch is not totally fair due to different input languages, solvers, and tool objectives.

Analysis of some Repairs

In this section, we take a closer look on the repair process for some programs. We start with our running example (cf. Example 2). For CMP=$\{c_1, c_2\}$, our tool identifies $\{c_2\}$ as the only diagnosis. The expressions `y`, `y+1`, `y+2`, etc., are computed as possible replacements of c_2. If the condition is considered as a third component c_3 (cf. Example 2) our tool finds the diagnoses $\{c_2\}$ and $\{c_1, c_3\}$. The former is repaired as before. For the latter, our tool computes the replacements

c_1	c_3
`y`	`x - y >= 0,`
`y+1`	`2*x - 2*y > 0,`
`y+2`	`3*x - 3*y > 0,`
`y+3`	`-x + y < 0`, and
`y+4`	`4*x - 4*y >= 0.`

In the mutant `tcas2` from Table I, the function

```
1 InhibitBiasedClimb() {
2   return (ClimbInhibit ? UpSep +
        NOZCROSS : UpSep);
3 }
```

has been modified: The constant `NOZCROSS` = `100` has been replaced by the constant `MINSEP` = `300`. The front-end of CREST simplifies the body of this function to:

```
1   if (ClimbInhibit) {
2     tmp = UpSep + 300;
3   } else {
4     tmp = UpSep;
5   }
6   return (tmp);
```

Our tool identifies the RHS of Line 2 as a diagnosis. For this diagnosis, the following repair candidates are computed.

Iteration	Candidate expression	Correct
1	`0`	no
2	`UpSep`	no
3	`UpSep + 100`	yes
4	`2*UpSep + 101`	no
5	`UpSep + 99`	no
6	`OtherTrAlt + UpSep + 99`	no
7	`-DwnSep + 2*UpSep + 199`	yes

Finally, the repair process aborts due to a time-out. The repair of Iteration 3 corresponds to the original program and is thus correct. The one found in Iteration 7 is correct because `InhibitBiasedClimb()` is only used in comparisons of the form `InhibitBiasedClimb() > DwnSep`. Since `UpSep` and `DwnSep` are integer variables, `-DwnSep + 2*UpSep + 199 > DwnSep` is true iff `UpSep + 100 > DwnSep` is true.

In the mutant `tcas18` from Table I, the statement

```
1 PosRAAltThresh[2] = 640;
```

has been modified by replacing the constant `640` with `640+50`. The RHS of this assignment is among the computed diagnoses. Our tool computes the following sequence of repair candidates for this diagnosis.

Iteration	Candidate	Correct
0	`0`	no
1	`400`	no
2	`500`	no
3	`640`	yes
4	`-OwnTrackedAltRate + 639`	no
5	`UpSep - 1`	no
6	`UpSep - 1000`	no
7	`-OwnTrackedAlt - 1`	no
8	`AltLayerValue + 638`	yes
9	`-AltLayerValue + 642`	yes
10	`2*AltLayerValue + 636`	yes
11	`-2*AltLayerValue + 644`	yes

The repair computed in Iteration 3 corresponds to the original program. The repairs found in the iterations 8 to 11 render the

program correct because the array `PosRAAltThresh` is only read at index `AltLayerValue`. Hence, the modification in `tcas18` affects the behavior only for `AltLayerValue` = 2. For this case, the expressions computed in the iterations 8 to 11 are equal to `640`. Thus they render the program correct.

These examples demonstrate that our method is able to find nontrivial corrections also for nontrivial programs.

VI. Conclusion

In this paper, we presented a novel method for automatic error localization and correction in imperative programs. It offers a wide range of different trade-offs between accuracy and resource requirements. Our method is based on symbolic execution, abstracting the debugging problem into the domain of logic. We showed how model-based diagnosis can be applied to locate errors using this abstraction. Our correction method is based on templates, a technique borrowed from the field of synthesizing loop invariants. This ensures that repairs are readable. We compute repairs with iterative refinements and presented a heuristic to speed this process up. This heuristic additionally prefers simple repairs. We implemented our debugging method for C programs. Although the implementation is still in a proof-of-concept state, experimental results demonstrate that the method works and can be used not just for toy examples.

In the future, we plan to investigate extensions of the fault model, develop methods to obtain more useful repairs for sketchy specifications, combine our method with other debugging approaches, and extend our tool to support more theories and solvers.

References

[1] A. Arcuri. On the automation of fixing software bugs. In *30th International Conference on Software Engineering (ICSE'08)*, pages 1003–1006. ACM, 2008.

[2] J. Burnim and K. Sen. Heuristics for scalable dynamic test generation. In *23rd International Conference on Automated Software Engineering (ASE'08)*, pages 443–446. IEEE, 2008.

[3] K.-H. Chang, I. L. Markov, and V. Bertacco. Fixing design error with counterexamples and resynthesis. In *Asia and South Pacific Design Automation Conference (ASP-DAC'07)*, pages 944–949, 2007.

[4] E. M. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'04)*, pages 168–176, 2004.

[5] L. A. Clarke. A system to generate test data and symbolically execute programs. *IEEE Trans. Software Eng.*, 2(3):215–222, 1976.

[6] M. Colón, S. Sankaranarayanan, and H. Sipma. Linear invariant generation using non-linear constraint solving. In *Proc. Computer Aided Verification (CAV'03)*, pages 420–432. Springer, 2003. LNCS 2725.

[7] L. Console, G. Friedrich, and D. Theseider Dupré. Model-based diagnosis meets error diagnosis in logic programs. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI'93)*, pages 1494–1499. Morgan-Kaufmann, 1993.

[8] V. Debroy and W. E. Wong. Using mutation to automatically suggest fixes for faulty programs. In *Third International Conference on Software Testing, Verification and Validation (ICST'10)*, pages 65–74. IEEE, 2010.

[9] H. Do, S. G. Elbaum, and G. Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering*, 10(4):405–435, 2005.

[10] B. Dutertre and L. M. de Moura. A fast linear-arithmetic solver for DPLL(T). In *Proc. Computer Aided Verification (CAV'06)*, pages 81–94. Springer, 2006. LNCS 4144.

[11] A. Felfernig, G. Friedrich, D. Jannach, and M. Stumptner. Consistency-based diagnosis of configuration knowledge bases. *Artificial Intelligence*, 152:213–234, 2004.

[12] S. Forrest, T. Nguyen, W. Weimer, and C. Le Goues. A genetic programming approach to automated software repair. In *Genetic and Evolutionary Computation Conference (GECCO'09)*, pages 947–954. ACM, 2009.

[13] G. Friedrich and K. M. Shchekotykhin. A general diagnosis method for ontologies. In *International Semantic Web Conference*, pages 232–246. Springer, 2005. LNCS 3729.

[14] G. Friedrich, M. Stumptner, and F. Wotawa. Model-based diagnosis of hardware designs. *Artificial Intelligence*, 111(1-2):3–39, 1999.

[15] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *Conference on Programming Language Design and Implementation (PLDI'05)*, pages 213–223. ACM, 2005.

[16] R. Greiner, B. A. Smith, and R. W. Wilkerson. A correction to the algorithm in Reiter's theory of diagnosis. *Artificial Intelligence*, 41(1):79–88, 1989.

[17] A. Griesmayer, R. Bloem, and B. Cook. Repair of Boolean programs with an application to C. In *18th Conference on Computer Aided Verification (CAV'06)*, pages 358–371, 2006. LNCS 4144.

[18] A. Griesmayer, S. Staber, and R. Bloem. Fault localization using a model checker. *Software Testing, Verification and Reliability*, 20(2):149–173, 2010.

[19] B. Jobstmann, A. Griesmayer, and R. Bloem. Program repair as a game. In *17th Conference on Computer Aided Verification (CAV'05)*, pages 226–238. Springer, 2005. LNCS 3576.

[20] B. Jobstmann, S. Staber, A. Griesmayer, and R. Bloem. Finding and fixing faults. *Journal of Computer and System Sciences*, 2011. In Press.

[21] M. Jose and R. Majumdar. Cause clue clauses: error localization using maximum satisfiability. In *Conference on Programming Language Design and Implementation (PLDI'11)*, pages 437–446. ACM, 2011.

[22] U. Junker. QUICKXPLAIN: Preferred explanations and relaxations for over-constrained problems. In *Proc. National Conference on Artificial Intelligence (AAAI'04)*, pages 167–172. AAAI Press/MIT Press, 2004.

[23] J. C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[24] J. de Kleer and B. C. Williams. Diagnosing multiple faults. *Artificial Intelligence*, 32:97–130, 1987.

[25] R. Koenighofer, G. Hofferek, and R. Bloem. Debugging unrealizable specifications with model-based diagnosis. In *Proc. Haifa Verification Conference (HVC'10)*, pages 29–45. Springer, 2010. LNCS 6504.

[26] C. Mateis, M. Stumptner, D. Wieland, and F. Wotawa. Model-based debugging of Java programs. In *Proc. Fourth International Workshop on Automated Debugging (AADEBUG'00)*, 2000.

[27] R. Reiter. A theory of diagnosis from first principles. *Artificial Intelligence*, 32:57–95, 1987.

[28] K. Sen, D. Marinov, and G. Agha. CUTE: A concolic unit testing engine for C. In *Proc. 10th European Software Engineering Conference and 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE'05)*, pages 263–272. ACM, 2005.

[29] A. Solar-Lezama. The sketching approach to program synthesis. In *Programming Languages and Systems, 7th Asian Symposium (APLAS'09)*, pages 4–13. Springer, 2009. LNCS 5904.

[30] A. Solar-Lezama, L. Tancau, R. Bodik, V. Saraswat, and S. A. Seshia. Combinatorial sketching for finite programs. In *Proc. Architectural Support for Programming Languages and Operating Systems (ASPLOS'06)*, pages 404–415. ACM, 2006.

[31] S. Srivastava, S. Gulwani, and J. S. Foster. From program verification to program synthesis. In *Symposium on Principles of Programming Languages (POPL'10)*, pages 313–326. ACM, 2010.

[32] M. Stumptner and F. Wotawa. Debugging functional programs. In *Proceedings on the 16th International Joint Conference on Artificial Intelligence (IJCAI'99)*. Morgan Kaufmann, 1999.

[33] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 28(2):183–200, 2002.

Optimal Redundancy Removal without Fixedpoint Computation

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

***Abstract*— Industrial verification and synthesis tools routinely identify and eliminate redundancies from logic designs. In the former case, redundancy removal yields critical speedups to the overall verification process. In the latter case, redundancy removal constitutes a primary mechanism to optimize the final fabricated circuit. Redundancy identification frameworks often utilize a greatest-fixedpoint iteration, initially postulating a set of candidate redundancies to be conjunctively proved then refining candidates based upon failed proof attempts. Such procedures generally do not yield *any* soundly-proved redundancies until a fixedpoint is achieved. In this paper, we overcome this drawback by augmenting the fixedpoint procedure with a set of efficient techniques to track dependencies between candidate redundancies. This approach enables the identification of an optimal subset of valid redundancies before the fixedpoint is reached, and may also be used to reduce the number of computations within the fixedpoint procedure. We apply our techniques to enhance k-induction as well as a more general transformation-based verification flow. For induction, we demonstrate up to 75% reduction in runtime and 97% reduction in the number of inductive proofs. For the more general flow, we demonstrate up to 90% reduction in runtime and 80% reduction in the total number of proof obligations.**

I. Introduction

Industrial gate-level designs are often rife with redundancy. Logic synthesis tools attempt to eliminate redundant structure as a way of improving the area, delay, or power of the final fabricated circuit. Verification tools eliminate redundancy to reduce the size of the design under verification, often yielding dramatic speedups to the overall verification process, e.g. [1], [2]. In *equivalence checking* frameworks, internal equivalences between two designs can be viewed as a set of redundancies, which once identified and eliminated, effectively decompose an otherwise intractable monolithic problem for greater scalability.

Redundancy identification frameworks often operate through a greatest-fixedpoint iteration to yield a maximal set of equivalent gates which can be proved to assume identical values in every reachable state[1]. Such frameworks often postulate a superset of candidate equivalences, e.g. identified using simulation signatures or structural heuristics, then iteratively attempt to prove the conjunction of this postulated set. Any inaccurate or unprovable equivalences are discarded, and the process repeats until a fixedpoint is achieved. The benefit of the fixedpoint procedure is that it enables cross-leveraging postulated equivalences, i.e., *assuming* one set of postulated equivalences when *proving* another. This often yields dramatic speedups, e.g., through enabling inductive proofs of redundancy which otherwise may require reachability analysis [3], [4], [1]. *Speculative reduction* may leverage assumptions to further reduce proof complexity by *merging* fanout references of postulated-equivalent gates, trivializing many proof obligations and simplifying the remainder [2], [5]. The drawback of cross-leveraging equivalences in this manner is that until a fixedpoint is achieved, no redundancy may be inferred because any successfully-completed equivalence proofs may be jeopardized by inaccurate candidate equivalences.

k*-Induction* is commonly used to scalably prove candidate equivalences [3], [1]. k-Induction first validates the *base case* by checking that the postulated equivalences hold on every state reachable in k or less steps from the initial states. Next, the *inductive step* validates that for all sequences of k consecutive states on which the postulated equivalences hold, they also hold in all successor states. If either check fails, the inaccurate or unprovable equivalences are discarded, and the fixedpoint process is repeated on the remaining equivalences.

More generally, redundancies may hold in a design which cannot be readily proved using induction. To identify such redundancies, one may need to leverage an arbitrary sequence of reduction, abstraction, and proof techniques to adequately simplify and ultimately prove postulated gate equivalences – often represented as verification properties termed *miters*. The use of verification-oriented transformations such as min-area retiming [6] and temporal decomposition [7] are particularly valuable in a redundancy removal framework, as they may eliminate structural differences between the logic being checked for equivalence. Speculative reduction is furthermore often critical to simplify the resulting set of proof obligations, both in enhancing the utility of other transformations and abstractions, as well as in simplifying the final proof obligation for a technique such as interpolation [8]. We refer to such a verification paradigm as *Transformation-Based Verification (TBV)* [6]. As with induction, if any miter is falsified or unproved by a given TBV algorithm sequence, the corresponding equivalences must be discarded, and the fixedpoint process is repeated on the remaining equivalences.

In this work we address the optimal identification of redundancies in an assume-then-prove framework without requiring fixedpoint computations. In particular, we present efficient techniques to track proof dependencies within inductive and TBV-based redundancy identification frameworks. Our techniques enable the identification of a subset of true redundancies before the fixedpoint is reached, despite any

[1] *Constant gates* and *antivalent gates* may be identified using a straightforward extension of such frameworks.

Alg. 1 . Redundancy Removal Fixedpoint Algorithm

```
 1: function identifyRedundancyFixedpoint()
 2:    Postulate redundancy candidates, represented as equivalence classes
 3:    loop
 4:       Attempt to prove each redundancy candidate as accurate
 5:       if (all redundancy candidates are proved) then
 6:          return equivalence classes as redundancies that may be merged
 7:       else
 8:          refine the equivalence classes
 9:       end if
10:    end loop
11: end function
```

cross-leveraged assumptions. This has several benefits: **(1)** we can soundly identify redundancies even when resource limits prevent *every* candidate equivalence from being proved or disproved; **(2)** we reduce effort within a fixedpoint procedure by not requiring candidate equivalences to be repetitively proved across iterations; and **(3)** within each iteration of the fixedpoint computation, we allow the proofs of unsolved equivalences to be deferred or discarded when we detect that it is not possible to mark this redundancy as soundly proved.

Section II describes the preliminaries. Section III describes the Proof Graph, our datastructure which tracks dependencies among equivalences. Sections IV and V describe the integration of Proof Graph techniques in inductive and TBV frameworks, respectively. In Section VI we provide proofs that our techniques are sound and optimal. Lastly, we provide experimental results in Section VII.

II. Preliminaries

We assume that the design under analysis is represented as a gate-level netlist, consisting of combinational gates of various types as well as sequential elements with associated *initial values* and *next-state functions*. Our implementation uses an And/Inverter Graph [9], [10] format, though our techniques are applicable to other netlist formats as well. In a verification setting, the netlist may also comprise logic expressing environmental assumptions and correctness properties. In an equivalence checking setting, the netlist may represent the composition of two designs being compared, with safety properties checking pairwise equivalence of primary outputs.

Algorithm 1 illustrates a traditional redundancy removal fixedpoint algorithm [3], [1], [5]. Such algorithms first postulate a superset of redundancy candidates, represented as *equivalence classes* wherein all gates within the same class are postulated to behave identically in all reachable states. A set of safety properties termed *miters* is constructed which represent the underlying equivalences candidates, and a set of proof techniques is then used to establish the validity of the miters and the candidates they represent. If any candidates are demonstrated to be inaccurate, or if the chosen proof techniques cannot yield a conclusive result for some candidates, the equivalence classes are *refined* by discarding the unprovable equivalences. This process repeats until finally all equivalence classes are demonstrated correct. When this fixedpoint is reached, the netlist may be simplified by *merging* gates which are proved equivalent. In particular, within each

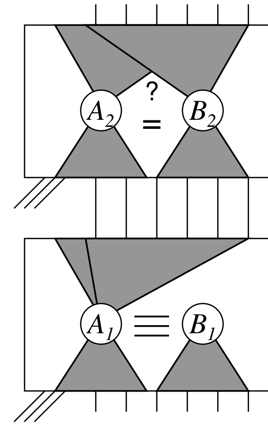


Fig. 1. Speculative reduction simplifies the unrolled netlist.

equivalence class, a *representative gate* is chosen, and each other gate in that equivalence class will be replaced with its representative in the netlist.

There are two fundamental techniques to enable the scalability of sequential redundancy removal. The first is the use of induction to establish the correctness of the *conjunction* of the postulated equivalences, which individually would often be non-inductive and require substantially more expensive proof techniques [3], [1]. Even if heavier-weight proof techniques are ultimately needed for maximal redundancy removal, conjunctive induction is often able to efficiently solve most of the proof obligations. The second is the use of *speculative reduction*, which reduces the size of the miter-annotated netlist by reconnecting the fanout of a given candidate equivalence gate (refer to gate B_1 in Figure 1) to its representative (gate A_1). This reduces the complexity of the logic in the fanout of the speculatively-merged gate and often trivializes downstream miters. Speculative reduction is capable of yielding orders of magnitude speedups in both inductive- and TBV-based approaches for redundancy removal [2], [5]. However, as a result of these two techniques, Algorithm 1 cannot generally be used to identify redundancies before a fixedpoint is reached, as one incorrect candidate may invalidate the soundness of the proof of the other candidates.

III. The Proof Graph

To deduce sound redundancies prior to achieving a fixedpoint, we record *dependencies* between candidate equivalences. Two sources of dependencies may arise in a sequential redundancy removal framework. **(1)** Speculative reduction may simplify the netlist under the assumption that $A \equiv B$, e.g. by merging the fanout B onto A as in Figure 1. If the merged gate B is in the *cone-of-influence (COI)* of some other postulated equivalence $C \equiv D$, then $C \equiv D$ depends on $A \equiv B$. **(2)** If using induction, the inductive hypothesis constrains the SAT solver to only explore state sequences where $A \equiv B$ and $C \equiv D$ on the first k time steps. If the solver utilizes these inductive hypotheses, then $A \equiv B$ depends on $C \equiv D$ and vice-versa. If $C \equiv D$ depends on $A \equiv B$ and we can

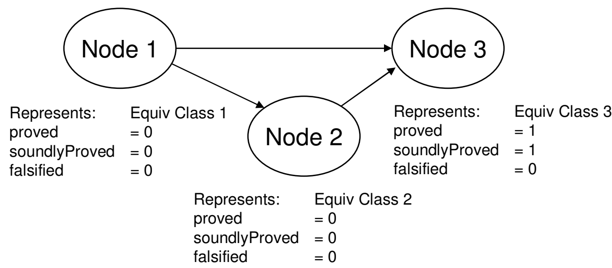


Fig. 2. An example Proof Graph

Alg. 2. Proof callback function

1: **function** informProved(*proofGraph*, *class*)
2: ensure that the *proofGraph* is condensed
3: *node* = the Proof Graph node containing *class*
4: *node.proved*[*class*] = 1
5: **if** ((∀ classes $C \in$ *node*, *node.proved*[C] == 1) **and** (∀ children D of *node*, D*.soundlyProved* == 1)) **then**
6: *node.soundlyProved* = 1
7: inform the calling application that *node*'s classes are soundly proved
8: **for all** parents P of *node*, P*.proved*[*] == 1 **do** informProved(P)
9: **end if**
10: **end function**

demonstrate that $A \not\equiv B$ then a proof of $C \equiv D$ does not soundly indicate that C and D are equivalent.

Dependencies are recorded in a directed graph called the *Proof Graph*. Each node in the Proof Graph represents a set of one or more equivalences. An edge $node_1 \rightarrow node_2$ represents the dependency $node_1$ "depends on" $node_2$. An example Proof Graph is shown in Figure 2.

We initially construct the Proof Graph to represent a single equivalence class per node. The resulting Proof Graph is cyclic in general, though we may render it acyclic without jeopardizing the optimality of identified redundancies in two ways. First, all *strongly connected components (SCCs)* [11] are identified, and the graph is *condensed* by collapsing the nodes in each SCC into a single Proof Graph node. Second, self-edges are suppressed. In this way, each Proof Graph node thus represents a set of equivalence classes.

We use the Proof Graph within a redundancy identification framework to identify when a proof represents a soundly-proved redundancy. In our algorithms we use three types of flags within the Proof Graph: **(1)** *proved* means that a given equivalence class has been proved relative to the other (possibly incorrect) redundancy candidates; **(2)** *soundlyProved* means that the proof of the corresponding equivalence class(es) is sound; and **(3)** *falsified* means that either this node contains a falsified equivalence, or it has a falsified dependency. A Proof Graph node has a single *soundlyProved* and *falsified* flag, and a *proved* flag for each equivalence class within that node. Because the topology of the Proof Graph depends upon the nature of the equivalence classes, the Proof Graph and its flags generally must be recomputed at each iteration of the fixedpoint Algorithm 1.

Algorithm 2 is called when all miters corresponding to a postulated equivalence class are proved. We set the *proved* flag on this class and conclude that this proof is sound iff all

Alg. 3. Falsification callback function

1: **function** informFalsified(*proofGraph*, *class*)
2: *node* = the Proof Graph node containing *class*
3: **if** (*node.falsified* == 1) **then return**
4: *node.falsified* = 1
5: **for all** parents P of *node* **do** informFalsified(*proofGraph*, P)
6: **end function**

classes in the same SCC are proved and all dependencies are soundly proved. Whenever Algorithm 2 deduces that a proof is sound, it recurses to the parents in the Proof Graph as the proofs of these parent classes may now be sound as well. As an example of Algorithm 2, if we call `informProved` on Class 2 of Figure 2 then we deduce that this proof is sound because all classes in Class 2's SCC are proved and the only dependency, Node 3, is soundly proved.

Algorithm 3 is called whenever an equivalence class is falsified. This sets the falsified flag on the corresponding Proof Graph node and propagates this flag to all ancestors in the Proof Graph. This flag is used to inform the higher-level algorithms that an equivalence class cannot be soundly proved and therefore need not be checked. As an example, calling `informFalsified` on Class 2 of Figure 2 will result in the falsified flag being set on Proof Graph Nodes 2 and 1. Node 1 can thereafter never be soundly proved, and the higher-level algorithms can use this information to forgo any attempts to prove the equivalences from Node 1.

Using the Proof Graph within a redundancy removal framework will not alter the set redundancies that are proved, as will be established in Theorem 2. Instead, the Proof Graph is used to improve the performance of the associated redundancy removal framework.

The Proof Graph is a general way to track dependencies. In this work, we apply this datastructure in the context of induction (Section IV) and TBV (Section V).

IV. Induction and the Proof Graph

In this section we enhance inductive redundancy identification frameworks using the Proof Graph. In induction, there are two types of dependencies that must be recorded in the Proof Graph: combinational structural dependencies, and proof dependencies.

An inductive proof unrolls the transition relation, performing speculative reduction to simplify the unrolled logic [2]. For example, in Figure 3A, the lower time frame will be simplified by assuming $A_1 \equiv B_1$ and $C_1 \equiv D_1$. If these assumptions are invalid, the behavior of the downstream logic may be altered, implying that downstream miters are dependent on these speculatively-reduced equivalences.

Algorithm 4 describes the process to infer such dependencies, called *combinational structural dependencies*. Given an equivalence class, the unfolding depth k used for induction, and a set of gates whose fanout was merged due to speculative reduction, we first mark the COI of all miters within the class. Then within this COI, we find gates that have been merged by speculative reduction. The given equivalence class will be

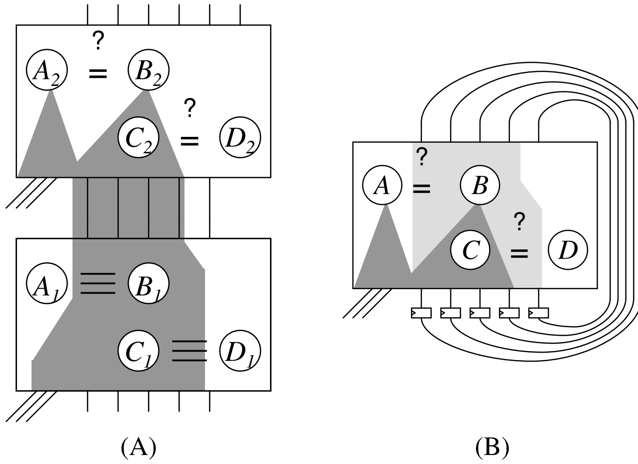


Fig. 3. (A) A combinational structural dependency, affecting induction. (B) A sequential structural dependency, affecting TBV.

Alg. 4. Discovery of combinational structure dependencies

1: **function** getCombStructureDeps(*class, k, specReduction*)
2: *coi* = ∅
3: **for all** *gate* in *class* **do**
4: *u* = unrolled instance *gate* in frame *k*
5: *coi* = *coi* ∪ combinational cone of influence of *u*
6: **end for**
7: **for all** *simplifiedGate* in *specReduction* ∩ *coi* **do**
8: *C* = equivalence class that spec-reduces *simplifiedGate*
9: record the dependency "*class* → *C*"
10: **end for**
11: **end function**

marked as dependent upon all classes responsible for these simplifications.

A second type of dependency arises from the inductive hypothesis. In k-induction we hypothesize that all equivalences hold at times $0, \ldots, k-1$. These hypotheses are typically implemented by passing additional constraints to the SAT solver, causing it to only explore paths for which the equivalences hold in the first k steps. If the proof of a miter depends on the inductive hypothesis, then the miter and its associated equivalence class have an additional dependency.

Algorithm 5 shows how to prove the miters from an equivalence class and extract the resultant dependencies, termed *proof dependencies*. A SAT solver is used to test the conjunction of all miters along with all inductive hypotheses. If the result is unsatisfiable, meaning the miters are proved, we extract an unsatisfiable core from the solver and inspect it to determine which hypotheses were utilized in the proof.

Alg. 5. Discovery of proof dependencies

1: **function** proveAndGetDeps(*class, hypotheses*)
2: *result* = SAT_solve($\bigwedge_{miter \in class}$ *miter* ∧ *hypotheses*)
3: **if** (*result* is "unsatisfiable") **then**
4: *core* = extract an unsatisfiable core from the SAT solver
5: **for all** *hyp* in *hypotheses* ∩ *core* **do**
6: *C* = equivalence class responsible for *hyp*
7: record the dependency "*class* → *C*"
8: **end for**
9: **end if**
10: **return** *result*
11: **end function**

Alg. 6. Determining the order in which to prove equivalences

1: **function** getProofObligations(*proofGraph*)
2: ensure that the *proofGraph* is condensed
3: *classesToProve* = ∅
4: **for all** condensed *node* ∈ *proofGraph*, *node.falsified* == 0 **do**
5: **if** ∀ children C of *node*, C.*soundlyProved* == 1 **then**
6: *classesToProve* = *classesToProve* ∪ {*node*'s classes}
7: **end if**
8: **end for**
9: **return** *classesToProve*
10: **end function**

Each hypothesis has an associated equivalence class C, and we record the dependence on each such C. Techniques to minimize the unsatisfiable core [12] may be employed to minimize these dependencies if desired.

Note that proof dependencies render the Proof Graph a dynamic datastructure when it is used for induction. Edges may be added after any single SAT call, and the topology of the Proof Graph can thus change. This is why Algorithm 2 may need to re-condense the Proof Graph.

With combinational structural dependencies and proof dependencies identified, we may use the Proof Graph techniques from Section III to reason about the equivalences that have been soundly proved in a single iteration of induction. Soundly-identified redundancies can be obtained despite inaccurate or unproved candidate equivalences, before the inductive fixedpoint is reached. This gives us partial results in the case that computational resources are exhausted before induction converges. In addition, if an equivalence is soundly proved during one induction iteration, the equivalence doesn't need to be re-tested during later induction iterations. As our experiments demonstrate, this dramatically reduces the number of SAT calls without jeopardizing the optimality of the final derived set of redundancies.

The Proof Graph can also be used to detect equivalence classes that cannot be soundly proved because they have a falsified dependency. We can skip these proof attempts during induction, further reducing the overall number of SAT calls without sacrificing the optimality of soundly proved equivalences as per Theorem 2.

Algorithm 6 may be used to derive an optimal ordering of equivalence classes to be proved by an induction framework. The induction framework repeatedly calls this function until no more equivalence classes need to be tested in the current induction iteration. The algorithm traverses the Proof Graph to look for nodes that are not falsified and have no unproved children. These represent the equivalence classes that if proved are most likely to yield sound equivalences, hence induction is directed to test these classes first. Because the Proof Graph is maintained to be acyclic, this algorithm is guaranteed to return a nonempty set of equivalence classes if any unsolved classes may yield a soundly-proved equivalence. Note that if Algorithm 3 sets the *falsified* flag, then induction will entirely skip any proof attempts for the corresponding candidate equivalences.

Alg. 7. Discovery of sequential structure dependencies

```
1: function getSeqStructureDeps(class, specReduction)
2:     coi = sequential cone of influence of all gate in class
3:     for all simplifiedGate in specReduction ∩ coi do
4:         C = equivalence class that spec-reduces simplifiedGate
5:         record the dependency "class → C"
6:     end for
7: end function
```

V. TBV AND THE PROOF GRAPH

Like induction, TBV can be used to prove that equivalences hold on every reachable state. Here the set of algorithms used to carry out a proof may be arbitrary. The netlist is transformed by adding miters for the suspected equivalences, speculatively reducing the (sequential) netlist, and passing this sub-problem to another user-specified algorithm or sequence of algorithms.

When the Proof Graph is used in a TBV context, there is only one type of dependency: those arising from speculative reduction. An example of this speculative reduction is shown in Figure 3B, where **(1)** the netlist is simplified assuming $A \equiv B$ and $C \equiv D$ by moving fanouts of A to B and fanouts of D to C, and **(2)** miters are added to test $A \equiv B$ and $C \equiv D$.

Algorithm 7 is used to extract speculative reduction dependencies, termed *sequential structure dependencies*, for TBV. This function is called once on each equivalence class, and it is passed the class and the set of gates merged using speculative reduction. The sequential COI of all miters in the class is marked, and simplifications within this COI are explored. For each simplification, a dependence on the associated class C is recorded.

As with Algorithm 6, it is advantageous to prove miters associated with leaves of the Proof Graph before attempting to prove other miters. In our implementation, we influence the proof order by associating assigning a priority to each miter. Additionally, we instruct downstream algorithms to skip proofs of miters associated with Proof Graph nodes that have the *falsified* flag set.

VI. SOUNDNESS AND OPTIMALITY

Our first theorem establishes the validity of any redundancy identified using our techniques.

Theorem 1 (Soundness): Any redundancy identified as "soundly proved" using the Proof Graph is valid.

Proof: If no speculative reduction or conjunctive induction is used within the underlying proof framework, the Proof Graph is unconnected hence this theorem trivially holds.

Speculative reduction may jeopardize the validity of a proof, since the corresponding fanout merge may alter netlist behavior if the corresponding postulated equivalence is incorrect. Note however that a speculative merge only may alter the behavior of gates in the fanout of the *merged* gate: not the *representative* onto which it was merged. Any miter in the fanout of this merged gate will have an associated edge in the Proof Graph, hence such fanout miters will be marked as *falsified* if the speculatively-merged gate is demonstrated inaccurate. Furthermore, no proved miter in the fanout of this speculatively-merged gate will be identified as "soundly proved" until the corresponding candidate equivalence is soundly proved and it is thereby guaranteed that the speculative merge does not alter netlist behavior. This theorem thus follows for speculative reduction given the results of [2], particularly that speculative reduction preserves the ability to identify invalid equivalences.

If using conjunctive induction, recall that a Proof Graph edge is added to any postulated equivalence upon which another equivalence proof is determined to rely. This will ensure that no proved equivalence will be identified as sound until the corresponding source of the necessary inductive hypothesis has been proved as accurate, thereby validating the soundness of using that hypothesis. ■

The following theorem establishes the optimality of the identified redundancies when using *fine-grained equivalence classes*, wherein each equivalence class contains a pair of gates: one to be merged onto the other representative. Coarser-grained equivalence classes are possible, though may trade reduction optimality for performance.

Theorem 2 (Optimality): Given fine-grained equivalence classes, the set of redundancies derived when using the Proof Graph is *optimal*. In particular, any proof discarded via use of the Proof Graph could not correlate to a soundly-identified redundancy under the chosen proof framework.

Proof: First consider the use of speculative reduction. Every miter in the fanout of a speculatively-merged gate will have an associated dependency identified in the Proof Graph, and thus will not be demonstrated as soundly proved until the speculatively-merged gate itself is demonstrated accurate. We note that this set of dependencies is minimal in that dependencies are limited to precisely those gates whose behavior would be altered if the corresponding postulated equivalence is invalid. Note that collapsing SCCs within the Proof Graph does not affect the minimality of this transitive dependency.

Next consider the use of conjunctive induction as the chosen proof framework, where the Proof Graph might additionally contain proof dependencies. If a candidate equivalence $e1$ cannot be proved, and another candidate proof $e2$ is proved using the inductive hypothesis of $e1$, the proof of $e2$ will be discarded along with all other candidates which transitively depend on $e1$. Such invalidation is necessary for soundness, since otherwise a potentially-invalid hypothesis would be used as the basis of a proof. Use of an unsatisfiable core furthermore ensures a *minimal* set of such dependencies and hence invalidations, whereas a traditional framework would require invalidating *all* proofs due to risk of such unsoundness. In general, transitive dependencies may include edges of both types. Optimality of identified redundancy follows noting that both types of dependencies are minimally identified. ■

VII. EXPERIMENTAL RESULTS

All techniques described in this paper have been implemented in the IBM internal verification tool *SixthSense* [9]. We utilize two disjoint benchmark suites:

- 1300 industrial property checking and sequential equivalence checking benchmarks. These designs are derived primarily from IBM high-performance microprocessors

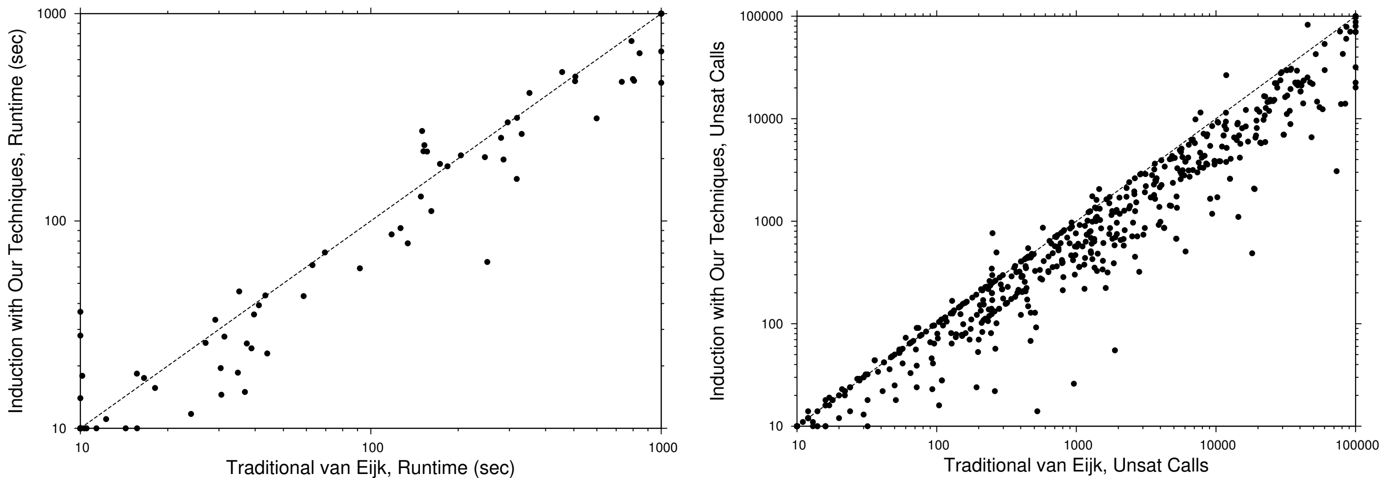


Fig. 4. Finding redundancies with $k = 1$ induction on 1300 IBM designs. Left: runtime, Right: number of unsatisfiable miters

and range in size up to 5.3M AIG AND gates and 330k registers.

- The publicly available HWMCC'10 benchmarks [10].

All experiments were run on a cluster of 16 GB, 2 GHz POWER 5 workstations.

A. Induction Results

We first examine the impact of the Proof Graph on induction. We preprocessed each netlist with combinational simplifications [13], light-weight sequential simplifications, phase abstraction [14], transient elimination [7], and input reparameterization [15]. Next, we use 640 passes of 32-cycle random simulation to derive candidate gate equivalences, and we use $k = 1$ induction to prove these equivalences. This flow was repeated twice: with the techniques presented in this paper, and without our techniques in a more traditional flow referred to as "van Eijk" below.

Figure 4 shows the difference in induction runtime of the van Eijk flow vs. our proposed algorithms on the IBM designs. Our techniques improve the runtime on almost all designs, and the maximum reduction in runtime is 75%. The occasional slowdowns are cases where the order of SAT calls imposed by the Proof Graph (Algorithm 6) is disadvantageous[2]. With our proposed methods, the order in which the miters are tested is influenced by the structure of the Proof Graph, while in the van Eijk flow we test the miters in topological order. In our implementation, we utilize incremental SAT which makes the ordering significant.

The runtime improvement is primarily due to the reduction in the number of SAT calls made by the induction package. Methods exist to reduce the number of SAT calls that are satisfiable – re-simulation of inductive counterexamples to quickly detect satisfiable miters [5]. Using the techniques described in this paper we are able to furthermore reduce the number of unsatisfiable calls. We do this by **(1)** avoiding re-testing soundly proved equivalences in the later induction iterations, and **(2)** skipping SAT calls for equivalences that cannot be soundly proved. Figure 4 shows a comparison of the number of unsatisfiable SAT calls on the IBM designs. Our techniques reduce the number of unsatisfiable calls by 25% on average and 97% in cases[3].

Figure 5 analyzes the performance of our induction implementation on a subset of the most challenging HWMCC'10 benchmarks. In most cases, our techniques improves runtime significantly, by 11% on average and 70% in cases. As with the IBM benchmarks, the runtime improvement is primarily due to a reduction in the number of unsatisfiable SAT calls, 37% on average and 92% in cases.

When we enable our Proof Graph algorithms the number of iterations increases slightly, 17% on average. The reason is that because SAT calls are skipped, inductive counterexamples may not be seen in the earlier iterations. This causes the equivalence classes to not be refined as aggressively as in a traditional flow. However, the net decrease in the number of SAT calls makes up for the slight increase in induction iterations.

Figure 5 also shows the number of merges. When our induction package deduces that an equivalence is soundly proved, it merges the equivalence and simplifies the design. *Early merges* are merges that are performed before the fixedpoint is reached. We can perform a significant percentage of the merges early, 37% on average. In one case, `bjrb07amba10andenv`, we hit an induction timeout of 1200 seconds and thus all merges were early merges – there was no fixedpoint.

B. TBV Results

Next we examine the impact of our algorithms on TBV. As in Section VII-A we aggressively pre-process the design and use random simulation to postulate register equivalences. We prove these suspected equivalences with 1-induction and apply TBV on those equivalences which are suspected to hold but

[2]Note that there is overhead associated with maintaining the Proof Graph. In our implementation, this overhead is minimal, and the change in the ordering of SAT calls is responsible for any slowdowns in the cases have studied.

[3]Because other aspects of the redundancy removal framework consume significant runtime, e.g. formulation of the SAT problem and resimulation of counterexamples, the reduction in the number of unsatisfiable SAT calls is not directly proportional to the reduction in the total runtime.

	Preprocessed Size		van Eijk					Our Techniques				
Benchmark	Ands	Reg.	Time	Iter.	Total Sat Calls	Unsat. Sat Calls	Total Merges	Time	Iter.	Total Sat Calls	Unsat. Sat Calls	Early Merges
bj08amba5g62	12411	39	91.6	4	27639	27490	4532	59.0	5	20237	20095	0
bjrb07amba10andenv	63127	58	1204.5	2	38230	38066	138	1200.3	3	29634	29454	138
bjrb07amba3andenv	5473	30	8.1	3	7727	7697	1912	8.4	3	4691	4662	12
bjrb07amba4andenv	13478	33	30.5	3	7955	7926	2473	14.5	4	4207	4175	2463
bjrb07amba5andenv	15063	38	134.0	4	22827	22738	4234	78.0	4	12786	12718	23
bjrb07amba6andenv	23622	41	295.9	3	26527	26437	5759	299.1	4	22323	22228	32
bjrb07amba7andenv	22198	45	173.0	3	16493	16383	4113	188.3	4	12285	12174	40
bjrb07amba9andenv	45539	52	1200.5	5	81087	80956	11065	657.7	5	43015	42894	106
bob1u05cu	12201	2146	6.7	34	25027	24305	746	5.2	87	15177	14753	63
bobmitersynbm	31015	5984	43.3	31	79981	78684	3225	43.7	58	15152	13905	3223
bobsmcodic	18447	1850	15.6	5	6212	6056	954	6.0	8	596	507	954
bobsmmem	55105	3584	18.0	8	13021	12740	1873	15.6	10	4350	4063	1852
bobsmrisc	9422	1323	2.7	5	8666	8587	7329	8.4	5	6813	6732	0
bobsynthetic2	2387	24	161.5	45	85792	85706	1345	111.6	44	60293	60208	0
bobuns2p10d20l	2229	20	351.7	2	226	223	283	414.5	2	231	228	0
mentorbm1and	17628	3138	9.4	44	41483	40990	3536	8.8	44	21372	21105	3536
mentorbm1p02	12255	2111	10.3	42	38285	37746	1231	6.2	43	22532	21954	252
mentorbm1p03	12254	2111	7.1	43	39087	38578	1229	8.9	42	22750	22186	252
mentorbm1p04	12282	2117	8.6	43	39012	38495	1260	5.6	42	22742	22177	252
mentorbm1p05	12290	2119	7.3	43	39415	38891	1270	5.8	43	22823	22258	252
mentorbm1p07	17465	3109	11.3	40	37433	36960	3413	9.0	41	23033	22490	280
mentorbm1p08	12273	2115	7.4	44	39863	39342	1251	6.2	42	22711	22147	252
mentorbm1p09	12253	2111	7.8	43	39104	38576	1230	5.9	43	23114	22533	252
mentorbm1p10	12253	2111	6.8	42	38176	37646	1225	4.9	45	22079	21790	1225
mentorbm1p12	12288	2114	7.5	43	39069	38575	1231	5.8	43	21421	21149	1231
neclaftp1001	35903	5360	204.2	7	78296	77352	24234	207.1	8	71842	70920	96
neclaftp1002	35734	5360	280.8	8	86909	85985	24167	251.7	8	79691	78766	287
neclaftp2001	21240	3478	12.2	4	29614	29596	23381	11.0	4	28528	28510	0
neclaftp2002	21891	3478	4.0	4	29112	29095	23682	4.7	4	27802	27785	0
pdtpmsviper	15066	574	10.1	2	10364	10277	5960	17.9	3	9396	9304	0
pj2002	16769	686	4.6	3	9766	9756	3250	1.4	3	3882	3872	2935
pj2003	16769	686	4.4	3	9766	9756	3250	1.7	3	3882	3872	2935
pj2006	16855	702	4.5	3	9773	9756	3248	1.8	3	3589	3572	2935
			1.00	1.00	1.00	1.00	1.00	0.89	1.17	0.63	0.62	0.37

Fig. 5. Finding redundancies with $k = 1$ induction on a subset of the HWMCC'10 designs

were unproved with induction. This TBV flow speculatively reduces the sequential netlist, annotates it with miters, and passes it downstream to first a combinational simplification engine and then an interpolation engine. We repeat this flow twice: once using our new techniques, and again with the Proof Graph disabled.

Figure 6 shows the TBV runtime on 93 of the most difficult IBM designs[4]. Nearly all runtimes are greatly improved by our Proof Graph techniques. We improve the runtime by 90% in cases and 18% cumulatively. The primary causes for these improvements are: **(1)** early merging prevents later TBV iterations from needing to re-prove what was soundly proved in earlier iterations, and **(2)** we use the proof graph to guide the downstream algorithms, only attempting proofs where a proved equivalence can lead directly to a merge, similar to Algorithm 6.

Figure 6 also shows the number of times interpolation was used to solve a miter. Our techniques are able to reduce the number of properties that interpolation attempts to prove by 80% in cases, 13% on average. Because each interpolation call has a 30-second time limit, by reducing the number of interpolation calls we improve the runtime substantially.

[4]Our aggressive pre-processing proves all properties in many of our benchmark designs.

VIII. Related Work

There has been much work in the field of sequential redundancy identification. Due to space limitations, we limit our focus to more recent work which transitively subsumes prior foundational work.

[16] proposes an incremental version of a redundant latch fixedpoint similar to Algorithm 1, using 1-step induction to correlate latches for combinational equivalence checking (CEC) frameworks. The induction itself is performed using an off-the-shelf CEC tool. The authors propose that the effort of the CEC tool in finding internal equivalence points at each iteration may be simplified by avoiding re-verification of internal equivalence points driven solely by latches which did not change in correlation since they were last proved. This result relates to our ability to infer soundly-proved equivalences before all proofs are completed. However, there are several differences from our work: **(1)** We may soundly identify and leverage redundancy even before a fixedpoint is reached, whereas their technique requires a fixedpoint in being leveraged solely for CEC. **(2)** Our approach is designed to handle general k-induction as well as arbitrary TBV flows to identify redundancies over arbitrary gates in the netlist, while their approach is focused upon 1-induction to identify latch equivalence using a CEC tool. **(3)** Our approach is robust enough to handle inductive hypothesis constraints while their

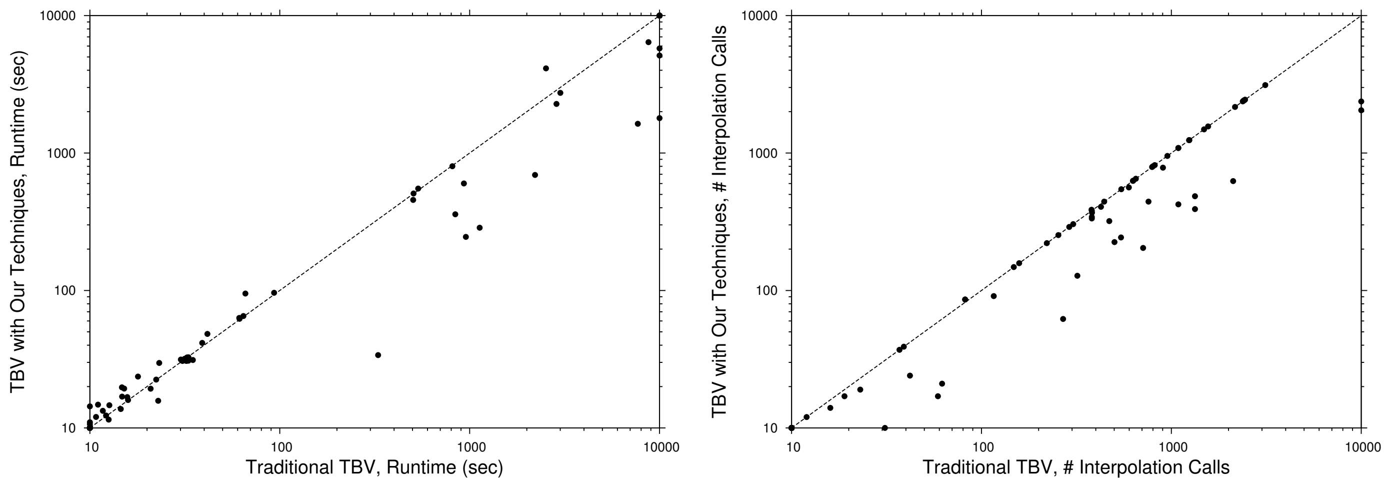


Fig. 6. Finding non-inductive redundancies with TBV on the 93 most difficult IBM designs. Left: runtime, Right: number of calls to interpolation [8]

approach need not consider them, as such hypotheses in their more limited settings are effectively "latch mappings."

[2] discusses how one may leverage postulated equivalences through speculative reduction, enabling greater simplification of the resulting netlist for enhanced bounded or unbounded proof analysis. However, this work does not provide a method to soundly simplify the netlist until all equivalence proof obligations are proved – hence does not offer early merging capability. In addition [2] is typically implemented by using a SAT solver test each suspected equivalence at every iteration of the fixedpoint procedure, a complexity we strive to avoid.

[5] describes a method to minimize the number of satisfiable SAT calls through re-simulation of induction counterexamples, which combined with speculative reduction yields up to 5 orders of magnitude speedup on a cumulative benchmark suite. However, aside from eliminating "implied" proofs via speculative reduction, this work does not address how to minimize the number of unsatisfiable calls, which is a primary contribution of this paper. This work is nonetheless complementary to ours, as we have also found it useful to aggressively resimulate induction counterexamples to rule out satisfiable induction queries.

IX. Conclusion

We have presented a method to improve the efficiency of redundancy identification frameworks by tracking dependencies between redundancy candidates. The dependencies are tracked using a datastructure called the Proof Graph, which is applied to enhance both inductive and transformation-based redundancy identification frameworks. Our techniques provide numerous benefits to redundancy identification frameworks.

- Many redundancies may be determined to be soundly proved before reaching a fixedpoint, allowing for useful reduction in the design size in the event that computational resources are exhausted, or an *incomplete* proof method is used.
- The total proof burden is reduced because soundly proved redundancies need not be re-proved in later fixedpoint iterations.
- The proof burden is additionally reduced because the Proof Graph allows us to identify redundancies which can never be soundly proved under a given set of candidates. The proofs of such redundancies can be skipped.

Experiments confirm that our techniques reduce the number of attempted proofs by up to 97%, and improve runtime by up to 75%, for redundancy identification frameworks on industrial as well as public benchmark sets.

References

[1] P. Bjesse and K. Claessen, "SAT-based verification without state space traversal," in *FMCAD*, Nov. 2000.

[2] H. Mony, J. Baumgartner, V. Paruthi, and R. Kanzelman, "Exploiting suspected redundancy without proving it," in *DAC*, June 2005.

[3] C. A. J. van Eijk, "Sequential equivalence checking without state space traversal," in *DATE*, Feb. 1998.

[4] D. Stoffel and W. Kunz, "Record & play: A structural fixed point iteration for sequential circuit verification," in *Int'l Conference on Computer-Aided Design*, Nov. 1997.

[5] H. Mony, J. Baumgartner, A. Mishchenko, and R. K. Brayton, "Speculative reduction-based scalable redundancy identification," in *DATE*, pp. 1674–1679, IEEE, 2009.

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

[7] M. Case, H. Mony, J. Baumgartner, and R. Kanzelman, "Enhanced verification through temporal decomposition," in *FMCAD*, Nov. 2009.

[8] K. McMillan, "Interpolation and SAT-based model checking," in *CAV*, July 2003.

[9] H. Mony, J. Baumgartner, V. Paruthi, R. Kanzelman, and A. Kuehlmann, "Scalable automated verification via expert-system guided transformations," in *FMCAD*, Nov. 2004.

[10] A. Biere and K. L. Claessen, "Hardware Model Checking Competition (HWMCC) 2010 benchmarks." http://fmv.jku.at/hwmcc10, 2010.

[11] R. Tarjan, "Depth-first search and linear graph algorithms," *SIAM Journal on Computing*, vol. 1, no. 2, pp. 146–160, 1972.

[12] L. Zhang and S. Malik, "Extracting small unsatisfiable cores from unsatisfiable boolean formula," in *SAT*, 2003.

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

[14] P. Bjesse and J. Kukula, "Automatic generalized phase abstraction for formal verification," in *ICCAD*, Nov. 2005.

[15] J. Baumgartner and H. Mony, "Maximal input reduction of sequential netlists via synergistic reparameterization and localization strategies," in *CHARME*, Oct. 2005.

[16] K. Ng, M. R. Prasad, R. Mukherjee, and J. Jain, "Solving the latch mapping problem in an industrial setting," in *Design Automation Conference*, June 2003.

Approximate Reachability With Combined Symbolic And Ternary Simulation

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

***Abstract*— Logic synthesis and formal verification both rely on scalable reachable state characterization for numerous purposes. One popular technique is over-approximate reachability analysis using an iterative ternary simulation. This method trades precision of reachability characterization for a high degree of computational efficiency. Although effective on many industrial designs, it breaks down when the design has registers that have complex initial states or has extremely deep deterministic subcircuits. In this paper, we improve upon the precision of ternary simulation-based approximate reachability while retaining its scalability by representing certain variables as symbols vs. *unknowns*, and by selectively saturating subcircuits which would otherwise preclude convergence. These techniques are particularly beneficial for enhancing the scalability of industrial sequential equivalence checking problems, occasionally solving such problems outright with no need for more costly and precise analysis.**

I. Introduction

Reachability analysis has many applications in contemporary verification and synthesis tools. For example, a design may be optimized using information about gates which are redundant in the reachable states; behavioral characteristics such as *oscillators* and *transients* may be identified and exploited for specific abstraction strategies; and properties may be solved using reachability information.

Unfortunately, exact reachability analysis is often computationally impractical, even for moderately sized designs. Approximate reachability analysis is thus often necessary, trading the precision of reachable state characterization for computational efficiency. Even when precise reachability analysis is ultimately necessary, it is often computationally beneficial to first apply faster approximate techniques to reduce the design before exact reachability analysis is performed.

One may perform approximate reachability analysis with ternary simulation [1] by letting signals take values in $\{0, 1, X\}$ as follows. Primary inputs are assigned X, and registers are assigned their initial values. Ternary simulation is then used to derive the next state. Computation proceeds in this way until a repetition of state values has been witnessed, indicating that an over-approximate reachability analysis has converged.

Reachability analysis with ternary simulation requires little runtime, often executing in seconds even on the largest industrial designs. Its reachability approximation is coarse but is precise enough to identify common artifacts in industrial verification and synthesis frameworks: inputs and registers that are constant due to testbench assumptions, simple internal equivalences, oscillating clocks, and transient signals. For this reason, ternary simulation-based reachability analysis is implemented in many logic synthesis and verification systems.

One key weakness of reachability analysis with ternary simulation is its inability to precisely characterize designs with complex initial values. Any registers with non-deterministic initial values are assigned X in the first iteration of reachability analysis, and because of the conservative nature of ternary simulation this X propagates to *all* fanout logic. For designs with non-deterministic initialization, this often precludes *any* useful reachable state characterization with this analysis.

A secondary weakness of reachability analysis with ternary simulation is that it may require an infeasible number of simulation steps to converge; designs containing large counters or "linear feedback shift register" type logic are often particularly problematic. This lack of convergence precludes any reachable state approximation.

In this paper we improve the precision and conclusiveness of reachability analysis with ternary simulation in two ways:

1) We utilize a symbolic representation and use this to represent initial states. We define reachability analysis over both ternary simulation and symbolic simulation. In practice, this adds little to the total runtime of approximate reachability and it improves the resolution substantially, particularly for industrial *Sequential Equivalence Checking (SEC)* problems.
2) We introduce a *saturation* technique to enable convergence without loss of useful reachable state characterization.
3) We additionally introduce extensions to ternary simulation-based application domains of redundancy removal, phase abstraction, and transient elimination to generalize them accordingly given our symbolic techniques.

Section II describes the related work in this field. In Section III we provide preliminaries, including an overview of reachability approximation using ternary simulation. Section IV describes symbolic simulation, and Sections V and VI incorporate symbolic simulation into approximate reachability. Section VII introduces our saturation technique which helps convergence in cases. Finally, experimental results are given in Section VIII.

II. Related Work

A significant amount of work exists in the field of approximate reachability characterization. Due to space limitations, we focus only upon those which rely upon ternary simulation vs. more expensive and precise techniques.

The use of reachable state characterization via ternary simulation and its applications within general model checking

was proposed in [1]. They note that this technique can identify a useful subset of redundant gates, whose simplification greatly enhances the scalability of subsequent verification. They also use this analysis for identifying oscillating subcircuits which may be leveraged for *phase abstraction*.

The work of [2] proposed another use of this analysis: to identify *transient* signals which settle to a reducible (e.g., constant) behavior after several timesteps. The verification process may then be decomposed, leveraging bounded techniques to analyze the first several timesteps before the transients settle, then time-shifting the design and simplifying the transients for unbounded verification thereafter. This work is similar in spirit to the primary application domain considered in this paper: to enable the reduction of designs with intricate initial values. However, these are complementary techniques and we have found them both useful in conjunction.

Symbolic Trajectory Evaluation (STE) [3] is a related simulation technique. In STE, ternary simulation is combined with symbolic simulation, by encoding ternary value functions using a pair of BDDs or other *dual-rail* based expressions. Users specify assertions of the form $A \Rightarrow C$, where A is the *antecedent* that specifies the values with which to drive the simulation, and C is the *consequent* that specifies the expected results of the simulation. The advantage of STE over scalar simulation is that it can cover large input spaces efficiently and precisely. The complexity of STE is dependent on the number of symbolic variables in the the antecedent, not necessarily the size of the design, so it can scale to large designs such as complex datapaths and memory arrays. While using related analysis methods, our approaches are distinct in numerous ways. First, our application domain is approximate reachability analysis to facilitate model checking, wherein we do not have an antecedent to dictate where symbols should be introduced nor a consequent against which we may attempt to refine lossy X values. Our analysis is intended to efficiently facilitate subsequent verification algorithms, and without the heuristics described in this paper there may be blowups in runtime or memory if too many symbols are introduced, vs. too coarse of reachability approximation if inadequate symbols are introduced.

Our form of symbolic simulation uses the same value domain as *quasi-symbolic simulation* [4]. In quasi-symbolic simulation, value functions are restricted to the set $\{0, 1, X, X_A, \neg X_A, X_B, \neg X_B, \ldots\}$, where $\{X_A, X_B, \ldots\}$ are symbolic variables corresponding to netlist inputs. Instead of supporting arbitrary symbolic functions, quasi-symbolic simulation employs case-splitting to eliminate symbolic variables and remove conservatism (i.e., propagation of X to a checked output). In contrast, our technique does not seek a complete symbolic simulation of the netlist, and does not employ any case-splitting. Rather, we introduce symbolic variables selectively to enhance approximate reachability analysis. Also, our technique may introduce symbolic variables at gates internal to the netlist in addition to the netlist inputs, which often tightens the approximation.

Alg. 1. Approximate reachability with ternary simulation

```
 1: function approxReachability(design)
 2:     for all (primary inputs I in design) do I = X
 3:     for all (registers R in design) do R = X
 4:     ternarySimulate(design)
 5:     state = vector of register initial state valuations
 6:     seen = { state }
 7:     for time = 0; ; ++time do
 8:         Assign registers their corresponding values in state
 9:         ternarySimulate(design)
10:         state = vector of register next state valuations
11:         if (state ⊆ seen) then seen over-approximates the reachable states
12:         seen = seen ⋃ { state }
13:     end for
14: end function
```

III. Preliminaries

We consider gate-level sequential logic designs, and for convenience we assume the netlist is expressed as an And-Inverter Graph (AIG). That is, every gate in the design is either a constant, an AND gate, inverter, or primary input. We also consider *registers* which hold the state of the design. Registers have an associated next state function that defines their value in the next time step.

In our model, registers also have an explicit initial value function. For registers with a constant initial value, this function maps to the corresponding constant gate. For more complex initial values, this function maps to a a combinational subcircuit which is used to encode a set of initial states. Such complex initial values arise in many contexts. For example, they may be necessary in SEC applications to represent an arbitrary power-on state. Even for designs with simpler initial values, a more complex initial state may arise through a verification-enhancing transformation such as retiming [5] or temporal decomposition [2]. While the commonly used netlist format *AIGER* assumes that every initial value is constant-0 [6], more complex initial values are supported through synthesizing a multiplexor at the output of every latch which may drive the desired initial value at time 0.

Ternary simulation is a way to approximate netlist behavior. Inputs are assigned values in $\{0, 1, X\}$, and simple rules govern how these values propagate through the logic. We use ternary simulation on AIGs, and in this context `and(A,B)` is 0 if either `A` or `B` are 0. `and(A,B)` is X if either `A` or `B` are X. Otherwise, when `A` and `B` are both 1, `and(A,B)` is 1. We define inversion in the normal way, though `not(`X`)` = X.

Ternary simulation can be employed to perform approximate reachability analysis, as shown in Algorithm 1. Primary inputs and registers are initially assigned X values, and ternary simulation is used to derive the initial state values. A set of *seen* states is maintained, initially equal to just this ternary initial state. The algorithm then iterates, **(1)** assigning the current state values to registers while retaining the X values on primary inputs, **(2)** using ternary simulation to derive the next state values, and **(3)** using the set of *seen* states to detect convergence.

Convergence is detected through checking if the next state cube is contained in the set of *seen* states. There are several

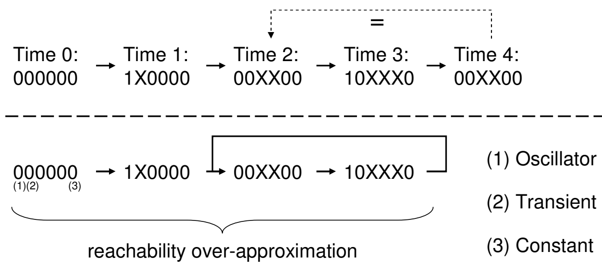


Fig. 1. Ternary simulation example for a design with six registers

ways this could be implemented. For example, BDDs [7] can be used to represent the set of seen state cubes, and the containment check can be implemented using BDD operations. In practice, the performance of such approach is prohibitive. Instead, we simply let the *seen* states be a list of ternary states, and instead of *state* $\subseteq$ *seen* we test if there exists an s in the *seen* list such that *state* $=$ s. Such a check can be implemented efficiently using a hash table, and this approximation to containment usually does not affect the convergence of Algorithm 1.

An example of reachability analysis with ternary simulation is shown in Figure 1. Five iterations are performed before it is determined that the time 4 state is equal to the time 2 state. This results in the approximate reachable state graph shown in the bottom half of the figure.

Ternary simulation-based reachability analysis has many applications. We implement this in a library which is used to characterize the netlist in various ways:

Oscillators: In Figure 1, register (1) is an oscillator, meaning that it periodically oscillates between 0 and 1 valuations. Designs that have oscillators may be simplified using phase abstraction [1], enabling significant verification benefits such as yielding a smaller netlist, enabling greater reduction potential through other transformations, and reducing diameter.

Transients: In Figure 1, register (2) is transient, meaning that after the initial time steps its value settles and remains constant forever after. Verification of designs that have transients can be simplified [2] through time-shifting their behavior, enabling reduction of the transient signals.

Redundancies: Gates that act as constants, or pairs of gates that are equivalent/antivalent in every reachable state, may be directly merged. In Figure 1, register (3) is constant, and in our implementation we replace this register with a constant-zero gate. Note that such a reduction may generally enable other reductions, such as constant propagation and cone-of-influence reduction.

IV. Symbolic Simulation

In this paper we strengthen reachability approximation by considering symbolic simulation as well as ternary simulation, similar to quasi-symbolic simulation [4]. In this section we define our notion of symbols and how they can be handled in simulation.

Symbols, written in the form X_A for some subscript A, represent concrete values that are not being precisely modeled. In contrast, the ternary X represents a *completely* unknown symbol. If two signals evaluate to X we conservatively conclude that the signals may not be equal, but if the signals both evaluate to X_A they are treated as equivalent.

We can expand ternary simulation to include a set of symbols $\{X_A, X_B, \ldots\}$ by letting signals take values in $\{0, 1, X, X_A, \neg X_A, X_B, \neg X_B, \ldots\}$. We retain the traditional ternary simulation evaluation from Section III for conjunction, with rules listed in order of precedence:

0 Identity: If $a \equiv 0$ then $a \cdot b = 0$. Likewise for $b \equiv 0$.
1 Identity: If $a \equiv 1$ then $a \cdot b = b$. Likewise for $b \equiv 1$.
X **Identity:** If $a \equiv X$ then $a \cdot b = X$. Likewise for $b \equiv X$.

If none of the above rules apply then both signals are symbolic: $a = X_A$, $b = X_B$. We apply the following symbolic rules:

Idempotence: If $X_A \equiv X_B$ then $X_A \cdot X_B = X_A$.
If $X_A \equiv \neg X_B$ then $X_A \cdot X_B = 0$.

Peephole Optimization: If $X_B \equiv X_A \cdot X_Z$ then $X_A \cdot X_B = X_B$.

Hashing: If $X_A \cdot X_B$ is in the hash table, return the previously-stored result.

New Symbol: Create a symbol X_C to represent $X_A \cdot X_B = X_C$. Store X_C in the hash table.

First idempotence is used to handle cases where a symbolic value is conjoined with itself. Next, peephole optimization is used to find cases of nested conjunctions with shared arguments. A hash table is used to determine if the result of $X_A \cdot X_B$ was previously computed. If all other checks fail then we create a new symbol X_C to represent the result of a conjunction.

V. Incorporating Symbols Into Reachability

In this section we discuss how to incorporate symbols into approximate reachability analysis. Symbolic simulation offers greater resolution than ternary simulation, but symbols must be applied judiciously. If every signal was handled symbolically then Algorithm 1 would perform exact reachability analysis – though likely with an explosion in the number of symbols represented, leading to unacceptable runtime or memory consumption.

We are motivated by designs that have complex initial values, and we would like to use symbolic simulation to represent these initial values. In our model, the initial values are derived from combinational functions over the primary inputs. For this reason, we assign the inputs symbolic values and trust that these symbols will propagate to the initial values, allowing us to represent these initial values more precisely.

One risk is that the number of symbols can explode. Specifically, new symbols are introduced for each primary input and with each application of the *New Symbol* rule of Section IV. If the current state of the design contains one new symbol for each step of approximate reachability, then Algorithm 1 can never converge. We limit the number of

Alg. 2. Approximate reachability with ternary and symbolic simulation

```
1: function approxReachability_symbols(design)
2:    for all (primary inputs I in design) do I = new symbol
3:    for all (registers R in design) do R = X
4:    symbolicTernarySimulate(design)
5:    state = vector of register initial state valuations
6:    seen = { state }
7:    for time = 0; ; ++time do
8:       Assign registers their corresponding values in state
9:       if (time = 0) then
10:         symbolicTernarySimulate(design)
11:      else
12:         for all (primary inputs I in design) do I = X
13:         symbolicTernarySimulate_noNewSymbols(design)
14:      end if
15:      state = vector of register next state valuations
16:      if (state ⊆ seen) then seen over-approximates the reachable states
17:      seen = seen ⋃ { state }
18:   end for
19: end function
```

symbols by only handling primary inputs symbolically at time 0. In addition, we only allow the *New Symbol* rule to apply in time 0. At all other times, we consider $X_A \cdot X_B = X$.

Algorithm 2 illustrates our framework for approximate reachability using ternary and symbolic reachability. We create new symbols to represent primary inputs at time-0, and at all other times we assign primary inputs the value X. Algorithm 2 utilizes two simulation routines, *symbolicTernarySimulate* and *symbolicTernarySimulate_noNewSymbols*. The function *symbolicTernarySimulate* is used only at time-0 and simulates the design as described in Section IV. We have found it effective in our desired application domain to *not* introduce any additional symbols after time 0, and this is accomplished by using the function *symbolicTernarySimulate_noNewSymbols* which implements the methods of Section IV but treats $X_A \cdot X_B = X$ when the value of this conjunction cannot be otherwise determined through idempotence, peephole optimization, or hash lookup.

Algorithm 2 introduces new symbols only at time-0. For all time > 0 no additional symbols are created, but the symbols created at time-0 can continue to propagate through the logic at later times.

By restricting the application of symbolic simulation, we derive an approximate reachability algorithm that retains most of the performance of Algorithm 1 but is significantly more precise for numerous classes of important verification problems. This increased precision allows the efficient characterization of reachability information – constant or equivalence signals, oscillators, and transients – that otherwise are undetectable.

VI. Generalized Simplification using Symbols

Algorithm 2 returns an over-approximation to the set of reachable states. Our implementation uses this information in several application domains, some of which become more complex when the reachability information contains symbols. In particular, the simplification of symbolic constant gates, oscillating registers, and transients is affected.

Symbolic constant gates are those that always evaluate to the same symbolic value in every reachable state. Such gates may be simplified by replacing them with a subcircuit that represents the given symbol.

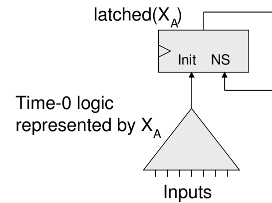


Fig. 2. A subcircuit that represents the symbol X_A

When oscillating registers are identified, we use phase abstraction to simplify the netlist. Phase abstraction will unfold the transition relation modulo a detected periodicity, and simplify the netlist by injecting constant values in place of the corresponding oscillator. However, when the reachability information contains symbols, the detected oscillators may assume symbolic periodic behavior. For example, we have observed period-two oscillators with signature $0, X_A, 0, X_A, \ldots$. To simplify such symbolic oscillators, we must replace a register in the unfolded transition relation with a reference to a subcircuit that represents the given symbol.

When transients are identified, temporal decomposition is able to simplify the design by time shifting and replacing each transient gate with the corresponding redundant value to which it settled. When using symbols, in cases we find transients that settle to a symbolic values. Temporal decomposition can simplify these by replacing each transient with a reference to a subcircuit that represents the given symbol.

Recall that in Section V we use symbols to represent the value of inputs, or combinational functions thereof, at time 0. We may obtain a subcircuit that represents such a symbol by simply latching the time-0 value; we fabricate a register whose initial value is that corresponding signal, and whose next-state function holds its current value. This corresponding logic may be used in the above three application domains to simplify the netlist, extending our ability to simplify a netlist using the enhanced reachable-state characterization enabled through using symbols. This logic depicted in Figure 2 shows a subcircuit that represents the symbol X_A.

Simplifying logic using this procedure is often beneficial to reduce overall netlist size and verification complexity. However, this procedure does entail synthesizing registers, which may be undesirable in cases. Our implementation uses several heuristics to minimize this impact: **(1)** when multiple two subcircuits $\mathrm{latched}(X_A)$ and $\mathrm{latched}(X_B)$ are synthesized, we try to share registers across these two symbol representations, and **(2)** we disallow simplifications in cases where the total number of registers would increase.

VII. Accelerating Convergence with Saturation

Approximate reachability, shown in Algorithm 1, is an iterative procedure which successively explores sets of states until it detects a fixedpoint. For deep and complex industrial designs, the number of iterations required for convergence may be prohibitive. Although we improve on the precision

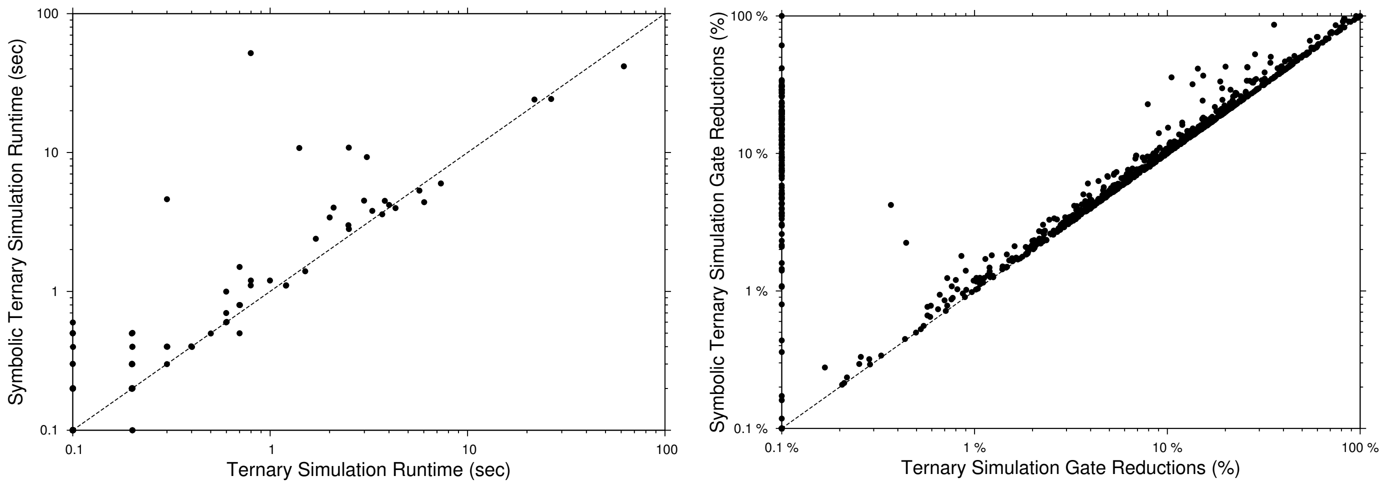


Fig. 3. Approximate reachability runtime on the IBM designs. Left: runtime, Right: gate reductions

Alg. 3. Approximate reachability with X-saturation

1: **function** approxReachability_symbols_saturation(*design*, *cycleLimit*)
2: setup approximate reachability as in Algorithm 2, lines 2-6
3: **for** *time* = 0; ; ++*time* **do**
4: Assign registers their corresponding values in *state*
5: **if** (*time* $\geq$ *cycleLimit*) **then**
6: **for all** $r \in$ {registers not oscillating or constant} **do** $r = X$
7: **end if**
8: do one iteration as in Algorithm 2, lines 9-17
9: **end for**
10: **end function**

of approximate reachability by using symbols in Algorithm 2, this convergence problem remains and in cases worsens due to the extra precision. In this section we detail our solution to accelerate convergence when a pre-determined resource threshold is exceeded.

The approximate reachability loop does not converge if new register valuations are encountered, and we can accelerate convergence by limiting the register valuations. Specifically, it is always conservative to further over-approximate the computation by overwriting register valuations with X. We refer to this process as *X-saturation*. If all registers are assigned X then the reachability approximation process will immediately converge; however, it would contain no useful information. The difficulty with effective X-saturation is thus in determining which subset of registers to X-saturate, and when to perform this saturation, balancing precision vs. runtime.

As in Section VI, approximate reachability may be used to detect oscillating registers and constants which may be used for phase abstraction and redundancy removal. Furthermore, we have found that such gates are high-fanout registers that influence much of the netlist behavior. We are motivated to always preserve constants and oscillating registers by not X-saturating them because **(1)** such saturation would limit results that are useful for phase abstraction and redundancy removal, and **(2)** X values injected on such gates would quickly propagate through the netlist and dramatically weaken the resulting reachability approximation.

Algorithm 3 is an extension of Algorithm 2 which includes X-saturation. It takes one additional argument *cycleLimit* which is the number of iterations that are allowed before registers are X-saturated. After *cycleLimit* is exceeded, non-oscillating and non-constant registers are forced to have the value X. This further approximation causes the algorithm to converge quickly, with the total number of cycles usually being only slightly larger than *cycleLimit* in practice.

Algorithm 3 requires constant and oscillating registers to be detected. In our implementation, we efficiently identify oscillators using a sliding window technique which is able to identify oscillators with period ≤ 128. In addition, we also detect *delayed oscillators* which may assume variable behavior during the design's initialization phase but thereafter act as oscillators. Both true- and delayed-oscillators influence large sub-circuits in the netlist, and to preserve a useful reachability approximation it is vital to not X-saturate these registers.

VIII. Experimental Results

All techniques described in this paper have been implemented in the IBM internal verification tool *SixthSense* [8]. In these experiments we utilize two benchmark sets:

IBM SEC: We use a suite of 1122 challenging industrial SEC problems. These designs come primarily from high performance microprocessors, and the largest such design has 5.3M AIG AND gates and 330k registers. In our framework, pairs of registers are often initialized with the same non-deterministic random value in order to check equivalence modulo any initialization sequence.

HWMCC'10 SEC: To enable evaluation against publicly available benchmarks, we evaluated our techniques against a subset of benchmarks from the Hardware Model Checking Competition (HWMCC) [9]. However, none of these benchmarks directly exhibits the complexities often faced in industrial SEC benchmarks. We thus emulated the industrial challenges in these problems as follows:
(1) We simulated each design for 1000 cycles starting from the initial state, and inferred register equivalences from the simulation data.

Benchmark	Reg. Equivalences	Gates	Ternary Simulation Time	Ternary Simulation Iterations	Ternary Simulation Gates Reduced	Symbolic Ternary Simulation Time	Symbolic Ternary Simulation Iterations	Symbolic Ternary Simulation Gates Reduced
139464p22_sec	1	20768	0.0	6	2	0.0	6	2
139464p23_sec	1	20800	0.0	6	2	0.0	6	2
139464p24_sec	1	20832	0.0	6	2	0.0	6	2
bob1u05cu_sec	17	51650	0.0	37	29627	0.0	37	29813
bobmitersynbm_sec	1507	47416	0.0	7	0	0.0	8	4385
bobsmmem_sec	1068	42351	0.0	7	0	0.0	12	2884
mentorbm1and_sec	28	46286	0.1	37	4279	0.1	37	7190
mentorbm1p02_sec	19	51450	0.0	37	8521	0.0	37	10662
mentorbm1p03_sec	19	51447	0.1	37	8525	0.0	37	10664
mentorbm1p04_sec	21	51444	0.1	37	8442	0.1	37	10599
mentorbm1p05_sec	21	51351	0.1	37	8060	0.1	37	10279
mentorbm1p07_sec	28	46478	0.1	37	4921	0.0	37	7724
mentorbm1p08_sec	19	51435	0.1	37	8511	0.1	37	10658
mentorbm1p09_sec	19	51450	0.1	37	8525	0.1	37	10666
mentorbm1p10_sec	19	51447	0.0	37	8525	0.0	37	10664
mentorbm1p12_sec	19	51429	0.1	37	8462	0.1	37	10601
pj2006_sec	1	37411	0.0	6	1826	0.0	6	1826
pj2013_sec	1	37518	0.0	7	2134	0.1	7	2134
pj2015_sec	1	41424	0.0	7	2076	0.0	7	2076
pj2017_sec	1	41580	0.0	7	1315	0.0	7	1315
Average Performance					1.00			1.18

Fig. 4. Detailed comparison of approximate reachability-based design simplifications on the HWMCC'10 SEC benchmarks

(2) For each register equivalence class, we constructed a new primary input to model the non-deterministic initial value for that class. For all registers in this class, we replaced the initial value with the new primary input.

We have made the modified HWMCC benchmarks publicly available [10].

In our experiments we simplify each design by applying approximate reachability in order to find constant and equivalent signals. This emulates the default flow in *SixthSense* where approximate reachability is the first algorithm applied to a design under verification, due to its speed, scalability, and capability to significantly simplify the design for subsequent more-precise analysis. We repeat this flow twice: once using only ternary simulation, and again using techniques presented in this paper. All experiments were run on a cluster of 4 GB, 2 GHz POWER5 workstations.

A. IBM SEC Results

Figure 3 examines the performance of reachability analysis with symbolic and ternary simulation on the IBM SEC benchmarks.

The first plot in Figure 3 shows the runtime of approximate reachability. Introducing symbolic simulation almost always slows approximate reachability, though this slowdown is negligible with most runs completing in less than 10 seconds. Given the large sizes of these industrial benchmarks (up to 5.3M ANDs), we are satisfied with this minimal runtime overhead.

The second plot in Figure 3 shows reductions enabled using the corresponding approximate reachability information. We show the number of gates eliminated during design simplification as a percentage to the original number of gates. Most designs see greater reductions with symbolic simulation. In addition, many designs were not simplified at all with ternary simulation but now are simplified with ternary and symbolic simulation. In some cases, symbolic simulation adds sufficient resolution to prove the properties outright.

B. HWMCC'10 SEC Results

Figure 4 examines the performance of reachability analysis with ternary and symbolic simulation on the HWMCC'10 SEC benchmarks. Recall that these benchmarks were created from the HWMCC'10 benchmarks by finding suspected equivalent registers and transforming their corresponding initial states. Column 2 shows the number of equivalent register pairs that were found during that process.

In Figure 4 we can see that introducing symbolic simulation did not affect the runtime of approximate reachability in any measurable way.

Next examine the number of iterations. This is the number of time steps processed by approximate reachability before convergence. We expect this number to increase when symbols are utilized due to the more precise state representation. However, using our methods of minimally-introducing symbols, the additional number of iterations imposed is minimal, and usually enabling symbolic simulation does not increase the number of iterations at all.

Figure 4 also shows the number of gates that were reduced through approximate reachability-based design simplification. Enabling symbolic simulation allows for more gate reductions, 18% on average. In cases, approximate reachability is unable to simplify the design without the additional resolution provided by symbolic simulation.

C. Saturation Results

Figure 5 examines X-saturation. On the combined set of 1200 benchmarks, 21 benchmarks (1.75%) failed to converge within 512 iterations. All of these benchmarks are IBM designs (the HWMCC designs converged quickly), and are labeled ibm1 through ibm21 in the table.

As a baseline, we consider Algorithm 2 which is limited to 1200 seconds and 1M iterations. Eight of our designs hit one of these limits, and approximate reachability stopped abruptly with no useful reachability information with which to reduce

Benchmark	Gates	Baseline Time	Iterations	Gates Reduced	With Saturation Heuristics Time	Iterations	Gates Reduced
ibm1	15853	1.1	4120	1052	0.2	519	900
ibm2	2507	5.9	49153	937	0.1	569	348
ibm3	25805	725.0	1000000	-	0.6	631	1563
ibm4	30058	763.1	1000000	-	0.4	521	2102
ibm5	11686	291.6	1000000	-	0.1	521	281
ibm6	13760	68.5	131101	2787	0.2	516	2438
ibm7	35377	180.8	131299	3148	0.7	519	3056
ibm8	30056	0.5	551	2616	0.5	523	2416
ibm9	916576	42.1	1060	66443	21.8	540	63849
ibm10	11802	55.0	196612	10	0.2	522	10
ibm11	2697713	1200	6320	-	105.6	561	1620533
ibm12	626060	154.6	32920	63693	2.5	532	60402
ibm13	11686	295.9	1000000	-	0.2	521	281
ibm14	1698160	35.0	525	1367988	33.7	521	1366566
ibm15	760354	18.8	525	601602	19.0	521	601598
ibm16	30056	0.5	551	2616	0.5	523	2416
ibm17	385244	3.0	523	219	3.4	523	186
ibm18	395964	10.9	523	7375	10.1	523	7372
ibm19	242756	1200	163470	-	4.0	516	0
ibm20	411849	1200	93004	-	6.7	518	17731
ibm21	20229	1200	490270	-	1.2	544	4456
Average Performance		1.00		1.00	0.33		0.94
Cummulative Performance		1.00		1.00	0.03		1.77

Fig. 5. X-Saturation on the IBM benchmarks

the size of the design. The remaining 13 designs converged and the resulting reachability approximation was effective at reducing the design size, though the runtimes were very long.

The final columns in Figure 5 show Algorithm 3 with *cycleLimit* set to 512. That is, a subset of the registers are X-saturated starting at iteration 512. In most cases, this allows the algorithm to converge with just a few extra iterations. On the 13 designs that previously converged with a high number of iterations the runtime is reduced by 67%. X-saturation does compromise the resolution of the reachability approximation slightly, though on this subset of designs we preserve 94% of the reductions, on average.

It is most interesting to examine X-saturation on the 8 designs that previously hit either the 1200 second limit or the 1M iteration limit. With X-saturation, approximate reachability converges on all of these designs, and the algorithm is very fast. In all cases, the reachability approximation is suitable for design reductions, and our best example is ibm11 where approximate reachability is able to reduce the design size by 60%. Without X-saturation we cannot realize such reductions.

Cummulatively we are able to reduce the runtime by 97% while increasing the reductions by 77%, considering the designs that previously had no reductions because of computational resource limits.

IX. Conclusion

In this paper we enhance techniques for approximate reachability analysis using ternary simulation in two ways:

- We use ternary simulation enhanced with symbols to allow greater precision in modeling registers that have complex initial values. The more precise reachability approximation enables considerably greater reduction opportunity, especially on industrial SEC models.
- We introduce X-saturation as a way to force approximate reachability into convergence on complex industrial designs. In cases of slow convergence, this helps to dramatically reduce the runtime. In cases of no convergence, this helps to provide a useful reachable state approximation where previously we had none.

We additionally introduce extensions to ternary simulation-based application domains of redundancy removal, phase abstraction, and transient elimination to generalize them accordingly given our symbolic techniques. All of these techniques are implemented in IBM internal verification tool *SixthSense*, and we have found them to be indispensable on large and complex industrial SEC problems.

References

[1] P. Bjesse and J. Kukula, "Automatic generalized phase abstraction for formal verification," in *ICCAD*, Nov. 2005.

[2] M. Case, H. Mony, J. Baumgartner, and R. Kanzelman, "Enhanced verification through temporal decomposition," in *FMCAD*, Nov. 2009.

[3] C. johan H. Seger and R. E. Bryant, "Formal verification by symbolic evaluation of partially-ordered trajectories," in *Formal Methods in System Design*, 1993.

[4] C. Wilson and D. L. Dill, "Reliable verification using symbolic simulation with scalar values," in *DAC*, 2000.

[5] C. Leiserson and J. Saxe, "Retiming synchronous circuitry," *Algorithmica*, vol. 6, 1991.

[6] A. Biere, "The AIGER And-Inverter Graph (AIG) format, version 20070427." http://fmv.jku.at/aiger/FORMAT-20070427.pdf.

[7] R. E. Bryant, "Graph-based algorithms for Boolean function manipulation," *IEEE Transactions on Computers*, vol. C-35, August 1986.

[8] H. Mony, J. Baumgartner, V. Paruthi, R. Kanzelman, and A. Kuehlmann, "Scalable automated verification via expert-system guided transformations," in *FMCAD*, Nov. 2004.

[9] A. Biere and K. L. Claessen, "Hardware Model Checking Competition (HWMCC) 2010 benchmarks." http://fmv.jku.at/hwmcc10.

[10] M. L. Case, "Sequential equivalence checking variants of the Hardware Model Checking Competition 2010 benchmarks." http://case-home.com/publications/hwmcc10_sec.tgz.

Learning Conditional Abstractions

Bryan A. Brady[1]
IBM
Poughkeepsie, NY 12601

Randal E. Bryant
Carnegie Mellon University
randy.bryant@cs.cmu.edu

Sanjit A. Seshia
UC Berkeley
sseshia@eecs.berkeley.edu

***Abstract*—Abstraction is central to formal verification. In term-level abstraction, the design is abstracted using a fragment of first-order logic with background theories, such as the theory of uninterpreted functions with equality. The main challenge in using term-level abstraction is determining what components to abstract and under what conditions. In this paper, we present an automatic technique to conditionally abstract register transfer level (RTL) hardware designs to the term level. Our approach is a layered approach that combines random simulation and machine learning inside a counter-example guided abstraction refinement (CEGAR) loop. First, random simulation is used to determine modules that are candidates for abstraction. Next, machine learning is used on the resulting simulation traces to generate candidate conditions under which those modules can be abstracted. Finally, a verifier is invoked. If spurious counter-examples arise, we refine the abstraction by performing a further iteration of random simulation and machine learning. We present an experimental evaluation on processor designs.**

I. Introduction

Designs are usually specified at the register-transfer-level (RTL). For formal verification, however, RTL can be a rather low level of abstraction where data are represented as bits and bit vectors, and operations on data are accomplished by bit-level manipulation. In verification tasks that involve proving strongly data-dependent properties, such as equivalence or refinement checking, bit-level RTL quickly leads to state-space explosion, necessitating additional abstraction.

Term-level modeling can make formal verification of data-intensive properties tractable by abstracting away details of data representations and operations, viewing data as symbolic *terms* and operations as *uninterpreted* functions. Term-level abstraction has been found to be especially useful in microprocessor design verification [14], [18], [20], [21]. The precise functionality of units such as instruction decoders and the ALU are abstracted away using *uninterpreted functions*, and decidable fragments of first-order logic are employed in modeling memories, queues, counters, and other common constructs. Efficient satisfiability modulo theories (SMT) solvers [5] are then used as the computational engines for term-level verifiers.

A major obstacle for term-level verification is the need to generate term-level models from bit-vector-level (word-level) RTL. Two recent efforts have sought to automate the generation of term-level models. Andraus and Sakallah [4] were the first to address the problem, proposing a counterexample-guided abstraction refinement (CEGAR) approach. While the CEGAR technique works in some cases, it can require very many iterations of abstraction-refinement in other situations. Brady et al. [8] proposed ATLAS, an approach that combines random simulation with static analysis to compute *interpretation conditions* — conditions under which a functional block is replaced with an uninterpreted function. ATLAS avoids the need for several abstraction-refinement iterations by computing conservative interpretation conditions using static analysis. However, in some cases, these conditions are so large as to negate the advantages of term-level verification over word-level methods.

In this paper, we present CAL, a new technique for *automatically generating a term-level verification model* from a word-level description. The main focus of this work is function abstraction. Similar to ATLAS, CAL conditionally abstracts functional blocks in the original design with uninterpreted functions. In contrast with previous work, CAL uses a novel layered approach based on a combination of *random simulation*, *machine learning*, and *counterexample-guided abstraction-refinement*. In the first stage, we exploit the module structure specified by the designer using random simulation to identify functional blocks corresponding to module instantiations that are suitable for abstraction with uninterpreted functions. Replacing functional blocks with uninterpreted functions is always sound, that is, the correctness of the resulting abstract design implies the correctness of the original design. However, this abstraction loses information and can lead to spurious counterexamples. In the second stage, we use machine learning inside a CEGAR loop to rule out such spurious counterexamples. First, a verifier is invoked on the unconditionally abstracted verification model. If spurious counterexamples arise, machine learning is used to compute conditions under which abstraction can be performed without loss of precision; i.e., if the resulting term-level design is in correct, then so is the original word-level design. This process is repeated until we arrive with a term-level model that is valid or a legitimate counterexample is found. Fig. 1 illustrates the CAL approach.

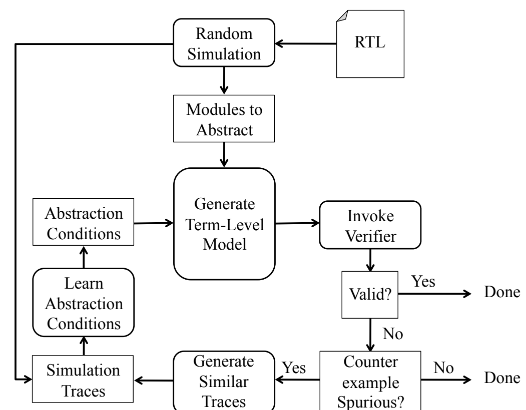


Fig. 1. **The** CAL **approach** A CEGAR-based approach, CAL identifies candidate abstractions with random simulation and uses machine learning to refine the abstraction if necessary.

[1]This work was conducted while the author was affiliated with the University of California, Berkeley.

We present experimental evidence that our approach is efficient and that the resulting term-level models are easier to verify. Moreover, we show that the abstraction conditions that we learn are as good or better than the previous best-known conditions.

The rest of this paper is organized as follows. We discuss some background material and related work in Section II. In Section III, we present the formal model for our work as well as some relevant ideas borrowed from our previous work on ATLAS [8]. Our new approach, CAL, is described in Section IV. Case studies are discussed in detail in Section V. We conclude in Section VI.

II. Background and Related Work

Background material on term-level abstraction is presented in Sec. II-A, function abstraction in Sec. II-B, and related work in Sec. II-C.

A. Term-Level Abstraction

Informally, a (word-level) design is said to be abstracted to the *term level* if one or more of the following three abstraction techniques is employed [8]:

1. *Function Abstraction:* In function abstraction, bit-vector operators and modules computing bit-vector values are treated as "black-box," *uninterpreted* functions constrained only by functional consistency. That is, they must evaluate to the same values when applied to the same arguments. It is possible for the inputs and outputs of uninterpreted functions to be bit vectors or to be abstract terms (say, interpreted over $\mathbb{Z}$). Function abstraction (applied selectively) is the focus of this paper, and we limit ourselves to uninterpreted functions that map bit vectors to bit vectors.
2. *Data Abstraction:* Bit-vector expressions are modeled as abstract terms that are interpreted over a suitable domain (typically a subset of $\mathbb{Z}$). Data abstraction is effective when it is possible to reason over the domain of abstract terms far more efficiently than it is to do so over the original bit-vector domain, through use of small-domain or bit-width reduction techniques. Data abstraction is not the focus of this paper.
3. *Memory Abstraction:* In memory abstraction, memories and data structures are modeled in a suitable theory of arrays or memories, such as by the use of special **read** and **write** functions [14] or lambda expressions [12]. We do not address automatic memory abstraction in this paper.

B. Function Abstraction

The concept of function abstraction is illustrated using a toy ALU design [8]. Consider the simplified ALU shown in Figure 2(a). Here a 20-bit instruction is split into a 4-bit opcode and a 16-bit data field. If the opcode indicates that the instruction is a jump, the data field indicates a target address for the jump and is simply passed through the ALU unchanged. Otherwise, the ALU computes the square of its 16-bit input and generates as output the resulting 16-bit result.

Using very coarse-grained term-level abstraction, one could abstract the entire ALU module with a single uninterpreted function (UF), as shown in Figure 2(b). However, we lose the precise mapping from `instr` to `out`.

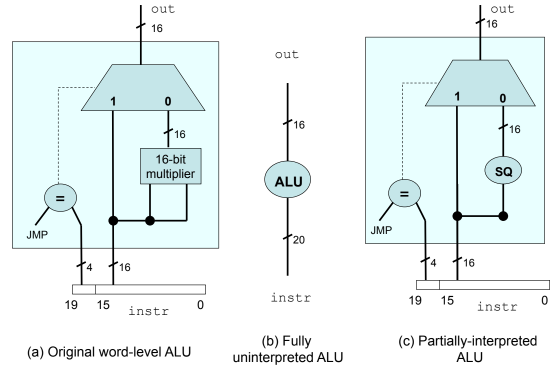


Fig. 2. **Three versions of an ALU design.** Boolean signals are shown as dashed lines and bit-vector signals as solid black lines [8].

Such a coarse abstraction is quite easy to perform automatically. However, this abstraction loses information about the behavior of the ALU on jump instructions and can easily result in spurious counterexamples. In Section III-B, we will describe a larger equivalence checking problem within which such an abstraction is too coarse to be useful.

Suppose that reasoning about the correctness of the larger circuit containing this ALU design only requires one to precisely model the difference in how the jump and squaring instructions are handled. In this case, it would be preferable to use a partially-interpreted ALU model as depicted in Figure 2(c). In this model, the control logic distinguishing the handling of jump and non-jump instructions is precisely modeled, but the datapath is abstracted using the uninterpreted function SQ. However, creating this fine-grained abstraction by hand is difficult in general and places a large burden on the designer. It is this burden that we seek to mitigate using the approach presented in this paper.

C. Related Work

The first automatic term-level abstraction tool was Vapor [4], which aimed at generating term-level models from Verilog. The underlying logic for term-level modeling in Vapor is CLU, which originally formed the basis for the UCLID system [12]. Vapor uses a counterexample-guided abstraction-refinement (CEGAR) approach [4]. Vapor has since been subsumed by the Reveal system [2], [3] which differs mainly in the refinement strategies in the CEGAR loop. Both Vapor and Reveal start by completely abstracting a Verilog description to the UCLID language by modeling all bit-vector signals as abstract terms and all operators as uninterpreted functions, and then iteratively rule out spurious counterexamples. While the CEGAR approach has shown much promise [3], in many cases, however, several abstraction-refinement iterations are needed to infer fairly straightforward properties of data, thus imposing a significant overhead [8]. While the approach presented in this paper is also counterexample-guided, we require very few refinement iterations in practice.

A more recent approach to automatic abstraction is ATLAS [8]. ATLAS exploits the module structure specified by the designer and uses random simulation to determine module instantiations that are candidates for function abstraction. ATLAS uses static analysis to heuristically compute *interpretation conditions* that specify when a functional block must be represented precisely.

While this works in many cases, for some benchmarks the interpretation conditions can grow extremely large, leading to poor performance [8]. Our approach, CAL, addresses this limitation by using a dynamic approach based on machine learning. As is the case with ATLAS, the CAL approach can be combined with bit-width reduction techniques (e.g. [7], [19]) to perform combined function and data abstraction.

To our knowledge, Clarke, Gupta et al. [15], [17] were the first to use machine learning to compute abstractions for model checking. Our work is similar in spirit to theirs. One difference is that we generate abstract, term-level models for SMT-based verification, whereas their work focuses on bit-level model checking and localization abstraction. Consequently, the learned concept is different: CAL learns Boolean interpretation conditions whereas their technique learns sets of variables to make visible. Additionally, our use of machine learning is more direct — e.g., while Clarke et al. [15] also use decision tree learning, they only indirectly use the learned decision tree (all variables branched upon in the tree are made visible), whereas we use the Boolean function corresponding to the entire tree as the learned interpretation condition.

III. PRELIMINARIES

We adopt the formal model used in [8]. In Sec. III-A, we present only the elements of this formal model necessary for the rest of the paper. An illustrative example is given in Sec. III-B.

A. Basic Definitions

We model a design at the word level as a *word-level netlist* $\mathcal{N}$ = ($\mathcal{I}$, $\mathcal{O}$, $\mathcal{S}$, $\mathcal{C}$, *Init*, $\mathcal{A}$) where

- $\mathcal{I}$ is a finite set of input signals;
- $\mathcal{O}$ is a finite set of output signals;
- $\mathcal{S}$ is a finite set of intermediate sequential (state-holding) signals;
- $\mathcal{C}$ is a finite set of intermediate combinational (stateless) signals;
- *Init* is a set of initial states, i.e., initial valuations to elements of $\mathcal{S}$, and
- $\mathcal{A}$ is a finite set of assignments to outputs and to sequential and combinational intermediate signals. An assignment is an expression that defines how a signal is computed and updated. We elaborate below on the form of assignments.

Input and output signals are assumed combinational, without loss of generality. A *combinational assignment* is a rule of the form $v \leftarrow e$, where v is a signal in the disjoint union $\mathcal{C} \uplus \mathcal{O}$ and e is an expression that is a function of $\mathcal{C} \uplus \mathcal{S} \uplus \mathcal{I}$. Combinational loops are disallowed. Similarly, a *sequential assignment* is a rule of the form $v := u$, where u is a signal. Signals v, u and expressions e are of three types: bit-vector, Boolean, or memory. For brevity, we omit the detailed syntax (see [8] for this), and present only the notation used in the paper. In word-level netlists, a memory is modeled as a flat array of bit-vector signals.

A *word-level design* $\mathcal{D}$ is a tuple $\langle \mathcal{I}, \mathcal{O}, \{\mathcal{N}_i \mid i = 1, \ldots, N\} \rangle$, where $\mathcal{I}$ and $\mathcal{O}$ denotes the set of input and output signals of the design, and the design is partitioned into a collection of N word-level netlists. A *well-formed* design is one where (i) every output of a netlist is either an output of the design or an input to some netlist (including itself) – i.e., there are no dangling outputs; and (ii) every input of a netlist is either an input to the design or exactly one output of some netlist. We refer to the netlists $\mathcal{N}_i$ as *functional blocks*, or *fblocks*.

A *term-level netlist* is a generalization of a word-level netlist where bit-vector and Boolean expressions can include applications of uninterpreted functions and predicates, written $UF(v_1, \ldots, v_k)$ and $UP(v_1, \ldots, v_k)$ for $k \geq 0$, and memory operations can be modeled in a suitable theory of arrays/memories using the usual **read** and **write** functions or lambda expressions [12].

A term-level netlist that has at least one expression of the form $UF(v_1, \ldots, v_k)$ or $UP(v_1, \ldots, v_k)$ is referred to as a *strict term-level netlist*. A *term-level design* $\mathcal{T}$ is a tuple $(\mathcal{I}, \mathcal{O}, \{\mathcal{N}_i \mid i = 1, \ldots, N\})$, where each fblock $\mathcal{N}_i$ is a term-level netlist.

Given a word-level design $\mathcal{D} = (\mathcal{I}, \mathcal{O}, \{\mathcal{N}_i \mid i = 1, \ldots, N\})$, we say that $\mathcal{T}$ is a *term-level abstraction* of $\mathcal{D}$ if $\mathcal{T}$ is obtained from $\mathcal{D}$ by replacing some word-level fblocks $\mathcal{N}_i$ by strict term-level fblocks $\mathcal{N}'_i$.

The verification problems of interest in this paper are *equivalence checking* and *refinement checking*.

Given two word-level designs $\mathcal{D}_1$ and $\mathcal{D}_2$, the *word-level equivalence* (*word-level refinement*) checking problem is to verify that $\mathcal{D}_1$ is *sequentially equivalent* to (*refines*) $\mathcal{D}_2$.

The definition is similarly extended to a pair of term-level designs $\mathcal{T}_1$ and $\mathcal{T}_2$. We also consider bounded equivalence checking problems, where the designs are to be proved equivalent for a bounded number of cycles from the initial state.

The *term-level abstraction problem* we consider in this paper is as follows.

> Given a pair of word-level designs $\mathcal{D}_1$ and $\mathcal{D}_2$, abstract them to term-level designs $\mathcal{T}_1$ and $\mathcal{T}_2$, such that $\mathcal{D}_1$ is equivalent to (refines) $\mathcal{D}_2$ if and only if $\mathcal{T}_1$ is equivalent to (refines) $\mathcal{T}_2$.

We follow the approach taken by ATLAS and generate the term-level abstraction by computing an *interpretation condition* — a condition under which we will retain the precise fblock in the term-level model (i.e, we replace the fblock by an uninterpreted function under the negation of the interpretation condition). The idea of conditional function abstraction is illustrated in Figure 3. The original word-level circuit is shown in Fig. 3(a) and the conditionally abstracted version with interpretation condition c is shown in Fig. 3(b).

In Section IV, we present our CAL approach to automatically generate term-level abstract models. In Section V, we show that using CAL can scale up verification by orders of magnitude.

B. Illustrative Example

Figure 4 depicts the equivalence checking problem that we will use as a running example [8]. Two variants of the same circuit, denoted Design A and Design B, are to be checked for output equivalence.

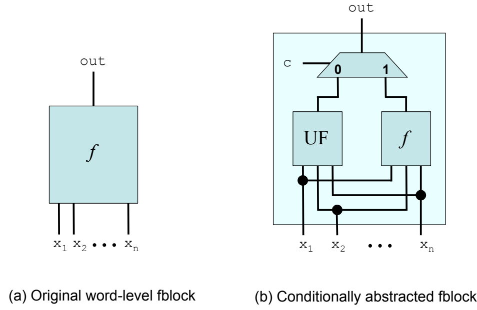


Fig. 3. **Conditional abstraction** (a) Original word-level fblock f. (b) Conditionally abstracted version of f with interpretation condition c

Consider Design A. This design models a fragment of a processor datapath. PC models the program counter register, which is an index into the instruction memory denoted as IMem. The instruction is a 20-bit word denoted instr and is an input to the ALU. The ALU is similar to the ALU design shown in Figure 2(a) – both ALUs pass the target address through unaltered when the instruction is a jump. The top four bits of instr are the operation code. If the instruction is a jump instruction (instr[19 : 16] equals JMP), then the PC is set equal to the ALU output out; otherwise, it is incremented by 4.

Design B is virtually identical to Design A, except in how the PC is updated. For this version, if instr[19 : 16] equals JMP, the PC is directly set to be the jump address instr[15 : 0].

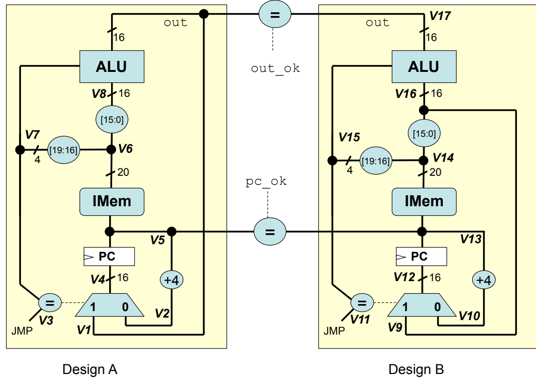


Fig. 4. **Equivalence checking of two versions of a portion of a processor design.** Boolean signals are shown as dashed lines and bit-vector signals as solid lines [8].

Note that we model the instruction memory as a read-only memory using an uninterpreted function IMem. The same uninterpreted function is used for both Design A and Design B. We also assume that Designs A and B start out with identical values in their PC registers.

The two designs are equivalent iff their outputs are equal at every cycle, meaning that the Boolean assertion out_ok $\wedge$ pc_ok is always **true**.

It is easy to see that this is the case, because from Figure 2(a) we know that A.out always equals A.instr[15 : 0] when A.instr[19 : 16] equals JMP. The question is whether we can infer this without the full word-level representation of the ALU.

Consider what happens if we use the abstraction of Figure 2(b). In this case, we lose the relationship between A.out and A.instr[19 : 16]. Thus, the verifier comes back to us with a spurious counterexample, where in cycle 1 a jump instruction is read, with the jump target in Design A different from that in Design B, and hence A.PC differs from B.PC in cycle 2.

However, if we instead used the partial term-level abstraction of Figure 2(c) then we can see that the proof goes through, because the ALU is precisely modeled under the condition that A.instr[19 : 16] equals JMP, which is all that is necessary.

The challenge is to be able to generate this partial term-level abstraction automatically. We describe our approach to solving this problem below.

IV. The CAL Approach

The main contribution of this paper is presented in this section. The goal of this step is to compute conditions under which it is precise to abstract using a machine-learning-based CEGAR loop.

A. Identifying Candidate fblocks

The first step in CAL is the same as in ATLAS: to use syntactic matching and random simulation to identify a set of fblocks that are candidates for replacement with uninterpreted functions. We review this procedure in this section since it is crucial to understand the rest of the CAL procedure.

The first step in identifying candidates for abstraction is to identify *replicated fblocks*. A replicated fblock is an fblock in $\mathcal{D}_1$ that has an isomorphic counterpart in $\mathcal{D}_2$. A formal definition can be found in [8]. In equivalence and refinement checking problems, identifying replicated fblocks is typically a matter of finding instances of the same RTL module present in both designs.

The fblock identification process generates a collection of sets of fblocks $\mathcal{FS} = \{\mathcal{F}_1, \mathcal{F}_2, ..., \mathcal{F}_k, \}$. Each set $\mathcal{F}_j$ contains replicated fblocks that are isomorphic to each other. $\mathcal{F}_j$ can be viewed as an equivalence class of the fblocks it contains. In later steps when function abstractions are computed, it is important to note that the same function abstraction is used for each fblock in $\mathcal{F}_j$.

The next step in the abstraction candidate identification process is to determine which equivalence classes $\mathcal{F} \in \mathcal{FS}$ will be considered for abstraction. This is accomplished using random simulation.

Fix an equivalence class $\mathcal{F}$. Let its cardinality be l. Let $f_i \in \mathcal{F}$ be an arbitrary fblock with m bit-vector output signals $\langle v_{i1}, \dots, v_{im} \rangle$, and n input signals $\langle u_{i1}, \dots, u_{in} \rangle$. Then, we term the tuple of corresponding output signals $\chi_j = (v_{1j}, v_{2j}, \dots, v_{lj})$, for each $j = 1, 2, \dots, m$, as a tuple of *isomorphic output signals*.

Given a tuple of isomorphic output signals $\chi_j = (v_{1j}, v_{2j}, \dots, v_{lj})$, we create a *random function* RF_{χ_j} unique to χ_j that has n inputs (corresponding to input signals $\langle u_{i1}, \dots, u_{in} \rangle$, for fblock f_i).

For each fblock $f_i \in \mathcal{F}$, $i = 1, 2, \dots, l$, we replace the assignment to the output signal v_{ij} with the random assignment $v_{ij} \leftarrow RF_{\chi_j}(u_{i1}, \dots, u_{in})$. This substitution is performed for all output signals $j = 1, 2, \dots, m$. The resulting designs $\mathcal{D}_1$ and $\mathcal{D}_2$ are then verified through simulation. Note that all other fblocks

not in $\mathcal{F}$ are interpreted precisely. This process is repeated for $\mathcal{F}$ using T different random functions RF_{χ_j}.

If the fraction of failing verification runs is greater than a threshold τ, then we drop the equivalence class $\mathcal{F}$ from further consideration. Otherwise, we retain $\mathcal{F}$ for further analysis, as described in the following section. It is important to note that we have not yet decided to replace fblocks in $\mathcal{F}$ with uninterpreted functions – this will be determined later using the counterexample-guided loop. We denote the set of equivalence classes that are to be considered for abstraction as $\mathcal{FS}_{\mathcal{A}}$.

B. Top-Level CAL *Procedure*

The top-level CAL procedure, VERIFYABS, is shown in Algorithm 1. VERIFYABS takes two arguments, the design $\mathcal{D}$ being verified (which includes both designs – e.g., it is the miter for equivalence checking) and the set of equivalence classes being abstracted $\mathcal{FS}_{\mathcal{A}}$. Initially, the interpretation conditions $c_i \in \mathcal{IC}$ are set to **false** meaning that we start by unconditionally abstracting the fblocks in $\mathcal{D}$. The procedure CONDABS creates the abstracted term-level design $\mathcal{T}$ from the word-level design $\mathcal{D}$, the set of equivalence classes to be abstracted $\mathcal{FS}_{\mathcal{A}}$, and the set of interpretation conditions $\mathcal{IC}$. Next, we invoke a term-level verifier on $\mathcal{T}$. If VERIFY ($\mathcal{T}$) returns "Valid", we report that result and terminate. If a counterexample arises, we evaluate the counterexample on the word-level design. If the counterexample is non-spurious, we report the counterexample and terminate, otherwise we store the counterexample in $\mathcal{CE}$ and invoke the abstraction condition learning procedure, LEARNABSCONDS ($\mathcal{D}, \mathcal{FS}_{\mathcal{A}}, \mathcal{CE}$).

We say that VERIFYABS is *sound* if it reports "Valid" only if $\mathcal{D}$ is correct. It is *complete* if it reports a true counterexample when $\mathcal{D}$ is incorrect. We have the following guarantee for the procedure VERIFYABS:

Theorem 1: If VERIFYABS terminates, it is sound and complete.

Proof: Any term-level abstraction is a sound abstraction of the original design, since any partially-interpreted function (for any interpretation condition) is a sound abstraction of the fblock it replaces. Thus VERIFYABS is sound. Moreover, VERIFYABS terminates with a counterexample only if it deems the counterexample to be non-spurious, by simulating it on the concrete design $\mathcal{D}$. Therefore VERIFYABS is complete. ■

In order to guarantee termination of VERIFYABS, we must impose certain constraints on the learning algorithm LEARNABSCONDS. This is formalized in the theorem below.

Theorem 2: Suppose that the learning algorithm LEARNABSCONDS satisfies the following properties:

- (i) If c_i denotes the interpretation condition for an fblock learned in iteration i of the VERIFYABS loop, then $c_i \implies c_{i+1}$ and $c_i \neq c_{i+1}$;
- (ii) The trivial interpretation condition $\texttt{true}$ belongs to the hypothesis space of LEARNABSCONDS, and
- (iii) The hypothesis space of LEARNABSCONDS is finite.

Then, VERIFYABS will terminate and return either Valid or a non-spurious counterexample.

Proof: Consider an arbitrary fblock that is a candidate for function abstraction. Let the sequence of interpretation conditions generated in successive iterations of the VERIFYABS loop be $c_0 = \textbf{false}, c_1, c_2, \ldots$. By condition (i), $c_0 \implies c_1 \implies c_2 \implies \ldots$ where $c_i \neq c_{i+1}$. Since no two elements of the sequence are equal, and the hypothesis space is finite, no element of the sequence can repeat. Thus, the sequence (for any fblock) forms a finite chain of implications. Moreover, since $\texttt{true}$ belongs to the hypothesis space, in the extreme case, VERIFYABS can generate in its final iteration the term-level design $\mathcal{T}$ identical to the original design $\mathcal{D}$, which will yield termination with either Valid or a non-spurious counterexample. ■

In practice, the conditions (i)-(iii) stated above can be implemented on top of any learning procedure. The most straightforward way is to set an upper bound on the number of iterations that LEARNABSCONDS can be invoked, after which the interpretation condition is set to $\texttt{true}$. Another option is to set c_{i+1} to $c_i \vee d_{i+1}$ where d_{i+1} is the condition learned in the $i+1$th iteration. Yet another option is to keep a log of the interpretation conditions generated, and if an interpretation condition is generated for a second time, the abstraction procedure is terminated by setting the interpretation condition to $\texttt{true}$. Many other heuristics are possible; we leave an exploration of these to future work.

Algorithm 1 Procedure VERIFYABS ($\mathcal{D}, \mathcal{FS}_{\mathcal{A}}$): Top-level CAL verification procedure.

1: // Input: Combined word-level design (miter) $\mathcal{D} := \langle \mathcal{I}, \mathcal{O}, \{\mathcal{N}_i \mid i = 1, \ldots, N\}\rangle$
2: // Input: Equivalence classes of fblocks $\mathcal{FS}_{\mathcal{A}} := \{\mathcal{F}_j \mid j = 1, \ldots, k\}$
3: // Output: Verification result $Result \in \{\mathsf{Valid}, \mathsf{CounterExample}\}$
4: Set $c_i = \textbf{false}$ for all $c_i \in \mathcal{IC}$.
5: **while true do**
6: $\quad \mathcal{T}$ = CONDABS ($\mathcal{D}, \mathcal{FS}_{\mathcal{A}}, \mathcal{IC}$)
7: $\quad Result$ = VERIFY ($\mathcal{T}$)
8: $\quad$ **if** $Result = \mathsf{Valid}$ **then**
9: $\quad\quad$ Return Valid.
10: $\quad$ **else**
11: $\quad\quad$ Store counterexample in $\mathcal{CE}$.
12: $\quad\quad$ **if** $\mathcal{CE}$ is spurious **then**
13: $\quad\quad\quad \mathcal{IC} \leftarrow$LEARNABSCONDS ($\mathcal{D}, \mathcal{FS}_{\mathcal{A}}, \mathcal{CE}$)
14: $\quad\quad$ **else**
15: $\quad\quad\quad$ Return CounterExample.
16: $\quad\quad$ **end if**
17: $\quad$ **end if**
18: **end while**

Procedure CONDABS ($\mathcal{D}, \mathcal{FS}_{\mathcal{A}}, \mathcal{IC}$) is responsible for creating a term-level design $\mathcal{T}$ from the original word-level design $\mathcal{D}$, the set of equivalence classes to be abstracted $\mathcal{FS}_{\mathcal{A}}$, and the set of interpretation conditions $\mathcal{IC}$. CONDABS operates by iterating through the equivalence classes in $\mathcal{FS}_{\mathcal{A}}$. A fresh uninterpreted function symbol UF_j is created for each tuple of isomorphic output signals χ_j associated with equivalence class $\mathcal{F}_i \in \mathcal{FS}_{\mathcal{A}}$. Each output signal $v_{ij} \in \chi_j$ is conditionally abstracted with UF_j as illustrated in Fig. 3. More formally, if $c_{v_{ij}} \in \mathcal{IC}$ denotes the interpretation condition associated with v_{ij}, then we replace the assignment $v_{ij} \leftarrow e$ in fblock f_i with the assignment $v_{ij} \leftarrow$

ITE$(\mathsf{c}_{v_{ij}}, e, UF_j(i_1, ..., i_k))$, where *ITE* denotes the if-then-else operator. See [8] for a more detailed description.

C. Learning Conditional Abstractions

Spurious counterexamples arise due to imprecision introduced during abstraction. More specifically, when a spurious counterexample arises, it means that at least one fblock $f_i \in \mathcal{F}$ (where $\mathcal{F} \in \mathcal{FS_A}$) is being abstracted when it needs to be modeled precisely. In the context of our abstraction procedure VERIFYABS, if VERIFY $(\mathcal{T})$ returns a spurious counterexample $\mathcal{CE}$, then we must invoke the procedure LEARNABSCONDS $(\mathcal{D}, \mathcal{FS_A}, \mathcal{CE})$.

The LEARNABSCONDS procedure invokes a *decision tree learning* algorithm on traces generated by replacing fblocks $f_i \in \mathcal{F}$ by a tuple of random functions RF_{χ_j}. Traces are classified as being "bad" or "good" depending on whether the replacement with a random function results in a property violation or not. The learning algorithm generates a classifier in the form of a decision tree to separate the good traces from the bad ones. The classifier is essentially a Boolean function over signals in the original word-level design. More information about decision tree learning can be found in Mitchell's textbook [22].

There are three main steps in the LEARNABSCONDS procedure:

1. Generate good and bad traces for the learning procedure;
2. Determine meaningful features that will help decision tree learning procedures compute high quality decision trees, and
3. Invoke a decision tree learning algorithm with the above features and traces.

The data input to the decision tree software is a set of tuples where one of the tuple elements is the target attribute and the remaining elements are features. In our context, a target attribute α is either Good or Bad. Our goal is to select features such that we can classify the set of all tuples where $\alpha = \mathsf{Bad}$ based on the rules provided by the decision tree learner. Since we use an off-the-shelf decision tree learning tool, we omit a description of how this works. However, it is very important to provide the decision tree learning with quality input data and features, otherwise, the rules generated will not be of use. The data generation procedure is described in Sec. IV-D and feature selection is described in Sec. IV-E.

D. Generating Data

In order to produce a meaningful decision tree, we must provide the decision tree learner with both good and bad traces. We use random simulation to generate *witnesses* and *counterexamples* and describe these procedures in detail below.

1) Generating Witnesses: Good traces, or *witnesses*, are generated using a modified version of the random simulation procedure described in Sec. IV-A. Instead of simulating the abstract design when only a single fblock has been replaced with a random function, we replace all fblocks with their respective random functions *at the same time* and perform verification via simulation. Replacing *all* the fblocks to be abstracted with the respective random function ensures diversity in the set of traces fed to the decision tree learner.

After replacing each fblock to be abstracted with the corresponding random functions, we perform simulation by verification, repeating the process for N different random functions for each fblock. N is chosen heuristically similar to T in Sec. IV-A (we discuss typical values for N in Sec. V-D). The initial state of design $\mathcal{D}$ is set randomly before each run of simulation. This usually results in simulation runs that pass, and hence in good traces — recall that at this stage we only consider fblocks that produce failing runs in a small fraction of simulation runs. Now, instead of only logging the result of the simulation, we log the value of every signal in the design for every cycle of each passing simulation. It is up to the feature selection step, described in Sec. IV-E, to decide what signals are important.

2) Generating Similar Counterexamples: Whenever LEARNABSCONDS is called, there is a spurious counterexample stored in $\mathcal{CE}$. We generate many counterexamples similar to $\mathcal{CE}$ using random simulation in a manner similar to that used while identifying abstraction candidates. If more than one equivalence class of fblocks has been abstracted, the counterexample $\mathcal{CE}$ can be the result of abstracting any individual equivalence class, or a combination of them.

Consider the situation where $\mathcal{CE}$ is the result of only abstracting a single equivalence class. In this situation, we replace each fblock in that class with a random function in the word-level design, just as we did when identifying abstraction candidates in Sec. IV-A. Next, verification via simulation is performed, and this process is iterated for N different random functions, for heuristically-chosen N. A main point of difference between generating similar counterexamples and generating witnesses is that in generating similar counterexamples, we set the initial state of design $\mathcal{D}$ to be *consistent with the initial state in* $\mathcal{CE}$, whereas we randomly set the initial state of design $\mathcal{D}$ when generating witnesses. We log the values of every signal in the design for each failing simulation run. It is possible that none of the simulation runs fail, because the counterexample could be the result of abstracting a different equivalence class. We repeat this process for each fblock that is being abstracted.

If replacing individual equivalence classes with random functions does not result in any failing simulation run, we must take into account combinations of equivalence classes. In this case, we try pairs of equivalence classes, then triples, and so on. Clearly, there is a potential exponential blowup here; however, this has not occurred in our experiments. In fact, considering a single equivalence class at a time sufficed for all examples considered in this work. We leave the exploration of heuristics that determine how to choose interpretation conditions for combinations of fblocks for future work.

As noted above, the witness generation and the counterexample generation procedures can both generate good and bad traces. Denote the set of all bad traces by Bad and the good traces as $Good$. We label each trace in Bad with the Bad attribute and each trace in $Good$ with the Good attribute.

E. Choosing Features

The quality of the decision tree generated is highly dependent on the features used to generate the decision tree. We use two heuristics to identify features:

1. Include input signals to the fblock being abstracted.
2. Include signals encoding the "unit-of-work" being processed by the design, such as the instruction being executed.

Input signals. Suppose we wish to determine when fblock f must be interpreted. It is very likely that whether or not f must be interpreted is dependent on the inputs to f. So, if f has input signals $(i_1, i_2, ..., i_n)$ it is almost always the case that we would include the input arguments as features to the decision tree learner.

Unit-of-work signals. There are cases when the input arguments alone are not enough to generate a quality decision tree. In these cases, human insight can be provided by defining the unit-of-work being performed by the design. For example, in a microprocessor design, a unit-of-work is an instruction. Similarly, in a network-on-a-chip (NoC), the unit-of-work is a packet, where the relevant signals could include the source address, destination address, or possibly the type of packet being sent across the network. Once a unit-of-work signal is identified at one part of the design, automatic dataflow analysis can identify all signals derived from it and label these also as features for the learning algorithm. For instance, in the case of a pipelined processor, the registers storing instructions in *each stage* of the pipeline are relevant signals to treat as features.

In rare cases, the above heuristics are not enough to generate quality decision trees; we discuss these scenarios and give additional features in Sec. V.

V. CASE STUDIES

We performed two case studies to evaluate CAL. Both of these case studies have also been verified using ATLAS. Additionally, each case study requires a non-trivial interpretation condition (i.e., an interpretation condition different from **false**). The first case study involves verifying the example shown in Fig. 4. In the second case study, we verify, via correspondence checking, two versions of the Y86 microprocessor.

All experiments were run on a MacBook Pro with a 2.4 GHz Intel Core 2 Duo processor with 4GB RAM. The term-level verification engine used for the experiments was the UCLID verification system [1], [9] with Minisat2 [16] and Boolector [11] as the SAT and SMT backends, respectively. Random simulation was performed using Icarus Verilog [25]. The decision tree learner we used in the experiments is C5.0 [23]. We compared our term-level abstraction-based approach with the state-of-the-art bit-level equivalence checker, ABC [6], [10]. The benchmarks used in these experiments as well as the results can be found at [24].

A. The Illustrative Example

In this experiment, we perform equivalence checking between Design A and B shown in Fig. 4. First, we initialize the designs to the same initial state and inject an arbitrary instruction. Then we check whether the designs are in the same state. The precise property that we wish to prove is that the ALU and PC outputs are the same for design A and B. Let $\mathtt{out_A}$ and $\mathtt{out_B}$ denote the ALU outputs and $\mathtt{pc_A}$ and $\mathtt{pc_B}$ denote the PC outputs for designs A and B, respectively. The property we prove is: $\mathtt{out_A} = \mathtt{out_B} \wedge \mathtt{pc_A} = \mathtt{pc_B}$. Aside from the top-level modules, the design consists of only two modules, the instruction memory (IMEM) and the ALU. We do not consider the instruction memory for abstraction because we do not address automatic memory abstraction. The ALU passes the random simulation stage, so it is an abstraction candidate.

The features we use in this case are arguments to the ALU; the instruction and the data arguments. The interpretation condition learned from the trace data is $\mathtt{op} = \mathsf{JMP}$ where $\mathtt{op}$ is the top 4 bits of the instruction. As shown in Table I, the runtime for CAL is comparable with that of ABC.

Interpretation Condition	Runtime (sec)		
		UCLID	
	ABC	SAT	SMT
true	0.02	28.51	27.01
$\mathtt{op} = \mathsf{JMP}$	—	**0.31**	**0.01**

TABLE I
Performance comparison Runtime comparison between ABC and UCLID for the processor fragment shown in 4. The runtime associated with the model abstracted with CAL is shown in **bold**.

B. The Y86 Processor

In this experiment, we verify two versions of the well-known Y86 processor model introduced by Bryant and O'Hallaron [13]. The Y86 processor is a pipelined CISC microprocessor styled after the Intel IA32 instruction set. While the Y86 is relatively small for a processor, it contains several realistic features, such as a dual read, dual write register file, separate data and instruction memories, branch prediction, hazard resolution, and an ALU that supports bit-vector arithmetic and logical instructions. Note that we have extended the ALU to include multiplication in order to create a harder verification problem. Of the several variants of the Y86 processor we focus on two that have different versions of branch prediction logic: NT and BTFNT. These versions are the only versions where the ALU cannot be fully abstracted (i.e., partial abstraction is required). In NT branches are predicted as not taken, whereas in BTFNT branches backwards in the address space are predicted as taken, while branches forward in the address space are predicted as not taken. NT and BTFNT were the designs that the ATLAS approach had the most difficulty abstracting [8]. The property we wish to prove on the Y86 variants is Burch-Dill style correspondence-checking [14].

Both NT and BTFNT versions have the same module hierarchy and differ only in the logic pertaining to branch prediction. The following modules are candidates for abstraction: register file (RF), condition code (CC), branch function (BCH), arithmetic-logic unit (ALU), instruction memory (IMEM), and data memory (DMEM). The RF module is ruled out as a candidate for abstraction during the random simulation stage due to a large number of failures during verification via simulation. This occurs because an uninterpreted function is unable to accurately model a mutable memory. We do not consider IMEM and DMEM for automatic abstraction because they are memories and we do not address automatic memory abstraction in this work. Instead, we manually model IMEM and DMEM with completely uninterpreted functions. The CC and BCH modules are also removed from consideration due to the relatively simple logic contained within them. Abstracting these modules is unlikely to yield substantial

verification gains and may even hurt performance due to the overhead associated with uninterpreted functions. This leaves us with the ALU module.

1) Decision tree feature selection: In the case of both BTFNT and NT using only the arguments of the abstracted ALU is not sufficient to generate a useful decision tree. The ALU takes three arguments, the op-code op and two data arguments a and b. Closer inspection of the data provided to the decision tree learner reveals a problem. In almost every cycle of both good and bad traces, the ALU op is equal to ALUADD and the b argument is equal to 0.

In this situation, the arguments to the ALU are not good features by themselves (i.e., there is not enough diversity within the traces to learn a useful classifier). Conceptually, the unit-of-work that we are performing in a pipelined processor is a sequence of instructions, specifically the instructions that are currently in the pipeline. The most relevant instruction is the instruction currently in the execute stage (i.e., the stage containing the ALU). When $Instr_E$, op, a, and b are used as features, the resulting decision tree yields the interpretation condition: $\mathsf{c}_{E,b} := Instr_E = \mathsf{JXX} \wedge b = 0$.

The main reason we need a partial abstraction is so that the target address of a jump instruction can pass through the ALU unaltered. Thus, this is the best interpretation condition we can hope for. In fact, in previous attempts to manually abstract the ALU in the BTFNT version, we used: $\mathsf{c}_{Hand} := op = \mathsf{ALUADD} \wedge b = 0$.

When we compare the runtimes for verification of the Y86-BTFNT processor, we see that verifying BTFNT with the interpretation condition $\mathsf{c}_{E,b}$ outperforms the unabstracted version and the previously best known abstraction condition (c_{Hand}). Table II compares the UCLID runtimes for the Y86 BTFNT model with the different versions of the abstracted ALU.

Interpretation Condition	Runtime (sec)		
		UCLID	
	ABC	SAT	SMT
true	> 1200	> 1200	> 1200
c_{Hand}	—	133.03	105.34
$\mathsf{c}_{E,b}$	—	**101.10**	**65.52**

TABLE II

Performance comparison Runtime comparison between ABC and UCLID for Y86-BTFNT for different interpretation conditions. The runtime associated with the model abstracted with CAL is shown in **bold**.

2) Abstraction-refinement: The NT version of the Y86 processor requires an additional level of abstraction refinement. In general, requiring multiple iterations of abstraction refinement is not interesting by itself. However, it is interesting to see how the interpretation conditions change using this machine learning-based approach.

Attempting unconditional abstraction of the ALU in the NT version results in a spurious counterexample. The interpretation condition learned from the traces generated in this step is $\mathsf{c} := a = 0$. It is interesting that the same interpretation condition is generated regardless of whether we consider all of the instructions as features, or only the instruction in the same stage as the ALU. Not surprisingly, the second attempt at verification using the interpretation condition c results in another spurious counterexample. In this case, the interpretation condition learned is $\mathsf{c}_E := Instr_E = \mathsf{ALUADD}$, which states that we must interpret anytime an addition operation is present in the ALU. Similarly with the first iteration, the interpretation condition learned is the same regardless of whether we use all of the instructions as features, or only the instruction in the execute stage. Verification is successful when c_E is used as the interpretation condition.

A performance comparison for the NT variant of the Y86 processor is shown in Table III. Unlike the BTFNT case, the abstraction condition we learn for the NT model is not quite as precise as the previously best known interpretation condition, and the performance isn't as good. However, the runtimes for conditional abstraction, including the time spent in abstraction-refinement, are smaller than that of verifying the original word-level circuit. That is, the runtime when the interpretation condition is c_E is accounting for two runs of UCLID that produce a counterexample and an additional run when the property is proven Valid. Note that the most precise abstraction condition is the same for both BTFNT and NT. The best performance on the NT version is obtained when the interpretation condition $\mathsf{c}_{BTFNT} := Instr_E = \mathsf{JXX} \wedge b = 0$ is used.

Condition	Runtime (sec)		
		UCLID	
	ABC	SAT	SMT
true	> 1200	> 1200	> 1200
c_{Hand}	—	154.95	89.02
c_E	—	**191.34**	**187.64**
c_{BTFNT}	—	94.00	52.76

TABLE III

Performance comparison Runtime comparison between ABC and UCLID for Y86-BTFNT for different interpretation conditions. The runtime associated with the model abstracted with CAL is shown in **bold**.

The reason the interpretation condition for BTFNT differs from that of NT is because the root cause of the counterexamples are different. The counterexample generated for the BTFNT model arises because the branch target that would pass through the ALU unaltered, gets mangled when the ALU is abstracted. The counterexample generated for the NT model arises because the abstracted ALU incorrectly squashes a properly predicted branch.

C. *Comparison with* ATLAS

ATLAS and CAL compute the same interpretation conditions for the processor fragment described in Sec. V-A. Thus, the only interesting comparison with regard to the interpretation conditions is for the Y86 design.

ATLAS is able to verify both BTFNT and NT Y86 versions with one caveat—the multiplication operator was removed from the ALU to create a more tractable verification problem. When multiplication is present inside the ALU, the ATLAS approach can not verify BTFNT or NT in under 1200 seconds. In the case where the multiplication operator is removed, the interpretation conditions generated by ATLAS simplify to **true**. In this case, ATLAS actually takes longer to verify BTFNT, with the abstracted version taking 1390 seconds and the word-level version taking only 1077 seconds. This behavior highlights the main drawback of ATLAS. The static analysis procedure blindly takes into account the structure of the design, giving equal importance to every

signal. This was the inspiration behind using machine learning to compute interpretation conditions. Not only is CAL able to verify the BTFNT and NT Y86 versions when multiplication *is* included in the ALU, but it does so with an order of magnitude speedup over the unabstracted version.

D. Remarks

The runtimes listed in Tables I, II, and III focus only on the time taken by ABC and UCLID. The remaining runtime taken by the other components of the CAL procedure is, in comparison, negligible. First, the runtime of the decision tree learner is less than 0.1 seconds in every case. Second, the simulation time is quite small. For instance, simulating 1000 correspondence checking runs for the Y86 model takes less than 5 seconds. However, we are unable to verify the original word-level Y86 designs within 1200 seconds, so the CAL runtime is negligible. Similarly, the runtime of generating an AIG for input to ABC is less than 1 second.

The number of good and bad traces required to produce a quality decision tree for the processor fragment example in Sec. V-A is 5 (10 total). For the Y86 examples, the number of good and bad traces was 50 (100 total). Thus, in every example, it takes only a fraction of a second to generate enough data for the machine learning algorithm to be able to produce useful results.

VI. Conclusion

In this paper, we present CAL, an automatic abstraction procedure based on a combination of random simulation and machine learning. We evaluate the effectiveness and efficiency of our approach on equivalence and refinement checking problems in the context of pipelined processors. We have shown that we are able to automatically learn conditional abstractions that lead to better verification performance. Additionally, we learned abstraction conditions that were better than the previously best known abstraction conditions for two variants of the Y86 microprocessor design.

Acknowledgements. This research was supported in part by SRC contract 2045.001 and an Alfred P. Sloan Research Fellowship.

References

[1] UCLID Verification System. Available at http://uclid.eecs.berkeley.edu.

[2] Z. S. Andraus, M. H. Liffiton, and K. A. Sakallah. Refinement strategies for verification methods based on datapath abstraction. In *Proceedings of ASP-DAC*, pages 19–24, 2006.

[3] Z. S. Andraus, M. H. Liffiton, and K. A. Sakallah. CEGAR-based formal hardware verification: A case study. Technical Report CSE-TR-531-07, University of Michigan, May 2007.

[4] Z. S. Andraus and K. A. Sakallah. Automatic abstraction and verification of Verilog models. In *Proceedings of the 41st Design Automation Conference (DAC)*, pages 218–223, 2004.

[5] C. Barrett, R. Sebastiani, S. A. Seshia, and C. Tinelli. Satisfiability modulo theories. In *Handbook of Satisfiability*, volume 4, chapter 8. IOS Press, 2009.

[6] Berkeley Logic Synthesis and Verification Group. ABC: A System for Sequential Synthesis and Verification. http://www.eecs.berkeley.edu/~alanmi/abc.

[7] P. Bjesse. A practical approach to word level model checking of industrial netlists. In *CAV '08: Proceedings of the 20th international conference on Computer Aided Verification*, pages 446–458, Berlin, Heidelberg, 2008. Springer-Verlag.

[8] B. A. Brady, R. E. Bryant, S. A. Seshia, and J. W. O'Leary. ATLAS: automatic term-level abstraction of RTL designs. In *Proceedings of the Eighth ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE)*, July 2010.

[9] B. A. Brady, S. A. Seshia, S. K. Lahiri, and R. E. Bryant. *A User's Guide to UCLID Version 3.0*, October 2008.

[10] R. K. Brayton and A. Mishchenko. ABC: An academic industrial-strength verification tool. In *CAV*, pages 24–40, 2010.

[11] R. D. Brummayer and A. Biere. Boolector: An efficient SMT solver for bit-vectors and arrays. In *In* Proc. of TACAS, pages 174–177, March 2009.

[12] R. E. Bryant, S. K. Lahiri, and S. A. Seshia. Modeling and verifying systems using a logic of counter arithmetic with lambda expressions and uninterpreted functions. In *Proc. Computer-Aided Verification (CAV'02)*, LNCS 2404, pages 78–92, July 2002.

[13] R. E. Bryant and D. R. O'Hallaron. *Computer Systems: A Programmer's Perspective*. Prentice-Hall, 2002. Website: http://csapp.cs.cmu.edu.

[14] J. R. Burch and D. L. Dill. Automated verification of pipelined microprocessor control. In D. L. Dill, editor, *Computer-Aided Verification (CAV '94)*, LNCS 818, pages 68–80. Springer-Verlag, June 1994.

[15] E. M. Clarke, A. Gupta, J. H. Kukula, and O. Strichman. SAT based abstraction-refinement using ilp and machine learning techniques. In *Proc. Computer-Aided Verification (CAV)*, volume 2404 of *Lecture Notes in Computer Science*, pages 265–279, 2002.

[16] N. Eén and N. Sörensson. The MiniSAT Page. http://minisat.se.

[17] A. Gupta and E. M. Clarke. Reconsidering CEGAR: Learning good abstractions without refinement. In *Proc. International Conference on Computer Design (ICCD)*, pages 591–598, 2005.

[18] W. A. Hunt. Microprocessor design verification. *Journal of Automated Reasoning*, 5(4):429–460, 1989.

[19] P. Johannesen. BOOSTER: Speeding up RTL property checking of digital designs through word-level abstraction. In *Computer Aided Verification*, 2001.

[20] S. K. Lahiri and R. E. Bryant. Deductive verification of advanced out-of-order microprocessors. In *Proc. 15th International Conference on Computer-Aided Verification (CAV)*, volume 2725 of *LNCS*, pages 341–354, 2003.

[21] P. Manolios and S. K. Srinivasan. Refinement maps for efficient verification of processor models. In *Design, Automation, and Test in Europe (DATE)*, pages 1304–1309, 2005.

[22] T. M. Mitchell. *Machine Learning*. McGraw-Hill, 1997.

[23] R. Quinlan. Rulequest research. http://www.rulequest.com.

[24] The CAL Approach. CAL: Conditional Abstraction through Learning. http://uclid.eecs.berkeley.edu/cal.

[25] S. Williams. Icarus Verilog. http://www.icarus.com/eda/verilog/.

Efficient Implementation of Property Directed Reachability

Niklas Een, Alan Mishchenko, Robert Brayton
{een,alanmi,brayton}@eecs.berkeley.edu

Berkeley Verification and Synthesis Research Center
EECS Department
University of California, Berkeley, USA.

***Abstract*—Last spring, in March 2010, Aaron Bradley published the first truly new bit-level symbolic model checking algorithm since Ken McMillan's interpolation based model checking procedure introduced in 2003. Our experience with the algorithm suggests that it is stronger than interpolation on industrial problems, and that it is an important algorithm to study further. In this paper, we present a simplified and faster implementation of Bradley's procedure, and discuss our successful and unsuccessful attempts to improve it.**

I. Introduction

Sequential verification is hard, both model checking and equivalence checking. Difficult instances are typically solved using several simplification steps followed by multiple verification engines scheduled sequentially or concurrently. Despite all the available tools, numerous practical instances remain unsolved. Therefore, research in formal verification is always on the lookout for methods that can handle difficult cases.

In 2003, a new verification method based on *interpolation* [7], was proposed by Ken McMillan to address hard UNSAT instances. Over time it was perfected and is currently considered one of the most valuable formal verification methods.

More recently, another novel method was pioneered by Aaron Bradley [1], [2]. He named his implementation IC3, but gave no name to the method itself. We choose to call it *property directed reachability* (PDR) to connect it to Bradley's earlier work on property directed invariant generation.

It came as a surprise that IC3 won the third place in the hardware model checking competition (HWMCC) at CAV 2010. It was marginally outperformed by two mature integrated verification systems, both carefully tuned to balance several different engines. As such, the new method appears to be the most important contribution to bit-level formal verification in almost a decade.

Although PDR has been generally known for less than a year, while interpolation has been around long enough for numerous improvements and extensions to be proposed, for example [4], [3], an up-to-date implementation of PDR can solve more instances from HWMCC than interpolation can. This is also true for the benchmarks our group has received from our industrial collaborators. Another remarkable property of PDR is its capability of finding deep counterexamples. Although on average BMC does better than PDR, there are many benchmarks where PDR can find counterexamples that elude both BMC and BDD reachability. Finally, PDR lends itself naturally to parallel implementation, as was explained in Bradley's original work.

In this paper, we explore PDR and try to understand the reason for its effectiveness. We propose a number of changes to the algorithm to improve its performance and to simplify its implementation. In particular:

- We achieve a significant speedup by using three-valued simulation to reduce the burden on the SAT-solver.
- We eliminate a tedious and error-prone special-case handling of counterexamples of length 0 or 1.
- We show experimentally that two elements of the original algorithm give no speedup: (i) variable activity and (ii) cube generalization beyond non-inductive regions.
- We separate the main algorithm from the handling of SAT queries through a clean interface. This separation reduces the overall complexity.
- We refute some potential improvements experimentally.
- We present detailed pseudo-code to fully document our implementation.

II. Preliminaries

This paper considers the verification of systems modeled using finite state machines (FSMs). Each state of the FSM is identified with a boolean assignment to a set of state variables. The FSM further defines a set of *initial states* and a set of *property states*. The algorithm to be presented verifies that there exists no sequence of transitions from an initial state to a non-property state ("bad" state).

In the presentation, a state variable or its negation is referred to as a *literal*, a conjunction of literals as a *cube*, and the negation of a cube (a disjunction of literals) as a *clause*. If a cube contains all the state variables, it is called a *minterm*. It is assumed that the FSM is represented symbolically in a way that can be translated into propositional logic for a SAT-solver.

III. Overview of PDR

The PDR algorithm can be understood on several levels. This section addresses:

(1) How it works.
(2) Why it is complete.
(3) What makes it effective.

In particular the first two points can be understood in terms of approximate reachability analysis. For the third point one must also consider the inductive flavor of the algorithm.

A. Notation

Let $\mathbf{I}$ and $\mathbf{P}$ be predicates over the FSM's state variables, denoting the initial states and the property states respectively. Also let $\mathbf{T}$ denote the transition relation over the current and next state variables. Given a cube s, a call to the underlying SAT solver will be expressed as:

$$SAT?\big[\mathbf{P} \wedge \neg s \wedge \mathbf{T} \wedge s'\big]$$

using primes to denote next states. This query asks "starting in a state where the property holds, but outside the cube s, can you get inside the cube s in one transition". If the answer is UNSAT, then it has been proved that $(\mathbf{P} \wedge \neg s \wedge \mathbf{T}) \rightarrow \neg s'$, and $\neg s$ is said to be inductive *relative* to $\mathbf{P}$.

In the algorithm, some cubes will be proved unreachable from the initial states in k steps or less. Such cubes will be referred to as *blocked cubes* of frame k.

B. Mechanics

PDR maintains a list of facts which we will call the *trace*: [$\mathbf{R}_0$, $\mathbf{R}_1$, ... $\mathbf{R}_N$]. The first element $\mathbf{R}_0$ is special; it is simply identified with the initial states. For $k > 0$, $\mathbf{R}_k$ is a set of clauses that AND-ed together represent an over-approximation of the states reachable from the initial states in k steps or less. The trace is maintained in such a way that $\mathbf{R}_i$ is contained in $\mathbf{R}_{i+1}$. In fact, this relation is syntactic: every clause of $\mathbf{R}_{i+1}$ is also present in $\mathbf{R}_i$, except for $i = 0$ ($\mathbf{R}_0$ has no clauses).

Together with the trace, the PDR algorithm maintains a set of *proof-obligations*. A proof-obligation consists of a frame number k and a cube s, where s is either a set of bad states or a set of states that can all reach a bad state in one or more transitions. The frame number k indicates a position in the trace where s must be proved unreachable, or else the property fails.

By manipulating the trace and the set of proof-obligations according to a scheme detailed below, PDR derives new facts and adds them to the trace until it either (i) produced an inductive invariant proving the property, or (ii) added a proof-obligation at frame 0 with a cube that intersects the initial states. Such a cube cannot be blocked and entails the existence of a counterexample.

(1) **PROOF-OBLIGATIONS***:* The core of PDR lies in how proof-obligations are handled, and how new facts are derived from them. All reasoning in PDR take place on one transition relation; there is no unrolling of the FSM as in, e.g., BMC. Given the proof-obligation (s, k), consider the query:

$$SAT?\big[\mathbf{R}_{k-1} \wedge \mathbf{T} \wedge s'\big] \qquad (Q_1)$$

If it is UNSAT, then the facts of $\mathbf{R}_{k-1}$ are strong enough to block s at frame k, and we can add the clause $\neg s$ to $\mathbf{R}_k$. However, the syntactic containment relation of the trace requires us also to add the same clause to all preceding $\mathbf{R}_i$, $i < k$. Is it sound to do this? Consider replacing $\mathbf{R}_{k-1}$ with $\mathbf{R}_{k-2}$ in the query. Containment states that $\mathbf{R}_{k-2}$ is stronger than $\mathbf{R}_{k-1}$, so the query remains UNSAT. Likewise for $\mathbf{R}_{k-3}$ and so on, all the way back to the initial states. The only thing left to check is whether s intersects the initial states or not. If s is not blocked by $\mathbf{R}_0$, then we cannot strengthen the trace by $\neg s$. In the algorithm, this query will not be used if s overlaps with the initial states.

Using this approach, the quality of the learned clause depends on the size of the cube in the proof-obligation. In practice, these cubes often have many literals, and the negation $\neg s$ is a weak fact. It turns out to be crucial for the performance of PDR to try to learn stronger facts, i.e. cubes with fewer literals. To achieve this, the above learning scheme is improved in two ways:

Improvement 1 – "Generalize s". Many modern SAT-solvers do not simply return UNSAT, but also give a reason for the unsatisfiability; either through an UNSAT-core or through a final conflict-clause. Both these mechanisms can be used to extract precise information about which clauses were actually used in the proof. Since s is a conjunction inside the query, it translates into a set of unit clauses. Not all of those clauses may actually be needed when proving UNSAT. Any literal of s corresponding to an unused clause can be removed without affecting the UNSAT status. This provides a virtually free mechanism of removing literals that just happen not to be used by the SAT-solver.

A more directed, but also more expensive, approach is to *explicitly* try to remove the literals one by one. If the query remains UNSAT in the absence of a literal, good riddance. If not, put the literal back. Although the order in which literals are probed affects the outcome, the procedure is monotone in the sense that removing a literal cannot make a satisfiable query UNSAT. Note that we cannot remove a literal if it makes s intersect with the initial states, even if the query is UNSAT.

Improvement 2 – "Add $\neg s$ to the query". A key insight of Bradley was to realize that the query could be extended by the term $\neg s$:

$$SAT?\big[\mathbf{R}_{k-1} \wedge \neg s \wedge \mathbf{T} \wedge s'\big] \qquad (Q_2)$$

Adding an extra conjunct means the query is more likely to be UNSAT, which improves chances of removing a literal, or indeed learning a clause at all. This extended query is depicted in *Figure 1*. Having s on both sides of the transition breaks monotonicity: as s gets weaker, $\neg s$ gets stronger. A query that is SAT may become UNSAT if more literals are removed—which makes the task of finding a minimal cube much harder (exponential in the size of s). Heuristics for minimizing s are discussed in [2].

But why is it sound to add $\neg s$ to the query? It can be viewed as a bounded inductive reasoning: The base case $\mathbf{R}_0 \rightarrow \neg s$ holds by construction (s does not intersect the initial states). We have proved that $(\mathbf{R}_{k-1} \wedge \neg s \wedge \mathbf{T}) \rightarrow \neg s'$, but because $\mathbf{R}_i$ is stronger than $\mathbf{R}_{k-1}$ for $i < k-1$, we have also proved that $\neg s$ is preserved by every transition up to frame k.

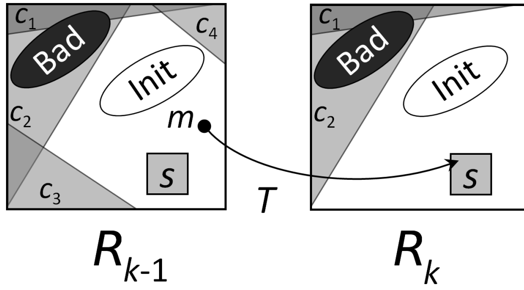


Figure 1. *Is s inductive relative to* $\mathbf{R}_{k-1}$*?* In the SAT query, we try to find a minterm m in the white region of the first frame, that in one transition can reach a point inside the cube s. The white region satisfies $\mathbf{R}_{k-1} \wedge \neg s$, illustrated by the four blocked cubes c_1 through c_4 and the cube s. If the query is UNSAT, it has been proved that a point outside s stays outside s for the first k transitions from the initial states. When generalizing s, we must make sure that the cube does not grow to intersect the initial states. This property, together with UNSAT, proves s to be unreachable in the first k frames. Note that the figure also illustrates how $\mathbf{R}_k$ contains a subset of the cubes of $\mathbf{R}_{k-1}$.

(2) **Satisfiable queries***:* We now turn to the case where the query (original or extended) is SAT. This means $\mathbf{R}_{k-1}$ was not strong enough to block s at frame k, and something new must be learned at frame $k-1$. From the satisfying assignment, we can extract a minterm m in the pre-image of s, which gives us a new proof-obligation (m, $k-1$).

The above learning scheme can now be applied to this proof-obligation, drawing from $\mathbf{R}_{k-2}$ to learn clauses in $\mathbf{R}_{k-1}$. If $\mathbf{R}_{k-2}$ is not strong enough, the procedure may recursively go further back into the trace and learn a whole cascade of facts over many time-frames. Eventually the procedure returns to the original proof-obligation (s, k) and may either succeed in blocking it this time, or generate a new minterm in the pre-image of s.

As noted in the previous section, learning short clauses is crucial for PDR to work. Indeed, most of the runtime is spent on generalizing cubes by removing literals. Because a minterm is maximally long, it is a particularly undesirable starting point for this process. To alleviate this situation, we propose to shrink the proof-obligations by using three-valued (ternary) simulation.[1] It requires the FSM to be in circuit form, but in practice this is often the case.

Reducing proof-obligations by ternary simulation. For a satisfiable query, extract the minterm m from the satisfying assignment, giving values to the flop outputs as well as the primary inputs. Simulate this assignment through one time-frame. Now, probe each flop by changing its value to X and propagate the effect of this using ternary simulation. If an X does not appear at any flop input among the flops in s, then the probed flop (state variable) can be safely removed from the proof-obligation. If the X do reach a flop in s, undo the propagation and the probing, and move on to the next flop.

The resulting cube has the property that all the states it represents can reach s in one transition, and hence the entire cube must be blocked.

[1]Ternary logic has three values: 0, 1, and X. The binary semantics is extended by: $(X \wedge 0 = 0)$, $(X \wedge 1 = X)$, $(X \wedge X = X)$, $(\neg X = X)$.

C. The Algorithm

For clarity, we state the precise properties of the trace:

(1) $\mathbf{R}_0$ = $\mathbf{I}$.
(2) All $\mathbf{R}_i$ except $\mathbf{R}_0$ are sets of clauses.
(3a) $\mathbf{R}_i \rightarrow \mathbf{R}_{i+1}$.
(3b) The clauses $\mathbf{R}_{i+1}$ is a subset of $\mathbf{R}_i$ for $i > 0$.
(4) $\mathbf{R}_{i+1}$ is an over-approximation of the image of $\mathbf{R}_i$.
(5) $\mathbf{R}_i \rightarrow \mathbf{P}$, except for the last element $\mathbf{R}_N$ of the trace.

We note that (5) is different from Bradley's original presentation, which also required the property to hold for $\mathbf{R}_N$. The change eliminates the need for the special BMC check of length 0 and 1, performed in Bradley's implementation of PDR.

At the start of the algorithm the trace has just one element $\mathbf{R}_0$. It then runs the following main loop:

while $SAT?\big[\mathbf{R}_N \wedge \neg\mathbf{P}\big]$ **do**
(a) extract a bad state m from the SAT model
(b) generalize m to a cube s using ternary simulation
(c) recursively block the proof-obligation (s, k)

When the loop terminates, the property holds for $\mathbf{R}_N$, and an empty frame is added to the trace. The algorithm will be repeated for this new frame, but first a *propagation phase* is executed, where learned clauses are pushed forward in the trace:

for $k \in [1, N-1]$ and $c \in \mathbf{R}_k$ **do**
if c holds in frame $k+1$, add it to $\mathbf{R}_{k+1}$

During the propagation phase it is important to do syntactic subsumption. If a clause c was moved forward from frame k to $k+1$, and frame $k+1$ has a weaker clause $d \supseteq c$, then d should be removed. Subsumed clauses accumulate quickly, but serves no purpose except to slow down the SAT-solver.

(1) **Queue of proof-obligations***:* Section III-B1 suggests a recursive clause-learning scheme. However, PDR can be improved by reusing proof-obligations of one time-frame in all future time-frames. After all, if a cube is bad, it should be blocked everywhere. This requires a queue, as the algorithm now can have many outstanding proof-obligations in each frame. The elements should be dequeued from the smallest time-frame first. This change has the added benefit of making PDR capable of finding counterexamples longer than the trace.

(2) **Terminatation***:* PDR can terminate in one of two ways: either (i) a proof-obligation at frame 0 intersects with the initial states, which implies that the property fails (in this case, a counterexample can be extracted with some additional bookkeeping); or (ii) the clause sets of two adjacent frames become syntactically identical: $\mathbf{R}_i \equiv \mathbf{R}_{i+1}$. Since $\mathbf{R}_i \rightarrow \mathbf{P}$ by (5); $\mathbf{R}_i \wedge \mathbf{T} \rightarrow \mathbf{R}'_{i+1}$ by (4); $\mathbf{I} \rightarrow \mathbf{R}_i$ by (1) and (3a); then $\mathbf{R}_i$ is an inductive invariant that proves the property.

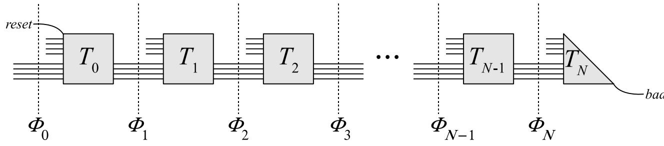


Figure 2. *BMC unrolling of length N.* The design is reset in the first frame and *Bad* is asserted in the last frame. The last frame is drawn partially because the next-state logic for the flops is not needed.

D. Convergence

Must the main loop terminate for some finite trace length? When generated, each proof-obligation $(s,\, k)$ contains at least one state that is not previously blocked. If the proof-obligation is immediately handled, or if the generalization procedure first checks that this still holds when the proof-obligation is dequeued, then every clause created by the learning algorithm must block at least one more state of frame k. Because there is a finite number of frames, and a finite number of states in the FSM, the main loop is guaranteed to terminate.

Can the length of the trace grow indefinitely? If the syntactic termination check ($\mathbf{R}_i \equiv \mathbf{R}_{i+1}$) were done semantically instead ($\mathbf{R}_i = \mathbf{R}_{i+1}$), then clearly this cannot happen. $\mathbf{R}_{i+1}$ would have to block at least one state less than $\mathbf{R}_i$. Suppose therefore $\mathbf{R}_i = \mathbf{R}_{i+1}$ but $\mathbf{R}_i \not\equiv \mathbf{R}_{i+1}$. During the propagation phase, all clauses of $\mathbf{R}_i$ will be moved into $\mathbf{R}_{i+1}$, making them syntactically identical and the algorithm terminates.

We note that the bound ($2^{|S|}$ frames with at most $2^{|S|}$ clauses in each) implied by the above argument is very large, and does in no way explain why the algorithm performs well in practice.

E. What makes PDR so effective?

The experimental analysis of Section VI shows that PDR represents a major performance improvement over interpolation based model checking (IMC) [7], hitherto regarded as the strongest bit-level engine. Why is this?

Consider the BMC unrolling depicted in *Figure 2*. Assume for simplicity that the design can non-deterministically return to the initial states at any time.[2] This guarantees that the set of reachable states grows monotonically with the frame number.

The first version of IMC, never published,[3] considered such an unrolling, and from an UNSAT proof computed interpolants Φ_i between every adjacent time-frames. This sequence of interpolants has the property:

(1) $\mathbf{I} = \Phi_0$
(2) $\Phi_k \wedge \mathbf{T} \rightarrow \Phi'_{k+1}$
(3) $\Phi_N \rightarrow \mathbf{P}$
(4) symbols(Φ_i) $\subseteq$ *state-variables*

If N is chosen large enough, one of the interpolants Φ_k must be an inductive invariant proving the property: if the suffix after Φ_k is longer than the backward diameter of the system, it cannot contain any state that can reach *Bad*; the prefix before Φ_k grows monotonically and for a finite system must eventually repeat itself.

This method is not as effective as the published version of IMC. So what is wrong with it? One can argue that the important feature of interpolation is its *generalizing* capability.[4] For instance, the interpolant Φ_1 can be viewed as an *abstraction* of the first time-frame, containing just the facts needed for the suffix to be unsatisfiable (this interpretation is particularly in accord with McMillan's asymmetric interpolant computation).

Even though logically (2) implies that each interpolant can be derived from its predecessor, this is not how the SAT-solver constructs them. During its search, the solver is free to roam all over the unrolling. We argue that this may deteriorate the generalizing capability of interpolation.

In the published algorithm, McMillan used the insight that (a) interpolants are smaller and more general toward the ends of the unrolling, and (b) repeatedly applying interpolation on its own output will improve the generalizing capability. In his algorithm, the interpolant Φ_1 is therefore repeatedly used to replace the initial states constraint, resulting in interpolants that are less and less dependent on the initial states and in an increasingly more general way imply the unsatisfiability of the suffix.

In a way, the procedure can be viewed as *committing* to the abstraction that was computed. It disallows the SAT-solver from going back to the real initial states and learning more facts. For this to work, the suffix must be long enough to prevent any state that can reach the bad states from entering into the interpolants. If it fails to prevent this, the algorithm has to start over from scratch, typically with a longer unrolling (although randomizing the SAT-solver and restarting with the same length works sometimes).

We now compare this to how PDR works. First note that at the end of each major cycle, just before pushing clauses forward, the $\mathbf{R}_i$ are in fact interpolants; all the facts in frame k and future frames are derived from $\mathbf{R}_k$.

During the computations, PDR completely commits to its current abstractions $\mathbf{R}_i$. The localized reasoning prevents it from learning new facts from earlier time-frames *unless* it has been proved that new facts *must* be learned. In a way, the whole procedure can be viewed as one big SAT-solving process, where the solver is carefully controlled to make sure it does *not* roam all over the unrolling. Further, when new facts are brought in from previous frames, a lot of effort is spent on simplifying those facts (the literal removing consumes $\sim$80% of the runtime). There is no similar mechanism in IMC, it must use whatever proof the SAT-solver happened to give it. Also, PDR constantly removes subsumed clauses, especially during the forward-propagation phase.

To summarize, PDR sticks to the facts it has learned as long as possible, similar to the way IMC commits to its interpolants. If what PDR has learned at a frame is too weak, it can repair the situation by learning new clauses rather than scrapping all

[2]This behavior can be achieved by rewiring the flops, or, alternatively, be made part of the verification algorithm.

[3]Private conversation with Ken McMillan.

[4]Indeed, interpolation based model checking is probably better understood as a method for "guessing" an inductive invariant rather than, as often done, an approximate reachability analysis.

the work done so far and starting over, as IMC does. PDR has a very targeted approach to producing small facts by its literal removing scheme, and it constantly weeds out redundant clauses by subsumption checking and forward propagation.

A possible drawback of PDR, however, is the strong *inductive bias* of its learning: it can only learn clauses in terms of state variables. But this bias is also the very reason it can efficiently do generalization. It might be that future improvements to the algorithm will allow it to work efficiently on a different domain.

IV. IMPLEMENTATION

This section details our implementation of PDR. In the pseudo-code, only cubes are used and not clauses. In particular we represent the trace as sets of blocked cubes rather than learned clauses. Furthermore, we only store a cube in the *last* time-frame where it holds (to avoid duplication). We call this delta-encoded trace $\mathbf{F}$, and it relates to $\mathbf{R}$ through:

$$\mathbf{R}_k = \bigwedge_{i \geq k} \neg \mathbf{F}_i$$

We also extend $\mathbf{F}$ by a special element $\mathbf{F}_\infty$ which will hold cubes that have been proved unreachable from the initial states by any number of transitions. In the code, the following data-types are used:

- **Vec**. A dynamic vector with methods:

uint ***size***()	– returns size of the vector
T& ***op***[](uint i)	– returns the ith element
void ***push***(T elem)	– pushes an element at the end
T ***pop***()	– pops and returns last element

- **Cube**. A fixed-size vector of literals (no push/pop).
- **TCube**. A pair (cube $\in$ Cube, frame $\in$ uint) referred to as a *timed cube*. Two special constants are defined for the frame component:

FRAME_NULL	– cube has no time component
FRAME_INF	– cube belongs in $\mathbf{F}_\infty$

  Function ***next***(TCube *s*) returns *s* with the frame number incremented by one.

An overview of the functions implementing PDR, and the program state they work on is given in *Figure 3*. The FSM is assumed to be given in circuit form, containing one safety property to be proved. The special frame $\mathbf{F}_\infty$ is stored as the last element of the vector *F*.

An outline of the execution: Function ***pdrMain***() gets a bad state in the last frame and calls ***recBlockCube***() to block it, using the helper function ***isBlocked***() (which checks if a proof-obligation has already been solved) and ***generalize***() (which shortens a cube). When the property has been proved for the last frame, ***propagateBlockedCubes***() pushes cubes of all time-frames forward while doing subsumption, handled by ***addBlockedCube***().

A. Separation of concerns

Our PDR implementation abstracts the handling of SAT calls through the interface in *Figure 4*. The semantics of the interface is defined as follows:

```
Program State:
    Netlist            N;        – Netlist with property
    Vec⟨Vec⟨Cube⟩⟩     F;        – Blocked cubes of each frame
    PdrSat             Z;        – Supporting SAT solver(s)

Main Function:
    bool  pdrMain();

Recursive Cube Generation:
    bool  recBlockCube(TCube s0);
    bool  isBlocked(TCube s);
    TCube generalize(TCube s0);

Cube Forward Propagation:
    bool  propagateBlockedCubes();

Small Helpers:
    uint  depth();
    void  newFrame();
    bool  condAssign(TCube& s, TCube t);
    void  addBlockedCube(TCube s);
```

Figure 3. *Overview of PDR algorithm.* "***pdrMain***()" will use "***recBlockCube***()" to recursively block bad states of the final time frame until the property holds, then call "***propagateBlockedCubes***()" to push blocked cubes from all frames in the trace forward to the latest frame where they hold.

```
interface PdrSat {
        Cube   getBadCube();
        bool   isBlocked(TCube s);
        bool   isInitial(Cube c);
        TCube  solveRelative(TCube s, uint params = 0);

        void   blockCubeInSolver(TCube s);
};
```

Figure 4. *Abstract interface for the SAT queries of PDR.* These methods can be implemented using either a monolithic SAT-solver, or one SAT-solver per time-frame. The roles of "Init" and "Bad" can be exchanged within this SAT abstraction to obtain the dual PDR procedure based on backward induction (although ternary simulation cannot be used backwards). The first four functions corresponds to actual SAT queries (although for some common restriction on initial states, "***isInitial***()" can be implemented by a syntactic analysis). The fifth function, "***blockCubeInSolver***()", merely informs the SAT implementation that a new cube has been added to the vector "*F*".

Method ***getBadCube***() returns a bad cube not yet blocked in the last frame. Method ***isBlocked***(*s*) returns TRUE if the cube *s.cube* is blocked at *s.frame*. Method ***isInitial***(*c*) returns TRUE if the cube *c* intersects with the initial states. Method ***blockCubeInSolver***(*s*) reports to *PdrSat* that a cube has been added to the vector *F*.

Finally, method ***solveRelative***(*s*) tests if *s.cube* can be blocked at frame *s.frame* using the extended query (Q2) of Section III-B1. If the answer is UNSAT, then the implementation returns a new cube *z* where:

$$\begin{aligned} z.cube &\subseteq s.cube \\ z.frame &\geq s.frame \end{aligned}$$

The method guarantees that not only is *s.cube* blocked at frame *s.frame*, but that actually the subset *z.cube* is blocked at a later frame. The SAT solver may learn these more general facts by

```
bool pdrMain() {
        F.push();                                        – push "F∞"
        newFrame();                                      – create "F[0]"

        Z = createPdrSat(N, F);

        forever{
                Cube c = Z.getBadCube();
                if (c != CUBE_NULL){
                        if (!recBlockCube(TCube(c, depth())))
                                – failed to block 'c' ⇒ CEX found
                                return FALSE;
                }else{
                        newFrame();
                        if (propagateBlockedCubes())
                                – invariant found, may store it here
                                return TRUE;
                }
        }
}
```

Figure 5. *Main procedure.* The last element of $\mathbf{F}$ (referred to as "$\mathbf{F}_\infty$") contains all the cubes that have been proved to be unreachable for all k. Their negation constitutes a proper inductive invariant. Function "***newFrame***()" inserts a new frame into $\mathbf{F}$ just before $\mathbf{F}_\infty$.

inspecting the final conflict-clause of the solver (or the UNSAT core), and taking this "free" information into account.

If instead the query is satisfiable, then the implementation returns a generalization, using ternary simulation, of a minterm in the pre-image of *s.cube*. All states of the returned cube *z.cube* can reach *s.cube* in one transition. The time component *z.frame* is set to FRAME_NULL.

The behavior of ***solveRelative***() can be altered by the *params* argument. Default value **"0"** means: do not extract a model if the query satisfiable, just return (CUBE_NULL, FRAME_NULL). Parameter **"EXTRACTMODEL"** means: work as described above. Parameter **"NOIND"** means: use the original query (Q1) instead of (Q2).

V. SAT SOLVING

In this section, we discuss the details of implementing ***solveRelative***() of the *PdrSat* interface using MINISAT and a single SAT instance. The other methods of the *PdrSat* interface can be implemented in a similar way.

There are two features that are particularly important: (i) MINISAT allows incremental SAT through *assumption literals*; a set of unit clauses that are temporarily assumed during one SAT call. After the call, the assumptions are undone and new regular clauses can be added before the next call. (ii) For UNSAT calls, MINISAT returns the subset of assumptions that were used in the proof.

The netlist is transformed to CNF using the standard Tseitin transformation [9] plus variable elimination [6]. Logic cones are added to the solver on demand, starting with just the transitive fanin of *Bad*. Whenever a new frame is added to the trace, a new activation literal act_i is reserved. All clauses learned in that frame will be extended by $\neg act_i$ in ***blockCubeInSolver***().

Given a cube $s = (s_1 \wedge s_2 \wedge \ldots \wedge s_n)$, procedure ***solveRelative***() does the following:

```
bool recBlockCube(TCube s0) {
  PrioQ⟨TCube⟩ Q;       – orders cubes from low to high frames
  Q.add(s0);

  while (Q.size() > 0){
    TCube s = Q.popMin();

    if (s.frame == 0)
      – Found counterexample, may extract it here
      return FALSE;

    if (!isBlocked(s)){
      assert(!Z.isInitial(s.cube));
      TCube z = Z.solveRelative(s, EXTRACTMODEL);

      if (z.frame != FRAME_NULL){
        – Cube 's' was blocked by image of predecessor:
        z = generalize(z);
        while (z.frame < depth()−1
          && condAssign(z, Z.solveRelative(next(z))));

        addBlockedCube(z);
        if (s.frame < depth() && z.frame != FRAME_INF)
          Q.add(next(s));

      }else{
        – Cube 's' was not blocked by image of predecessor:
        z.frame = s.frame − 1;
        Q.add(z);
        Q.add(s);
      }
    }
  }
  return TRUE;
}
```

Figure 6. *Recursively block a cube.* The priority queue "*Q*" stores all pending proof-obligations: a cube and a time frame where it should be blocked. In a practical implementation, it may also store the proof-obligation from which the element was generated (this facilitates extraction of counterexamples). We noticed (or think we noticed) a small performance gain by giving "PrioQ" a stack-like behavior for proof-obligations of the same frame. We left one of our program assertions in the pseudo code because this invariant is important and non-obvious. Finally, note the line "***Q.add***(***next***(*s*))" line (just above the "**else**"). Adding the current proof-obligation in the next frame is not necessary, but it improves performance for UNSAT problems *and* allows PDR to find counterexamples longer than the length of the trace—sometimes much longer.

(1) Reserve a new activation literal a and add the clause $\{\neg a, \neg s_1, \neg s_2, \ldots, \neg s_n\}$ (unless NOIND is given).

(2) Call the solve method with the following assumptions: [a, act_k, act_{k+1}, ..., act_{N+1}, s'_1, s'_2, ..., s'_n], where s'_i denotes a flop input.

(2u) If UNSAT:

- Remove all literals of s whose corresponding assumption s'_i was not used, unless doing so makes the new cube overlap with the initial states.
- Find the lowest act_i that was used. Return the timed cube $(s_{new}, i + 1)$.

(2s) If SAT and EXTRACTMODEL is specified:

- Extract a minterm m from the satisfying assignment.
- Shorten m to cube by ternary simulation. Return $(m_{new}, \text{FRAME_NULL})$.

```
bool isBlocked(TCube s) {
        – Check syntactic subsumption (faster than SAT):
        for (uint d = s.frame; d < F.size(); d++)
                for (uint i = 0; i < F[d].size(); i++)
                        if (subsumes(F[d][i], s.cube))
                                return TRUE;

        – Semantic subsumption thru SAT:
        return Z.isBlocked(s);
}

TCube generalize(TCube s) {
        for all literals p ∈ s {
                TCube t = "s minus literal p"
                if (!Z.isInitial(t.cube))
                        condAssign(s, Z.solveRelative(t));
        }
        return s;
}
```

Figure 7. *Helper functions for recursive cube blocking.* Function "***isBlocked***()" semantically checks if s is already blocked, which could have happened after the proof-obligation was enqueued. For efficiency reasons, it first does a syntactic check. This check is so effective that we did not notice any performance loss by disabling the semantic SAT check at the end (but we kept it to ensure convergence, as argued in Section III-D). In fact, deriving a new cube from s, even if s is blocked, may be a good idea, as the new cube can subsume several old cubes. We note that function "***generalize***()" iterates over s while s is being modified, which the implementation must handle.

(2s') else if SAT, return (CUBE_NULL, FRAME_NULL).
(3) Add unit clause $\{\neg a\}$ permanently.

The last step (3) forever disables the temporary clause added in (1). The periodic cleanup of MINISAT will reclaim the memory. However, the variable index reserved for the activation literal cannot be reused. For that reason we recycle the solver when more then 50% of the variables currently in use are disabled activation literals. This has the added benefit of cleaning up cones of logic that may no longer be in use. We note that the previous activation literal can be reused if s is a subset of the cube of the previous call, which happens quite frequently.

```
uint depth() { return F.size() − 2; }

void newFrame() {
        – Add frame to 'F' while moving 'F∞' forward:
        uint n = F.size();
        F.push();
        F[n−1].moveTo(F[n]);
}

bool condAssign(TCube& s, TCube t) {
        if (t.frame != FRAME_NULL){
                s = t;
                return TRUE;
        }else
                return FALSE;
}

void addBlockedCube(TCube s) {
        uint k = min(s.frame, depth() + 1);

        – Remove subsumed clauses:
        for (uint d = 1; d ≤ k; d++){
                for (uint i = 0; i < F[d].size();){
                        if (subsumes(s.cube, F[d][i])){
                                F[d][i] = F[d].last();
                                F[d].pop();
                        }else
                                i++;
                }
        }

        – Store clause:
        F[k].push(s.cube);
        Z.blockCubeInSolver(s);
}
```

Figure 8. *Small helper functions.* Function "***addBlockedCube***()" will add a cube both to $\mathbf{F}_a$ nd the PdrSat object. It will also remove any subsumed cube in $\mathbf{F}$. Subsumed cubes in the SAT-solver will be removed through periodical recycling.

VI. EXPERIMENTAL ANALYSIS

A number of experiments have been performed to evaluate our PDR implementation, both on public benchmarks from the Hardware Model Checking Competition of 2010 (HWMCC10) and on industrial benchmarks.[5] This section summarizes the most interesting results we have found.

A. Comparison of IC3 and PDR

This experiment was performed using 274 hard problems from our industrial collaborators. We simplified the designs by running the ABC command "dprove"(see Figure 4.1 of [8]). With a timeout of 10 minutes, 42 problems were solved by either IC3 or PDR; included in *Table I*. From the table we see that our implementation solves almost twice as many instances as the original IC3 (38 vs. 21), but there are also 4 instances where IC3 solves them and our PDR does not. The last column shows for comparison the results of interpolation based model checking (IMC).

Figure 10 shows the behavior of the implementations for increasing timeout limits. For space reasons we included two more PDR runs discussed in the next section.

Figure 11 shows the performance of PDR, IC3 and IMC on the HWMCC10 benchmarks. Looking closer at PDR vs. IMC reveals that the difference is mostly on UNSAT problems, where PDR solves 420 vs. 362 for IMC (14% difference). On satisfiable instance, numbers are 303 vs. 294 (3% difference).

B. Ternary simulation and Generalization

The third column of *Table I*, and the corresponding curve in *Figure 10*, show the performance of PDR without ternary simulation. It is clear that ternary simulation has a big impact. Without it, our implementation drops way below IC3. One reason for this may be that IC3 never had ternary simulation, and Bradley implemented some other tricks that compensates

[5]Although we cannot distribute the industrial benchmarks, we will make our implementation of PDR available at http://bvsrc.org

```
bool propagateBlockedCubes() {
  for (uint k = 1; k < depth(); k++){
    for all cubes c ∈ F[k] {
      TCube s = Z.solveRelative(TCube(c, k+1), NOIND);
      if (s.frame != FRAME_NULL)
        addBlockedCube(s);
    }

    if (F[k].size() == 0) return TRUE;          – Invariant found
  }

  return FALSE;
}
```

Figure 9. *Propagating blocked cubes forward.* All cubes in $\mathbf{F}$ are revisited to see if they now hold at a later time-frame. If so, they are inserted into that frame. The subsumption of "***addBlockedCube***()" will remove the cube from its current frame (and possible other cubes in the later frame). Note that in a practical implementation, the iteration over cubes in $\mathbf{F}_k$ must be aware of these updates. Because c is already present in frame k, we can use (Q1) instead of (Q2) in the call to ***solveRelative***().

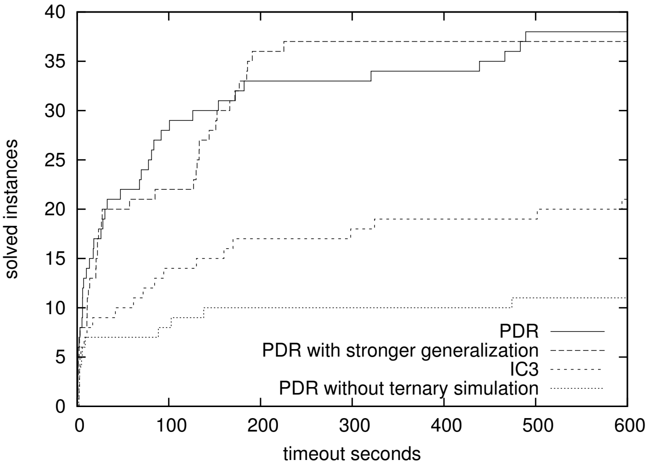


Figure 10. *Comparison of IC3 and PDR on industrial problems.* Two modifications to PDR are also evaluated.

Benchmark	IC3	PDR	NoSim	StGen	IMC
design01_prop1	–	–	–	–	249.5
design01_prop2	4.1	**0.3**	102.5	0.4	0.2
design01_prop3	–	**81.2**	–	126.9	–
design01_prop4	–	**70.0**	–	191.0	–
design01_prop5	–	**91.6**	–	166.5	–
design01_prop6	–	**100.7**	–	176.7	–
design01_prop7	–	–	–	–	168.8
design01_prop8	160.1	**6.1**	–	11.1	21.9
design01_prop9	130.1	**5.9**	–	10.7	42.8
design01_prop10	71.9	**7.1**	–	12.3	44.2
design02_prop1	594.0	**30.2**	–	144.0	–
design02_prop2	–	**489.2**	–	–	–
design02_prop3	–	**68.0**	–	–	–
design03_prop1	–	**466.4**	–	129.8	–
design03_prop2	–	**483.3**	–	130.8	–
design04	**84.5**	–	–	–	–
design05_prop1	–	**172.5**	–	152.5	–
design05_prop2	–	**182.1**	–	172.0	–
design06_prop1	2.7	**0.8**	1.8	1.0	–
design06_prop2	**3.1**	**3.1**	5.6	0.8	–
design07	94.4	**6.0**	88.6	13.8	–
design08	298.3	**83.6**	–	133.1	–
design09	–	**77.8**	–	151.2	–
design10_prop1	2.0	**1.0**	2.3	1.4	–
design10_prop2	2.6	**1.0**	2.7	2.7	–
design11_prop1	324.4	**28.1**	474.0	27.4	–
design11_prop2	7.7	**2.1**	8.9	3.3	–
design12	–	–	–	–	62.6
design13_prop1	–	**126.1**	–	85.0	–
design13_prop2	–	**47.2**	–	57.2	–
design13_prop3	–	**26.0**	–	22.2	–
design13_prop4	–	**17.6**	–	22.1	–
design13_prop5	–	**18.1**	–	26.4	–
design14_prop1	**41.7**	–	–	–	–
design14_prop2	**61.5**	–	–	–	–
design15_prop1	–	**5.3**	–	20.7	4.7
design15_prop2	–	**32.8**	–	10.9	595.8
design16	2.2	**0.9**	2.6	2.2	286.6
design17	–	–	–	185.8	
design18	10.8	**0.7**	4.9	1.4	409.7
design19	501.7	**13.4**	–	23.3	–
design20	17.1	**10.0**	138.0	20.4	–
design21	**169.9**	–	–	225.4	–
design22	–	**154.1**	–	185.0	–
design23_prop1	–	**438.7**	–	–	–
design23_prop2	–	**320.5**	–	133.0	–
Total solved	21	38	11	37	11

Table I. *Comparison of IC3 and PDR on industrial problems.* Two modifications to PDR are also evaluated (disabling ternary simulation "NoSim", and stronger cube generalization "StGen"). Interpolation (IMC) is also included for comparison. All benchmarks are UNSAT except for *design02* (3 properties) and *design14* (2 properties). Boldfaced figures indicates winner between IC3 and PDR only.

for this loss, notably removing multiple literals per SAT call in the cube generalization.

The fourth column shows the effect of stronger cube generalization, as proposed in the paper on IC3. The modified procedure will try to remove a literal even if the SAT query is satisfiable by exploiting the non-monotonicity. As in IC3, this is done for three random literals. Our conclusion from looking at the results is that this technique was *not* helpful. Although we do not have room to present the data here, the result holds for the HWMCC10 benchmarks as well.

C. Effect of changing the semantics of $\mathbf{R}_N$

As pointed out in Section III-C, we diverge from IC3 by not requiring the last frame of the trace to fulfill the property. The approach of IC3 has two effects compared to ours:

(1) When a new frame is opened, the property is known to hold, so $\mathbf{P}$ can be added to the relative induction SAT query. This means that the final invariant will be of the form $\mathbf{R}_i \wedge \mathbf{P}$ rather than just $\mathbf{R}_i$, and that the clauses in $\mathbf{R}_\infty$ may depend on $\mathbf{P}$.

(2) Seeding the recursive cube-blocking with minterms of the pre-image of $\mathbf{P}$ rather than with minterms of $\mathbf{P}$ corresponds to a one-step target-enlargement.

The second difference, target-enlargement, can be implemented by preprocessing the design (unroll the property cone for one frame and combine the new and the old property outputs). *Figure 12* shows the effect it has on the HWMCC10 benchmarks. Note that it improves the performance for simple satisfiable problems solved in less than 100 seconds. The difference is substantial enough to motivate the use of target-enlargement.

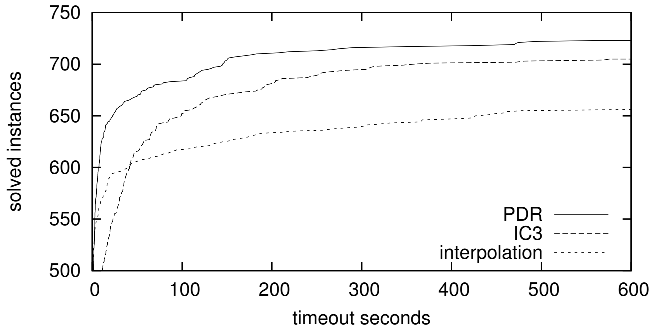


Figure 11. *Comparison of IC3 and PDR on HWMCC10 problems.*

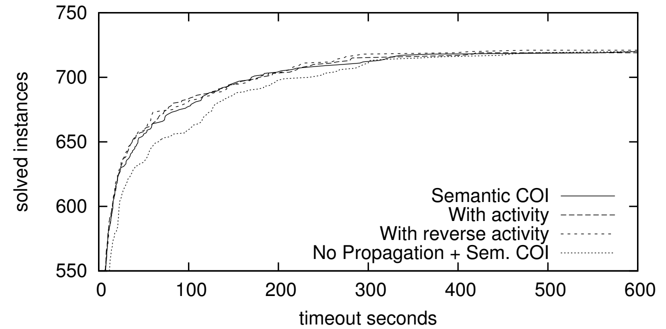


Figure 13. *Refuting activity / Semantic cone-of-influence.*

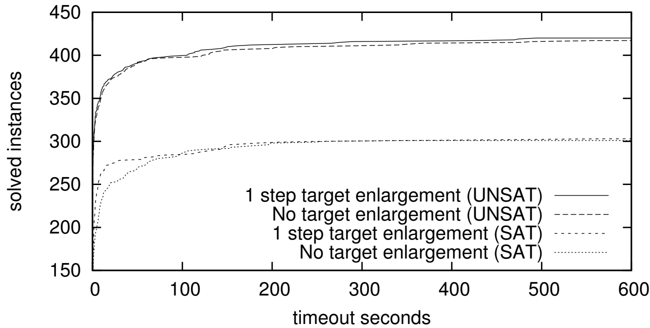


Figure 12. *Effect of target enlargement.*

We also investigated if the first difference above had any effect by running our previous PDR implementation which had the same behavior as IC3 in this respect (but includes ternary simulation and other improvements). For space reason we do not include the graph here, but the curves of the new implementation (with target enlargement) and the previous implementation match exactly on both SAT and UNSAT problems. We also tried target enlargement of 2 steps, but there was no additional benefit.

We conclude that there is no performance loss due to our modification of the original algorithm. It makes the implementation simpler, and it has the extra benefit that $\mathbf{R}_\infty$ is a proper invariant, which can be used to strengthen other proof-engines running in parallel, or be useful for synthesis.

D. Runtime breakdown

In order to identify directions for future improvements, we ran an instrumented version of our PDR on a handful of examples. Our findings suggests that about 20% of the runtime is spent in ***propagateBlockedCubes***() and 80% in ***recBlockCube***()—most of which is in ***generalize***(), but a substantial portion also in the first call to ***solveRelative***(). Satisfiable calls to the SAT-solver are about twice as common as unsatisfiable ones, and 5x more expensive.

E. Other things we tried

— We evaluated the effect of the extended query (Q2) vs. the original (Q1) (Section III-B1). Although the (Q2) gave a clear performance boost, PDR works remarkably well even without it (it solved 704 instead of 723 problems; more than interpolation, which solved 656 problems).

— We evaluated the proposed activity scheme of IC3, which controls the order in which literals are tried for removal. We ran it against itself with the activity reversed ("worst" order) and could see no difference (*Figure 13*), and no difference to a static order either (not in the graph).

— We implemented a technique we call *semantic cone-of-influence*. At the end of each major round, all cubes in the trace that is not needed to prove the property of the final frame are removed. This analysis can be done through a series of SAT calls of roughly the same cost as forward-propagation. The method removes many cubes. However, running PDR with this turned on did not give any noticeable speedup, but it also did not degrade performance (thus the cost of doing semantic COI was amortized by the improvement). But a really interesting result is that running semantic COI, while turning off forward-propagation, works almost as well as the standard version of PDR (*Figure 13*). In contrast, turning off forward-propagation *without* semantic COI is a disaster! This shows that an important feature of the forward-propagation is the cleansing effect it has through the subsumption mechanism.

— Because most of the time is spent in satisfiable SAT calls, and this partly is a result of MiniSat always returning complete models, we made a modified version of MiniSat that only does BCP in the cone-of-influence of the flops in the query. With this version, a few more benchmarks (728 instead of 723) were solved. However, we think a justification based variable order should do even better. We are currently working on a circuit based SAT-solver with this feature.

We have also implemented a non-monolithic version of PDR (one solver instance per time-frame) that helps to localizing the SAT solving better, especially together with frequent solver recycling. For large benchmarks, where the relevant logic is small compared to the size of the design, this version does very well. It is worth noting that most of the work in PDR takes place in the last couple of time-frames where the COI is the smallest. In a monolithic PDR, early time-frames may pollute these calls.

— We made a version that finds an inductive subset of $\mathbf{R}_N$ after propagating the cubes forward. This will find true

inductive invariants that can be put into $\mathbf{F}_\infty$. Although the cost of this procedure did not quite amortize over the gains, having more clauses in $\mathbf{F}_\infty$ can be useful if those facts are exported to other engines.

— We made an extension that allows PDR to develop and use an abstraction, where some flops are considered as primary inputs. This is relatively straight-forward to implement. The only tricky part in using localization abstraction is when it is combined with proof-based abstraction [5], which can shrink the current abstraction in the middle of PDR's operations. The reason is that the assertion in *Figure 6* will not hold if we apply a smaller abstraction to the initial states. The way to address this is to introduce a reset signal that gives the correct value at the flop outputs of frame 0, and then let all flops be uninitialized.

References

[1] A. R. Bradley. **SAT-based model checking without unrolling**. In *Proc. VMCAI*, 2011.

[2] A. R. Bradley and Z. Manna. **Checking safety by inductive generalization of counterexamples to induction**. In *Proc. FMCAD*, 2007.

[3] G. Cabodi, L. A. Garcia, M. Murciano, S. Nocco, and S. Quer. **Partitioning interpolant-based verification for effective unbounded model checking**. In *IEEE TCAD*, 2010.

[4] G. Cabodi, M. Murciano, S. Nocco, and S. Quer. **Boosting interpolation with dynamic localized abstraction and redundancy removal**. In *ACM TODAES*, 2008.

[5] N. Een, A. Mishchenko, and N. Amla. **A Single-Instance Incremental SAT Formulation of Proof- and Counterexample-Based Abstraction**. In *FMCAD*, 2010.

[6] Niklas Een and Armin Biere. **Effective Preprocessing in SAT through Variable and Clause Elimination**. In *SAT*, 2005.

[7] K. L. McMillan. **Interpolation and SAT-based Model Checking**. In *CAV*, 2003.

[8] A. Mishchenko, M. L. Case, R. K Brayton, and S. Jang. **Scalable and scalably-verifiable sequential synthesis**. In *Proc. ICCAD*, 2008.

[9] G. Tseitin. **On the complexity of derivation in propositional calculus**. *Studies in Constr. Math. and Math. Logic*, 1968.

Incremental Formal Verification of Hardware

Hana Chockler, Alexander Ivrii, Arie Matsliah, Shiri Moran, Ziv Nevo
IBM Research – Haifa
E-mail:{hanac,alexi,ariem,shirim,nevo}@il.ibm.com

***Abstract*—Formal verification is a reliable and fully automatic technique for proving correctness of hardware designs. Its main drawback is the high complexity of verification, and this problem is especially acute in regression verification, where a new version of the design, differing from the previous version very slightly, is verified with respect to the same or a very similar property. In this paper, we present an efficient algorithm for *incremental verification*, based on the ic3 algorithm, that uses stored information from the previous verification runs in order to improve the complexity of re-verifying similar designs on similar properties. Our algorithm applies both to the positive and to the negative results of verification (that is, both when there is a proof of correctness and when there is a counterexample). The algorithm is implemented and experimental results show improvement of up to two orders of magnitude in running time, compared to full verification.**

I. Introduction

Today's rapid development of complex hardware designs requires reliable verification methods. In *formal verification*, we verify the correctness of a design with respect to a desired behavior by checking whether a labeled state-transition graph that models the design satisfies a specification of this behavior, expressed in terms of a temporal logic formula or a finite automaton [CGP99]. The main advantages of formal verification tools are their reliability (if a design passes verification, then it is 100% correct with respect the specification), full automation of the verification process, and the ability of the tools to accompany a negative answer to the correctness query by a counterexample to the satisfaction of the specification in the design [CGMZ95].

The main drawback of the formal verification technology, and the one that prevents it from being even more widely used in the hardware industry, is that it requires significant computational effort, even for moderately sized designs. Moreover, when small changes are introduced into the design or the specification, for example due to a bug fix or an upgrade, the whole design needs to be re-verified, generally requiring the same amount of resources as for the initial verification. The problem is especially acute in *regression verification*, where a new version of a hardware design is re-verified with respect to the same (or very similar) specification. Since regression verification is a only a preliminary (albeit necessary) stage in functional verification of a new version, the time and effort allocated to it are usually much lower than for the initial verification; in reality, since the amount of effort is the same as for the initial verification, our experience is that regression verification is often not performed thoroughly enough, thus possibly leading to lower quality designs.[1] A better option would be to verify the changes *incrementally*, that is, to reuse the results from the previous execution and only verify the change.

Another area in which *incremental verification* techniques are in dire need is *coverage computation* in formal verification. Most of the existing work on coverage in formal verification is based on the notion of *mutation coverage*, where small mutations are introduced to the design and the mutant designs are checked with respect to the original specification [HKHZ99], [CKV06b], [CKV06a], [KLS08], [CKP10], with the goal of checking thoroughness of formal specifications. Efficient incremental verification techniques can reduce the cost of computing mutation coverage, where a large number of slightly modified designs need to be checked with respect to the same property.

Several papers view the problem of incremental verification as an instance of *dynamic graph algorithms*. In this setting, a design is represented as a graph and incremental verification checks the influence of small changes in the graph (edge insertion and removal) on the properties that were previously satisfied in this graph, thus reducing the problem of incremental verification to a dynamic graph problem. However, dynamic graph connectivity, one of the main problems in dynamic graph algorithms, and the one that is most relevant to verification, is an open problem, hence this reduction is of limited value in practice [SBS95], [CK03]. A somewhat related direction is using the reduction to dynamic graph

This work is partially supported by the European Community under the call FP7-ICT-2009-5 – project PINCETTE 257647.

[1] Our experience is based on participating in the formal verification of IBM hardware designs, as well as on discussions with formal verification engineers in other companies.

problems in order to prove complexity results for LTL model checking of evolving designs with non-changing properties [KW03].

The idea of saving the result of model checking in order to use it for subsequent model-checking queries is extensively used in counter-example-guided abstraction refinement (CEGAR) approach [CGJ+03], where the state-explosion problem is addressed by iterative verification and refinement of an abstract design. Abstract counterexamples are analyzed and, if spurious, are used in order to guide the refinement process of the abstraction for the next iteration of the verification process (see also [LBBO01]).

In this paper, we present an approach for re-using the result of model-checking a design (either a proof or a counterexample) for verification of the same or a slightly different property on the same or a slightly modified design. Our method applies both for the case where the result of the model-checking query is negative, in which case we re-use a counterexample, and where it is positive, and we re-use the correctness proof. In fact, the later scenario is very common in regression verification (since the previous version of the design is assumed to pass the verification successfully).

The basis of our work is the novel ic3 ("Incremental Construction of Inductive Clauses for Indubitable Correctness") model checking algorithm, recently proposed by Aaron Bradley [Bra11]. In addition to being one of the the fastest bit-level verification methods [BEM11], [BBC+11], the ic3 algorithm lends itself very naturally to incremental verification.

We describe an algorithm for saving the relevant parts of the proof obtained by ic3, and using them to reproduce the proof (or counterexample) on a new, possibly modified version of the model. This requires very little computational effort in case the same proof works for the new version as well; in case the same proof does not apply, our method attempts to "patch" it by extracting the maximum valid part of the previously saved information and using it as a basis for proving the new version. The latter case, where the original proof does not apply "as is" is the main strength of our method. Roughly speaking, the main idea of our algorithm is as follows. First we observe that producing (and saving) re-usable information does not incur any significant overhead on top of the standard execution of ic3; in addition, the saved part is usually very small. We also describe a query-efficient SAT-based algorithm that we call an *invariant finder* for extracting the maximum valid part of the previously saved information. Then we describe how the valid parts extracted by the invariant finder can be used as a starting point of subsequent ic3's execution, with minor modifications to the algorithm.

Similarly, our algorithm allows to save a small part of a counterexample produced by ic3 that is, nevertheless, sufficient in order to easily reproduce a concrete counterexample and then use this part in order to compute counterexamples for a modified design (or a modified property), by re-using the saved part "as is" or "patching" it to produce a valid counterexample for a modified version. To improve the performance of the algorithm for reusing counterexamples, we propose a simple technique that reduces the size of the partial assignments produced by ic3 by $\approx 30\%$.

The algorithm is implemented in the model-checking engine IVE (*incremental verification engine*), which is a part of the formal verification platform of IBM [Rul], [Six] and is checked on the Hardware Model Checking Competition (HWMCC'10) [HWM10] benchmarks as well as on a real IBM hardware design. Our results show a significant speed-up (up to three orders of magnitude) compared to the re-run of the same model-checking procedure. We note that the performance of IVE is on par with the state-of-the art model checking tools, which makes the speed-up achieved by our incremental verification algorithm even more significant.

The rest of this paper is organized as follows. The necessary definitions and an overview of the ic3 algorithm are provided in Section II. We describe the main contribution of this paper – the invariant finder and the incremental verification algorithms – in Section III, and present the experimental results of executing our implementation on known benchmarks and on an IBM design in Section IV. In Section V we summarize our contributions and discuss possible directions for future work. The complete table of running times and speed-ups achieved by our algorithm on all benchmarks from the HWMCC'10 competition appears in the full version of this paper [Rul].

II. PRELIMINARIES

In this section, we give the necessary definitions for our algorithm and provide overviews of a SAT solver with incremental capabilities and of the ic3 algorithm.

A. Definitions

Throughout this paper we consider verification of safety properties on finite state machines (FSMs). An FSM M is a tuple $\langle X, I, T\rangle$, where X is a set of Boolean state variables, such that each assignment $s \in \{0,1\}^X$ corresponds to a state of M, and the predicates $I \subseteq \{0,1\}^X$ and $T \subseteq \{0,1\}^X \times \{0,1\}^X$ define the initial states and the transition relation of M, respectively. A

predicate $P \subseteq \{0,1\}^X$ defines a property to be verified on M.

State variables and their negations are called *literals*, and disjunctions of literals are called *clauses*. A CNF formula is a conjunction of clauses. (We sometimes refer to CNF formulas as sets of clauses as well.)

We follow the standard notation of $X' = \{x' : x \in X\}$ representing the state variables in the next step, and we assume that the FSMs are given in a representation that allows encoding pairs of states $\langle s, s' \rangle$ into a CNF formula ψ on variables $X \cup X'$, so that $\langle s, s' \rangle \in T$ if and only if ψ is satisfiable (containment in I and P should be similarly expressible with a CNF formula on variables in X).

A sequence π of states $t_0, \ldots, t_n$ is a *path* in M if for each $0 \leq i < n$, $\langle t_i, t_{i+1} \rangle \in T$, that is, there is a transition between each subsequent pair of states in π. A path that starts from an initial state is called an *initialized path*. A state $t \in \{0,1\}^X$ is *reachable* if there is an initialized path that ends in t. Let R denote the set of all reachable states, and for $k \geq 0$, let R_k denote the set of states reachable by initialized paths of length at most k. In particular, $R_0 = I$ and $R_{2^{|X|}} = R$. The goal of a (formal) verification algorithm is to prove $R \subseteq P$, that is, to prove that the property P holds in all reachable states of M.

In all that follows we make definitions and claims with respect to some FSM M on state variables X, with its initial and transition relations I, T, and some property P, without explicitly mentioning them.

Definition 2.1 (invariants and inductive invariants):

- A CNF formula φ is an *invariant* if $s \in R \implies s \models \varphi$. Furthermore, φ is a *k-step invariant* if $s \in R_k \implies s \models \varphi$.
- A CNF formula φ is an *inductive invariant* if $I \implies \varphi$ and $(s \models \varphi \wedge \langle s, s' \rangle \in T) \implies s' \models \varphi$. Similarly, φ is a *k-step inductive invariant* if $I \implies \varphi$ and $(s \in R_{k-1} \wedge s \models \varphi \wedge \langle s, s' \rangle \in T) \implies s' \models \varphi$.

Note that inductive invariance implies invariance (but not vice versa).

Observation 2.1: If φ is an inductive invariant and $\varphi \implies P$, then P holds in all reachable states.

B. SAT solver with incremental capabilities

Our algorithm invokes SAT-based procedures for verifying P on M. In order to allow efficient incremental verification, we use an extended version of *mage*, an IBM SAT solver with incremental capabilities (see the homepage of the formal verification platform of IBM [Rul], [Six], for the description of mage). Incremental capabilities of the extended version of mage are similar to those of MiniSAT [ES03], [EMA10], specifically, the following query is supported (φ is a CNF formula and $\mathcal{A}$ is a set of clauses (assumptions)): $Sat(\varphi, \mathcal{A})$?

- if $\varphi \wedge \mathcal{A}$ is unsatisfiable return UNSAT and a minimal subset $\mathcal{B} \subseteq \mathcal{A}$ so that $\varphi \wedge \mathcal{B}$ is still unsatisfiable[2];
- if $\varphi \wedge \mathcal{A}$ is satisfiable return SAT and a satisfying assignment $\alpha \in \{0,1\}^X$;

For example, the following queries (referring to the given FSM) can be translated to a single query to the SAT solver:

- $s \in I$? (is s an initial state?)
- $\varphi \wedge T \wedge \neg\varphi'$? (Is there a pair of states, $\langle s, s' \rangle \in T$, so that s satisfies φ but s' does not? If the answer is yes we can also extract a witness pair of states $\langle s, s' \rangle$ from the assignment returned by the solver.)

C. Overview of ic3

In this section we provide a brief overview of the ic3 algorithm and highlight the features of ic3 that are relevant to incremental verification. For a more in-depth description of ic3 the reader is refered to the original paper by Bradley [Bra11] and to the paper of Brayton et al. [BEM11], who provide an overview of ic3 and present *pdr* – an improved version of ic3.

The main advantage of ic3 is its ability to perform unbounded SAT-based model checking without unfolding the transition relation. Given a model checking instance consisting of an FSM M and a property P as defined in Section II-A, the ic3 algorithm decides whether P is an invariant in M, producing an inductive strengthening if so, and a counterexample trace if not. The algorithm proceeds by incrementally refining and extending a sequence $\mathcal{F}_1, \ldots, \mathcal{F}_k$ of sets of clauses, each $\mathcal{F}_i$ forming an i-step inductive invariant CNF formula. Initially $k = 1$, and it gradually grows until termination. Furthermore, if M satisfies P (P holds in all reachable states) then on termination of ic3, for some $i \leq k$, the set $\mathcal{F}_i$ [3] forms an inductive invariant CNF formula that implies P:

- $I \implies c$ for every clause $c \in \mathcal{F}_i$;
- $\mathcal{F}_i \wedge T \implies \mathcal{F}_i'$;
- $\mathcal{F}_i \implies P$;

If M does not satisfy P then ic3 produces a *set of counter-examples* in form of a sequence $\alpha_0, \ldots, \alpha_k$ of partial assignments to X. In this sequence

[2]Although obtaining the minimal subset is hard, there are efficient ways to compute subsets that tend to be quite small in practice.

[3]$\mathcal{F}_i$ is the set that becomes empty during "clause pushing".

- all α_0 states (states formed by extending α_0 to a full assignment) are in I;
- all α_k states are **not** in P;
- all α_i states lead to some α_{i+1} state;

A concrete counter-example (CEX) may be extracted from such a sequence using $k+1$ calls to a SAT solver.

D. Additions to ic3

a) Shrinking partial assignments: Given a pair (s, α), where s is a full assignment to X (describing a state), α is a partial assignment to X' (describing a set of states) and there is an α-state t such that $\langle s, t\rangle \in T$, "shrinking" is the process of generalizing s into a set of states (represented by a sub-assignment of s) all leading to some α-state in one step.

The optimization from [BEM11] shrinks assignments using ternary simulation, and it is indeed very efficient. Here we propose a different method to further shrink the assignments using a SAT solver. This is described in detail in Section III-C.

b) Injecting invariants: We note that instead of starting "from scratch", ic3 can take, as input, a set $\mathcal{I}$ of invariant clauses, and use it as an absolute invariant: during its entire execution, all clauses from $\mathcal{I}$ can be directly injected into each of the sets $\mathcal{F}_i$.

III. Algorithm

In this section we present the main contribution of this paper – an algorithm for efficient incremental verification. We start with an overview of the algorithm, divided into an overview of the invariant finder and an overview of the incremental verification algorithm, which combines the invariant finder and ic3. Then we describe the algorithm in more detail, including some additional (smaller) contributions that further improve its performance.

A. Overview

We start with an overview of the *invariant finder*. Note that in addition to playing a crucial role in our algorithm for incremental verification, the invariant finder might be of independent interest. Invariant finder takes a model M and an arbitrary set $\mathcal{C}$ of (candidate invariant) clauses as input, and finds the *maximum* subset $\mathcal{I} \subseteq \mathcal{C}$ that is an inductive invariant with respect to M, i.e.:

- $I \implies c$ for every clause $c \in \mathcal{I}$;
- $\mathcal{I} \wedge T \implies \mathcal{I}'$;

The general idea of generating and exploiting inductive invariants in formal verification is not new (see e.g. [CCG+09], [CNQ09], [CMB07], [BMC+09]). The novelty of our algorithm is in the way it extracts the maximum set of inductive invariants (from an arbitrary set of candidates) using a SAT solver with incremental capabilities described in Section II-B; in particular, as the experimental results show, our algorithm is very efficient in practice (see Section IV).

The *incremental verification* algorithm combines ic3 and the invariant finder for storing and re-using information from previous verification runs in order to speed up subsequent verification on modified models and properties as follows.

If the result of the verification run of P on M is positive, the ic3 algorithm produces an invariant set $\mathcal{I}$ of clauses that implies the property P on M. We use the invariant finder to extract from $\mathcal{I}$ the largest invariant subset that holds on a modified model and provide this subset as a starting point to ic3 (see Section II-D).

If the verification of P fails on M, the set of counterexamples generated by ic3 is saved in a form of partial assignments, together with the clauses $\mathcal{C} \triangleq \bigcup_{i=1}^{k} \mathcal{F}_i$. Then, in subsequent runs we check whether the saved partial assignments can be extended to full assignments that produce a counterexample that is valid in a modified model with a modified property. If so, we have found a valid counterexample; otherwise, we extract the maximal inductive invariant from $\mathcal{C}$, and provide this subset as a starting point to ic3.

B. Detailed description of the algorithm

Invariant finder: Recall that our task is: Given a candidate set $\mathcal{C}$ of clauses, find its maximal subset $\mathcal{I}$ that is inductive invariant with respect to a given model M. To this end, we first check if the whole set $\mathcal{C}$ is inductive invariant by making the query $\mathcal{C} \wedge T \implies \mathcal{C}'$. If so, we are done; otherwise, there are clauses $c \in \mathcal{C}$ not implied by $\mathcal{C} \wedge T$. We then update $\mathcal{C}$ by removing (possibly a subset of) such clauses, and repeat.

To make this straightforward process more practical, we encode the SAT queries using auxiliary variables so that 1) the learnt information in the solver can be re-used from iteration to iteration; 2) $\mathcal{C}$ gets updated quickly – by detecting many non-implied clauses c simultaneously.

Specifically, we propose the following algorithm:

1) Remove from $\mathcal{C}$ all clauses not in I;
2) For each clause $c_i \in \mathcal{C}$ (whose literals refer to current cycle)
   - introduce two auxiliary variables x_i and y_i;
   - introduce the "shifted" copy c_i' of c_i (whose variables refer to the next cycle);
   - add the clause $(\neg x_i \vee c_i)$ to the solver (this is equivalent to $x_i \implies c_i$);
   - for each literal $a_{i,j}'$ of c_i', add the binary clause $(y_i, \neg a_{i,j}')$ to the solver (these clauses are equivalent to $c_i' \implies y_i$)

3) Initialize $it \leftarrow 0$, $\mathcal{I}_{it} \leftarrow \mathcal{C}$;
4) while $\mathcal{I}_{it} \neq \emptyset$ do:
   a) If $Sat(\{(x_1), ..., (x_{|\mathcal{I}_{it}|}), (\neg y_1 \vee \ldots \vee \neg y_{|\mathcal{I}_{it}|})\})$ is UNSAT report "$\mathcal{I} \triangleq \mathcal{I}_{it}$ is invariant";
   b) Else, let α denote the satisfying assignment that respects the assumptions $(x_1), ..., (x_{|\mathcal{I}_{it}|}), (\neg y_1 \vee \ldots \vee \neg y_{|\mathcal{I}_{it}|})$. Form $\mathcal{I}_{it+1}$ by removing from $\mathcal{I}_{it}$ all clauses with indices corresponding to each y_i assigned to 0 in α (see Remark 3.1), update x_i's and y_i's accordingly, and proceed with $it \leftarrow it + 1$;
5) Report "no invariant";

Remark 3.1:

- Observe that in Step 4b there must be at least one such y_i assigned to 0, thus in the worst case the number of iterations and SAT calls until termination is bounded by $|\mathcal{C}| - |\mathcal{I}|$. In practice, however, many y_i's may be assigned to 0 at once, making the loop terminate faster.
- Note that all SAT queries involve only a single copy of transition relation.
- Note that the invariant finder can take any set of clauses as the candidate set $\mathcal{C}$, thus there is no need to worry about validity of the previously saved information.

Claim 3.1: Invariant finder always outputs the maximum inductive invariant subset $\mathcal{I} \subseteq \mathcal{C}$ (which may be an empty set).

Proof: First, observe that there is a unique maximum inductive invariant subset; indeed, it is easy to verify that if $\mathcal{A}$ and $\mathcal{B}$ are inductive invariant subsets of $\mathcal{C}$, then so is $\mathcal{A} \cup \mathcal{B}$.

Let $\mathcal{I}$ be the output of the invariant finder upon termination. By definition, since in every iteration we check if $\mathcal{I}_{it} \wedge T \implies \mathcal{I}'_{it}$, the set $\mathcal{I}$ with which the loop terminates is clearly inductive invariant, thus we only need to argue that it is maximal. Let $\mathcal{I}^*$ denote the maximal invariant subset of $\mathcal{C}$, and assume towards a contradiction that $\mathcal{I}^* \not\subseteq \mathcal{I}$. Let it denote the first iteration in which some clause of $\mathcal{I}^*$ was removed from $\mathcal{I}_{it}$, that is, $\mathcal{I}^* \subseteq \mathcal{I}_{it}$ but $\mathcal{I}^* \not\subseteq \mathcal{I}_{it+1}$ (such it must exist since $\mathcal{I}^*$ was initially contained in $\mathcal{I}_0$). This means that $\mathcal{I}_{it} \wedge T$ did not imply $\mathcal{I}^*$, contradicting the inductiveness of $\mathcal{I}^*$. ■

Incremental verification: First, let us consider the (more challenging) case where the design passes the verification, namely, P holds in all reachable states of M. As explained in Section II-C, ic3 produces an invariant set $\mathcal{I}$ of clauses, and we can save it (in form of a standard CNF file) as the "proof of correctness" (see Observation 2.1).

Now assume that we want to *re-verify* P on the same model; this amounts to verifying the validity of $\mathcal{F}_i$, which is done in just three SAT calls:

1) $I \implies \mathcal{I}$?
2) $\mathcal{I} \wedge T \implies \mathcal{I}'$?
3) $\mathcal{I} \implies P$?

To summarize, if the saved proof is re-used for the same model, the verification is completed almost immediately (see Table I).

In case a model or a property are modified, our algorithm invokes the invariant finder to extract from $\mathcal{I}$ the largest invariant subset $\hat{\mathcal{I}}$ (with respect to the updated model), and injects it into ic3's data-structure as described in Section II-C. In particular, after this step, ic3 does not need to rediscover all the invariant clauses that hold both in the original and modified models. We note here that discovering a single invariant clause in ic3 requires several (often more than its size) SAT calls; hence, quite naturally, the amount of work that is saved by re-using the previous verification results corresponds to the amount of overlap between the two models.

Now let us consider the case where the verification fails and a counterexample is produced. We can store the set of counter-examples generated by ic3, in form of the aforementioned partial assignments $\alpha_0, \ldots, a_k$, and the set of clauses $\mathcal{C} \triangleq \bigcup_{i=1}^{k} \mathcal{F}_i$. To check if a concrete counter-example can be extracted in a future run (on a possibly modified model) we make $k+1$ SAT calls, essentially asking for a sequence $s_0, \ldots, s_k$ of full assignments to X with the following properties:

1) s_i is an extension of α_i for all $i \leq k$, and in particular, $s_0 \in I$;
2) $\langle s_i, s_{i+1} \rangle \in T$ for all $i < k$;
3) $s_k \notin P$.

If a counterexample cannot be extracted, we proceed with the usual execution of ic3, but first we attempt, using the invariant finder, to find an inductive invariant subset in $\mathcal{C}$ to use it as a starting point of ic3.

It is important to make the partial assignments $\alpha_0, \ldots, \alpha_k$ as small as possible (in the initial run), so that extracting a concrete counter-example becomes possible even when the modified model significantly differs from the original model. In Section III-C we discuss the improvements we added to the ic3 algorithm that enable to "shrink" the partial assignments computed by ic3.

C. Shrinking partial assignments with solver

We introduce an additional improvement of the algorithm allowing us to further shrink the partial assignments produced by ic3. The goal of this improvement is

to enlarge the set of valid counter-examples as discussed in Section II-D.

For some $i < k$, all α_i states can reach some α_{i+1} state in one step. Shrinking α_i results in an increase of the number of states that can lead to some α_{i+1} state, and is done using a single SAT call (and in fact this SAT call only involves BCP).

Specifically, given a pair $\langle s, s' \rangle \in T$ and the corresponding input values β (under which the transition $(s = \alpha \wedge inp = \beta) \rightarrow s'$ holds), we can shrink α to a partial assignment so that all α states lead to s' in one transition. To this end, we query the solver (knowing in advance that the answer is negative) if $(s = \alpha) \wedge (inp = \beta) \wedge \langle s, \hat{s}' \rangle \in T \wedge (\hat{s}' \neq s')$ is satisfiable. The first condition is passed to the solver in form of $|\alpha|$ assumptions, so that it also returns the minimal subset of those assumptions required for the conflict (see Section II-B). Then we can safely remove from α all those indices that are not required for reaching the conflict.

We conjecture that the reason why this additional step is effective is that instead of only looking at the structure of the model (as in ternary simulation) it takes into account all learned information (invariants from ic3 and the learned clauses from solver) to rule out unreachable states that do not lead to any α_{i+1} state.

IV. Experimental results

We implemented our algorithm for incremental verification in IVE (*incremental verification engine*), which is a part of the formal verification platform of IBM [Rul], [Six], and measured its performance on known benchmarks and on a real IBM hardware design.

HWMCC'10 benchmarks

The first set of experiments is based on the benchmarks used in the Hardware Model Checking Competition [HWM10] that was a part of the first Hardware Verification Workshop (HVW'10), affiliated with Computer-Aided Verification (CAV) Conference in 2010. We note that, out of the 758 publicly available benchmarks, IVE successfully verified 713 within 15 minutes, while the winner of the HWMCC'10, an engine *abcsuperprove* [abc] from Berkeley, verified 717 benchmarks within 15 minutes on comparable machines[4]. In other words, the performance of IVE is comparable to the state-of-the-art model-checking tools.

For each benchmark, we measured the running time of the verification procedure (setting 1 hour time limit), and the running time of incremental verification of the *same benchmark* (in other words, re-verification based on the results of the previous verification procedure). This setting emulates the most common scenario in incremental verification, where the changes introduced into the design do not, in fact, affect the verification at all, either because they fall outside of the cone of influence of the verified properties, or because they are "filtered out" during the preliminary reductions.

For each benchmark, we also measured the running time of IVE with incremental verification after small changes were introduced into the design. We simulated introduction of a small change by a *random mutation* in 1% of the assignments in the original instances (represented in AIG form) as follows. An assignment of the form $res = \ell_1 \& \ell_2$ was selected with probability 0.01 and mutated into one of the following assignments: $\{res = 0,\ res = 1,\ res = \neg\ell_1 \& \ell_2,\ res = \ell_1 \& \neg\ell_2,\ res = \neg\ell_1 \& \neg\ell_2,\ res = \ell_1,\ res = \neg\ell_1,\ res = \ell_2,\ res = \neg\ell_2\ \}$ with equal probability. In the absence of a domain-specific knowledge about the structure of the design, random mutations are the best approximation of small changes introduced into the design, and they also ensure that the experimental results are not biased towards any specific type of changes. We then measured the running time of IVE with incremental verification for the original benchmarks after the mutated ones and of the mutated ones after the original ones. In each case, the results of the previous verification were saved and used by the subsequent verification. The detailed results are presented in the full version of this paper [Rul]; Table I contains their summary. The row *"overall"* contains the results summarized for all benchmarks together, thus the speed-up represents a speed-up that is achieved by model-checking all benchmarks one after another. The overall speed-up is by the factor of 76 for re-verification of the same instance, and is by the factor of 3 after a small mutation was introduced. The median speed-up shows only a very slight improvement of our technique compared to re-verification without using the previous results. However, this is mostly due to the fact that median is dominated by very light instances, that constitute the vast majority of HWMCC'10 benchmarks. The most significant improvement in the running time was achieved for heavy instances: indeed, the median speedup (rerun vs. original) computed on instances that take > 60 seconds to solve is larger by two orders of magnitude (277 vs. 1.2).

IBM hardware design

In the second set of experiments, based on a real and up-to-date IBM hardware design, we measured the

[4]HWMCC'10 used Intel Quad Core 2.6 GHz with 8 GB; we used Intel Quad Core 2.6 GHz with 2 GB.

	original	rerun	speedup	original after mutated	speedup	mutated	mutated after original	speedup
Overall:	30597.01	402.89	75.92	10070.84	3.04	50294.46	37348.26	1.35
Average	42.49	0.55	114.06	13.98	1.80	69.85	51.87	2.95
Median	0.175	0.11	1.20	0.13	1.43	0.43	0.15	3.00

TABLE I
SUMMARY OF RUNTIMES ON 721 BENCHMARKS FROM HWMCC'10.

benefit of our algorithm when the changes between the two verification runs are significant and represent real changes in the design. Namely, we checked our algorithm on two versions of a model composed of a hardware design, augmented with a driver and a set of properties. The design implements logic that responds to requests from several threads. The two versions differ only in the driver: In the first ($1T$: 19,822 state variables and 299,185 gates before reductions; 2,187 state variables and 55,756 gates after reductions) the driver allows requests from a single thread, disabling all others. In the second version ($8T$: 19,831 state variables and 300,316 gates before reductions; 2,249 state variables and 56,458 gates after reductions), the driver enables 8 requesting threads. Therefore, in the first version all interleavings and priority-based decisions between threads are disabled, creating a significant difference in behaviors between the first and the second version. The verification suite for the design consists of 17 temporal logic properties. Table II presents the results of executing IVE with incremental verification implementation for both versions on all properties, including the original running time, the re-verification running time, and the running time of verifying one model after another.

The most important number in Table II is the accumulated speed-up, presented in the row ''overall". This number represents the performance gain of incremental verification in the scenario where the whole verification suite is re-checked in the design, which is a typical scenario after a bug fix and in regression verification.

The accumulated speed-up between the original run of model $1T$ and its re-run, presented in the row ''overall", is by the factor of ≈ 30, compared with re-run of the verification "from scratch"; the accumulated speed-up between the original run of model $8T$ and its re-run is by the factor of ≈ 61. These numbers model a performance gain in a typical scenario of regression verification, when the changes do not affect the properties at all.

The accumulated speed-up between the original run of model $1T$ and the run of model 1T after model $8T$ is by the factor of ≈ 3, and the accumulated speed-up of model $8T$ after model $1T$ is by the factor of ≈ 2. These numbers show that even if a change in the model is wide and applies to all behaviors of the design, using the incremental verification techniques has the potential of reducing the running time quite significantly. The difference in speed-up between verifying the $8T$ model after $1T$ and the $1T$ after the $8T$ is due to the fact that $1T$ has a small fraction of behaviors of $8T$, and hence the proof of $1T$ is only a small step in proving $8T$; on the other hand, the correctness of $1T$ for most part follows from the correctness of $8T$.

It is clear that in some cases, incremental verification techniques will not be beneficial in improving performance; after all, our algorithm incurs an additional computational cost in analyzing stored data. It is easy to see that this situation occurs when the instance is solved almost immediately, thus making the time required for analyzing the stored data very significant in the overall performance estimation – see, for instance, some of the results for properties $\varphi 07$ and $\varphi 08$ in Table II. We also note that the speed-up of this set of experiments is in most cases significantly smaller than the speed-up of the HWMCC'10 benchmarks; this is, again, due to the contribution of the time required to analyze the stored data in this relatively big design.

Overall vs. median and average speed-up: The median and average speed-ups in Tables I and II are affected by the (negligible) speed-up achieved on very light instances, where the analysis of the stored data is the major component in the overall verification. On the other hand, the *overall* speed-up is computed by dividing the sum of the running times of the original verification by the sum of the running times of incremental verification of all instances. Thus, the speed-up achieved on heavy instances, which are also the most important target for the application of incremental verification, is more accurately represented by the overall speed-up column.

The order of verification: Table I presents the results of executing incremental verification of the original designs after the mutant designs and vice versa. It is easy to see that the speed-up achieved by re-verifying the original design after the mutant design is larger (by the factor of 2) than the speed-up achieved by re-verifying the mutant design after the original. It is hard to say whether this difference is meaningful; SAT solvers are based on heuristic techniques, and the correlation of the size of an instance with the time required to solve it is not always clear. We conjecture that the difference may stem from the fact that mutations create SAT instances

property	1T	1T after 1T	speedup	1T after 8T	speedup	8T	8T after 8T	speedup	8T after 1T	speedup
ϕ01	1	0	∞	1	1.00	1	2	0.50	2	0.50
ϕ02	2330	299	7.79	3336	0.70	5603	129	43.43	1742	3.22
ϕ03	95	96	0.99	124	0.77	86	48	1.79	58	1.48
ϕ04	4913	91	53.99	1777	2.76	13458	234	57.51	1231	10.93
ϕ05	204	12	17.00	238	0.86	786	83	9.47	630	1.25
ϕ06	77	19	4.05	863	0.09	112	7	16.00	13	8.62
ϕ07	1	3	0.33	1	1.00	6	2	3.00	2	3.00
ϕ08	0	2	0.00	1	0.00	0	1	0.00	0	∞
ϕ09	13636	332	41.07	3848	3.54	13602	121	112.41	10291	1.32
ϕ10	8	29	0.28	174	0.05	15	1	15.00	0	∞
ϕ11	13823	79	174.97	2	6911.50	17421	55	316.75	9658	1.80
ϕ12	17	96	0.18	37	0.46	9	5	1.80	3	3.00
ϕ13	129	1	129.00	139	0.93	106	21	5.05	2	53.00
ϕ14	177	72	2.46	220	0.80	108	13	8.31	3	36.00
ϕ15	135	8	16.88	170	0.79	250	1	250.00	5	50.00
ϕ16	651	6	108.50	30	21.70	1814	36	50.39	32	56.69
ϕ17	407	93	4.38	749	0.54	779	116	6.72	772	1.01
Overall	36605	1238	29.57	11710	3.13	54160	883	61.34	24447	2.22

TABLE II
TWO VERSIONS OF AN IBM DESIGN – ONE THREAD AND EIGHT THREADS. ORIGINAL AND RERUN TIMES IN SECONDS.

that do not always correspond to real designs. Since SAT solvers are fine-tuned to efficiently solve real designs, introducing a small mutation can cause a significant increase in the number of clauses that SAT solver is required to learn in order to solve the instance.

V. CONCLUSIONS

We described a novel algorithm for incremental model-checking of hardware. Our algorithm is partially based on an improved version of the ic3 algorithm and it relies on the results of the previous verification procedure in order to improve the complexity of verification after a small change was introduced in either the design or the property. Our algorithm applies both to the case where the original model-checking procedure failed producing a counter-example, and to the case where the original model-checking was successful producing a proof of correctness. The algorithm requires storing a minimal amount of information from the proof or a counterexample, hence the overhead for the initial verification is negligible. We implemented our algorithm in IVE, an engine which is a part of the formal verification platform of IBM. We measured the performance improvements obtained by our implementation on publicly available benchmarks and on a real IBM design. The performance of IVE engine is on par with the state-of-the-art model checkers on known benchmarks, and we demonstrate that with our implementation we are able to achieve a speed-up of up to two orders of magnitude on "heavy" instances for re-verification after a small change.

To conclude, our technique clearly presents significant performance improvements, even when the difference between the original model and the changed model is significant. In fact, when we compare the original model-checking execution and re-verification of the same model, the speed-up is usually huge. We consider this result to be especially significant, because in the standard re-verification scenario – regression verification – the changes in the design are usually very small, and are often outside the cone of influence of the verification procedure.

REFERENCES

[abc] Abc homepage. http://www.eecs.berkeley.edu/ alanmi/abc/.

[BBC+11] Jason Baumgartner, Robert Brayton, Gianpiero Cabodi, Niklas Eén, Alexander Ivrii, Arie Matsliah, Alan Mishchenko, and Hari Mony. Personal communication. 2011.

[BEM11] Robert Brayton, Niklas Eén, and Alan Mishchenko. Efficient implementation of property directed reachability. In *IWLS*, 2011.

[BMC+09] Jason Baumgartner, Hari Mony, Michael L. Case, Jun Sawada, and Karen Yorav. Scalable conditional equivalence checking: An automated invariant-generation based approach. In *FMCAD*, pages 120–127, 2009.

[Bra11] Aaron R. Bradley. Sat-based model checking without unrolling. In *VMCAI*, pages 70–87, 2011.

[CCG+09] Gianpiero Cabodi, Paolo Camurati, Luz Garcia, Marco Murciano, Sergio Nocco, and Stefano Quer. Speeding up model checking by exploiting explicit and hidden verification constraints. In *DATE*, pages 1686–1691, 2009.

[CGJ+03] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM*, 50(5):752–794, 2003.

[CGMZ95] E.M. Clarke, O. Grumberg, K.L. McMillan, and X. Zhao. Efficient generation of counterexamples and witnesses in symbolic model checking. In *Proc. 32nd DAC*, pages 427–432. IEEE Computer Society, 1995.

[CGP99] E.M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.

[CK03] G. Cohen and O. Kupferman. Incremental LTL model checking. In *Proc. Workshop on semantics and verification of hardware and software systems*, 2003.

[CKP10] Hana Chockler, Daniel Kroening, and Mitra Purandare. Coverage in interpolation-based model checking. In *DAC*, pages 182–187, 2010.

[CKV06a] H. Chockler, O. Kupferman, and M.Y. Vardi. Coverage metrics for formal verification. *STTT*, 8(4-5):373–386, 2006.

[CKV06b] H. Chockler, O. Kupferman, and M.Y. Vardi. Coverage metrics for temporal logic model checking. *Formal Methods in System Design*, 28(3):189–212, 2006.

[CMB07] Michael L. Case, Alan Mishchenko, and Robert K. Brayton. Automated extraction of inductive invariants to aid model checking. In *FMCAD*, pages 165–172, 2007.

[CNQ09] Gianpiero Cabodi, Sergio Nocco, and Stefano Quer. Strengthening model checking techniques with inductive invariants. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 28(1):154–158, 2009.

[EMA10] Niklas Eén, Alan Mishchenko, and Nina Amla. A single-instance incremental sat formulation of proof- and counterexample-based abstraction. *CoRR*, abs/1008.2021, 2010.

[ES03] Niklas Eén and Niklas Sörensson. Temporal induction by incremental sat solving. *ENTCS*, 89(4), 2003.

[HKHZ99] Y. Hoskote, T. Kam, P.-H Ho, and X. Zhao. Coverage estimation for symbolic model checking. In *Proc. 36th DAC*, pages 300–305, 1999.

[HWM10] Hardware model checking competition, 2010. http://fmv.jku.at/hwmcc10/.

[KLS08] Orna Kupferman, Wenchao Li, and Sanjit A. Seshia. A theory of mutations with applications to vacuity, coverage, and fault tolerance. In *FMCAD*, pages 1–9, 2008.

[KW03] Detlef Kähler and Thomas Wilke. Program complexity of dynamic LTL model checking. In *Proceedings of CSL*, pages 271–284, 2003.

[LBBO01] Yassine Lakhnech, Saddek Bensalem, Sergey Berezin, and Sam Owre. Incremental verification by abstraction. In *TACAS*, pages 98–112, 2001.

[Rul] Rulebase homepage. http://www.haifa.il.ibm.com/projects/verification/RB_Homepage.

[SBS95] Gitanjali Swamy, Robert K. Brayton, and Vigyan Singhal. Incremental methods for FSM traversal. In *ICCD*, 1995.

[Six] Sixthsense homepage. http://domino.research.ibm.com/comm/research_projects.nsf/pages/sixthsense.index.html.

An Incremental Approach to Model Checking Progress Properties

Aaron R. Bradley, Fabio Somenzi, Zyad Hassan, Yan Zhang
Dept. of Electrical, Computer, and Energy Engineering
University of Colorado at Boulder
Email: bradleya@colorado.edu, fabio@colorado.edu

***Abstract*—An incremental algorithm for model checking progress properties is proposed. It follows from the following insight: any SCC-closed region of a system's state graph can be represented by a sequence of inductive assertions. Each iteration of the algorithm selects a set of states, called a skeleton, that together satisfy all fairness conditions; it then applies safety model checkers to attempt to connect the states into a reachable fair cycle. If this attempt fails, the resulting learned lemma takes one of two forms: an inductive reachability assertion that shows that at least one state of the skeleton is unreachable, or an inductive wall that defines two SCC-closed regions of the state graph. Subsequent skeletons must be chosen entirely from one side of the wall. Because a lemma often applies more generally than to the one skeleton from which it was derived, property-directed abstraction is achieved. The algorithm is highly parallelizable.**

I. Introduction

An incremental-style analysis, one that generates many intermediate lemmas on the way to a proof, yields property-focused abstraction, speed, and the possibility of parallelism. IC3 demonstrated the power of incrementality for safety model checking [1]. In this paper, we introduce an incremental algorithm for model checking progress properties [2] that harnesses safety model checkers.

While alternatives exist for lifting safety model checkers to progress properties [3], incrementality in itself is a worthwhile goal—whether one is using parallel resources to implement a portfolio of many safety model checkers [4] or applying the resources to accelerate computation [1]. In addition to using computational resources well, an incremental-style model checker generalizes from specific cases of why the property might not hold to intermediate lemmas about aspects of the system that are relevant to proving the property. In this way, it achieves property-focused abstraction of the system, and like a human verifier, it invests relatively little computation into discovering each lemma.

An *SCC-closed* region of the state graph is such that every SCC (*strongly connected component*) is either entirely contained in the region or entirely disjoint from the region. The fundamental insight for making an incremental progress model checker is that any SCC-closed region of a system's state graph can be represented by a sequence of inductive assertions. In other words, intermediate lemmas to characterize the SCCs of the state graph can take the form of inductive assertions. Each assertion defines a *one-way wall* that transects the state SCC graph. Given a selection of states, called a *skeleton*, that together satisfy the fairness conditions, one can prove via safety queries that (1) at least one of the states is unreachable from the system's initial condition; that (2) one of the skeleton states cannot reach another, providing a one-way wall; or that (3) the skeleton can actually be completed to form a "lasso"-shaped counterexample. How to use the second outcome is the crux of the algorithm.

Suppose that P is the one-way wall, an inductive proof that one skeleton state cannot reach another. Any fair cycle must occur completely on one side of the wall: all of its states must either satisfy P, or they must all satisfy $\neg P$. For once a path crosses the wall, it cannot return. Hence, when finding fair cycles, the transition relation can be strengthened by the constraint $P \leftrightarrow P'$, which excludes transitions that cross the wall P. (Technically, because P is inductive, only $\neg P \rightarrow \neg P'$ is necessary.) This constraint is the incremental information expressed by the lemma P. Subsequent skeletons must be chosen from one side of the wall or the other, and eventually every reachable *arena* defined by the sequence of walls must become unfair, if the progress property indeed holds. A crucial characteristic of a proof, when IC3 is used as the safety model checker, is that it potentially splits many arenas, not just the arena from which the skeleton was selected. Hence, not every arena need be examined explicitly.

After introducing the problem domain (Section II), Section III describes the algorithm in detail. Then Section IV relates the proposed algorithm to previous work. Finally, Section V investigates empirical characteristics of the algorithm in relation to other well-known techniques.

II. Background

Following standard practice, we represent a *finite-state system* as a tuple $S : (\overline{i},\ \overline{x},\ I(\overline{x}),\ T(\overline{i}, \overline{x}, \overline{x}'))$ consisting of primary inputs $\overline{i}$, state variables $\overline{x}$, a propositional formula $I(\overline{x})$ describing the initial configurations of the system, and a propositional formula $T(\overline{i}, \overline{x}, \overline{x}')$ describing the transition relation. Primed state variables $\overline{x}'$ represent the next state.

A state of the system is an assignment of Boolean values to all variables $\overline{x}$ and is described by a *cube* over $\overline{x}$, which, generally, is a conjunction of literals, each *literal* a variable or its negation. An assignment s to all variables of a formula F either satisfies the formula, $s \models F$, or falsifies it, $s \not\models F$. If s is interpreted as a state and $s \models F$, we say that s is

an F-*state*. A formula F *implies* another formula G, written $F \Rightarrow G$, if every satisfying assignment of F satisfies G.

A *clause* is a disjunction of literals. A subclause $d \subseteq c$ is a clause d whose literals are a subset of c's literals.

A *run* of S, $s_0, s_1, s_2, \ldots$, which may be finite or infinite in length, is a sequence of states such that $s_0 \models I$ and for each adjacent pair (s_i, s_{i+1}) in the sequence, $\exists \bar{i}.(\bar{i}, s_i, s'_{i+1}) \models T$. That is, a run is the sequence of assignments in an execution of the transition system. A state that appears in some run of the system is *reachable*.

An *invariance property* $P(\overline{x})$, a propositional formula, asserts that only P-states are reachable. P is *invariant* for the system S (that is, S-invariant) if indeed only P-states are reachable. If P is not invariant, then there exists a finite *counterexample* run $s_0, s_1, \ldots, s_k$ such that $s_k \not\models P$. An invariance property $P(\overline{x})$ is *inductive* if

1) (*initiation*) every initial state satisfies the property: $I(\overline{x}) \Rightarrow P(\overline{x})$; and
2) (*consecution*) every transition from a P-state leads to a P-state: $P(\overline{x}) \wedge T(\bar{i}, \overline{x}, \overline{x}') \Rightarrow P(\overline{x}')$.

An assertion F is *inductive relative to* another assertion G, possibly containing primed variables, if

1) every initial state satisfies F: $I(\overline{x}) \Rightarrow F(\overline{x})$; and
2) F satisfies consecution under assumption G: $G(\overline{x}, \overline{x}') \wedge F(\overline{x}) \wedge T(\bar{i}, \overline{x}, \overline{x}') \Rightarrow F(\overline{x}')$.

Relative induction is useful for gaining knowledge about a system in an incremental fashion [2].

Checking a *safety property* of S is reducible to checking an invariance property. While the work described in this paper makes use of safety model checkers, the primary focus is on analyzing *progress properties* [2]. For this purpose, we need to introduce *fairness* into our system models. A *Büchi fairness condition* $B(\overline{x})$ of a system S is a propositional formula that constrains the infinite runs of S: infinite run $s_0, s_1, s_2, \ldots$ is a *computation* of S if infinitely many s_i satisfy B, $s_i \models B$. We represent a system with fairness conditions as the augmented tuple $S : (\bar{i},\ \overline{x},\ I(\overline{x}),\ T(\bar{i}, \overline{x}, \overline{x}'),\ \mathcal{B} : \{B_1(\overline{x}), \ldots, B_\ell(\overline{x})\})$. The fundamental question that this paper addresses is that of *language emptiness*: *Does S lack computations?*

Model checking LTL properties of systems motivates this problem. Deciding whether a system S satisfies LTL property P is reducible to checking language emptiness of the system constructed as the parallel composition of S and the *Büchi automaton* A for $\neg P$. The resulting system inherits the fairness conditions of S as well as one additional fairness condition, the Büchi acceptance condition of A.

A fairness condition B of S is *weak* [5] if for every computation of S there exists k such that $i \geq k \Rightarrow s_i \models B$. Weak fairness conditions correspond to *persistence properties* [2]. Multiple weak conditions can be reduced to just one weak condition so that the search for fair cycles can be restricted to the reachable states that lie on some cycle where the weak condition holds globally. When a fairness condition of S is inherited from a Büchi automaton, its strength is at most the strength of the fairness condition of the automaton [6].

Because S is finite-state, a nonempty language described by system S with fairness conditions must have a computation that takes the form of a *reachable fair cycle*: a "lasso" consisting of a "stem" (a finite run) from an initial state s_0, $s_0 \models I$, to an intermediate state s_i, and a "loop" (also a finite run) from s_i back to itself that contains at least one state s_j satisfying each fairness condition B_j. Our algorithm searches for such reachable fair cycles.

III. Fair: An Incremental Algorithm

A. The Basic Algorithm

The algorithm works in the following manner. It iteratively executes a *skeleton query* to obtain a set of states that together satisfy all fairness conditions. If the query is ever unsatisfiable, the algorithm concludes that the language of S is empty. It next attempts to complete the skeleton into a reachable fair cycle by executing a set of safety model checking queries to connect the initial states to one state of the skeleton, and each state of the skeleton to another in such a way as to create a cycle. If it succeeds, then it has found a reachable fair cycle and thus concludes that the language of S is not empty. Otherwise, one of the safety queries fails and returns an inductive proof. If the *stem query*, which attempts to connect an initial state to a skeleton state, fails, then the proof provides new global unreachability information. If a *cycle query*, which attempts to connect one skeleton state to another, fails, then the proof yields new information about the SCC structure of S. In particular, the proof says that a fair cycle, if one exists, must occur completely on one side or the other of the inductive proof (that is, all states of the cycle must satisfy the proof, or all states must falsify it). Both situations thus cause the algorithm to make progress, so that it eventually must find a reachable fair cycle or conclude that one does not exist.

In detail, consider system $S : (\bar{i},\ \overline{x},\ I(\overline{x}),\ T(\bar{i}, \overline{x}, \overline{x}'),\ \mathcal{B} : \{B_1, \ldots, B_\ell\})$. Let $\mathcal{R}$ denote a growing list of global reachability assertions, each of which is inductive relative to its predecessors and provides information about unreachable states. Let $\mathcal{W}$ denote a growing list of *walls* that no fair cycle can cross, each of which satisfies consecution relative to previously generated walls and $\mathcal{R}$, as discussed in detail later. Walls represent learned information about the SCC structure of the state graph of S. A set of walls $\mathcal{W}$ defines $2^{|\mathcal{W}|}$ (possibly empty) arenas; each arena (and, consequently, any union of arenas) is SCC-closed. Both lists are empty initially.

The *skeleton query* returns a *skeleton* or, if unsatisfiable, indicates that the language of S is empty. A skeleton consists of a set of states that together satisfy all fairness conditions. The query requires (in its complete form, but see Section III-B) one copy of $\overline{x}$ for each fairness condition $B \in \mathcal{B}$:

$$\bigwedge_{B \in \mathcal{B}} \left[\begin{array}{l} \quad B(\overline{x}_B) \wedge \bigwedge_{R \in \mathcal{R}} R(\overline{x}_B) \\ \wedge \quad \bigwedge_{W \in \mathcal{W}} (c_W \rightarrow W(\overline{x}_B)) \wedge (\neg c_W \rightarrow \neg W(\overline{x}_B)) \end{array} \right]$$

The first line of the query requires that the states of a model be such that each fairness condition is represented by states

not known to be unreachable. The second line requires that all states of a model come from the same *arena*, that is, are on the same side of each wall $W \in \mathcal{W}$. The *choice variables* c_W for $W \in \mathcal{W}$ achieve this requirement: a model can only have one assignment to each choice variable, and that assignment determines on which side of each wall the skeleton appears.

If the skeleton query is satisfiable, any model describes some set of states $\{s_0, \ldots, s_{n-1}\}$, where $n \leq |\mathcal{B}|$, that satisfy the fairness conditions, that are not known to be unreachable, and that are not separated by any previously discovered wall ($n < |\mathcal{B}|$ if some state satisfies multiple fairness conditions and appears more than once). The task, then, is to attempt to complete the skeleton into a reachable fair cycle or, if the attempt fails, to learn new information in the form of an inductive reach assertion or a wall about why any reachable cycle of S cannot contain all of the states of the skeleton.

Any safety model checker that produces counterexample runs or inductive proofs can address this task, although we discuss later why proofs from certain model checkers, like IC3, make better walls. Let $\mathsf{reach}(S, C, F, G)$ be a function that accepts a system S, a set of constraints $C(\overline{x}, \overline{x}')$ on the transition relation, an initial condition F, and a target G; and that returns either a counterexample run from an F-state to a G-state, or an inductive proof $P(\overline{x})$ separating F from G, that is, such that

- $F(\overline{x}) \Rightarrow P(\overline{x})$,
- $P(\overline{x}) \Rightarrow \neg G(\overline{x})$, and
- $C(\overline{x}, \overline{x}') \wedge P(\overline{x}) \wedge T(\overline{i}, \overline{x}, \overline{x}') \Rightarrow P(\overline{x}')$.

Notice that P is inductive relative to the constraints C.

For an n-state skeleton, $n + 1$ reach-queries are required. One *stem query* determines if the skeleton is reachable, given the learned reachability information $\mathcal{R}$:

$$\mathsf{reach}\left(S,\ \bigwedge_{R \in \mathcal{R}} R(\overline{x}),\ I,\ s_0\right)\ . \qquad (1)$$

This query asks whether s_0 is reachable from an I-state. While the previous reachability information is not necessary, it is provided to restrict the search. One could instead pose the more general query in which the disjunction of all skeleton states, $\bigvee_{i=0}^{n-1} s_i$, is the target; or pose n queries, one for each state s_i, depending on computational resources. If an instance of a stem query is unsatisfiable, the proof is added to $\mathcal{R}$.

The remaining n queries are *cycle queries*, which determine if each state s_i can reach a successor $s_{i \oplus_n 1}$, where $\oplus_n$ is addition modulo n. One can pose up to n^2 queries if the computational resources are available. These queries are more complicated than the stem query because more previously-derived information can be used.

A naive cycle query takes the following form:

$$\mathsf{reach}\left(S,\ \mathsf{true},\ s_i,\ s_{i \oplus_n 1}\right)\ . \qquad (2)$$

If s_i cannot reach $s_{i \oplus_n 1}$, then the query returns an inductive proof P: $s_i \Rightarrow P$, $P \wedge T \Rightarrow P'$, and $P \Rightarrow \neg s_{i \oplus_n 1}$. P is a wall: no cycle can cross it because no P-state has a $\neg P$-state successor. While a $\neg P$-state can have a P-state successor, crossing the wall is pointless when searching for a cycle since it cannot be crossed again. P can thus be added to $\mathcal{W}$, the list of walls that no fair cycle can cross.

However, this query does not exploit known information. For a cycle query, each wall $W \in \mathcal{W}$ constrains the transition relation as follows:

- If no W*-skeleton* (a skeleton whose states are W-states) exists, then $\neg W \wedge \neg W'$.
- If no $\neg W$-skeleton exists, then $W \wedge W'$.
- Otherwise (if both sides contain skeletons), $W \leftrightarrow W'$.

Unfortunately, encoding the full constraints in the cycle queries requires a quantifier alternation. Instead, each new wall W is tested to learn a new constraint on T; such constraints are collected in the *constraint list* $\mathcal{C}$:

- If no W-skeleton exists, then add $\neg W \wedge \neg W'$. (Technically, because W is inductive, $\neg W'$ is sufficient.)
- If no $\neg W$-skeleton exists, then add $W \wedge W'$. (Technically, W is sufficient.)
- Otherwise, add $W \leftrightarrow W'$. ($W' \rightarrow W$ is sufficient.)
- Optionally, if W is determined (heuristically) to be uninteresting for constraining T, do not add a constraint.

It is also possible to exclude regions defined by multiple walls—even individual arenas—that lack fair skeletons. However, this more general heuristic, while potentially useful at the beginning of the analysis, is too expensive for general use. The list $\mathcal{C}$ is used to constrain T during the cycle query:

$$\mathsf{reach}\left(S,\ \bigwedge_{R \in \mathcal{R}} R \wedge \bigwedge_{C \in \mathcal{C}} C,\ s_i,\ s_{i \oplus_n 1}\right)\ . \qquad (3)$$

This query is satisfiable precisely when the naive cycle query (2) is satisfiable. However, a proof discovered during evaluating this query need only be inductive relative to the information contained in $\mathcal{R}$ and $\mathcal{C}$ rather than on its own.

There is one technicality: when there is only one state in the skeleton, the form of the single cycle query is different. A *single-state skeleton cycle query* determines if a state s_0 can reach itself nontrivially, which is stated as a query determining whether the successors of s_0 can reach s_0:

$$\mathsf{reach}\left(S,\ \bigwedge_{R \in \mathcal{R}} R \wedge \bigwedge_{C \in \mathcal{C}} C,\ \mathsf{post}(S, s_0),\ s_0\right)\ . \qquad (4)$$

(Safety model checkers such as IC3 can be modified in such a way that the post-image does not have to be computed explicitly.) Additionally, a proof P does not eliminate the same skeleton from further consideration. P, as a wall, separates s_0 (which satisfies $\neg P$) from its successors (which satisfy P). However, s_0 can be selected as a skeleton again. There are several solutions to avoid this nontermination situation: (1) constrain the skeleton query so that only states with some successors in the same arena can be selected, (2) for a cycle proof P, construct a wall defined by $W = P$ and $\neg W = \neg P \wedge \neg s_0$ instead of the usual wall defined by $W = P$ and $\neg W = \neg P$. More powerful refinements of each of these solutions are discussed in Sections III-B and III-C.

If all queries (1) and (3/4) return counterexamples, the runs are assembled into a computation that takes the form of a lasso, proving that the language of S is nonempty. Otherwise, a proof P returned by the stem query provides new global reachability information, so P is added to $\mathcal{R}$; or a proof P returned by one of the cycle queries provides new information about the SCC structure of S, so P is added to the set of walls, $\mathcal{W}$, and a new constraint may be derived from P and added to $\mathcal{C}$. Then with this new information, the algorithm again executes the skeleton query. The new information is sufficient to exclude the same skeleton from being selected again.

Several aspects of this basic algorithm are nondeterministic and thus invite further detail and heuristics:

- Selection of the skeleton (Section III-B).
- The order in which the stem query and cycle queries are executed (Section III-F).
- The proofs themselves (Section III-D).
- Whether to derive a new constraint on T from a wall W. Technically, none are required for completeness; using some accelerates the search; and using all can slow the search. Our implementation derives a new constraint from W if one side of W lacks skeletons or if W consists of a single clause.

Section III-E discusses an incomplete but effective method of discovering information about the SCC structure independently of skeletons.

B. Choosing Skeletons

We discuss two enhancements to the basic algorithm. The first minimizes the number of states in skeletons by formulating the skeleton query to force states to satisfy multiple fairness conditions when possible. The intuition is that the discovered walls might explain more if the separated states satisfy multiple fairness conditions. The second enhancement adds constraints to the skeleton query to force a selected state to have at least K-step successor and predecessor sequences of different states within the arena, unless some state in these sequences is the state itself. This enhancement effectively reduces the number of skeletons to consider. For $K > 0$, single-state skeletons with no successors cannot be chosen, so that this enhancement addresses the termination issue raised in the previous section.

To (heuristically) minimize the number of states selected, let $j : \mathcal{B} \to \{1, \ldots, |\mathcal{B}|\}$ be a map from the Büchi fairness conditions of S to indices, where j can map different fairness conditions to the same index. The skeleton query then has the following form:

$$\bigwedge_{B\in\mathcal{B}} \left[\begin{array}{l} B(\overline{x}_{j(B)}) \wedge \bigwedge_{R\in\mathcal{R}} R(\overline{x}_{j(B)}) \\ \wedge \bigwedge_{W\in\mathcal{W}} (c_W \to W(\overline{x}_{j(B)})) \wedge (\neg c_W \to \neg W(\overline{x}_{j(B)})) \end{array} \right]$$

Potentially fewer copies of the assertions are required.

Of course, the query is only complete, in the sense that its unsatisfiability implies the emptiness of the language of S, when each condition is mapped to a unique index. Hence, the modified algorithm finds a map j that (heuristically) minimizes the number of unique indices while still producing a satisfiable query. If because of new information the query becomes unsatisfiable, a new map, which may have the same number of unique indices but must at least combine conditions differently, is generated. Only when the query corresponding to a bijective mapping is unsatisfiable does the algorithm conclude that the language of S is empty.

The second enhancement reduces the number of potential skeletons by requiring selected states to have nontrivial sequences of successors and predecessors. For each unique index, $2K$ unrollings of the transition system are asserted with time-steps ranging from $-K$ to K. An additional constraint asserts that either the predecessor sequence or the successor sequence includes $\overline{x}^0_{j(B)}$ itself, or otherwise that the predecessor sequence and the successor sequence are each loop-free, yielding the following skeleton query:

$$\bigwedge_{B\in\mathcal{B}} \left[\begin{array}{l} B(\overline{x}^0_{j(B)}) \wedge \bigwedge_{R\in\mathcal{R}} R(\overline{x}^{-K}_{j(B)}) \\ \wedge \bigwedge_{k\in\{-K,\ldots,K-1\}} T(\overline{i}^k_{j(B)}, \overline{x}^k_{j(B)}, \overline{x}^{k+1}_{j(B)}) \\ \wedge \left(\begin{array}{l} \bigvee_{k\in\{-K,\ldots,-1,1,\ldots,K\}} \overline{x}^k_{j(b)} = \overline{x}^0_{j(b)} \\ \vee \ (\mathsf{loopFree}_{<0} \wedge \mathsf{loopFree}_{>0}) \end{array} \right) \\ \wedge \bigwedge_{W\in\mathcal{W}} (c_W \to W(\overline{x}^{-K}_{j(B)})) \wedge (\neg c_W \to \neg W(\overline{x}^K_{j(B)})) \end{array} \right]$$

where

$$\mathsf{loopFree}_{<0} = \bigwedge_{k\in\{-K,\ldots,-2\}} \bigwedge_{\ell\in\{k+1,\ldots,-1\}} \overline{x}^k_{j(B)} \neq \overline{x}^\ell_{j(B)} ,$$

and $\mathsf{loopFree}_{>0}$ is similarly defined.

C. Single-State Skeletons

Recall that single-state skeletons must be handled via a single-state cycle query (4) that determines whether the successors of s_0 can reach s_0. If the query returns a proof P, then the successors of s_0 must satisfy P since they define the initial condition, while s_0 itself must falsify P since it defines the target. In other words, the proof P cuts the state space directly through the transitions between s_0 and its successors, so that s_0 is on the edge of an arena.

Consequently, the following propositional query, which asks if s_0 has any $\neg P$-successor, must be unsatisfiable:

$$s_0 \wedge \neg P \wedge T \wedge \bigwedge_{C\in\mathcal{C}} C \wedge \bigwedge_{R\in\mathcal{R}} R \wedge \neg P' .$$

From the unsatisfiable core one can extract a cube $d \subseteq s_0$ whose $\neg P$-states lack $\neg P$-state successors. Its negation can be conjoined to $\neg P$ to form one side of the wall: $\neg P \wedge \neg d$, which eliminates at least s_0 from consideration.

If a successor state t of s_0 is known, for example, when $K > 0$ (Section III-B), a similar query can test whether t has P-predecessors. If not, one can extract a cube $d \subseteq t$ from the core and strengthen P with $\neg d$.

D. *Refining IC3 Proofs*

IC3 discovers inductive proofs P in CNF [1]. While adequate as certificates of unreachability, which is what matters in the context of safety model checking, the proofs can be unnecessarily large and specific to the query. For example, a proof from a cycle query contains the clause $\neg s_{i \oplus_n 1}$. We describe several methods of manipulating an IC3 proof to make it more general and to reduce its size.

The property can be generalized. Let $P = F \wedge \neg s_{i \oplus_n 1}$. The MIC algorithm [7] is applied to $\neg s_{i \oplus_n 1}$ in the context of P to derive a subclause $c \subseteq \neg s_{i \oplus_n 1}$, yielding proof $\overline{P} = F \wedge c$.

The proof P can be strengthened by applying MIC iteratively to the clauses of P until no further changes are possible. We apply this manipulation to global reachability proofs.

The proof P can be weakened. Again, let $P = F \wedge c$, where $c \subseteq \neg s_{i \oplus_n 1}$. A MIC-like algorithm is applied to drop clauses of F. First, observe that one can use the unsatisfiable core of $F \wedge c \wedge T \wedge \neg P'$, corresponding to consecution, to reduce P: any clause of F that is not in the core is unnecessary. Second, observe that dropping an arbitrary clause d can result in a non-inductive assertion because d might be required to support other clauses. In this case, consecution fails with some counterexample states (t, t'). The set of clauses that t' falsifies in the next state must then be dropped, as they are no longer supported. Dropping these clauses may in turn require dropping other clauses, and so on. If ever c becomes unsupported (that is t' falsifies c'), the process must backtrack to the last inductive assertion; there, the same steps can be applied to a different clause unless all options have been explored. Alternately, if the process converges on an assertion for which consecution holds, the first observation can be used to further reduce the clause set. Then the clause-dropping process can be attempted again.

These manipulations can be combined to heuristically derive a minimally-sized proof: iteratively apply strengthening followed by weakening until no further changes can be made. Strengthening may reduce the number of literals, while weakening may reduce the number of clauses.

E. *Skeleton-Independent Proofs*

Skeletons serve to direct the exploration of the SCC structure of S; however, some important facts are not easily derived by this property-directed method.

Consider, for example, a system consisting of a single n-bit counter whose bits are named $b_0, \ldots, b_{n-1}$, where b_{n-1} is most significant; an output bit o that switches to 1 the first time that the counter reaches all 1s and then stays at 1; a fairness condition that asserts that infinitely often $\neg o$; and an initial condition in which all bits are 0. The system is unfair because $o = 0$ only for the first iteration through the counter's values. An ideal proof is constructed as follows:

- Inductive assertion o, since once o becomes 1, it stays 1. No skeleton exists among the o-states, so $\neg o \wedge \neg o'$ constrains future cycle queries.
- Inductive (relative to $\neg o$) assertion b_{n-1}, since once b_{n-1} becomes 1, it stays 1 in the $\neg o$ arena. Both sides of the proof have skeletons, so $b_{n-1} \leftrightarrow b'_{n-1}$ constrains future cycle queries.
- Inductive (relative to previous walls) assertion b_{n-2}, since once b_{n-2} becomes 1, it stays 1 in *every arena defined by the previous two proofs*.
- Similarly, inductive assertions $b_{n-3}, \ldots, b_0$ are derived in that order, each holding relative to prior information.

When $K > 0$ (see Section III-B), the skeleton query becomes unsatisfiable after these walls are generated: because of the learned constraint $b_0 \leftrightarrow b'_0$, each arena has only one state, and that state lacks a successor in its arena. The size of the proof is thus linear in the size of the counter. This proof sequence discovers the obvious ranking function.

Unfortunately, discovering the first fact with skeletons requires stumbling fortuitously upon the skeleton in which all $b_i = 1$ and $o = 0$. This state's only successor is the state in which all $b_i = 0$ and $o = 1$, so an inductive separating wall is indeed o. However, no other fair state has a successor in which $o = 1$, so the resulting walls cannot simply be o (since the successor must satisfy it) or $\neg o$ (since both the fair state and its successor satisfy it and thus are not separated). In other words, their walls must involve b_i literals and be less informative as a result. Discovering subsequent facts via skeletons requires similarly, although decreasingly, fortuitous selections; for example, to discover b_{n-1} requires examining precisely the one state for which $b_{n-1} = 0$ and whose successor has $b_{n-1} = 1$.

In contrast, iteratively testing whether any literal of the state variables of the system is itself a proof (that is, satisfies consecution relative to known information) produces the linear-sized proof quickly. Let ℓ be such a literal. Then if the formula

$$\bigwedge_{R \in \mathcal{R}} R \wedge \bigwedge_{C \in \mathcal{C}} C \wedge T \wedge \ell \wedge \neg \ell' \tag{5}$$

is unsatisfiable, ℓ obeys consecution: once ℓ is true, it is true henceforth. In this case, ℓ is a wall.

While this heuristic is incomplete, its effectiveness on counters suggests that such simple queries should be executed frequently, for example, after each addition to $\mathcal{R}$ or $\mathcal{C}$. Experiments show that on more complicated systems, several iterations of skeleton-based wall construction create opportunities to learn new non-skeleton-directed proofs.

In addition to counters, this technique quickly derives information about property automata for favorable encodings of the automata's transition relations. A one-hot encoding, for example, reveals structural information readily. Predicates derived from the system description may also be effective candidates for this heuristic.

F. *Executing Queries*

The ideal computational environment in which to run this algorithm is a highly parallel one:

- $n + 1$ queries must be analyzed until either all yield counterexample runs or one yields a proof.
- Each query can be analyzed by a portfolio of safety model checkers, even incomplete methods: based on BDDs [8],

- BMC [9], interpolation [10], IC3 [1], and simulation. While any counterexample run is informative, only proofs that are inductive are useful. However, proofs produced by non-approximating safety checkers (e.g., BDD-based) will cause fair to derive walls that are only useful in the arena from which the skeleton was drawn, thus hindering the algorithm's ability to generalize from skeletons, a key characteristic. Hence, we rely on IC3 for proofs.
- IC3 is itself parallelizable.
- As the overall methodology is incremental, multiple skeletons can be analyzed simultaneously in the same way that multiple counterexamples to induction can be analyzed simultaneously in IC3.

However, if parallel resources are unavailable, one observation has become clear from experimentation: queries must be analyzed in a time sharing fashion. Since only one query need be unsatisfiable to rule out a skeleton, a poor time allocation can cause excessive time to be wasted on finding irrelevant counterexample runs. Varying the order in which queries are executed also seems important, so that one fairness condition is not favored over others or over the stem query.

G. A Summary of the Algorithm

Figure 1 lists pseudocode for the fair algorithm.

Two forms of the skeleton query (`skelQ`) are used: the full query at lines 6, 44, and 46, based on the bijection ι between $\mathcal{B}$ and $\{1, \ldots, |\mathcal{B}|\}$ defined at line 4; and the skeleton-minimization version (Section III-B) at line 10, based on the map j defined at line 8. Notice that the latter version is only used to enforce a preference on skeletons and not, for example, to construct C at lines 44-49. In this pseudocode, all queries use the same K; however, it would be reasonable for the queries at lines 44 and 46 to use a different unrolling than K. In particular, since the full version has as many copies of T as $2K|\mathcal{B}|$, it may only be practical to use an unrolling of 0 or 1 for these queries, which are executed more frequently than the one at line 6.

Lines 13-16 correspond to finding a skeleton-independent proof (Section III-E); if none exist, then this choice is disabled. Lines 18-25 correspond to choosing a skeleton (Sections III-A and III-B) and executing the one stem (`stemQ`) and m cycle (`cycleQ`) queries (Section III-F).

Lines 27-50 act on the result of the search for a new proof. If all (safety) queries returned counterexample runs, then they can be formed into a "lasso" representing a computation of S (lines 27-28). Otherwise, if `stemQ` returned proof P, then P describes new reachability information (lines 30-32).

Otherwise, if either a skeleton-independent proof P is discovered (Section III-E) or a cycle query returned proof P, then P is a wall, and P and $\neg P$ are SCC-closed regions (lines 34-50). If the skeleton has just one state ($m = 0$) and $K = 0$, then it is necessary to augment $\neg P$ with additional information (Section III-C), and it might be useful to do so if $K > 0$ as well (lines 38-40). Line 40 takes liberties with logic: it says that $\neg P$ will henceforth be $\neg P \wedge \neg d$, so that $\neg P$ is no longer simply the negation of P. In other words, the list $\mathcal{W}$ of walls

```
bool fair(S : system, K : uint):                                  1
 R := ∅, W := ∅, C := ∅                                           2
 {for full skeleton query}                                        3
 ι := bijection between B and {1,...,|B|}                         4
                                                                  5
 while skelQ(R, W, ι, K) is sat:                                  6
  {for skeleton-minimization (§B)}                                7
  j := map(R, W, K)                                               8
                                                                  9
  while skelQ(R, W, j, K) is sat:                                 10
   result :=                                                      11
    heuristically choose:                                         12
     {skeleton-independent proof (§E)}                            13
     let ℓ be a literal or other predicate                        14
      such that query (5) is unsat                                15
     P := ℓ                                                       16
    alternately:                                                  17
     {skeleton-based analysis (§A)}                               18
     s_0,...,s_{m-1} := skelQ(R, W, j, K)                         19
     in parallel do until                                         20
       all yield counterexamples                                  21
       or one returns a proof:                                    22
      stemQ(R, s_0)                                               23
      for i ∈ {0,...,m-1}:                                        24
       cycleQ(R, C, s_i, s_{i⊕_m 1})                              25
                                                                  26
   if result is all counterexamples:                              27
    return true {non-empty language}                              28
                                                                  29
   elif result is a proof P from stemQ:                           30
    {new reachability information}                                31
    R := R ∪ {P}                                                  32
                                                                  33
   elif result is a proof P from                                  34
     a skeleton-independent search                                35
     or a cycleQ:                                                 36
    {P is a wall: P, ¬P are SCC-closed}                           37
    if m = 1: {§C}                                                38
     d := singleCube(R, C, s_0, P)                                39
     ¬P := ¬P ∧ ¬d                                                40
    W := W ∪ {P}                                                  41
    if heuristic(P):                                              42
     {c_P is the choice variable for P}                           43
     if skelQ(R, W, ι, K) ∧ c_P is unsat:                         44
      C := ¬P'                                                    45
     elif skelQ(R, W, ι, K) ∧ ¬c_P is unsat:                      46
      C := P                                                      47
     else                                                         48
      C := P' → P                                                 49
    C := C ∪ {C}                                                  50
                                                                  51
 return false {empty language}                                    52
```

Fig. 1. The fair algorithm: Does S have a computation?

must actually be implemented as two lists, one to hold positive proofs and the other to hold possibly modified negative proofs. If P is determined heuristically to be interesting (line 42), then a C constraint is constructed and added to $\mathcal{C}$ (lines 42-50, Section III-A). Lines 44-45 correspond to the case in which no skeleton exists on the P side of the wall; lines 46-47 correspond to the case in which no skeleton exists on the $\neg P$ side of the wall; and lines 48-49 correspond to the typical case in which both sides have skeletons but the wall cannot be crossed.

If the skeleton-minimization version of the skeleton query at line 10 is unsatisfiable, then the full version is tested at line 6; if it is satisfiable, then a new map is constructed at line 8. If, however, the full skeleton query at line 6 is also unsatisfiable, then S does not have a computation (line 52).

H. Correctness

We prove the correctness of the fair algorithm. The first three lemmas formalize the assumption that the safety model checker is correct.

Lemma 1: A proof is returned for query (1) iff s_0 is unreachable from I, and such a proof excludes s_0 and is S-inductive relative to $\mathcal{R}$.

Hence, no subsequent skeleton contains s_0.

Lemma 2: A proof is returned for query (3) iff $s_{i \oplus_n 1}$ is unreachable from s_i, and such a proof separates s_i from $s_{i \oplus_n 1}$ and is S-inductive relative to $\mathcal{R}$ and $\mathcal{C}$, with the exception that it satisfies initiation with respect to s_i rather than I.

Hence, no subsequent skeleton contains both s_i and $s_{i \oplus_n 1}$.

Lemma 3: A proof is returned for query (4) iff s_0 is unreachable from its successors, and such a proof separates the successors of s_0 from s_0 and is S-inductive relative to $\mathcal{R}$ and $\mathcal{C}$, with the exception that it satisfies initiation with respect to the successors of s_0 rather than I.

Combined with either $K > 0$ for the technique of Section III-B or the technique of Section III-C to exclude s_0 from the $\neg P$ side of the wall, no subsequent skeleton contains s_0.

Besides progress criteria, these lemmas together imply that a skeleton can be completed into a reachable fair cycle if and only if all queries return counterexample runs.

Lemma 4: No transition excluded by a constraint $C \in \mathcal{C}$ is on a reachable fair cycle.

This lemma is straightforward once one realizes that each C is derived from (relatively) inductive information. A proof W from a cycle query observes that no path allowed by the current $\mathcal{C}$ that passes from a $\neg W$-state to a W-state can be part of a cycle, as it can never return to a $\neg W$-state. This observation is encoded as $W \leftrightarrow W'$. Additionally, if W-states ($\neg W$-states) cannot satisfy every fairness condition, then no path that has a W-state ($\neg W$-state) can be part of a fair cycle. This observation is encoded as $W \wedge W'$ ($\neg W \wedge \neg W'$). Hence, induction on the list $\mathcal{C}$ proves the lemma.

Another perspective on this lemma is that a cycle query proof W, by its inductiveness, describes regions W and $\neg W$ that are SCC-closed *with respect to S constrained by $\mathcal{C}$*. The resulting constraint C excludes only transitions leaving an SCC-closed region or all transitions of an SCC-closed region that does not intersect some fair condition; hence, no transition of a fair cycle is excluded.

By similar reasoning, one concludes that, in general, any fair cycle must be entirely contained in an arena defined by $\mathcal{W}$-constraints: for each $W \in \mathcal{W}$, the entire cycle must satisfy either W or $\neg W$. Hence we have the following lemma.

Lemma 5: If the skeleton query is unsatisfiable, then S does not have a reachable fair cycle.

Together these lemmas imply correctness of the algorithm.

Theorem 1: The algorithm fair always terminates, and it returns a reachable fair cycle iff the language of S is nonempty.

As suggested in Section III-A, the constraints $\mathcal{C}$ that are used during cycle queries are unnecessary for completeness, although crucial for the algorithm to be effective in practice. Lemma 4 states that these constraints do not destroy soundness. In contrast, all constraints in the skeleton query corresponding to the members of the sets $\mathcal{R}$ and $\mathcal{W}$ are necessary for completeness, as suggested by Lemmas 1-3, which state how the algorithm makes progress. Each new reachability assertion $R \in \mathcal{R}$ eliminates at least one state from being returned henceforth from a skeleton query; and each new wall $W \in \mathcal{W}$ eliminates at least one pair of states (Lemma 2) or one state (Lemma 3) from further consideration.

IV. Related Work

Several fair cycle detection algorithms have been developed for symbolic model checking. In this section we compare the main ones to fair, focusing on two features: the identification of SCC-closed sets and the ability to generalize from facts learned about the model.

SCC decomposition algorithms [11]–[13] recursively divide the states into SCC-closed sets. In that respect, they are the closest to fair. However, the walls that they derive are local to the arenas from which SCCs are extracted. Therefore, if the language of a model is empty, SCC decomposition must break up all reachable arenas until they become trivial or unfair. In contrast, fair produces wall that transect the entire state space; hence, it can prove language emptiness by considering a number of skeletons that is much smaller than the number of nontrivial SCCs.

SCC hull algorithms [14], [15] compute an SCC-closed set that contains all fair SCCs and that is empty if no fair SCC exists. In its simplest form, an SCC hull is defined by one wall. (See [15] for hulls defined by two walls.) One side of the wall is known to contain no fair SCC, and the algorithms move the wall until the SCCs abutting the wall on the other side are all fair. While the wall may be moved across very large numbers of SCCs in one step of the procedure, the restriction to a small, fixed set of walls prevents SCC hull algorithms from learning important facts about the structure of the SCC graph. In addition, SCC hull algorithms converge to a hull before declaring a language nonempty. In contrast, fair is often able to home in on a reachable fair SCC well before the entire state space has been examined. Every skeleton that is examined

focuses the successive skeleton queries on where the fair SCCs may lie.

Among the first algorithms for BDD-based cycle detection is the one of [16] based on the computation of the transitive closure of the state graph by iterative squaring. The approach works well for counters, but unlike fair, it is often impractical because it computes too much information about the model.

In Bounded Model Checking (BMC) [9] cycle detection can be formulated as a SAT query such that a model of an appropriate formula describes a lasso-shaped path of prescribed length in the given finite-state system. Deciding that no lasso-shaped path exists regardless of length requires the computation of appropriate bounds (e.g., [17]). While this approach does not fix a skeleton in advance, failure to find a path of a given length does not directly translate into information about the SCC-closed sets of the model. By separating the choice of the skeleton from the attempt to flesh it out to a cycle, fair incrementally learns inductive lemmas.

The liveness-to-safety conversion of [3] is the most common approach to prove progress with interpolation-based model checking [4], [10]. While safety checking is more developed and arguably better understood than checking for progress properties, the transformation to safety has several drawbacks: first, the model's doubled number of state variables negatively affects some model checkers; second, the nature of the problem—cycle detection—is not obvious to the model checker from the encoding; third, the approach is inherently non-incremental, because it asks the safety model checker for a single, monolithic proof that there is no fair cycle.

In the D'n'C approach [18], SCC decomposition is applied to a sequence of increasingly refined abstractions of a system. If an effective way to choose the abstract models is given, this approach may be profitably combined with fair to initially provide it with simple lemmas about the abstractions. Both methods can leverage the weakness of fairness conditions; fair, however, can sometimes discover weakness even on large structures—even, that is, when weakness is not inherited from the acceptance condition of a small Büchi automaton.

V. Experimental Evaluation

An implementation of fair was evaluated against other cycle detection methods on a set of models. Even though fair is highly parallelizable, the implementation uses only one thread of execution but employs a time sharing scheme, as described in Section III-F.

The implementation of skeleton queries differs from the description of Section III-B: for $K = 0$, one forward and no backward unrolling is used; for $K = 1$, two forward and one backward unrollings are used; and so on. Therefore, it only adds a clause as in Section III-C if it provides additional information.

The skeleton-minimization heuristic of Section III-B is implemented as a search: map construction is guided by intermediate partial skeleton queries based on partial maps. If a partial map corresponds to an unsatisfiable query, the last assignment of an index to a fairness condition is incremented, potentially extending the range of the partial map by one. Of course, if the assignment is already onto $\{1, \ldots, |\mathcal{B}|\}$, then the standard skeleton query is also unsatisfiable, and the proof is complete. Once a map is constructed, it is used until the corresponding skeleton query becomes unsatisfiable, at which point a new map is constructed. A separate full skeleton query is used throughout execution, as described in Section III-G.

The implementation also checks if each proof returned by a cycle query is actually inductive with respect to the system, and if so, the proof is upgraded to a reachability proof. While the benchmarks did not reveal if this check is worthwhile, it is inexpensive. Finally, only IC3 is used to answer safety queries, and its proofs are refined as described in Section III-D.

Unlike the case of safety properties, there are no widely accepted benchmark sets for progress properties. Moreover, models of practical import are difficult to come by. The evaluation therefore relies on models that have been identified in the literature as challenging for certain approaches or that present features that one may find combined in real-life problem instances. The abq, cnt, and jc models were written for this evaluation; the remaining ones were adapted from [19].

Table I reports the results of the experiments, which were run on machines with one 2.67 GHz Intel Core i5 CPU and 8 GB of memory each. CPU times are in seconds. The timeout was set at 7200 s. For each model, the table shows whether the language is empty, the number of latches in the cone of influence of the fairness conditions, the number of 2-input AND gates after combinational simplification, and the number of fairness conditions (with the number of weak conditions in parentheses). Next, the results for fair are shown: in the latter three columns, for $0 \leq K \leq 2$ with the skeleton-minimization heuristic enabled, the CPU time and the number of skeletons examined are reported. If fair timed out (indicated by a dash) the number of skeletons examined up to that point is given. The first column for fair shows similar results for $K = 0$ with the skeleton-minimization heuristic disabled.

The remaining columns show results for other language emptiness algorithms. GSH, LS, and DnC are the SCC hull method of [15], the SCC decomposition method of [12], and the D'n'C algorithm of [18] as implemented in the lang_empty command of VIS 2.3 [20] (run with dynamic variable ordering enabled and default settings except that D'n'C is run without preliminary reachability analysis). These three methods were chosen for inclusion in the table because they represent well the gamut of BDD-based algorithms and because GSH and D'n'C without reachability performed better than the others that were tried.

Finally, the group of columns under LTS refers to the liveness-to-safety approach of [3], with reachability checked with interpolation as implemented in ABC [4] (ITP), with IC3, and with ABC. For ITP, the parameters controlling ABC were set to disable its IC3 implementation and to reduce the chance of inconclusive runs. A question mark in the ITP column signals that ABC nevertheless reported the problem as "undecided" before its time was up. For ABC, the parameter controlling its use of its IC3 implementation was set to allow

TABLE I
EXPERIMENTS

model	empty	latches	gates	$\lvert\mathcal{B}\rvert$	fair $K=0^*$	fair $K=0$	fair $K=1$	fair $K=2$	BDD-based GSH	BDD-based LS	BDD-based DnC	LTS ITP	LTS IC3	LTS ABC
abq2mf	yes	35	383	4(1)	1/24	1/12	1/8	1/5	1	1	1	–	2	8
abq4mf	yes	67	745	6(1)	3/37	3/39	2/8	3/9	3	–	7	–	11	40
abq8mf	yes	131	1469	10(1)	23/182	168/67	16/14	21/14	2794	–	–	–	373	157
abq2f	yes	61	747	4(1)	3/30	3/55	2/9	4/3	4	10	1	–	10	20
abq4f	yes	119	1471	6(1)	423/221	31/106	13/28	34/46	2890	–	213	–	388	–
abq8f	yes	235	2923	10(1)	–/75	–/116	5730/84	4384/65	–	–	–	–	6330	–
cnt12	yes	12	68	1(1)	1/0	1/0	1/0	1/0	1	1	0	1	1761	1
cnt32	yes	32	188	1(1)	1/0	1/0	1/0	1/0	–	–	–	?	–	–
cnt128	yes	128	764	1(1)	1/0	1/0	1/0	1/0	–	–	–	?	–	–
jc12	yes	13	231	1(1)	1/0	1/0	1/0	1/0	1	1	0	9	93	9
jc32	yes	33	631	1(1)	1/0	1/0	1/0	1/0	–	–	–	16	–	–
jc128	yes	129	2551	1(1)	2/0	2/0	2/0	3/0	–	–	–	805	–	–
jc128f	no	129	2170	1(1)	2/1	2/1	2/1	2/1	2	2	2	1	1	1
om1	yes	29	810	16(16)	–/99	–/202	–/244	–/274	272	–	356	–	–	–
om2	yes	29	806	16(16)	42/2082	39/2077	42/2083	45/2071	192	–	8	–	236	–
om3	yes	29	803	16(16)	1/0	1/0	2/0	5/0	35	–	25	–	105	–
nim1	yes	27	769	2(2)	1/29	1/32	1/0	1/0	1	174	1	–	20	117
nim2	yes	29	788	2(2)	1309/28	1264/28	1157/18	1457/18	2	120	1	–	1192	177
nim3	no	29	788	2(2)	1/32	1/28	1/3	1/3	1	309	1	1	1	1
gbak	yes	37	677	10(1)	25/182	12/172	74/184	26/125	3	7	14	–	97	90
tarb16	yes	79	1109	17(1)	18/166	15/101	17/72	70/79	–	–	–	60	58	31
tarb32	yes	159	2269	33(1)	146/582	75/204	214/146	956/147	–	–	–	?	–	209
sarb16	yes	50	141	1(1)	1/0	1/0	1/0	1/0	1	1	3	?	5	7
sarb32	yes	98	269	1(1)	1/0	1/0	1/0	1/0	3	1	–	?	157	79
tf1	yes	30	452	2(1)	1949/5174	393/2222	285/1278	288/1213	8	–	2	–	–	281
tf2	no	30	384	2(1)	1/9	1/5	1/2	1/2	2	60	1	1	1	1
tq1	yes	55	756	3(1)	2267/3072	3143/4208	2690/2775	5434/3172	1645	–	3	–	–	737
tq2	no	60	771	4(2)	5/27	4/28	4/22	5/26	3056	–	5	2	27	2
fq1	yes	105	1365	5(1)	–/2920	–/2596	–/2485	–/1771	–	–	336	–	–	–
fq2	no	120	1546	8(4)	21/41	15/39	25/55	29/44	–	–	–	–	374	30

it to run through the two-hour time limit.

The abq models are interconnected queues with bounded sources. The cnt models are counters and the jc models are the "forward jumping counters" of [3]. The om models are used in [15] to prove lower bounds on SCC hull algorithms. The nim models are NIM players. The gbak model is a finite-state version of the bakery protocol. The tarb models are tree arbiters, while the sarb models are McMillan synchronous arbiters. The tf, tq, and fq models are versions of the two-queue example in [21].

The cnt models illustrate fair's ability to find linear-size proofs for counters as discussed in Section III-E. This ability accounts for the good performance of fair on models like the om and nim (NIM player) sets—in which the original state graph has many SCCs—or like the jc and tarb (tree arbiter) sets, in which the composition with a Büchi automaton breaks the single SCC of the model into a myriad of SCCs. While computing the transitive closure would be effective for counters, it would not work on more complex examples.

The om set contains three models that differ only in the transitions out of unreachable states. For om3, fair quickly produces an inductive proof that there are no fair SCCs; for the other two models, however, it has to prove, at a much higher cost, that such fair SCCs are unreachable. Combining fair with a global reachability engine, perhaps based on BDDs, would benefit the analysis for om1 and om2, but was outside the scope of this evaluation. Yet not relying on full reachability analysis is partly responsible for fair's speed in detecting nonemptiness for tq2 and fq2.

For all four configurations, fair decided either 27 or 28 of the 30 language emptiness problems and was the only model checker to solve two of the problems. Behind it, each of GSH, DnC, and LTS/IC3 solved 21 problems, and LTS/ABC solved 20 problems. Together the BDD methods solved 22 problems, and the LTS methods solved 26 problems. On 11 models, one of the fair configurations, typically $K = 1$, was decidedly faster than the other methods; on 8 models, one of the other six methods was decidedly faster. Overall, fair was the clear winner on this set of models.

It is worth noting that the two models that fair failed to prove—om1 and fq1—were solved by BDD methods but not by LTS methods. Furthermore, fair generally dominated the LTS methods, with the exception of nim2 and tq1, both of which were, in any case, trivial for at least one BDD method. In short, fair seems to complement BDD methods and to dominate LTS methods.

As expected, LS suffered on models with many SCCs, while LTS/ITP had rather unpredictable performance. For many models the number of skeletons examined by fair decreased with increasing K, with the largest jump usually occurring between $K = 0$ and $K = 1$; however, on tarb16 and tarb32, which have many fairness conditions and thus require large skeleton queries, fair suffered as K increased.

The skeleton-minimization heuristic of Section III-B is

effective at finding small skeletons. For $K = 0$, six of the models on which the analysis finished required examining skeletons that have more than one state: nim1 (≤ 2), gbak (always 3, as there are three disjoint fairness conditions), tarb32 (≤ 2), tq1 (≤ 2), tq2 (≤ 2), fq2 (≤ 4). Furthermore, it typically resulted in fewer skeletons, as hypothesized; tarb32 and tf1 are extreme cases.

These illustrative benchmarks indicate the potential of the fair algorithm. However, only practical experience with a suite of industrial benchmarks will reveal the best use of skeleton-minimization, a method for choosing K dynamically, and a heuristic for choosing when to enrich the $\mathcal{C}$ constraint set.

VI. Conclusion

We have presented a new incremental algorithm for model checking progress properties that selects skeletons for fair cycles and, if it fails to flesh them out, learns inductive lemmas that divide the states into SCC-closed sets. An initial implementation shows promise, especially when one considers that one of the strengths of the proposed approach—that of being highly parallelizable—was not brought into play. Another important aspect that awaits exploration is the integration of the new approach into a multi-engine framework, which has been shown to be key to robust performance in the case of safety properties.

Acknowledgments. The first author thanks Barbara Jobstmann for a collaboration that inspired this work while the two were post-docs in Tom Henzinger's group at EPFL. Thanks also to the reviewers for their specific questions and suggestions, which aided us in improving the presentation. This material is based upon work supported in part by the National Science Foundation under grant No. 0952617 and by the Semiconductor Research Corporation under contract GRC 1859. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

[1] A. R. Bradley, "SAT-based model checking without unrolling," in *Verification, Model Checking, and Abstract Interpretation (VMCAI'11)*, Austin, TX, 2011, pp. 70–87, lNCS 6538.

[2] Z. Manna and A. Pnueli, *Temporal Verification of Reactive Systems: Safety*. Springer-Verlag, 1995.

[3] V. Schuppan and A. Biere, "Efficient reduction of finite state model checking to reachability analysis," *Software Tools for Technology Transfer*, vol. 5, no. 2–3, pp. 185–204, Mar. 2004.

[4] R. K. Brayton and A. Mishchenko, "ABC: An academic industrial-strength verification tool," in *Twentysecond Conference on Computer Aided Verification (CAV"10)*. Edinburgh, UK: Springer, 2010, pp. 24–40, lNCS 6174.

[5] D. E. Muller, A. Saoudi, and P. E. Schupp, "Weak alternating automata give a simple explanation of why most temporal and dynamic logics are decidable in exponential time," in *Proceedings of the 3rd IEEE Symposium on Logic in Computer Science*, Edinburgh, UK, Jul. 1988, pp. 422–427.

[6] R. Bloem, K. Ravi, and F. Somenzi, "Efficient decision procedures for model checking of linear time logic properties," in *Eleventh Conference on Computer Aided Verification (CAV'99)*, N. Halbwachs and D. Peled, Eds. Berlin: Springer-Verlag, 1999, pp. 222–235, lNCS 1633.

[7] A. R. Bradley and Z. Manna, "Checking safety by inductive generalization of counterexamples to induction," in *Formal Methods in Computer Aided Design (FMCAD'07)*, Austin, TX, 2007, pp. 173–180.

[8] 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.

[9] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu, "Symbolic model checking without BDDs," in *Fifth International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS'99)*, Amsterdam, The Netherlands, Mar. 1999, pp. 193–207, lNCS 1579.

[10] K. L. McMillan, "Interpolation and SAT-based model checking," in *Fifteenth Conference on Computer Aided Verification (CAV'03)*, W. A. Hunt, Jr. and F. Somenzi, Eds. Berlin: Springer-Verlag, Jul. 2003, pp. 1–13, lNCS 2725.

[11] A. Xie and P. A. Beerel, "Implicit enumeration of strongly connected components and an application to formal verification," *IEEE Transactions on Computer-Aided Design*, vol. 19, no. 10, pp. 1225–1230, Oct. 2000.

[12] R. Bloem, H. N. Gabow, and F. Somenzi, "An algorithm for strongly connected component analysis in $n \log n$ symbolic steps," *Formal Methods in System Design*, vol. 28, no. 1, pp. 37–56, Jan. 2006.

[13] R. Gentilini, C. Piazza, and A. Policriti, "Symbolic graphs: Linear solutions to connectivity related problems," *Algorithmica*, vol. 50, no. 1, pp. 120–158, 2008.

[14] E. A. Emerson and C.-L. Lei, "Efficient model checking in fragments of the propositional mu-calculus," in *Proceedings of the First Annual Symposium of Logic in Computer Science*, Jun. 1986, pp. 267–278.

[15] F. Somenzi, K. Ravi, and R. Bloem, "Analysis of symbolic SCC hull algorithms," in *Formal Methods in Computer Aided Design*, M. D. Aagaard and J. W. O'Leary, Eds. Springer-Verlag, Nov. 2002, pp. 88–105, lNCS 2517.

[16] H. J. Touati, R. K. Brayton, and R. P. Kurshan, "Testing language containment for ω-automata using BDD's," *Information and Computation*, vol. 118, no. 1, pp. 101–109, Apr. 1995.

[17] M. Awedh and F. Somenzi, "Termination criteria for bounded model checking: Extensions and comparison," *Electronic Notes in Theoretical Computer Science*, vol. 144, no. 1, pp. 51–66, 2006, presented at the Third International Workshop on Bounded Model Checking (BMC'05).

[18] C. Wang, R. Bloem, G. D. Hachtel, K. Ravi, and F. Somenzi, "Divide and compose: SCC refinement for language emptiness," in *International Conference on Concurrency Theory (CONCUR01)*. Berlin: Springer-Verlag, Aug. 2001, pp. 456–471, lNCS 2154.

[19] "VIS verification benchmarks. http://vlsi.colorado.edu/∼vis," University of Colorado at Boulder.

[20] "URL: http://vlsi.colorado.edu/∼vis."

[21] H. Jin, K. Ravi, and F. Somenzi, "Fate and free will in error traces," *Software Tools for Technology Transfer*, vol. 6, no. 2, pp. 102–116, Aug. 2004.

Hardware Model Checking: Status, Challenges, and Opportunities

Muralidhar Talupur
Strategic CAD Labs, Intel
murali.talupur@intel.com

It's been nearly 30 years since Model Checking was first proposed as a method to validate hardware designs. Since then there has been considerable progress in this area and an active research community has grown around it. While areas like Software Model Checking and Hybrid Systems Model Checking are thriving, it seems there hasn't been much progress or activity in the area of Hardware Model Checking (HMC) itself lately. If we examine the proceedings of conferences like CAV, TACAS, or even FMCAD the number of submissions dealing with hardware model checking has gone down drastically compared to 10-15 yrs ago. But verification of hardware designs continues to be a pressing problem – if anything it is only getting exacerbated with the continuing push towards larger designs and shorter turn-around times. This panel will examine the current status of HMC, what further challenges and opportunities exist going forward and what can be done to give a fresh impetus to the area. The panel will broadly have three sections 1) Taking stock of the area, 2) Challenges to wider industrial adoption, and 3) How to renew interest in HMC? Each section will have multiple talking points (as described below) which will serve as guides for the panelists in taking their positions.

A. *Taking stock of the area*

Recent advances in HMC seem to have come from "engineering" mostly (ignoring the newly proposed IC3 algorithm which is an exception to the general trend). In this section the panelists's will give their view of the current status of research in HMC.

1) *Talking points:*

- The problem is fundamentally hard and the pace of recent activity indicates we have reached a plateau.
- There has been significant advance in the last five years that has impacted practitioners.
- The current status of HMC is satisfactory.

B. *Challenges to industrial adoption*

The ultimate success of an applied field like HMC should be measured by its impact on the ground, which means adoption by the hardware industry. While there has been some adoption and Model Checking gets mentioned as a respectable validation option, simulation continues to be the main work horse for validation. This section will examine the various factors that might be preventing wider adoption of Model Checking and what can be done to alleviate them.

1) *Talking points:*

- The capacity of model checkers is still far short of what is required. Or perhaps capacity is not the bottleneck.
- Writing specifications and input constraints are the major barriers for adoption of model checking.
- Model Checking will always be a specialist activity.
- All or nothing nature of Model Checking preventing adoption.
- Model checking will replace simulation in the near future, say, for 50 percent of the units in the next 5 years.
- What problem should a starting researcher consider?

C. *How to renew interest in HMC?*

This section will consider what the causes are for the reduced research in this area of late. Given that hardware validation continues to be a pressing problem and will continue to be so in this era of SoCs and multi-cores, how do we renew the interest in this area? Are there other problems we should be looking at?

1) *Talking points:*

- Industry is not interested in funding research in this area anymore because of poor ROI.
- Lack of open source RTL infra-structure making entry barrier too high.
- There are low hanging fruits in the form of related problems like *Clock Domain Crossing*.

D. *Panelists*

The panelists are academic and industrial experts on Model Checking with several decades of combined experience. Pranav Ashar led NEC Lab's Formal Methods team for fourteen years before embarking on a startup career. He is currently the CTO of RealIntent. Jason Baumgartner has fourteen years of experience in the research, development and application of Model Checkers and Equivalence Checkers for large-scale industrial designs, culminating in the SixthSense project at IBM. Robert Brayton and his group at UC Berkeley have built the ABC verification and synthesis engine that has consistently placed among the top Hardware Model Checkers. Erik Seligman is an FV expert at Intel with a wide experience applying Model Checking and other Formal Methods to industrial designs.

Realtime Regular Expressions for Analog and Mixed-Signal Assertions

John Havlicek
Austin, TX 78728, USA
Email: jwhavlicek@yahoo.com

Scott Little
Freescale Semiconductor
Austin, TX 78729, USA
Email: scott.little@freescale.com

***Abstract*—Syntax and semantics are proposed for realtime (i.e., continuous-time) regular expressions, which extend and generalize existing SVA regular expressions. The extensions are motivated by practical needs for AMS circuit verification and were developed as part of the authors' contribution to analog assertions work in the Accellera committee standardizing Verilog-AMS. Given a suitable notion of sampling, we prove that the realtime semantics provided for the existing SVA clocked digital regular expressions is equivalent to the original discrete semantics. As a result, the existing digital operators can intermix freely with the new realtime operators, which is a major contribution of our framework. We also investigate the theoretical relationship between our framework and the timed regular expressions of Asarin, Caspi, and Maler. We provide a semantically faithful embedding of timed regular expressions into our realtime regular expressions, as well as a construction of timed automata recognizers for our realtime regular expressions. These constructions show that our realtime regular expressions are no less expressive than the timed regular expressions of [1] and no more expressive than the generalized timed regular expressions of [2]. The automata recognizers also provide the basis for an implementation strategy for our framework.**

I. INTRODUCTION

Over a number of years, assertion-based techniques have been growing in importance as part of functional verification methodologies for industrial semiconductor designs. This growth is evidenced in part by the standardization of industrially focused assertion languages, like SystemVerilog Assertions (SVA) [3] and Property Specification Language (PSL) [4]. SVA and the Foundation Language of PSL are both discrete-time temporal logics, based upon Linear Temporal Logic (LTL) [5] augmented with regular expressions. The reckoning of time in SVA and in clocked formulas of PSL is in terms of discretely occurring events.[1] The use of events to define units of discrete time works well for the majority of applications to digital circuit verification, although complex timing properties can be challenging to write using event-based assertions [6]. Applications to analog/mixed-signal (AMS) circuits require specification of relationships between events and event-based patterns, but also often involve direct timing requirements. For example, the notion of settling time is common in AMS circuits. Settling time is defined as the amount of time required for a signal to stabilize after a specific event. A property to check the settling time of a digital-to-analog converter (DAC) might ensure that the circuit's output has settled for an input pattern of all zeros, then change the input pattern to all ones and verify that the output settles to its new expected value within a specified time. There are many other examples of AMS properties involving direct timing requirements in the literature [7], [8], [9], [10]. For the expression of many AMS properties, a first class notion of time is needed in the assertion language.

Previous work [11], [12] provides a clear roadmap for extending the LTL features of SVA for AMS applications. In this paper, we define realtime extensions to SVA regular expressions.[2] Our extensions are motivated by practical needs for AMS circuit verification and were developed as part of the authors' contribution to the work of the Analog Assertions Subgroup of the Accellera Verilog-AMS Committee [13]. Many AMS properties rely either on continuously varying quantities whose changes are not confined to clock boundaries or on time constraints whose starting and ending points are not clock aligned. Current event-based assertion languages do not facilitate writing these properties carefully and succinctly. The realtime regular expressions presented in this paper take an important step toward enabling accurate verification of AMS properties. Because of its close alignment with SVA, PSL can be extended using the same approach.

Our semantic framework is based on bounded intervals of the real line. The semantics of realtime regular expressions is defined by a matching relation that specifies when a realtime regular expression matches over a bounded realtime interval of a realtime trace. The interval can be empty, open, closed, or half-open. A fundamental characteristic of our definition is that it is indeed an extension of the current SVA regular expressions. For simplicity of exposition, we omit local variables and the `first_match` operator from SVA. Our definition includes realtime semantics for the existing clocked digital regular expressions, and we prove that the new semantics is equivalent, through a suitable notion of sampling, to the original discrete semantics. In this way, the existing digital operators and forms intermix and combine freely with the new realtime operators and forms. The enablement of harmonious

See http://www.async.ece.utah.edu/~little/pubs/realtimeAppendix.pdf for supplementary material and proofs.

[1] An unclocked formula of PSL is also interpreted over discrete time, but the granularity of time is not specified in the formula.

[2] We would like to acknowledge Himyanshu Anand for his insightful comments during the development of the realtime regular expressions and their semantics.

interplay between discrete and realtime elements of the regular expressions is a major contribution of our work; we had to explore several variations on the semantic framework before we discovered one that achieved this goal.

Our definition adds only one basic realtime form (immediate Boolean) and one primitive realtime operator (Boolean smear) to the existing digital regular expressions.[3] We believe that the preponderance of realtime regular expressions of practical interest can be written using our extension. As in the discrete case, the realtime regular expressions are augmented with a number of useful derived operators. These include flexible concatenation, concatenation with realtime delay, and the realtime goto operator.

We provide a semantically faithful mapping from the timed regular expressions of [1] into our realtime regular expressions, which shows that our formulation is no less expressive. We also give a construction of timed automata recognizers for our realtime regular expressions, which shows that they are no more expressive than the generalized timed regular expressions of [2].

The rest of the paper is organized as follows. In Section 2 we introduce some preliminaries and the notation that will be used throughout the paper. In Section 3 we review digital sequences and their discrete-time semantics. We also define realtime semantics for the digital sequences and prove that the realtime semantics for digital sequences is a faithful generalization to realtime of the discrete-time semantics. In Section 4 we define realtime sequences, illustrate their use in some practical examples, and provide the semantically faithful mapping from the timed regular expressions of [1] into our realtime regular expressions. Section 5 describes the construction of timed automata recognizers for our realtime regular expressions. Section 6 discusses relationships with timed regular expressions, and the paper concludes with a brief discussion of future directions.

II. Preliminaries and Notation

As in SVA, we use the term *sequence* as a synonym for regular expression.

$\mathbb{R}$ denotes the set of real numbers, $\mathbb{R}_{\geq 0}$ denotes the set of non-negative real numbers, $\mathbb{B}$ denotes the set $\{0,1\}$ of Boolean values, and $\mathbb{N}$ denotes the set of non-negative integers. Let A and D be finite sets. A will be understood as the set of *analog variables*, and D will be understood as the set of *discrete variables*. A *state* (*of the variables*) is an assignment of an element of $\mathbb{R}$ to each analog variable and an element of $\mathbb{B}$ to each discrete variable. A state may be identified with an element of the set $\Sigma = \mathbb{R}^A \times \mathbb{B}^D$.

A *Boolean expression* (*over the variables*) assigns to each state of the variables an element in $\{0,1\}$. A Boolean expression may be identified with an element of the Boolean algebra $\mathbb{B}^\Sigma = \mathbb{B}^{\mathbb{R}^A \times \mathbb{B}^D}$. We may think of $\mathbb{B}^\Sigma$ as the set of functions $\Sigma \to \mathbb{B}$ in the usual way, so that if b is a Boolean expression and s is a state, then $b(s) \in \mathbb{B}$. We write $s \models b$ iff $b(s) = 1$. In this case, b is said to *occur* at s. An *event* is a Boolean expression from a designated class.[4] Events are denoted by κ and ζ in the remainder of this paper. In realtime, we require events to occur only at isolated points (see below).

A *discrete trace*, or *word*, is a function $w : \{i \in \mathbb{N} : i \leq n-1\} \to \Sigma$, where $0 \leq n \leq \infty$. n is said to be the *length* of the word, which is also denoted $|w|$. The empty word has length 0 and is denoted ε. Throughout, u, v, and w are used to denote words. The concatenation of u and v is denoted by uv. For $i < |w|$, we use w^i to denote $w(i)$, the $(i+1)^{\text{st}}$ letter of w, and we denote by $w^{i..}$ the suffix of w starting at index i. We denote by $w^{i..j}$ the finite sequence of letters starting from index i and ending in index j. That is, $w^{i..j} = (w^i w^{i+1} \cdots w^j)$. A Boolean expression b is said to *occur* in w at i iff $w^i \models b$. A *sampling* is a strictly increasing function $T : \mathbb{N} \to \mathbb{R}_{\geq 0}$ such that $\lim_{n\to\infty} T(n) = \infty$.

A *realtime trace* is a function $W : \mathbb{R}_{\geq 0} \to \Sigma$. Given a realtime trace W and a sampling T, $W \circ T$ is a discrete trace, where $\circ$ denotes the composition of functions. Given a Boolean expression b and a realtime trace W, we say that b *occurs* in W at t iff $W(t) \models b$. The set $\{t \in \mathbb{R}_{\geq 0} : W(t) \models b\}$ is the set of times at which b occurs in the trace W. If κ is an event, then we require that $\{t \in \mathbb{R}_{\geq 0} : W(t) \models \kappa\}$ have no limit point in $\mathbb{R}$. As a result, the points at which a given event occurs cannot be arbitrarily close together.

Throughout, I,J denote *bounded* intervals in the real line $\mathbb{R}$. They may be open, closed, or half-open.

Definition 1:
(a) $I \leq I'$ *iff* $\forall t \in I\ \forall t' \in I' : t \leq t'$.
(b) $I < I'$ *iff* $\forall t \in I\ \forall t' \in I' : t < t'$.

If I is non-empty, then we write $|I| = \sup I - \inf I$. We write $|\{\}| = 0$.

III. Digital Sequences

Digital sequences are the discrete regular expressions used in this paper. They are generated by the following grammar, where κ denotes an event and b denotes a Boolean expression:

$$\sigma ::= \texttt{@(}\kappa\texttt{)}\ (b) \mid \sigma\ \texttt{\#\#1}\ \sigma \mid \sigma\ \texttt{\#\#0}\ \sigma \mid \sigma\ \texttt{or}\ \sigma \\ \mid \sigma\ \texttt{intersect}\ \sigma \mid \sigma\texttt{[*0]} \mid \sigma\texttt{[+]}$$

Intuitively, `@(`κ`)` (b) specifies that the Boolean expression b occur at the nearest point where event κ occurs; `##1` and `##0` are the non-overlapping and overlapping concatenation operators, respectively; `or` is the union operator; `intersect` is the intersection operator; `[*0]` specifies zero repetitions (i.e., the empty word); and `[+]` specifies one or more repetitions.

Digital sequences mimic SVA syntax and are essentially the same as SVA sequences and PSL SEREs.[5] There are some differences regarding where events can be written. In a digital sequence, the event κ can be attached only to a Boolean expression, as in the form `@(`κ`)` (b). This restriction

[3]If an application or implementation restricts Boolean manipulation of events, then a second realtime operator (sequence without an event) may be considered primitive rather than derived.

[4]Events may be treated differently than other Boolean expressions in certain tool flows and verification applications, such as digital and analog simulation.

[5]*SERE* stands for *semi-extended regular expression* and is the regular expression sublanguage of PSL [4].

simplifies reasoning about and defining the semantics of digital sequences. SVA and PSL are less restrictive: they allow an event to be specified in more general positions and provide rules to determine the scope of an event. Let's say that a sequence is *basic* if it does not employ any of the following constructs: local variables, `first_match` (SVA only), or endpoint query methods (`triggered` and `matched` in SVA; `ended` in PSL). Any SVA sequence or PSL SERE that is basic can be rewritten as an equivalent digital sequence by eliminating derived operators ([3], Annex F.3.4; [4], Annex B.4); elaborating instances (cf. [3], Annex F.4.1); and eliminating reliance on the scoping rules for events (cf. [3], Annex F.5.1; [4], Annex B.5). For example, the SVA sequence $\texttt{@(}\kappa\texttt{)}\ x\ \texttt{\#\#0}\ y\texttt{[+]}\ \texttt{\#\#1}\ \texttt{@(}\zeta\texttt{)}\ z$ is equivalent to the digital sequence $\texttt{@(}\kappa\texttt{)(}x\texttt{)}\ \texttt{\#\#0}\ \texttt{(@(}\kappa\texttt{)(}y\texttt{))[+]}\ \texttt{\#\#1}\ \texttt{@(}\zeta\texttt{)(}z\texttt{)}$.

In a different sense, digital sequences are less restrictive than SVA sequences. SVA allows multiply clocked sequences to be joined only with `##1` or `##0`, while digital sequences can be combined freely, without regard to how events appear within them. For example, the digital sequence $\texttt{(@(}\kappa\texttt{)(}x\texttt{)}\ \texttt{or}\ \texttt{@(}\zeta\texttt{)(}y\texttt{))[+]}$ is not a legal sequence in SVA.[6] Every digital sequence can be regarded as a PSL SERE by straightforward syntactic translation.

A. Discrete-Time Semantics

Let $w = w^0 w^1 \cdots w^{|w|-1}$ be a finite word. The discrete-time semantics of a digital sequence σ is defined by the matching relation $\models_d$, which is given recursively as follows:

- $w \models_d \texttt{@(}\kappa\texttt{)}\ \texttt{(}b\texttt{)}$ iff $|w| > 0$ and b and κ occur at $w^{|w|-1}$ and κ does not occur at any earlier position of w.
- $w \models_d \sigma\ \texttt{\#\#1}\ \sigma'$ iff there exist u, u' such that $uu' = w$ and $u \models_d \sigma$ and $u' \models_d \sigma'$.
- $w \models_d \sigma\ \texttt{\#\#0}\ \sigma'$ iff there exist u, v, u' such that $uvu' = w$ and $|v| = 1$ and $uv \models_d \sigma$ and $vu' \models_d \sigma'$.
- $w \models_d \sigma\ \texttt{or}\ \sigma'$ iff either $w \models_d \sigma$ or $w \models_d \sigma'$.
- $w \models_d \sigma\ \texttt{intersect}\ \sigma'$ iff both $w \models_d \sigma$ and $w \models_d \sigma'$.
- $w \models_d \sigma\texttt{[*0]}$ iff w is empty.
- $w \models_d \sigma\texttt{[+]}$ iff there exist $n \geq 1$ and $u_1, \ldots, u_n$ such that $w = u_1 \cdots u_n$ and $u_i \models_d \sigma$ for all $1 \leq i \leq n$.

B. Realtime Semantics

This section defines our interval-based realtime semantics for digital sequences and presents a correspondence theorem between the realtime and the discrete-time semantics. Let W be a realtime trace and I be a bounded interval. The realtime semantics of digital sequence σ is defined by the relation $\models_r$, given recursively as follows:

- $W, I \models_r \texttt{@(}\kappa\texttt{)}\ \texttt{(}b\texttt{)}$ iff $\{t \in I \colon W(t) \models \kappa\} = \{\sup I\}$ and $W(\sup I) \models b$.
- $W, I \models_r \sigma\ \texttt{\#\#1}\ \sigma'$ iff there exist J, J' such that $I = J \cup J'$ and $J < J'$ and $W, J \models_r \sigma$ and $W, J' \models_r \sigma'$.
- $W, I \models_r \sigma\ \texttt{\#\#0}\ \sigma'$ iff there exist J, t, J' such that $I = J \cup J'$ and $\{t\} = J \cap J'$ and $J \leq \{t\}$ and $\{t\} \leq J'$ and $W, J \models_r \sigma$ and $W, J' \models_r \sigma'$.
- $W, I \models_r \sigma\ \texttt{or}\ \sigma'$ iff either $W, I \models_r \sigma$ or $W, I \models_r \sigma'$.
- $W, I \models_r \sigma\ \texttt{intersect}\ \sigma'$ iff both $W, I \models_r \sigma$ and $W, I \models_r \sigma'$.
- $W, I \models_r \sigma\texttt{[*0]}$ iff I is empty.
- $W, I \models_r \sigma\texttt{[+]}$ iff there exist $n \geq 1$ and $J_1, \ldots, J_n$ such that $J_i < J_j$ for all $1 \leq i < j \leq n$ and $I = J_1 \cup \cdots \cup J_n$ and $W, J_i \models_r \sigma$ for all $1 \leq i \leq n$.

If σ is a digital sequence and if $\texttt{@(}\kappa\texttt{)(}b\texttt{)}$ appears as a subsequence of σ, then we say that *κ is an event of σ*. The following theorem establishes the correspondence between the discrete-time and realtime semantics for digital sequences.

Theorem 1: *Let σ be a digital sequence, let W be a realtime trace, and let $T \colon \mathbb{N} \to \mathbb{R}_{\geq 0}$ be a sampling such that $T(\mathbb{N})$ contains all points of $\mathbb{R}$ at which any event of σ occurs in W. Let $w = W \circ T$. Let I be a bounded interval. If I is empty, then let v be the empty word. Otherwise, assume that I is right-closed with $\sup I \in T(\mathbb{N})$ and let $v = w^{i..j}$, where $i = \min T^{-1}(I)$ and $j = \max T^{-1}(I)$. Then $W, I \models_r \sigma$ iff $v \models_d \sigma$.* □

According to Theorem 1, the realtime semantics for digital sequences is a faithful generalization to realtime of the discrete-time semantics. The proof is by induction and makes use of the following

Lemma 2: *Let σ be a digital sequence. If $W, I \models_r \sigma$ and I is non-empty, then I is right-closed and at least one of the events of σ occurs in W at the right endpoint of I.* □

IV. Realtime Sequences

This section generalizes digital sequences by adding one new basic form and one new primitive operator. The new constructs are motivated by assertion-based applications to AMS verification. Realtime sequences are generated by the following grammar, where κ denotes an event, b denotes a Boolean expression, α denotes a non-negative rational constant, and β denotes either a non-negative rational constant or the special symbol `$`, representing ∞:

$$\begin{aligned} R ::= &\ \texttt{@(}\kappa\texttt{)}\ \texttt{(}b\texttt{)} \mid R\ \texttt{\#\#1}\ R \mid R\ \texttt{\#\#0}\ R \mid R\ \texttt{or}\ R \\ &\mid R\ \texttt{intersect}\ R \mid R\texttt{[*0]} \mid R\texttt{[+]} \\ &\mid b \mid b\texttt{[*}\alpha\texttt{[+]:}\beta\texttt{[-]]} \end{aligned}$$

The realtime semantics of the digital sequence forms and operators remains as before, while the semantics of the new constructs is given as follows:

- $W, I \models_r b$ iff there exists t such that $I = \{t\}$ and $W(t) \models b$.
- $W, I \models_r b\texttt{[*}\alpha\texttt{[+]:}\beta\texttt{[-]]}$ iff $\alpha \leq [<]\ |I| \leq [<]\ \beta$ and $W(t) \models b$ for all $t \in I$.

A. Derived Realtime Forms

Assertion languages for industrial use typically provide numerous derived forms, with the goal of improving the time efficiency of the engineers deploying them. Both SVA and PSL have many derived sequence forms, all of which can be thought of as extending the present digital and realtime sequences, with syntax adapted as necessary. For example,

[6] An equivalent non-basic SVA sequence can be written using method `triggered` [14].

R`[*]` $\equiv$ R`[*0] or` R`[+]` is the usual Kleene operator. There are also several derived realtime sequence forms that are useful for AMS assertion applications. These are defined as follows, where `!` denotes Boolean negation and the other notational conventions are as before:

- b`[*`α`]` $\equiv$ b`[*`$\alpha : \alpha$`]` [exact-length smear].
- b`[~>1]` $\equiv$ `!`b`[*0.0:$] ##1` b [realtime goto].
- R `without @(`κ`)` $\equiv$ R `intersect !`κ`[*0.0:$]` [sequence without an event].
- R `#0` R' $\equiv$ `(`R `##0` R'`) or (`R `##1` R'`)` [flexible concatenation].[7]
- R `#[`α`[+]:`β`[-]]` R' $\equiv$ R `#0 1[*`α`[+]:`β`[-]] #0` R' [concatenation with realtime delay].
- R `#[`α`]` R' $\equiv$ R `#[`$\alpha : \alpha$`]` R' [concatenation with exact-length delay].
- R`[*]` $\equiv$ R`[*0] or` R`[+]` [repetition]
- R `and` R' $\equiv$ `((`R `#0 1[*0.0:$]) intersect` R'`) or (`R `intersect (`R' `#0 1[*0.0:$]))` [flexible intersection].

In SVA and PSL, the discrete-time goto b`[->1]` is governed by an event and only checks the Boolean condition at occurrences of that event. It can be derived according to `@(`κ`)(`b`[->1])` $\equiv$ `@(`κ`)(!`b`)[*] ##1 @(`κ`)(`b`)`. The realtime goto checks the Boolean condition continuously and advances to the nearest point in time at which the condition is true. Its direct semantics is $W, I \models_r$ b`[~>1]` iff $\{t \in I : W(t) \models b\} = \{\sup I\}$.

The flexible concatenation is an important realtime operator. Its direct semantics is the following: $W, I \models_r R$ `#0` R' iff there exist J, J' such that $I = J \cup J'$, $J \leq J'$, $W, J \models_r R$, and $W, J' \models_r R'$. The intervals J and J' being joined must leave no gap and can overlap at most in a shared point. This capability is often needed because the intervals over which realtime sequences match can be open, closed, or half-open.

Consider, for example, `@(`κ`)(`b`) ##1` R. `@(`κ`)(`b`)` matches only a right-closed interval, and `##1` requires the interval over which R matches to abut but not overlap with this interval. If $R = b'$, then the overall sequence cannot match since the realtime Boolean b' matches only over a single point. This incompatibility can be avoided with the flexible concatenation: `@(`κ`)(`b`) #0` R.

Flexibility in matching is important to our semantics because it allows the user to be careful about including or excluding endpoints when needed and not to worry about accounting for endpoints when it is not important. The semantics for `##0` requires that it join a right-closed with a left-closed interval, while `##1` joins a right-closed (resp., -open) interval with a left-open (resp., -closed) interval. Digital sequences and smear-free realtime sequences match only over empty and right-closed intervals. The smear operator introduces the possibility of matching right-open intervals, but whether a right-open interval is actually matched depends on the trace. This flexibility is built into the smear operator, similar to the flexible concatenation operator.

[7]We are grateful to Dejan Nickovic for pointing out that flexible concatenation can be derived in this way.

B. Realtime Sequence Examples

To illustrate the use of realtime sequences we discuss two representative examples. The first illustrates the utility of intermingling digital and realtime sequences. The second illustrates the inadequacy of discrete approximations for realtime specifications.

Let's examine how the settling time specification mentioned in the introduction can be written using realtime sequences. We will make the specification more concrete by specializing it to an 8-bit DAC. The 8-bit DAC input, *in*, is latched on the rising edge of its clock, *clk*. Settling time measurement begins when *in* equals `8'h00`[8] on the input for five cycles, followed by a change to `8'hff` in the next clock cycle. The input is then required to remain `8'hff` throughout the remainder of the measurement. The DAC output, *out*, should then settle to $5\ V \pm 25\ mV$ after 50 ns of latching the `8'hff` input. We understand *settled* to mean that the output remains within the specified voltage range for 25 ns after the initial 50 ns period has passed. The following sequence captures this behavior:

```
@(posedge clk)(in == 8'h00)[*5] ##1
@(posedge clk)(in == 8'hff) #0
( (in == 8'hff)[*0.0:$] intersect
  1 #[50.0n](out < 5.25 && out > 4.75)[*25.0n])
```

The sequence begins by matching the Boolean expression *in* `== 8'h00` for five cycles followed by *in* `== 8'hff`, sampled at posedges of *clk*. The sequence then switches to matching a realtime subsequence where the input remains constant and the output stays within the specified range for 25 ns after the initial 50 ns period. This is an example of the usefulness of the intermixing of realtime and clocked operators within a single sequence.

Many common idioms for AMS circuit verification can be approximated using digital sequences. These approximations typically involve user management of the sampling clock and of auxiliary signals to represent inequalities involving continuously varying quantities. These approximations result in both imprecise assertions and usability challenges. Matching a glitch is an example of one commonly encountered idiom that illustrates user management of the sampling clock. Assume we want to write a sequence to match glitches of 25 ns or less on a signal *a*. For our purposes a glitch is a short positive pulse of a Boolean. The Boolean may represent a digital signal or a threshold crossing of a realtime signal. A digital sequence to capture these glitches is shown below.

```
@(posedge a)(1) ##1
@(posedge s)(a)[*0:25] ##1
@(posedge s)(!a)
```

s is a clock with a period of 1 ns, which functions as a sampling clock. This sequence provides a reasonable approximation to the specification, but it may miss glitches, as, for

[8]The syntax `8'h00` represents the 8-bit number whose hex value is 00. Similarly, `8'hff` is the 8-bit number whose hex value is ff.

example, in the case that the glitch is less than 1 ns in length and does not stay high across a posedge of s. In fact, the choice of sampling clock is a key decision that must be made by the user when coding the sequence. In this encoding, the sequence is accurate to a precision of 1 ns. Glitches less than 1 ns may be missed, and glitches nearly 27 ns long may result in an undesired successful match. Also, the user must provide the sampling clock, which may add additional complication to the verification environment.

A realtime sequence to match the same types of glitches is shown below.

```
@(posedge a) (1) #0 (!a[~>1] intersect
      1 [*0.0:25.0n])
```

The most notable difference is the time accounting. In the realtime sequence, no sampling clock is needed because the ability to describe time is provided in the language. This sequence matches all of the expected glitches. In a realistic simulation sampling will occur, but it is likely that the user will not directly manage the sampling. Simulator controls that manage sampling should be adequate for most AMS verification applications.

C. Mapping from Timed Regular Expressions

In this section, we show how to map from the timed regular expressions of [1] into our realtime sequences. Further relationships with timed regular expressions are discussed in Section VI. The definition of timed regular expressions in [1] uses the following grammar, where b denotes a Boolean expression and Z denotes an integer bounded interval:

$$\varphi ::= b \mid \varphi \cdot \varphi \mid \varphi \vee \varphi \mid \varphi \wedge \varphi \mid \varphi^* \mid \langle \varphi \rangle_Z$$

The semantics from [1] can be rendered in our notation by the satisfaction relation $\models_{tre}$ defined as follows:

- $W, I \models_{tre} b$ iff $|I| > 0$ and $W(t) \models b$ for every $t \in I$.
- $W, I \models_{tre} \varphi \cdot \varphi'$ iff there exist J, J' such that $J < J'$, $I = J \cup J'$, $W, J \models_{tre} \varphi$, and $W, J' \models_{tre} \varphi'$.
- $W, I \models_{tre} \varphi \vee \varphi'$ iff either $W, I \models_{tre} \varphi$ or $W, I \models_{tre} \varphi'$.
- $W, I \models_{tre} \varphi \wedge \varphi'$ iff both $W, I \models_{tre} \varphi$ and $W, I \models_{tre} \varphi'$.
- $W, I \models_{tre} \varphi^*$ iff there exist $n \geq 0$ and $J_1, \ldots, J_n$ such that $J_i < J_j$ for all $1 \leq i < j \leq n$, $I = J_1 \cup \cdots \cup J_n$, and $W, J_i \models_{tre} \varphi$ for all $1 \leq i \leq n$.
- $W, I \models_{tre} \langle \varphi \rangle_Z$ iff $W, I \models_{tre} \varphi$ and $|I| \in Z$.

This casting of the semantics of timed regular expressions leads directly to a linear syntactic map $\mathfrak{M}$ into realtime sequences:

- $\mathfrak{M}(b) =$ b`[*0.0+:$]`.
- $\mathfrak{M}(\varphi \cdot \varphi') = \mathfrak{M}(\varphi)$ `##1` $\mathfrak{M}(\varphi')$.
- $\mathfrak{M}(\varphi \vee \varphi') = \mathfrak{M}(\varphi)$ `or` $\mathfrak{M}(\varphi')$.
- $\mathfrak{M}(\varphi \wedge \varphi') = \mathfrak{M}(\varphi)$ `intersect` $\mathfrak{M}(\varphi')$.
- $\mathfrak{M}(\varphi^*) = \mathfrak{M}(\varphi)$`[*]`.
- $\mathfrak{M}(\langle \varphi \rangle_Z) = \mathfrak{M}(\varphi)$ `intersect` $\mathfrak{M}(Z)$, where
  - $\mathfrak{M}([\alpha, \beta]) =$ `1[*`α`:`β`]`
  - $\mathfrak{M}((\alpha, \beta]) =$ `1[*`α`+:`β`]`
  - $\mathfrak{M}([\alpha, \beta)) =$ `1[*`α`:`β`-]`
  - $\mathfrak{M}((\alpha, \beta)) =$ `1[*`α`+:`β`-]`

Semantic faithfulness of $\mathfrak{M}$ is given by the following

Proposition 3: *$W, I \models_{tre} \varphi$ iff $W, I \models_r \mathfrak{M}(\varphi)$.* □

Proposition 3 holds for any trace W and bounded interval I. It should be noted that in [1], timed regular expressions are interpreted over a restricted class of realtime traces, called *signals*, that are piecewise constant and left continuous. The piecewise constant condition requires the set of discontinuities of the trace to have no limit point in $\mathbb{R}$, so, in particular, there can be at most finitely many discontinuities in any bounded interval of the trace. The condition of left continuity implies that no event can occur in a signal. Furthermore, [1] restricts the domain of a signal to be a bounded interval that either is empty or is left open and right closed.

V. Automata Construction

This section provides a construction for timed automata recognizers for our realtime regular expressions. The automaton $\mathcal{A}$ constructed for sequence R recognizes R in the sense that for all W and I, $W, I \models_r R$ iff $\mathcal{A}$ has an accepting run whose trace is satisfied by W over the interval I. These notions will be made precise below. The construction provides the basis for an implementation strategy for our framework. The definition of timed automaton below is based on that in [1].

A. Definition of Timed Automaton

A timed automaton is a tuple $\mathcal{A} = (Q, C, \Delta, \Gamma, L, S, F)$ where Q is a finite set of states, C is a finite set of clocks, Δ is a transition relation (see below), Γ is an alphabet that we assume to be a Boolean algebra with multiplicative unit 1, $L : Q \to \Gamma$ is the state labeling, $S \subseteq Q$ is a set of initial states, and $F \subseteq Q$ is a set of final (accepting) states. The transition relation, Δ, consists of tuples of the form (q, ϕ, ρ, q') where $q, q' \in Q$, $\rho \subseteq C$ (the set of clocks to be reset), and ϕ is a Boolean combination of terms of the form $(c \in I)$, where $c \in C$ and I is an interval of $\mathbb{R}_{\geq 0}$ whose endpoints are rational numbers or ∞.

A *clock valuation* is a function $\mathbf{v} : C \to \mathbb{R}_{\geq 0}$. We denote the space of all clock valuations by $\mathcal{H}$. By using clocks as coordinates, a clock valuation can be identified with a vector in $\mathbb{R}^C$. $\mathbf{0}$ denotes the vector $(0, 0, \ldots, 0)$ and $\mathbf{1}$ denotes the vector $(1, 1, \ldots, 1)$. A *configuration* of the automaton is a pair $(q, \mathbf{v}) \in Q \times \mathcal{H}$. Every ρ creates a reset function $Reset_\rho : \mathcal{H} \to \mathcal{H}$ defined by

$$Reset_\rho(\mathbf{v})(c) = \begin{cases} 0 & \text{if } c \in \rho \\ \mathbf{v}(c) & \text{if } c \notin \rho \end{cases}$$

A *finite run* of the automaton on $[a, b]$ is a sequence

$$t_0, (q_0, \mathbf{v}_0) \xrightarrow[t_1]{d_1} (q_1, \mathbf{v}_1) \xrightarrow[t_2]{d_2} \cdots \xrightarrow[t_n]{d_n} (q_n, \mathbf{v}_n),$$

where $q_i \in Q$, $\mathbf{v}_i \in \mathcal{H}$, $d_i \in \Delta$, $t_i \in \mathbb{R}_{\geq 0}$ are such that

- $0 \leq a = t_0 \leq t_1 \leq t_2 \leq \cdots \leq t_n = b$ and $\mathbf{v}_0 = \mathbf{0}$.
- $d_i = (q_{i-1}, \varphi_i, \rho_i, q_i)$, where $\mathbf{v}_i = Reset_{\rho_i}(\mathbf{v}_{i-1} + (t_i - t_{i-1}) \cdot \mathbf{1})$ and φ_i is satisfied on $\mathbf{v}_{i-1} + (t_i - t_{i-1}) \cdot \mathbf{1}$.

The run is *accepting* if $q_0 \in S$ and $q_n \in F$. The *full trace* of the run is the function $T : [t_0, t_n] \to \Gamma$ defined by the following rules:

- If $t_i < t_{i+1}$, then for $t \in (t_i, t_{i+1})$, $T(t) = L(q_i)$.
- If $t_{i-1} < t_i = t_{i+1} = \cdots = t_{j-1} = t_j < t_{j+1}$, then $T(t_i) = L(q_i) \wedge \cdots \wedge L(q_{j-1}) \in \Gamma$.
- If $T(t)$ is not defined by the preceding conditions, then $T(t) = 1 \in \Gamma$.

We assume that each initial and final state is classified either as *inclusive* or *exclusive*. The *restricted trace* of a run is the function $T|_I$, where $T : [t_0, t_n] \to \Gamma$ is the full trace and $I \subseteq [t_0, t_n]$ is the interval obtained from $[t_0, t_n]$ by including or excluding each of the endpoints t_0 and t_n in accordance with the kind, inclusive or exclusive, of the initial and final states of the run, respectively.[9] If the initial and final states are both exclusive and $t_0 = t_n$, then $I = \{\}$.

Let W be a realtime trace, $\mathcal{A}$ be a timed automaton, and I be a bounded interval. For I non-empty, we define $W, I \models \mathcal{A}$ iff there exists a run of $\mathcal{A}$ on $[\inf I, \sup I]$ such that I is the domain of the restricted trace of the run and $W(t) \models T(t)$ for all $t \in I$, where T is the trace (full or restricted) of the run. We define $W, \{\} \models \mathcal{A}$ iff there exists $t_0 \in \mathbb{R}_{\geq 0}$ and a run of $\mathcal{A}$ on $[t_0, t_0]$ such that the domain of the restricted trace of the run is $\{\}$. We say that $\mathcal{A}$ *recognizes* the sequence R if for all W and I, $W, I \models_r R$ *iff* $W, I \models \mathcal{A}$.

B. Automata Convenience Features

To simplify the exposition of our automata construction we use additional features and notations described below. By definition, clock constraints ϕ appear only on transitions of a timed automaton. It can be convenient to specify a timing condition on a state, where the timing condition restricts the amount of time that a run may spend in a single visit to the state (distinct visits are treated independently). Such conditions can be implemented by (at worst) adding a single *state clock* η that is reset on every transition and such that the timing condition of a state is added to each outgoing transition.[10] The state timing condition "0" abbreviates "$\eta = 0$", i.e., no time elapse in the state. Such a state is called a 0-time state. The state timing condition "+" abbreviates "$\eta > 0$", i.e., positive time must elapse in each visit to the state. Such a state is called a +-time state and is annotated by a "+" in the lower half of the state in figures.

The label of a state conditions the trace of a run for the times that the run is in the state. It can be useful also to condition with a label the times at which a transition is taken. A transition label can be implemented by inserting a 0-time state and placing the label on the new state.

Ingresses and egresses provide a graphical notation for simplified initial and final states and their classification as inclusive or exclusive. An *ingress* is an initial state with no incoming and a single outgoing transition, while an *egress* is a final state with no outgoing and a single incoming transition. The ingress and egress states are 0-time and their state label is understood to be $1 \in \Gamma$. If needed, a labeling condition may be placed on the incident transition. Inclusive states are denoted by small closed circles, while exclusive states are denoted by small open circles. For example, the automaton in Fig. 1 has three ingresses, one exclusive and two inclusive. The transitions from the inclusive ingresses are labeled "$\neg\kappa$" and "$\kappa \wedge b$", respectively. The automaton has one egress, which is inclusive and has no label on its incident transition.

C. Automata Construction

The automata are built by induction on the structure of the sequences.

- The automaton for `@(`κ`)` (b) is shown in Fig. 1.

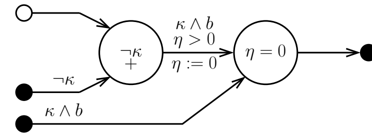


Fig. 1. Automaton for a clocked Boolean, `@(`κ`)` (b).

- The automaton for R `##1` R' is created by connecting the automata for R and R'. The rule for connection requires that an inclusive ingress/egress be connected to an exclusive egress/ingress. This rule ensures that there is no overlap and no gap in the interval matched. When an ingress/egress is connected it is no longer an ingress/egress and consequently is no longer an initial/final state, respectively. An example of how the connection works is shown in Fig. 2. This connection rule does not apply to a subautomaton for empty, as empty is an identity for `##1`. Instead, the subautomaton is combined as though it were an identity.
- The automaton for R `##0` R' is created by connecting the automata for R and R'. The rule for connection requires that inclusive egresses of R be connected to inclusive ingresses of R'. This ensures that there is a single point overlap between the matches for R and R', which is required by `##0`. The connection works in a manner similar to R `##1` R'.
- The automaton for R `or` R' is created using a standard union of the automata for R and R'.
- The automaton for R `intersect` R' is created as a product automaton for R and R'. The rules for the product construction are as follows:
  - (p, q) is an ingress (resp., egress) iff both p and q are ingresses (resp. egresses) and either both p and q are inclusive or both p and q are exclusive.
  - (p, q) is a +-time state iff both p and q are +-time states.
  - (p, q) is a 0-time state iff either p or q is a 0-time state.
  - *Parallel transitions* are transitions where both of the factor automata are changing state. If $p \xrightarrow{\alpha} p'$ is a

[9]Inclusion is understood to take precedence over exclusion in the case where $t_0 = t_n$ and the classifications of the initial and final states do not agree.

[10]A final state with a timing condition can be implemented by rendering the state non-final and adding a companion final state to which it transitions. The companion state matches the original in its label and classification.

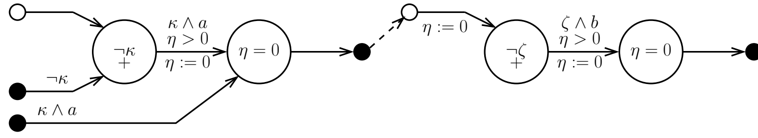


Fig. 2. An automaton for `@(`κ`)(`a`) ##1 @(`ζ`)(`b`)`. This demonstrates the connection rules for `##1`.

transition of R and $q \xrightarrow{\beta} q'$ is a transition of R', then $(p, q) \xrightarrow{\alpha\wedge\beta} (p', q')$ is a parallel transition of the product.

– *Stutter transitions* are transitions where only one of the factor automata is changing state. The non-changing state is said to be *stuttering*. There is a subtlety when the stuttering state is +-time. The stuttering transition can lead to gaps in the state label for this factor because a trace associates a state label with the interior of a positive length time interval spent in that state. To avoid gaps, the state label needs to be added to the stutter transition in certain circumstances. Intuitively, the case where the state label is added occurs when the changing state is transitioning from a 0-time state to a +-time state and the stuttering state has already been occupied for positive time in this visit. A complete definition of the rules is found below. Each factor state is either 0-time, denoted, e.g., ${}^0q$, or +-time, denoted, e.g., ${}^+q$. A mixed product state is one with one factor state 0-time and one factor state +-time. It is annotated with a *kind* 0 or 1. A kind 0 state indicates that the +-time factor has not yet been occupied for positive time in this visit, while a kind 1 state indicates that the +-time factor has been occupied for positive time in this visit. A parallel transition is denoted by two solid arrows, while a stutter transition is denoted by one solid and one dashed arrow. Rule g shows the addition of a state label to a stutter transition. Rules a-h below describe all the transition forms involving mixed product states of various kinds, up to swapping the factors of the tuples. Each rule has a dual which is also valid. In the description, * allows any possible qualifier and e is a variable used to represent kind 0 or 1.

a. $\begin{pmatrix} {}^*p \\ {}^*q \end{pmatrix}_* \begin{matrix} \longrightarrow \\ \longrightarrow \end{matrix} \begin{pmatrix} {}^0p' \\ {}^+q' \end{pmatrix}_0$ **b.** $\begin{pmatrix} {}^0p \\ {}^+q \end{pmatrix}_1 \begin{matrix} \longrightarrow \\ \longrightarrow \end{matrix} \begin{pmatrix} {}^*p' \\ {}^*q' \end{pmatrix}_*$

c. $\begin{pmatrix} {}^0p \\ {}^+q \end{pmatrix}_1 \begin{matrix} \dashrightarrow \\ \longrightarrow \end{matrix} \begin{pmatrix} {}^0p \\ {}^*q' \end{pmatrix}_*$ **d.** $\begin{pmatrix} {}^0p \\ {}^*q \end{pmatrix}_* \begin{matrix} \dashrightarrow \\ \longrightarrow \end{matrix} \begin{pmatrix} {}^0p \\ {}^+q' \end{pmatrix}_0$

e. $\begin{pmatrix} {}^0p \\ {}^+q \end{pmatrix}_e \begin{matrix} \longrightarrow \\ \dashrightarrow \end{matrix} \begin{pmatrix} {}^0p' \\ {}^+q \end{pmatrix}_e$ **f.** $\begin{pmatrix} {}^0p \\ {}^+q \end{pmatrix}_0 \begin{matrix} \longrightarrow \\ \dashrightarrow \end{matrix} \begin{pmatrix} {}^+p' \\ {}^+q \end{pmatrix}$

g. $\begin{pmatrix} {}^0p \\ {}^+q \end{pmatrix}_1 \begin{matrix} \longrightarrow \\ \underset{L(q)}{\dashrightarrow} \end{matrix} \begin{pmatrix} {}^+p' \\ {}^+q \end{pmatrix}$ **h.** $\begin{pmatrix} {}^+p \\ {}^+q \end{pmatrix} \begin{matrix} \longrightarrow \\ \dashrightarrow \end{matrix} \begin{pmatrix} {}^0p' \\ {}^+q \end{pmatrix}_1$

.

- The automaton for R`[*0]` is shown in Fig. 3(a).
- The automaton for R`[+]` follows the connection rule for R `##1` R'. A connected ingress/egress no longer functions as ingress/egress. In a repetition the initial/final state behavior must be maintained, so ingresses/egresses for the repetition must be duplicated for use in the connection. If R has an empty subautomaton, then so does R`[+]`, but the empty subautomaton does not play a role in the connections. See Fig. 3(b) for an example.
- The automaton for b is shown in Fig. 3(c).

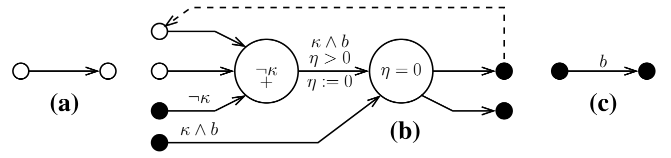


Fig. 3. Automata for (a) R`[*0]`, (b) `(@(`κ`)(`b`))[+]`, and (c) realtime Boolean, b.

- The automata for b`[*`α`[+]:`β`[-]]` are shown in Fig. 4.

VI. Further Relationships with Timed Regular Expressions

In [1], [2], Asarin, Caspi, and Maler define *timed regular expressions* and study their relationship to timed automata. The main results show that timed regular expressions, generalized to support renaming, have the same expressive power as timed automata. This section discusses relationships between our realtime sequences and timed regular expressions.

The syntax for timed regular expressions is essentially the same in [1] and [2] and has been given in Section IV-C. [2] adds syntax for the empty word, analogous to our form R`[*0]`. [2] also adds syntax to specify renamings, which have no analog in our framework.

Several different semantic models are presented in [1] and [2]. The piecewise constant, left continuous signals of [1] are, in many regards, the closest to our realtime traces. They underly the relation $\models_{tre}$ and provide the basis for the semantically faithful embedding $\mathfrak{M}$ presented in Section IV-C. Signals do not, however, support expression that a condition hold or an event occur at a specific point in time. These capabilities are important for applications to AMS verification and are supported in realtime sequences by the immediate Boolean (b), clocked Boolean (`@(`κ`)(`b`)`), and concatenations with overlap (`##0`, `#0`, etc.).

Time-event sequences are the primary semantic model in [2]. A *time-event sequence* is a sequence of terms, each of which is either an event or a non-negative real number, specifying a time elapse. For example, the time-event sequence $r \cdot \kappa \cdot s \cdot \zeta$ represents that event κ occur after r units of time and that event ζ occur after another s units of time. Simultaneity of multiple events is expressible, as in $r \cdot \kappa \cdot \zeta$,

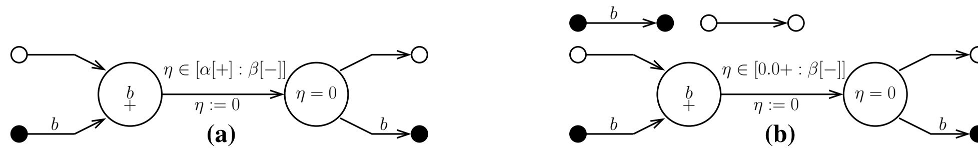


Fig. 4. Automata for (a) the Boolean smear, `b[*`α`[+]:`β`[-]]` with a positive α and (b) the Boolean smear, `b[*0.0:`β`[−]]` with α equal to zero.

$r \cdot \zeta \cdot \kappa$, $r \cdot \kappa \cdot \kappa$, which are all distinct. In [2], timed regular expressions can specify unbounded regular patterns of events, all occurring at the same time but with discrete ordering amongst themselves. For applications to AMS verification, we do not believe that such capabilities are needed. For example, we see no practical use for expressing the condition that the value of a particular analog variable cross a particular threshold five times, say, at a single instant of time. Our realtime traces do not admit this granularity of ordering at a single time. In a realtime trace, an event either occurs or does not at any particular time. There is no notion of multiplicity, and if two events occur at the same time, there is no distinction of their order. Our realtime sequences can express the condition that a fixed set of events occur simultaneously, as in `(@(`κ`)(1) ##0 @(`ζ`)(1)) intersect 1`. This form also shows a syntactic order of κ before ζ, but our realtime trace models do not resolve this order.

In Section V we showed a construction of timed automata recognizers for our realtime sequences. Assuming that suitable translation conventions are fixed to convert between the differing semantic models, this construction shows that our realtime regular expressions are no more expressive than the generalized extended timed regular expressions of [2]. Definitive comparison of the two regular expression languages seems to depend on precise reconciliation of the semantic models. In mapping from time-event sequences to realtime traces, multiplicity and ordering of simultaneous events need to be encoded using analog and discrete variables. In mapping the other direction, behaviors of analog and discrete variables to which the relevant Boolean expressions and events are sensitive need to be represented by regular patterns of events. The details of such analysis appear non-trivial and merit consideration in future work.

VII. Conclusion

Verification of AMS systems is becoming increasingly important as AMS designs become more popular and complex. To meet the needs of AMS verification, we must develop verification techniques and languages that support both the clocked and realtime domains. We have proposed syntax and semantics for realtime regular expressions. This has been done before [1], [2], but the key feature of our framework is that it generalizes the framework of the existing SVA regular expressions. This feature allows free intermingling of realtime and digital sequences, which enables our realtime regular expressions conveniently to represent complex properties that specify both clocked and realtime requirements. We have investigated how the new syntax and semantics relate to existing definitions of realtime regular expressions. We provide a semantically faithful mapping from the timed regular expressions of [1], which demonstrates that our formalism is not less expressive. We also provide a construction of timed automata recognizers for our realtime regular expressions. This construction demonstrates that our realtime regular expressions are no more expressive than the generalized timed regular expressions of [2] and provides a basis for an implementation strategy.

In the future, we plan to demonstrate how this semantics can be extended to local variables and the `first_match` operator. We also plan to develop similarly compatible semantics for the SVA property operators. When completed, these pieces will constitute a realtime extension to the full SVA language. This new realtime SVA language will provide engineers the ability to specify complex AMS properties accurately and in a single assertion language.

References

[1] E. Asarin, P. Caspi, and O. Maler, “A Kleene theorem for timed automata,” in *IEEE Symposium on Logic in Computer Science*. IEEE Press, 1997, pp. 160–171.

[2] ——, “Timed regular expressions,” *Journal of the ACM (JACM)*, vol. 49, no. 2, pp. 172–206, 2002.

[3] *IEEE Standard for SystemVerilog (1800-2009)*, IEEE Computer Society, Dec. 2009.

[4] *IEEE Standard for Property Specification Language (PSL) (1850-2010)*, IEEE Computer Society, Jun. 2010.

[5] A. Pnueli, “The temporal semantics of concurrent programs,” *Theor. Comput. Sci.*, vol. 13, pp. 45–60, 1981.

[6] D. Smith, “Asynchronous behaviors meet their match with SystemVerilog Assertions,” in *Design and Verification Conference (DVCON)*, 2010.

[7] D. Nickovic, O. Maler, A. Fedeli, P. Daglio, and D. Lena. (2007) Analog case study : PROSYD deliverable 3.4/2. [Online]. Available: http://www.prosyd.org/twiki/pub/Public/DeliverablePageWP3/prosyd3.4.2.pdf

[8] K. D. Jones, V. Konrad, and D. Nickovic, “Analog property checkers: a DDR2 case study,” *Formal Methods in System Design*, vol. 36, no. 2, pp. 114–130, 2010.

[9] R. Mukhopadhyay, S. K. Panda, P. Dasgupta, and J. Gough, “Instrumenting AMS assertion verification on commercial platforms,” *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 14, pp. 21:1–21:47, April 2009.

[10] S. Mukherjee and P. Dasgupta, “Incorporating local variables in mixed-signal assertions,” in *TENCON*, 2009, pp. 1–5.

[11] R. Alur, T. Feder, and T. A. Henzinger, “The benefits of relaxing punctuality,” *Journal of the ACM*, vol. 43, no. 1, pp. 116–146, 1996.

[12] D. Nickovic, “Checking timed and hybrid properties: Theory and applications,” Ph.D. dissertation, Université Joseph Fourier, 2009.

[13] H. Anand, J. Havlicek, and S. Little. (2010) Some notes on realtime semantics. [Online]. Available: http://www.vhdl.org/twiki/pub/VerilogAMS/RequirementsGatheringGroup/semantics.pdf

[14] E. Cerny, S. Dudani, J. Havlicek, and D. Korchemny, *The Power of Assertions in SystemVerilog*. Springer, 2010.

Formal Analysis of Fractional Order Systems in HOL

Umair Siddique
Research Center for Modeling and Simulation
National University of Sciences and Technology
(NUST), Islamabad, Pakistan
Email:umair.siddique@rcms.nust.edu.pk

Osman Hasan
School of Electrical Engineering and Computer Science
National University of Sciences and Technology
(NUST), Islamabad, Pakistan
Email:osman.hasan@seecs.nust.edu.pk

***Abstract*—Fractional order systems, which involve integration and differentiation of non integer order, are increasingly being used in the fields of control systems, robotics, signal processing and circuit theory. Traditionally, the analysis of fractional order systems has been performed using paper-and-pencil based proofs or computer algebra systems. These analysis techniques compromise the accuracy of their results and thus are not recommended to be used for safety-critical fractional order systems. To overcome this limitation, we propose to leverage upon the high expressiveness of higher-order logic to formalize the theory of fractional calculus, which is the foremost mathematical concept in analyzing fractional order systems. This paper provides a higher-order-logic formalization of fractional calculus based on the Riemann-Liouville approach using the HOL theorem prover. To demonstrate the usefulness of the reported formalization, we utilize it to formally analyze some fractional order systems, namely, a fractional electrical component Resistoductance, a fractional integrator and a fractional differentiator circuit.**

I. Introduction

In reality, many situations arise when integer order calculus is not sufficient to model all kind of dynamics. For example, an electrical component Resistoductance [10] exhibits an intermediate behavior between that of a resistor and inductor and thus its accurate modeling involves the differentiation of order between 0 and 1. Such systems that involve integration and differentiation of non integer order, or *fractional calculus* [26], for their modeling are usually referred to as fractional order systems. The idea of fractional calculus is as old as integer order calculus itself. The question which gave birth to fractional calculus was about the interpretation of $\frac{d^n y}{dx^n}$, if n is not an integer or more broadly if n is any real, irrational or even a complex number.

Accurate modeling of engineering and scientific systems have become imperative these days due to their extensive usage in safety-critical domains, such as, medicine and transportation. This fact has led to the widespread usage of fractional calculus in modeling physical systems. For example, in control engineering the concept of fractional operations is mostly used in fractional system identification [17], biomimetic control [6], fractional PI^α [22] and PD^μ controllers [8]. In signal processing, fractional operators are used in the design of fractional order differentiators and integrators [21] and for modeling the speech signals [20]. Other interesting applications of fractional calculus are in image processing [29], electromagnetic theory [13], chaotic communication [1], and circuit theory [10].

Traditionally, the analysis of fractional calculus based models has been done using paper-and-pencil proof methods. However, considering the complexity of present age engineering and scientific systems, such analysis is notoriously difficult if not impossible, and is quite error prone. Many examples of erroneous paper-and-pencil based proofs are available in the open literature, a recent one can be found in [7] and its identification and correction is reported in [27]. One of the most commonly used computer based analysis technique for fractional order systems is numerical computation of fractional integration and differentiation. Some examples include, chaos in fractional order volta systems [30], fractional PI^α controllers [22] and motion planning of redundant and hyper-redundant manipulators [23]. Fractional order systems are continuous in nature and thus the first step in their simulation based analysis is to construct a discretized system model with minimal error. Most of the numerical algorithms are based either on the Grünwald-Letnikov definition [12] or on the Power Series Expansion (PSE) method [30]. Both of them cannot provide reliable results due to the involvement of infinite summations in case of Grünwald-Letnikov definition and huge memory requirements in case of the PSE method. Similarly, the computation of the Gamma function $\Gamma(x)$ for large values of x is not possible in such numerical computation software packages. For example, MATLAB [24] returns `7.26e306` as the approximated value computed for $x = 171$ and returns `Inf` for all values beyond $x = 171$. Another alternative to analyze fractional order systems is computer algebra systems [3], which are very efficient for computing mathematical solutions symbolically, but they are not reliable [15] due to their limitations of dealing with side conditions. Another limitation of computer algebra systems related to fractional calculus is the uncertain simplification of singular expressions particularly in case of the Gamma function, which are frequently used in fractional calculus [18] . Another source of inaccuracy in computer algebra systems is the presence of unverified huge symbolic manipulation algorithms in their core, which are quite likely to contain bugs. Thus, these traditional techniques should not be relied upon for the analysis of fractional order systems, especially when they are used in safety-critical areas

(e.g., cardiac tissue electrode interface [9] which is modeled and analyzed using fractional calculus), where inaccuracies in the analysis may even result in the loss of human lives.

In the past couple of decades, formal methods have been successfully used for the precise analysis of a variety of hardware and software systems. The rigorous exercise of developing a mathematical model for the given system and analyzing this model using mathematical reasoning usually increases the chances for catching subtle but critical design errors that are often ignored by traditional techniques like numerical methods. Given the sophistication of the present age fractional order systems and their extensive usage in safety critical applications, there is a dire need of using formal methods in this domain. However, due to the continuous nature of the analysis and the involvement of transcendental functions, automatic state-based approaches, like model checking [19], cannot be used in this domain. On the other hand, we believe that higher-order-logic theorem proving [14] offers a promising solution for conducting formal analysis of fractional order systems. The main reason is being the highly expressive nature of higher-order logic, which can be leveraged upon to essentially model any system that can be expressed in a closed mathematical form. In fact, most of the classical mathematical theories behind elementary calculus, such as limits, differentiation, integration and transcendental functions, have been formalized in higher-order logic [15]. In this paper, we build upon the available theories of elementary calculus to formalize Riemann-liouville's [26] definitions of fractional integration and differentiation in higher-order logic. These definitions are then used to formally verify some classical properties of fractional calculus using the HOL theorem prover [34], which has been chosen due to the availability of Harrison's seminal work on the formalization of elementary calculus [15]. The formal verification of these classical properties of fractional calculus , such as, linearity, identity and the relationship with elementary calculus, not only ensures the correctness of our formal definitions of fractional integration and differentiation but also plays a vital role in the formal analysis of fractional order systems. To the best of our knowledge, the reported formalization is the first one of its kind and facilitates the formal analysis of fractional order systems, which is a novelty that has not been presented in the open literature so far using any formal technique.

The rest of the paper is organized as follows: Section II describes some fundamentals of fractional calculus , its commonly used definitions and the justification behind the choice of Riemann-Liouville approach for our formalization. Section III presents the proposed framework for the formal analysis of fractional order systems. Section IV presents our HOL formalization. In order to demonstrate the practical effectiveness and the utilization of proposed framework, we present the analysis of some real-world fractional order systems, i.e., a Resistoductance, a fractional differentiator and a fractional integrator circuit in Section V. Finally, Section VII concludes the paper.

II. Fractional Calculus

In 1695, L'Hôpital asked Leibnitz regarding his notation $\frac{d^n y}{dx^n}$, "*What if n is* $\frac{1}{2}$". Leibnitz prophesied in his letter [10] to L'Hôpital,"*... Thus it follows that* $d^{\frac{1}{2}}x$ *will be equal to* $x\sqrt{dx : x}$*. This is an apparent paradox from which, one day, useful consequences can be drawn ...*". Leibnitz's initial work on the problem of defining the derivative of arbitrary order gave birth to a new field of research in mathematics and attracted attention of many biologists, physicists, engineers and geometers. Initially more efforts were made for defining fractional derivatives and fractional integrals but Neils Henrik Abel [26] was the first one to use this idea in solving the famous Tautochrone problem. The other great mathematicians and physicists who touched the field of fractional calculus are Riemann, Liouville, Laurent, Heaviside, Al-Bassam, Davis Erdelyi, Riesz and Thomas J. Osler [26].

Fractional integrals and fractional derivatives are also referred to as Differintegrals [28] and there are more than ten well known definitions for Differintegrals [9]. We consider two of them, which are most widely used in analyzing real-world problems. These are the Riemann-Liouville and Grünwald-Letnikov definitions, which are also equivalent for a wide class of functions [31].

- Riemann-Liouville (RL) Definition:

$$J_a^v f(x) = \frac{1}{\Gamma(v)} \int_a^x (x-t)^{v-1} f(t) dt \tag{1}$$

Where $J_a^v f(x)$ represents fractional integration with order v and lower integration limit a. $a = 0$ gives the Riemann definition and $a = -\infty$ gives the Liouville definition of fractional integration [32]. Γ in the above definition denotes the Gamma function which is defined using the well-known improper integral as follows:

$$\Gamma(z) = \int_0^\infty t^{z-1} e^{-t} dt \tag{2}$$

for $z > 0$.
The fractional differentiation is given as follows:

$$D^v f(x) = (\frac{d}{dx})^m J_a^{m-v} f(x) \tag{3}$$

where m represents the ceiling of v, i.e., $\lceil v \rceil$.

- Grünwald-Letnikov (GL) Definition:

$${}_cD_x^v f(x) = \lim_{h \to 0} h^{-v} \sum_{k=0}^{[\frac{x-c}{h}]} (-1)^k \binom{v}{k} f(x-kh) \tag{4}$$

Grünwald-Letnikov definition caters for both fractional differentiation and integration, as positive values of v give fractional differentiation and negative values of v give fractional integration. Here, $\binom{v}{k}$ represents the binomial coefficient, which is described in terms of the Gamma function.

The Riemann-Liouville definition provides a way to find analytical solutions while Grünwald-Letnikov definition facilitates the numerical computation of solutions. There are two

motivations of using the Riemann-Liouville definition for our formalization: Firstly, it is widely used in the modeling and analysis of engineering fractional order systems [10], Secondly, the analysis carried out in this way is purely analytical and hence free from any kind of approximations. On the other hand, Grünwald-Letnikov definition is more suitable for numerical analysis based methods and thus provides approximate solutions.

III. Proposed framework

The proposed framework, given in Figure 1, outlines the main idea behind the theorem-proving-based fractional order system analysis. The grey shaded boxes in this figure represent the key contributions of the paper that serves as the fundamental requirements of conducting fractional order system analysis in a theorem prover. Like all the system analysis tools, the input to this framework, depicted by two rectangles with curved bottoms, is the description of the fractional order system that needs to be analyzed and a set of properties that are required to be checked for the given system.

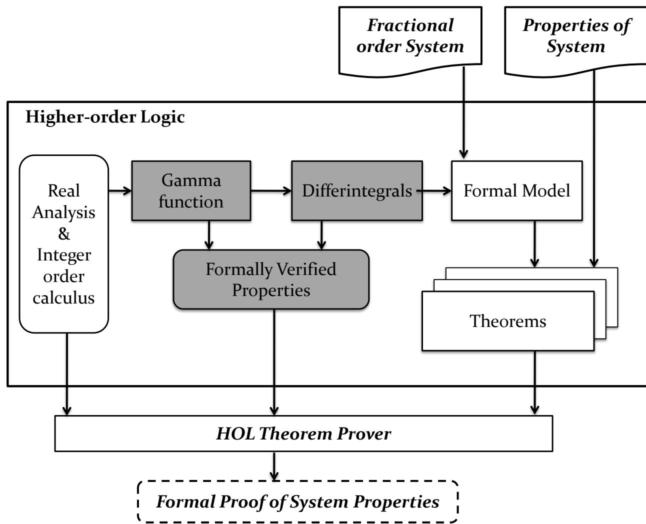


Fig. 1. Proposed Framework

The first step in conducting fractional order system analysis using a theorem prover is to construct a formal model of the given system in higher-order logic. For this purpose, the foremost requirement is the ability to formalize fractional derivatives and integrals (Differintegrals) as higher-order logic functions. The formalization of Differintegrals, given in Equations 1 and 3, requires the mathematical theories of real numbers, integer order calculus and the Gamma function. Harrison's work on the formalization of real numbers [15] provides the first two requirements and we built upon Harrison's work to formalize the Gamma function in this paper to fulfil the third requirement. Using these fundamentals, this paper also presents the formalization of Differintegrals, given in Equation 1 and 3, which in turn can be used to represent the dynamics of fractional order systems in higher-order logic. The second step in the theorem proving based fractional order system analysis is to utilize the formal model of fractional order system, developed in the first step, to express system properties as higher-order logic theorems.

The third step for conducting fractional order system analysis in a theorem prover is to formally verify the higher-order-logic theorems developed in the previous step using a theorem prover. For this verification, it would be quite handy to have access to a library of some pre-verified theorems corresponding to some commonly used properties of Gamma function and Differintegrals. To fulfil this requirement, this paper presents formal verification of the classical properties of Gamma function, such as, Pseudo-Recurrence Relation, Factorial Generalization and Functional Equation, and Differintegrals, such as, Identity and Linearity, using the HOL theorem prover. Building on such a library of theorems would minimize the interactive verification efforts and thus speed up the verification process. Finally, the output of the theorem proving based fractional order system analysis framework, depicted by the rectangle with dashed edges, is the formal proofs of system properties that certify that the given system properties are valid for the given fractional order system.

IV. HOL Formalization

This section presents the higher-order logic formalization of the main requirements of the proposed framework, depicted by the gray shaded boxes in Figure 1. We have arranged the information in two subsections. The first subsection presents the formalization of the Gamma function and the formal verification of its associated properties using the HOL theorem prover. While the second subsection presents formalization of Riemann-Liouville definition of Differintegrals and the formal verification of its associated properties in HOL.

A. Formalization of Gamma function

The applicability of Gamma function in fractional calculus is due to its unique characteristic of generalizing factorials over non-integer numbers. The theory of improper integrals [2] suggests that it is convenient to write Gamma function (Equation 2) as follows:

$$\Gamma(z) = \lim_{a \to 0_+, b \to \infty} \int_a^b t^{z-1} e^{-t} dt \quad (5)$$

We formalize Equation (5) as follows:

Definition 1: *Gamma Function*

$\vdash \forall$ z.gamma z =
  lim(λ n.(lim
    (λ b.$\int_{\frac{1}{2^n}}^{b}$ t rpow (z-1)exp(-t) dt))

The function rpow [33] is a power function with real exponent. It takes two real numbers x and y, and returns x^y. We used $\lim_{n \to \infty}(\frac{1}{2^n})$ to model 0_+ as $(\frac{1}{2^n})$ becomes very close to 0 as n becomes very large. The integral $(\int_a^b f)$ is used to represent HOL function integral(a,b) f, which represents the formalization of the Gauge integral in HOL [15]. Mhamdi [25] presented the higher-order logic formalization of Lebesgue integration theory, which is fundamental concept in many

TABLE I
PROPERTIES OF THE GAMMA FUNCTION

Property	HOL Formalization
Pseudo-Recurrence Relation	⊢ ∀ z.(0 < z) ⟹ (gamma (z + 1)= z gamma (z))
Functional Equation	⊢ gamma 1 = 1
Factorial Generalization	⊢ ∀ n ∈ ℕ. gamma(n + 1) = n!
Reconstruction of Gamma	⊢ ∀ x z.(0 < z)∧(0 < x) ⟹ gamma z = gamma_upper x z + gamma_lower x z)
Recurrence Lower_Gamma	⊢ ∀ z x.(0 < z)∧(0 < x) ⟹ gamma_lower x (z + 1)= (z)gamma_lower x z − x rpow (z)exp(-x)
Recurrence Upper_Gamma	⊢ ∀ z x.(0 < z)∧(0 < s) ⟹ gamma_upper s z = (z − 1)gamma_upper s (z-1)+ s rpow (z-1)exp(-s)

mathematical theories and allows a wider class of functions than the Riemann integration theory. In our formalization, we built upon Harrison's formalization of Gauge integral because the proposed framework is intended to be used by engineers and practitioners, who are normally not familiar with Lebesgue integration theory.

The lower and upper incomplete Gamma functions play a vital role in obtaining Differintegrals of periodic functions, such as, sinusoidal response study of fractional operators [10], and can be formalized as follows:

Definition 2: *Upper Incomplete Gamma Function*
$$\vdash \forall\ x\ s.\texttt{gamma_upper}\ s\ z = (\texttt{lim}\ (\lambda\ b.\ \int_s^b \texttt{t rpow (z-1)exp(-t) dt}))$$

Definition 3: *Lower Incomplete Gamma Function*
$$\vdash \forall\ x\ z.\texttt{gamma_lower}\ x\ z = (\texttt{lim}\ (\lambda\ n.\ \int_{\frac{1}{2^n}}^x \texttt{t rpow (z-1)exp(-t) dt}))$$

Next, we defined and verified some of the key properties of Gamma function in HOL using Definitions 1, 2, and 3. The formal verification of these properties not only ensures the correctness of our formal definition but also facilitates the formal reasoning about fractional order systems in higher-order logic as mentioned in Section III. The formally verified properties of Gamma function are given in Table I.

The first property in Table I represents the Pseudo-Recurrence Relation of the Gamma function and can be classified as the most important property of the Gamma function as it plays a vital role in verifying the other properties of Gamma function and Differintegrals. The verification of this property was also one of the most challenging part of our formalization as it involves the core concepts of improper integrals, limits and sequences. Its reasoning process involve ten main lemmas, such as, convergence of integral with respect to upper and lower limits, limits on infinity and zero, simplification of integrand by integration by parts and the continuity, differentiability and integrability of the integrand. The complete formalization details are provided in [33].

The second and third properties of Table I, i.e, Functional Equation and Factorial Generalization, are very important in establishing the link between fractional calculus and integer order calculus. The verification of these properties requires the Pseudo-Recurrence Relation of Gamma function along with some limit theory proofs and arithmetic reasoning in HOL. The fourth property, Reconstruction of Gamma function, shows that the Gamma function can be divided into two integrals, which are incomplete at one limit, i.e., upper and lower incomplete Gamma functions. The verification of this property requires lemmas used in the verification of Pseudo-Recurrence Relation along with the properties of the Gauge integral. The last two properties in Table I show the recurrence relation of the upper and lower incomplete Gamma functions. These relations are very important in fractional calculus because factional integration and differentiation of many important functions, e.g., Exponential function, is represented in terms of the incomplete Gamma functions and then these properties are utilized to evaluate such mathematical expressions. The verification of these properties is similar to that of the verification of Pseudo-Recurrence Relation.

This completes our formalization of the Gamma function, which to the best of our knowledge is the first one in higher-order logic. The main challenge in the reasoning process is to deal with improper integrals in higher-order logic. The Gamma function is useful in many domains, such as, probability theory (Gamma Distribution), and our formalization can be directly utilized in such applications. Our formalization of Gamma function can be generalized to formalize other improper integrals, such as, the Beta function. Next, we build upon the formalization of the Gamma function to formalize Differintegrals.

B. Formalization of Differintegrals

The second major requirement of formal reasoning about fractional order systems, is the formalization of Differintegrals, as depicted in Figure 1. We utilize Equations (1) and (3) to formally define fractional integration and differentiation, respectively.

Definition 4: *Fractional Integration*
$$\vdash \forall\ f\ v\ a\ x.\texttt{frac_int}\ f\ v\ a\ x = \texttt{if (v = 0) then f else}\ \texttt{lim}(\lambda n.\ \frac{1}{\texttt{gamma v}} (\int_a^{x-\frac{1}{2^n}} \texttt{((x - t) rpow (v-1)) f(t) dt})$$

Definition 5: *Fractional Differentiation*
$$\vdash \forall\ f\ v\ a\ x.\ \texttt{frac_diff}\ f\ v\ a\ x = \texttt{n_order_deriv}\ \texttt{(clg v) (frac_int f (clg v - v) a x)}$$

Where `f` is a function of type (`real` → `real`), `v` is a `real` number that indicates the order of integration/differentitiation, and `a` and `x` represent the lower and upper limits of integration, respectively. The function `n_order_deriv` returns the nth integer order derivative of its argument `f` as $\frac{d^n f}{dx^n}$. The function `clg` is the ceiling function, which returns the least greater

TABLE II
PROPERTIES OF DIFFERINTEGRALS

Property	HOL Formalization
Identity	⊢ ∀ f a x. (a < x) ⟹ (frac_int f 0 a x = f)∧ (frac_diff f 0 a x = f)
Generalized Integral	⊢ ∀ f a x v ∈ N. (a < x)∧ (1 < v) ⟹ frac_int f v a x = lim(λn. $\frac{1}{(v-1)!}\int_a^{x-\frac{1}{2^n}}$ (x - t) rpow (v-1)f(t) dt)
frac_int Linearity	⊢ ∀ f v x a b. (frac_exists f x v)∧ (frac_exists g x v) ⟹ frac_int (a f + b g) v 0 x = a(frac_int f v 0 x)+ b(frac_int g v 0 x)
frac_diff Linearity	⊢ ∀ f v x a b. (frac_exists f x v)∧ (frac_exists g x v)∧ (∀ m. (m <= clg v) ⇒ (n_order_deriv m (frac_int f v 0 x)) differentiable x)∧ (∀ m. (m <= clg v) ⇒ (n_order_deriv m (frac_int g v 0 x)) differentiable x)⟹ (frac_diff (a f + b g) v 0 x = a(frac_diff f v 0 x)+ b(frac_diff g v 0 x))

integer of its real number argument. It is important to note that we have explicitly defined the case for $v = 0$, which is justified based on integer order calculus and proves to be very convenient for further manipulations [11].

As mentioned in Section III, now we will use our formal definitions of Differintegrals to formally verify some of the classical properties of fractional calculus, given in Table II, using the HOL theorem prover. The first property of Differintegrals is the Identity property, which shows that the 0th order fractional operators return original functions. The proof of the first part of this property is obvious from the definition of fractional integration (Definition 4) and proof of the second part is done based on the fact that $\frac{d^n f}{dx^n}$ with order 0 returns the original function. The second property in Table II shows that fractional integration generalizes the integer order integration. The verification of this property utilizes the third property (Factorial Generalization) of Gamma function, given in Table I. The next property is about the linearity of fractional integration and helps in formal reasoning about fractional order systems with multiple inputs. In the HOL formalization of `frac_int linearity` property, the assumptions `frac_exists f x v` and `frac_exists g x v` ensure the existence of Differintegrals for function `f` and `g`, respectively. The verification of this property requires the properties of Gamma function, integer order integration and limits along with some arithmetic reasoning. The HOL formalization of last property of Differintegrals in Table II shows the linearity of fractional differentiation. From Definition 5 it is clear that fractional differentiation involves fractional integration followed by the nth order ordinary differentiation. So, the third and fourth assumptions of this property ensures the differentiability of $(clg(v)-v)^{th}$ order fractional integral of the functions $f(t)$ and $g(t)$, respectively. The formal verification of this property requires the linearity of the nth integer order derivative along with some arithmetic reasoning.

Due to inherent soundness of higher-order logic theorem proving, our verification results are exactly the same as produced by paper-and-pencil proof methods. It is interesting to note that we have been able to identify a couple of critical assumptions that are missed by almost all the paper-and-pencil based proof analysis, that we came across. For example, the assumption $0 < x$ in the last two properties of the Gamma function (Table I) have not been specified in anyone of the paper-and-pencil proof based analysis (e.g., [10]). Obviously the results do not hold without this assumption and this discrepancy in the paper-and-pencil based proofs may lead to disastrous consequences if these properties are used without considering $0 < x$ for designing safety-critical fractional order systems.

The formalization, presented in this section, had to be done in an interactive way due to the undecidable nature of higher-order logic and took around 7000 lines of HOL code and approximately 550 man hours. However, the main advantage of this rigorous exercise is that our results can be built upon to facilitate formal reasoning about fractional order systems. Our proof script is available for download [33] and thus can be utilized by other researchers to conduct the formal analysis of their fractional order systems.

V. APPLICATIONS

In order to illustrate the utilization and effectiveness of the proposed framework, we apply it to analyze three real-world fractional order systems, i.e., a fractional electrical component Resistoductance, a fractional integrator and a differentiator circuit. Resistoductance is used to extend the current-voltage relationship to non-integer order and this kind of fractional order model is usually used for modeling bio-electrodes for cardiac tissue interfacing [9]. Fractional integrators and differentiators are the most basic components in fractional order PID (proportional integrator differentiator) controllers and can achieve more robustness than integer order control [5]. These systems have been chosen as case studies in our work because of their wide usability in the field of circuit theory and control systems. To the best of our knowledge, currently, there is no formal technique available for the formal verification of such systems.

A. Resistoductance

Electrical components, such as, resistors, inductors and capacitors are largely used to perform integer order calculus operations for different engineering and scientific applications. However, actual electrical components do not posses ideal behavior and exhibit some fractional order characteristics. Ignoring these characteristics always results in modeling inaccuracies. Therefore, fractional calculus is being widely used to capture real world dynamics of electrical components these days [4]. Resistoductance is a linear electrical circuit element that posses the characteristics between an ohmic resistor and an inductor. Being a fractional order electrical component,

it exhibits fractional order dynamics, which can be modeled by Differintegrals. The model of a single Resistoductance is shown in Figure 2, and its governing voltage and current relationship is given as follows:

$$i(t) = \frac{1}{K} J^{\alpha} v(t) \tag{6}$$

where $v(t)$ is the voltage and $i(t)$ is the current through the circuit element at time t. The range of the α is between 0 and 1. If $\alpha = 0$ the circuit will be purely resistive with K = R ohms and if $\alpha = 1$ the circuit will be purely inductive with K = L henrys.

The two important characteristics of Resistoductance are the output current through the circuit element when constant input voltage V_0 is applied and the behavior of the output current for the cases when $\alpha = 0$ and $\alpha = 1$. These two properties are widely used in designing Resistoductance based fractional order systems for signal processing and control engineering applications [4].

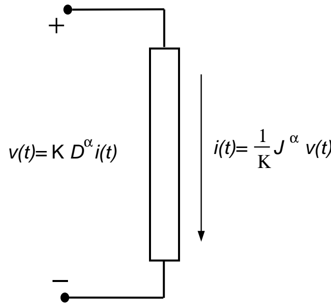


Fig. 2. Resistoductance

Now, we will present the formal verification of the above mentioned two properties of Resistoductance using our proposed framework given in Figure 1. The first step in conducting the formal analysis of Resistoductance is to construct its formal model in higher-order logic. Due to the availability of Definition 4, the formalization can be simply done as follows:

Definition 6: *Current through Resistoductance*

```
⊢ ∀ K v_i alpha x. i_t K v_i alpha x =
          (1/K)frac_int v_i(t) alpha 0 x
```

where `v_i` is input voltage, `i_t` is current through the circuit element, `alpha` is the order of integration, and the variable `x` represents the upper limit of integration. In the above definition the lower limit of integration is taken as `0` [10]. The next step, in the proposed framework, is to utilize the formal model of Resistoductance (Definition 6) to express the properties of interest as higher-order logic theorems as follows:

Theorem 1: *i_t for constant voltage V_0*

```
⊢ ∀ K v_0 alpha x.
     (0 < x) ∧ (0 < alpha) ⟹
(i_t K V_0 alpha x =
     (1/K (Gamma (alpha + 1))
                      (V_0(x rpow alpha))))
```

Theorem 2: *Special Cases for i_t*

```
⊢ ∀ x. (0 < x) ⟹
  (((alpha = 0) ⇒
      (i_t K V_0 alpha x = V_0 / K)) ∧
   ((alpha = 1)⇒
   (i_t K V_0 alpha x = (V_0 / K ) x)))
```

Theorem 1 shows the relationship of output current of Resistoductance when constant input voltage `V_0` is applied at $t = 0$. The formal verification of this theorem is based on the properties of Gamma function (Table I, Pseudo-Recurrence relation) and the definition of fractional integration. Since, these required properties have already been verified in HOL library, the interactive formal reasoning process only consists of verifying the continuity of fractional integral. Theorem 2 shows an interesting feature of Resistoductance, i.e., for $(\texttt{alpha} = 0)$ it behaves as a pure resistor and for $(\texttt{alpha} = 1)$ it exhibits the behavior of a pure inductor. The verification of Theorem 2 requires Theorem 1, the properties of the real power (`rpow`) function and some arithmetic reasoning.

This verification of Theorems 1 and 2 consumed approximately 350 lines of HOL code and about two man hours and thus was very short compared to the challenging verification of the theorems presented in the last section. The verification process, besides being compact, was also very straightforward and involved reasoning based on real analysis theories only and thus can be done with some basic know how of higher-order-logic theorem proving. The main reason for the above mentioned benefits is clearly the availability of formalized Gamma function and the Differintegrals.

B. Fractional Differentiator and Integrator Circuits

Proportional integrator (PI) and proportional integrator differentiator (PID) controllers are widely used in the industry. Numerous reliable and high performance controllers have been designed and deployed. In recent years, it has been observed that Fractional order (FO) controllers offer more flexibility in the adjustment of gain and phase characteristics than integer order controllers. Due to these flexibilities, there is a growing interest in using fractional order controllers in industry and academia [5]. The most fundamental components of PI and PID controllers are integrator and differentiator circuits, respectively. In this section, we will present the formal analysis of a fractional integrator and a differentiator circuit, [10] shown in Figure 3. The output voltage-current equations for a fractional integrator and a differentiator circuits are given as follows:

$$v_o(t) = -\frac{1}{RC} J^{\mu} v_i(t) \qquad Integrator \tag{7}$$

$$v_0(t) = -RCD^{\mu} v_i(t) \qquad Differentiator \tag{8}$$

where R and C denotes resistance and capacitance, respectively, and their values are used to define the reset rate of PID controllers. The variables, $v_o(t)$ and $v_i(t)$, in the

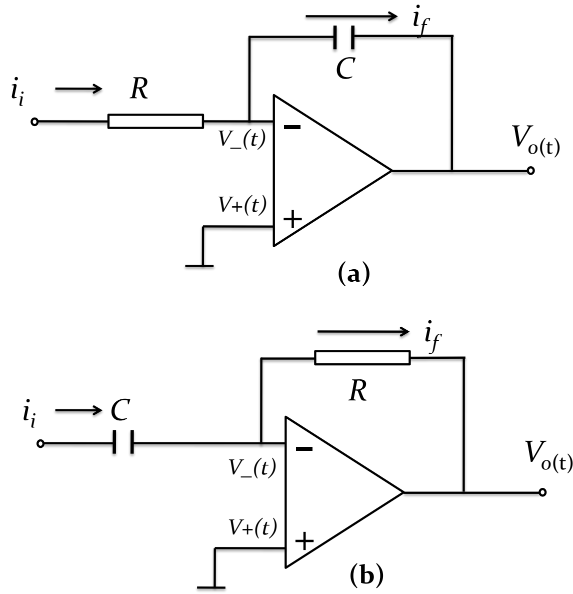


Fig. 3. (a) Integrator (b) Differentiator

above Equations represent output and input voltages at time t, respectively.

The output response of integrator and differentiator circuit is usually analyzed for benchmark input signals, such as, the unit step, which is defined as follows:

$$u(t) = \begin{cases} 0 & \text{if } t \leq 0; \\ 1 & \text{if } t > 1; \end{cases}$$

The first step in the formal analysis of integrator and differentiator circuits, when unit step signal is applied at the input, is to construct the formal model of these circuits and unit step signal in higher-order logic. Since the governing equations (Equations 7, 8) of integrator and differentiator circuits involve fractional integration and differentiation, thus we utilize our formalized definitions (Definition 4, 5) of Differintegrals as follows:

Definition 7: *Fractional Order Integrator*

```
⊢ ∀ R C v_i mu x. v_I_0 R C v_i mu x =
            -(1/RC)frac_int v_i(t) mu 0 x
```

Definition 8: *Fractional Order Differentiator*

```
⊢ ∀ R C v_i mu x. v_D_0 R C v_i mu x=
            -(RC)frac_diff v_i(t) mu 0 x
```

Definition 9: *Unit Step*

```
⊢ ∀ t. unit t = if (0 <= t) then 1 else 0
```

where `v_I_0` and `v_D_0` are output voltages of integrator and differentiator circuits, respectively. `v_i` is the input voltage, `R`, `C`, `mu` and `x` represent resistance, capacitance, order of integration/differention and upper limit of integration, respectively.

Now, the next step in the formal analysis of fractional integrator and differentiator, as mentioned in Fig 1, is to describe their properties of interest as higher-order logic theorems:

Theorem 3: *Output of Fractional Integrator Circuit*

```
⊢ ∀ R C mu x .
      (0 < x) ∧ (0 < mu) ∧ ( mu < 1)⟹
(v_I_0 R C (unit t) mu x =
      (-1/(RC Gamma (mu + 1))
                           ((x rpow ( mu))))
```

Theorem 4: *Output of Fractional Differentiator Circuit*

```
⊢ ∀ R C mu x .
      (0 < x) ∧ (0 < mu) ∧ ( mu < 1)⟹
(v_D_0 R C (unit t) mu x =
      ((-1/(RC (Gamma ((1 - mu))))
                         ((x rpow (- mu))))
```

The next step in the theorem proving based fractional order system analysis is the verification of above mentioned theorems using the already verified properties and lemmas of Section IV. Theorem 3 gives the output response of a fractional integrator circuit for unit step signal, and its formal verification certifies the output response under the given conditions. The availability of already verified properties of Gamma function and Differintegrals (Table I and Table II) led us to the simple subgoal, i.e., the proof of continuity of the integrand, which involves multiplication of power function and the unit step signal. We verified the continuity by differentiability using the classical definition of derivative formalized in HOL [15].

Theorem 4 describes the output response of the fractional differentiator circuit with unit step signal as an input. The second and third assumptions in Theorem 4 ensure that the order of the fractional differentiation `mu` is between 0 and 1 which means that `clg` of `mu` will always be `1`. So the verification of this theorem requires fractional integration of the order `1` $-$ `mu` followed by the fractional differentiation of order `1`. This requires Theorem 3 along with some arithmetic reasoning. Just like the case of the Resistoductance, the verification of Theorem 3 and Theorem 4 was very straightforward and took about 400 lines of HOL code and about 2.5 man hours.

The above case studies clearly demonstrate the effectiveness of the proposed theorem proving based fractional order system analysis technique. Due to the formal nature of the model and inherent soundness of higher-order logic theorem proving, we have been able to verify the properties of given fractional order systems with 100% accuracy; a feature that, to the best of our knowledge, is not available in any other computer based analysis technique. This additional benefit comes at the cost of the time and effort spent, while formalizing the Differintegrals and formally reasoning about their properties. But, the availability of such a formalized infrastructure significantly reduces the time required to analyze fractional order systems, as the verification task of the properties of Resistoductance and a fractional integrator and differentiator circuits took just a couple of man hours each.

VI. Conclusions

In this paper, we presented a novel application of formal methods in the area of analyzing fractional order systems. In

particular, we developed a framework for accurate and reliable analysis of fractional order systems within the sound core of the HOL theorem prover. This approach can thus be of great benefit for the analysis of fractional order systems used in safety-critical domains, such as, medicine and transportation. The paper provides the complete formalization details of Differintegrals along with the formal verification of their classical properties. For illustration purposes, we provided the formal analysis of Resistoductance, a fractional differentiator and a fractional integrator circuit. To the best of our knowledge, this is the first time that a formal method technique has been used to conduct the analysis of fractional order systems.

The reported formalization opens the doors to many interesting and novel directions of research. Some worth mentioning ones include enriching the library of the formally verified properties of Differentagrals with law of exponents and relationship with Beta function to broaden the scope of formal fractional order system analysis. Similarly, the reported formalization can be utilized to formalize the fractional Laplace transform theory, which in turn can be utilized for the the formal analysis of industrial fractional order control systems. Our formalization was done using real numbers and the same formalization can also be extended to cover the complex numbers using the higher-order-logic formalization of complex number theory [16], which would allow us to formalize fractional electromagnetic systems, such as, fractional rectangular waveguides [13].

VII. Acknowledgement

This work was supported by the National Research Program for Universities grant (number 1543) of Higher Education Commission (HEC), Pakistan.

References

[1] N. Pariz A. Kiani-B, K. Fallahi and H. Leung. A Chaotic Secure Communication Scheme Using Fractional Chaotic Systems Based on an Extended Fractional Kalman Filter. *Communications in Nonlinear Science and Numerical Simulation*, 14:863–879, Elsevier, 2009.

[2] E. Artin. *The Gamma Function*. Athena Series, 1964.

[3] G. Baumann. Fractional Calculus and Symbolic Solution of Fractional Differential Equations. In *Fractals in Biology and Medicine*, Mathematics and Biosciences in Interaction, pages 287–298. Birkhuser Basel, 2005.

[4] G. W. Bohannan, S. K. Hurst, and L. Spangler. Electrical Components with Fractional Order Impedence. Patent, US 2006/0267595 A1, November 2006.

[5] Y. Q. Chen, I. Petras, and D. Xue. Fractional Order Control - A Tutorial. pages 1397–1411. American Control Conference, 2009.

[6] Y. Q. Chen, D. Xue, and H. Dou. Fractional Calculus and Biomimetic Control. In *Robotics and Biomimetics (ROBIO 2004), IEEE*, pages 901 –906, 2004.

[7] Q. Cheng, T. J. Cui and C. Zhang. Waves in Planar Waveguide Containing Chiral Nihility Metamaterial. *Optics and Communication*, 274:317–321, Elsevier, 2007.

[8] F. Yu D. J. Zhuang and Y. Lin. Evaluation of a Vehicle Directional Control with a Fractional Order pd^{λ} Controller. *International Journal of Automotive Technology*, 9(6):679–685, Springer, 2008.

[9] M. Dalir and M. Bashour. Application of Fractional Calculus. *Applied Mathematical Sciences*, 4(21):12, Hikari Ltd, 2010.

[10] S. Das. *Functional Fractional Calculus for System Identification and Controls*. Springer, 2007.

[11] K. Diethelm. *The Analysis of Fractional Differential Equations*. Springer-Verlag Berlin / Heidelberg, 1st edition, 2010.

[12] K. Diethelm, N. J. Ford, A. D. Freed, and Y. Luchko. Algorithms for the Fractional Calculus: A Selection of Numerical Methods. *Computer Methods in Applied Mechanics and Engineering*, 194(6-8):743 –773, Elsevier, 2005.

[13] M. Faryad and Q. A. Naqvi. Fractional Rectangular Waveguide. *Progress In Electromagnetics Research, PIER*, 75:383–396, 2007.

[14] M. J. C. Gordon and T. F. Melham, editors. *Introduction to HOL: A Theorem Proving Environment for Higher-Order Logic*. Cambridge University Press, 1993.

[15] J. Harrison. *Theorem Proving with the Real Numbers*. Springer-Verlag, 1998.

[16] J. Harrison. Formalizing Basic Complex Analysis. In *From Insight to Proof: Festschrift in Honour of Andrzej Trybulec*, volume 10(23) of *Studies in Logic, Grammar and Rhetoric*, pages 151–165. University of Białystok, 2007.

[17] T. T. Hartley and C. F. Lorenzo. Fractional System Identification: An Approach Using Continuous Order Distributions. Technical report, National Aeronautics and Space Administration Glenn Research Center NASA TM, 1999,.

[18] Uncertain Singular Expressions in Mathematica. http://reference.wolfram.com/mathematica/ref/FullSimplify.html, 2011.

[19] E. M. Clarke Jr., O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, 1999.

[20] W.M. Ahmad K. Assaleh. Modeling of Speech Signals Using Fractional Calculus. *9th International Symposium on Signal Processing and Its Applications,*, pages 1–4, 2007.

[21] B. T. Krishna and K. V. V. S. Reddy. Design of Digital Differentiators and Integrators of Order $\frac{1}{2}$. *World Journal of Modelling and Simulation, UK*, 4:182–187, World Academic Press, 2008.

[22] G. Maione and P. Lino. New Tuning Rules for Fractional pi^{α} Controllers. *Nonlinear Dynamics*, 49, Springer, 2007.

[23] M. D. G. Marcos, J. A. Tenreiro Machado, and T. P. Azevedo-Perdicoúlis. A Fractional Approach for the Motion Planning of Redundant and Hyper-Redundant Manipulators. *Signal Process.*, 91:562–570, March 2011.

[24] MATLAB. http://www.mathworks.com/products/matlab/, 2011.

[25] T. Mhamdi, O. Hasan, and S. Tahar. On the Formalization of the Lebesgue Integration Theory in HOL. In *Interactive Theorem Proving*, volume 6172 of *Lecture Notes in Computer Science*, pages 387–402. Springer, 2010.

[26] K. S. Miller and B. Ross. *An Introduction to Fractional Calculus and Fractional Differential Equations*. John Willey and Sons, Inc, 1993.

[27] A. Naqvi. Comments on Waves in Planar Waveguide Containing Chiral Nihility Metamaterial. *Optics and Communication*, 284:215–216, Elsevier, 2011.

[28] K. B. Oldham and J. Spanier. *The Fractional Calculus*. New York, Academic Press, 1974.

[29] B. Mathieu, P. Melchior, A. Oustaloup and Ch. Ceyral. Fractional Differentiation for Edge Detection. *Signal Processing*, 83:2421–2432, 2003.

[30] I Petras. Chaos in the Fractional-Order Volta's System: Modeling and Simulation. *Nonlinear Dynamics*, 57:157–170, 2009.

[31] I. Podlubny. Fractional Differential Equations, Academic Press. 1999.

[32] B. Ross. A Brief History And Exposition of The Fundamental Theory of Fractional Calculus. In *Fractional Calculus and Its Applications*, volume 457 of *Lecture Notes in Mathematics*, pages 1–36. Springer, 1975.

[33] U. Siddique. Formal Analysis of Fractional Order Systems - HOL Proof Script. http://save.seecs.nust.edu.pk/students/umair/fc.html, 2011.

[34] K. Slind and M. Norrish. A Brief Overview of HOL4. In *Theorem Proving in Higher Order Logics, TPHOLs*, volume 5170 of *Lecture Notes in Computer Science*, pages 28–32. Springer, 2008.

Static Scheduling of Latency Insensitive Designs with Lucy-n

Louis Mandel
LRI, Université Paris-Sud 11
INRIA Paris-Rocquencourt

Florence Plateau
LRI, Université Paris-Sud 11
Presently at Prove & Run

Marc Pouzet
DI, École Normale Supérieure
INRIA Paris-Rocquencourt

***Abstract*—Lucy-n is a data-flow programming language similar to Lustre extended with a buffer operator. It is based on the n-synchronous model which was initially introduced for programming multimedia streaming applications. In this article, we show that Lucy-n is also applicable to model Latency Insensitive Designs (LID). In order to model latency introduced by wires, we add a delay operator. Thanks to this new operator, a LID can be described by a Lucy-n program. Then, the Lucy-n compiler automatically provides static schedules for computation nodes and buffer sizes needed in shell wrappers.**

I. Introduction

The theory of Latency Insensitive Design ([1], [2]) was introduced to cope with the problem of long wires in Systems on Chips (SoC). Due to the length of wires, data can take more than a single clock cycle to go from one computation node (also known as Intellectual-Property or IP) to another. It raises the issues of activating IPs only when all inputs are available and storing the inputs awaiting to be processed. They are treated by encapsulating each computation node in a process, called a *shell wrapper*, that is used as a communication interface. In order to synchronize all the inputs, the shell wrappers have a buffer on each input wire. To respect the desired clock period, long wires are split into shorter segments by inserting relay nodes, called *relay stations*.

Different dynamic scheduling protocols for the shell wrappers have been proposed [2], [3]. Those protocols use back pressure mechanisms to inform the producer node that the consumer node has no more room to keep the inputs waiting to be processed. When a producer node is not executed, the shell wrapper has to inform the consumer node through a control channel that no valid inputs have been sent.

To avoid the communication overhead introduced by these dynamic protocols, static scheduling methods have been proposed [4], [5], [6], [7], [8]. Schedules are represented as ultimately periodic binary words indicating the instants where nodes have to be executed.

The schedules of [4], [5] and [6] fire the execution of the nodes As-Soon-As-Possible (ASAP). While in [7] a well-balanced schedule is sought in order to minimize buffer sizes for a fixed rate. This method is semi-automatic: prefixes of schedules must be found by hand. The approach of [8] is to search for schedules that can be shared between several IPs. The advantage of this method is that it reduces the complexity of the algorithm which finds the schedules and simplifies the circuits needed to generate them.

The approaches for finding schedules are either analytic [4], [7] or based on simulation [6], [8].

This paper presents a novel analytic way to compute static schedules for Latency Insensitive Designs by encoding LID models in the n-synchronous language Lucy-n [9], [10]. The schedules obtained by the Lucy-n compiler are not yet as good as the ones obtained by previous techniques. However, the advantage of our technique is to be able to compose modularly IPs that have already been statically scheduled. This can be useful for example when IP blocks are provided as black boxes. These Statically Scheduled IPs (SSIPs) have the particularity of not reading all their inputs nor writing all their outputs at each activation.

The paper is organized as follows. Section II presents the Lucy-n language. Section III explains how to encode LID models in this language and presents the new `delay` operator that is mandatory for the encoding. Section IV defines the typing of our new operator and presents a way of solving the typing constraints. Section V discuss the composition of SSIPs. Section VI presents experimental results. Finally, section VII concludes.

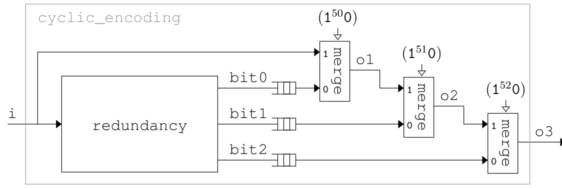
The Lucy-n compiler, the complete code of the paper examples and additional materials are available at http://www.lri.fr/~mandel/lucy-n/fmcad11.

II. A brief overview of Lucy-n

Lucy-n [10] is a programming language similar to the synchronous data-flow language Lustre [11] extended with a build-in buffer operator. The goal of this language is to relax synchrony constraints by inserting buffers without abandoning the guaranties given by synchrony, namely, determinism and execution in bounded time and memory. To this end, the compiler must compute both static schedules so that executions can be performed with finite buffer sizes and the buffer sizes themselves.

We present the language through the example of a cyclic encoder that takes as input a stream of bits and returns as output the same stream where after every 50 bits are added 3 redundancy bits. The program is given Figure 1.

The input flow `i` goes into a `redundancy` node that computes three flows of redundancy bits (line 2): `bit0`, `bit1`




```
let node cyclic_encoding i = o3 where
  rec (bit0, bit1, bit2) = redundancy i
  and o1 = merge (1^50 0) i (buffer bit0)
  and o2 = merge (1^51 0) o1 (buffer bit1)
  and o3 = merge (1^52 0) o2 (buffer bit2)
```

Fig. 1. Graphical and textual representation of a cyclic encoder in Lucy-n.

and `bit2` each producing 1 redundancy bit every 50 bits. The implementation of this node is based on a classical division circuit [12].

In parallel, the input flow `i` is merged with the first flow of redundancy bits `bit0`, following the condition $(1^{50}0)$ (line 3). It means that periodically 50 bits of `i` are read as input and transmitted as output in `o1`, then 1 bit of `bit0` is read and transmitted. Following the same principle, periodically, 51 bits of `o1` are merged with 1 bit of `bit1` to produce `o2` (line 4) and 52 bits of `o2` are merged with 1 bit of `bit2` to produce `o3` (line 5). The three flows `bit0`, `bit1` and `bit2` are bufferized such that their values are stored until the instant they are needed.

In synchronous languages, each flow is associated to a clock indicating the instants where a value is present. Clocks are infinite binary words where 1 represent the presence of a value on a flow and 0 the absence of value. Dedicated types, named clock types, specify this information. For example, the clock type of the node `redundancy` is: [1]

$$\forall\alpha.\,\alpha \texttt{ on } (1^{50}0) \rightarrow \alpha \texttt{ on } (0^{50}1) \times \alpha \texttt{ on } (0^{50}1) \times \alpha \texttt{ on } (0^{50}1)$$

Here, all the input and output types are expressed relative to a type variable α which represents the activation rhythm of the node `redundancy` and thus defines its notion of instant. The input type $\alpha \texttt{ on } (1^{50}0)$ means that whatever the base rhythm α, periodically, the input must be present for 50 instants of α, then absent for 1 instant. Each of the three outputs of the `redundancy` node emits a value on the last instant of every cycle of 51 instants.

Communication between two nodes is synchronous, *i.e.* it can be done without a buffer, if the flow is produced on the wire at the same clock that it is consumed. Equality of clocks is ensured by equality of types. So, such synchrony is guaranteed at compile time by a type system, called a clock calculus. For example, let us consider the typing rule for the `merge` operator where H is a typing environment which associates types to variables:

$$\frac{H \vdash ce : ct \quad H \vdash e_1 : ct \texttt{ on } ce \quad H \vdash e_2 : ct \texttt{ on not } ce}{H \vdash \texttt{merge } ce\; e_1\; e_2 : ct}$$

[1]In the rest of the article, we use the term *type* instead of *clock type* since data types are not considered here.

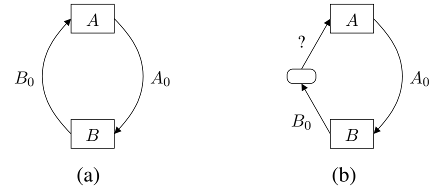


Fig. 2. First example of synchronous circuit and LID of [4].

This rule indicates that in the typing environment H, the expression `merge` $ce\; e_1\; e_2$ has type ct if: (1) the merging condition ce has type ct, (2) the expression e_1 has type $ct \texttt{ on } ce$ and (3) the expression e_2 has type $ct \texttt{ on not } ce$. It expresses the synchronous semantics of the `merge` operator: with respect to a rhythm ct, the presence instants of the flow e_2 (*i.e.* `not` ce) are the complement of the presence instants of the flow e_1 (*i.e.* ce), and the output flow of the `merge` is present at each instant of the reference rhythm ct.

In the n-synchronous language Lucy-n, the `buffer` operator relaxes the synchronous hypothesis by introducing of a subtyping rule to the clock calculus.

$$\frac{H \vdash e : ct \qquad ct <: ct'}{H \vdash \texttt{buffer } e : ct'}$$

This typing rule means that if an expression e has type ct and if clocks represented by ct are *adaptable* to clocks represented by ct', then we can use the results of e on the type ct' provided we store them in a buffer. The adaptability relation ensures that such buffers are bounded. It is denoted $w_1 <: w_2$ where w_1 is the clock of writes into the buffer and w_2 is the clock of reads, and it is defined as the conjunction of two conditions. First, no reads may occur on an empty buffer, *i.e.* the jth reading in a buffer must always occur after the jth writing. Second, there must be a bounded number of values in the buffer, *i.e.* the difference between the number of writes and reads since the beginning of an execution must be bounded. Theses conditions can be checked at compile time provided clocks are periodic.

The clock calculus automatically infers the type of the flows. For example, the type inferred for the entire `cyclic_encoding` node is: $\forall\alpha.\alpha \texttt{ on } (1^{50}0^3) \rightarrow \alpha$.[2] From such types, it is possible to build a static schedule for the program and to compute the buffer sizes needed. Here, the compiler finds that the node `cyclic_encoding` can be executed without buffering `bit0` and with buffers of size one for `bit1` and `bit2`. For more information about Lucy-n and its type system refer to [13], [10], [14].

III. LID Encoding in Lucy-n

To illustrate the encoding of Latency Insensitive Designs in Lucy-n, we use the first example given in [4]. We first model the synchronous circuit of Figure 2(a), and then we model the variation of Figure 2(b) where a relay station is added to model a communication latency from B to A.

[2]This type shows that the input must be absent during the insertion of the redundancy bits into the output.

A. Encoding of Computation Nodes

In the theory of LID, each computation node or IP reads all its input and produces all its outputs at each activation. Hence, IPs are encoded as Lucy-n nodes of clock $\forall\alpha.\,\alpha \times \cdots \times \alpha \rightarrow \alpha \times \cdots \times \alpha$. Since we do not have to know the behavior of the IPs to compute the scheduling of the system, the IPs can be represented as dummy nodes with the correct clock type. For example, the IPs of Figure 2 can be modeled as:

```
let node ip_A x = x
let node ip_B x = x
```

B. Encoding of Wires

In the synchronous circuits described in [4], data takes one clock cycle to go from one IP to another. Hence, a wire cannot be represented as a Lucy-n variable, because variables in synchronous languages model instantaneous communication channels. We thus model wires with a new operator, called `delay`, that transmits its input to its output with one instant of delay.[3] With this operator, a wire from a source `src` to a destination `dst` is written:

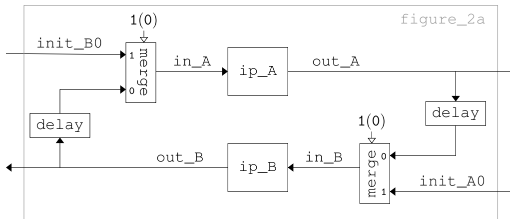
```
dst = delay src
```

To put an initial value on a wire as A_0 and B_0 in the example of Figure 2(a), we use the `merge` operator with the condition `1(0)`.

```
dst = merge 1(0) init (delay src)
```

The condition `1(0)` equals 1 at the first instant then 0 forever. So, at the first instant, the value `init` is transmitted to the destination `dst`. Then, thereafter, the values coming from the source `src` are transmitted.

Now, following the encoding of computation nodes and wires, we can program the example of Figure 2(a):




```
let node figure_2a (init_A0, init_B0) =
  (out_A, out_B) where
  rec out_A = ip_A in_A
  and out_B = ip_B in_B
  and in_A = merge 1(0) init_B0 (delay out_B)
  and in_B = merge 1(0) init_A0 (delay out_A)
```

In this node, `out_A` and `out_B` are the flows of values computed by the IPs A and B. The definitions of `in_A` and `in_B` describe the wires between A and B. The node `figure_2a` takes as inputs `init_A0` and `init_B0`, the initial values on the wires, and returns as outputs `out_A` and `out_B`, the values computed by the two IPs.

The Lucy-n compiler infers the following type for the node `figure_2a`:

$$\forall\alpha.\,\alpha \texttt{ on } 1(0) \times \alpha \texttt{ on } 1(0) \rightarrow \alpha \times \alpha$$

It means that the initial values need only be present at the first instant and that the two IPs produce some outputs at each instant. Since the presence instants of IPs outputs correspond to their activation instants, output types give the activation instants of the IPs, *i.e.* their static schedules. Here, we can verify that in the case of the synchronous circuits, all the IPs must be executed at the same instants.

C. Encoding of Shell Wrappers

In latency insensitive designs, each computation node is enclosed inside a shell wrapper that controls the activation of the node and bufferizes the inputs until this activation. To model this behavior, we need only put a buffer in front of each input and the activation condition will be automatically computed by the clock calculus. Hence, if we want to put the `ip_A` of the previous example in a shell wrapper, we only have to write:

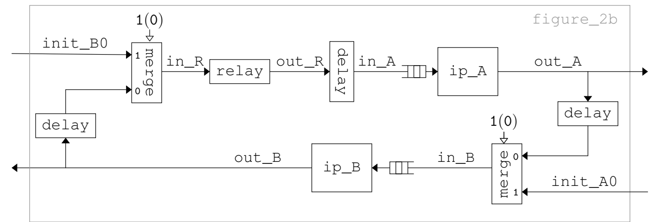
```
out_A = ip_A (buffer in_A)
```

D. Encoding of Relay Stations

A relay station is just a dummy node that splits a wire into two segments and thus introduces a delay.

```
let node relay x = x
```

Now that we have presented the encoding of computation nodes, wires, shell wrappers and relay stations, we can model the circuit of Figure 2(b):[4]




```
let node figure_2b (init_A0, init_B0) =
  (out_A, out_B) where
  rec out_A = ip_A (buffer in_A)
  and out_B = ip_B (buffer in_B)
  and out_R = relay in_R
  and in_A = delay out_R
  and in_B = merge 1(0) init_A0 (delay out_A)
  and in_R = merge 1(0) init_B0 (delay out_B)
```

The type of the node computed by the compiler is equal to $\forall\alpha.\,\alpha \texttt{ on } 1(0) \times \alpha \texttt{ on } 1(0) \rightarrow \alpha \texttt{ on } (01) \times \alpha \texttt{ on } (01)$. It means that the initial values are used only at the first instant and that the nodes A and B have to be executed every two instants from the second instant.

[3] Even if the `pre` operator of Lustre introduces a delay, it does not have exactly the same semantics as the one of `delay`. If we consider a flow `x` of clock (100), the values of `x` are output by `pre x` on the clock $000(100)$ whereas they are output by `delay x` on the clock $0(100)$. The `pre` operator outputs a value only when a new input arrives.

[4] Notice that there is no initial value on the wire between the relay station and the `ip_A` (line 6).

IV. Typing the delay operator

The typing rule for the `delay` operator is similar to the one for `buffer`. It introduces a constraint between the types of its input and output:

$$\frac{H \vdash e : ct \qquad ct' = \texttt{shiftr}\ ct}{H \vdash \texttt{delay}\ e : ct'}$$

The constraint $ct' = \texttt{shiftr}\ ct$ means that the clocks represented by ct' must be the same as the ones represented by ct delayed by one instant with respect to the activation rhythm of the node.

During typing, all the subtyping constraints introduced by the buffers and all the equality constraints introduced by the delays are collected. For example, the constraints generated by typing the node `figure_2b` are:

$$\left\{\begin{array}{lcl} \alpha_{in_A}\ \texttt{on}\ (1) & <: & \alpha_{out_A}\ \texttt{on}\ (1) \\ \alpha_{in_B}\ \texttt{on}\ (1) & <: & \alpha_{out_B}\ \texttt{on}\ (1) \\ \alpha_{in_A}\ \texttt{on}\ (1) & = & \texttt{shiftr}\ (\alpha_R\ \texttt{on}\ (1)) \\ \alpha_{in_B}\ \texttt{on}\ 0(1) & = & \texttt{shiftr}\ (\alpha_{out_A}\ \texttt{on}\ (1)) \\ \alpha_R\ \texttt{on}\ 0(1) & = & \texttt{shiftr}\ (\alpha_{out_B}\ \texttt{on}\ (1)) \end{array}\right\}$$

The collection and resolution of subtyping constraints is explained in [14]. Here, we extend this technique to cope with the constraints introduced by delays.

The resolution algorithm has the following structure:

1) Express all the constraints with respect to the same type variable. This variable represents the reference rhythm of the node. In our example, we state that $\alpha_{in_A} = \alpha\ \texttt{on}\ c_{in_A}$, $\alpha_{out_A} = \alpha\ \texttt{on}\ c_{out_A}$, $\alpha_{in_B} = \alpha\ \texttt{on}\ c_{in_B}$, $\alpha_{out_B} = \alpha\ \texttt{on}\ c_{out_B}$ and $\alpha_R = \alpha\ \texttt{on}\ c_R$ where c_{in_A}, c_{out_A}, c_{in_B}, c_{out_B} and c_R are unknown infinite binary words.
2) Since all constraints are now expressed with respect to the same type variable, simplify them into constraints over binary words. Here, we get:

$$\left\{\begin{array}{lcl} c_{in_A}\ \textit{on}\ (1) & <: & c_{out_A}\ \textit{on}\ (1) \\ c_{in_B}\ \textit{on}\ (1) & <: & c_{out_B}\ \textit{on}\ (1) \\ c_{in_A}\ \textit{on}\ (1) & = & 0(1)\ \textit{on}\ c_R\ \textit{on}\ (1) \\ c_{in_B}\ \textit{on}\ 0(1) & = & 0(1)\ \textit{on}\ c_{out_A}\ \textit{on}\ (1) \\ c_R\ \textit{on}\ 0(1) & = & 0(1)\ \textit{on}\ c_{out_B}\ \textit{on}\ (1) \end{array}\right\}$$

   and the variables c_{in_A}, c_{out_A}, c_{in_B}, c_{out_B} and c_R become the new unknowns of the system.
3) Translate constraints on words into linear constraints on integers representing the index of the `1`s of the unknown words of the system.
4) Solve these constraints using standard techniques.

The main difference with the former resolution algorithm is the presence of the `shiftr` operator. It does not affect step 1 of the algorithm. You can notice that at step 2, typing constraints of the form $\alpha_x\ \texttt{on}\ p_x = \texttt{shiftr}\ (\alpha_y\ \texttt{on}\ p_x)$ have become constraints on words of the form $c_x\ \textit{on}\ p_x = 0(1)\ \textit{on}\ c_y\ \textit{on}\ p_y$, where the *on* operator is defined as follows:

$$\begin{array}{lcl} 0.w_1\ \textit{on}\ w_2 & \stackrel{def}{=} & 0.(w_1\ \textit{on}\ w_2) \\ 1.w_1\ \textit{on}\ 1.w_2 & \stackrel{def}{=} & 1.(w_1\ \textit{on}\ w_2) \\ 1.w_1\ \textit{on}\ 0.w_2 & \stackrel{def}{=} & 0.(w_1\ \textit{on}\ w_2) \end{array}$$

If we consider w_1 and w_2 as clocks, the intuitive semantics of this operator is that $w_1\ \textit{on}\ w_2$ is the clock w_2 executed on the rhythm w_1. Therefore, $0(1)\ \textit{on}\ w$ is the clock w executed from the second instant. It corresponds to the clock w shifted by one instant.

Thanks to this translation, delays generate constraints that have the same nature as constraints coming from buffers and thus, steps 3 and 4 can be done similarly as before.[5]

V. Composition of Statically Scheduled IPs

To the best of our knowledge, the only work on the composition of Statically Scheduled IPs (SSIPs) is [15], where different composition techniques are proposed depending on the nature of SSIPs involved.

For SSIPs where the same number of values are consumed and produced on all ports, they propose using dynamic scheduling protocols like those used in the dynamic scheduling of LIDs. If it is not the case, they propose encoding the composition of SSIPs with Synchronous Data-Flow graphs (SDF) [16]. This encoding is similar to the encoding of Cyclo-Static SDF graphs [17] into SDF. It raises two problems. First, the initial phases of the schedules cannot be encoded. Second, the encoding is an abstraction where some information is lost and thus correct networks with cycles may be rejected.

In the context of Lucy-n, the composition of SSIPs is classical node composition. Indeed, in Lucy-n nodes, consumption and production patterns are arbitrarily complex ultimately periodic binary words, so SSIPs are Lucy-n nodes. We think that the main strength of our method is that it treats with IPs and SSIPs uniformly.

VI. Experiment

One of the LIDs that we have encoded is shown in Figure 3. It is the MPEG-2 video encoder from [2] where the relay stations are at the same places as in [4]. This program is compiled by the Lucy-n compiler in less than 0.1 seconds and has the following type:

$$\texttt{mpeg} :: \forall\alpha.\alpha\ \texttt{on}\ (10) \rightarrow \alpha\ \texttt{on}\ 0(10)$$

The throughput of the solution computed with our algorithm is $1/2$ whereas the one of the solution computed in [4] is $2/3$. Nevertheless, the programmer can give a hint to the compiler with an option `-nbones 2` asking it to seek a solution with twice as many `1`s in the periodic pattern of the schedule. With this option, the new solution is:[6]

$$\texttt{mpeg} :: \forall\alpha.\alpha\ \texttt{on}\ (110) \rightarrow \alpha\ \texttt{on}\ 00(101)$$

with a throughput of $2/3$.

[5] As $<:$ is antisymmetric, equality constraints can be translated into two inverse adaptability constraints.

[6] The compilation line is: `lucync -nbones 2 -obj r mpeg.ls`.

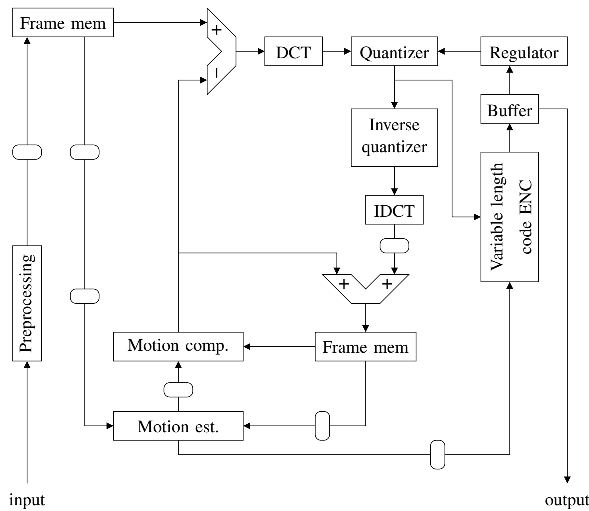


Fig. 3. MPEG-2 video encoder as found in [2], [4].

For the moment, only a few small-scale experiments have been performed, but thanks to our experiments on other Lucy-n programs [14], we think that it is possible to schedule systems with hundreds of elements.

VII. Conclusion

The contribution of this paper is the introduction of a delay operator to the language Lucy-n and a demonstration of its utility for modeling Latency Insensitive Designs. The benefit of modeling LIDs in Lucy-n is that it gives a new algorithm to automatically compute static schedules for the designs. The advantage of this method is that it allows designs that have already been scheduled to be incorporated in compositions.

We are working to improve the quality of the schedules. We can already influence the resolution algorithm by choosing the number of `1`s in the sought solution and by giving different objective functions during step 4 of the typing algorithm, for instance to privilege buffer sizes or throughput. Nevertheless, we do not yet have optimality results. Thus, we hope to adapt the results of the works cited in this article. Note, however, that our problem is more difficult because the consumption and production patterns are more complex. But, this is precisely why we can deal with IPs and SSIPs uniformly.

Acknowledgment

The authors would like to thank the anonymous reviewers for their useful remarks and Timothy Bourke for its careful reading of the paper.

References

[1] L. P. Carloni, K. L. McMillan, and A. L. Sangiovanni-Vincentelli, “Theory of Latency-Insensitive Design,” *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 20, no. 9, pp. 1059–1076, Sep. 2001.
[2] L. P. Carloni and A. L. Sangiovanni-Vincentelli, “Coping with Latency in SOC Design,” *IEEE Micro*, vol. 22, no. 5, pp. 24–35, 2002.
[3] J. Carmona, J. Cortadella, M. Kishinevsky, and A. Taubin, “Elastic circuits,” *IEEE Trans. on Computer-Aided Design*, vol. 28, no. 10, 2009.
[4] M. R. Casu and L. Macchiarulo, “A new approach to latency insensitive design,” in *Proc. of the Design Automation Conference*, 2004.
[5] J. Boucaron, “Modélisation formelle de systèmes insensibles à la latence et ordonnancement,” Ph.D. dissertation, Univ. de Nice Sophia-Antipolis, 2007.
[6] J. Boucaron, R. de Simone, and J.-V. Millo, “Formal Methods for Scheduling of Latency-Insensitive Designs,” *EURASIP Journal on Embedded Systems*, vol. 2007 , Issue 1, Jan. 2007.
[7] J.-V. Millo, “Ordonnancements Périodiques dans les Réseaux de Processus : Application à la Conception Insensible aux Latences,” Ph.D. dissertation, Univ. de Nice Sophia-Antipolis, 2008.
[8] J. Carmona, J. Júlvez, J. Cortadella, and M. Kishinevsky, “Scheduling synchronous elastic designs,” in *Int. Conf. on Application of Concurrency to System Design*, Jun. 2009.
[9] A. Cohen, M. Duranton, C. Eisenbeis, C. Pagetti, F. Plateau, and M. Pouzet, “N-Synchronous Kahn Networks: a relaxed model of synchrony for real-time systems,” in *ACM Int. Conf. on Principles of Programming Languages*, 2006.
[10] L. Mandel, F. Plateau, and M. Pouzet, “Lucy-n: a n-synchronous extension of Lustre,” in *Tenth Int. Conf. on Mathematics of Program Construction*, 2010.
[11] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud, “The synchronous dataflow programming language lustre,” *Proceedings of the IEEE*, vol. 79, no. 9, pp. 1305–1320, Sep. 1991.
[12] W. W. Peterson, *Error-Correcting Codes*. The M.I.T. Press, 1961.
[13] F. Plateau, “Modèle n-synchrone pour la programmation de réseaux de Kahn à mémoire bornée,” Ph.D. dissertation, Univ. Paris-Sud 11, 2010.
[14] L. Mandel and F. Plateau, “Typage des horloges périodiques en Lucy-n,” in *Journées Francophones des Langages Applicatifs*, 2011, English extended version available at http://www.lri.fr/~mandel/lucy-n/fmcad11.
[15] J. Boucaron and J.-V. Millo, “Compositionality of Statically Scheduled IP,” *ENTCS*, vol. 200, no. 1, pp. 71–87, May 2007.
[16] E. A. Lee and D. G. Messerschmitt, “Synchronous Data Flow,” *IEEE Trans. on Computers*, vol. 75, no. 9, Sep. 1987.
[17] T. M. Parks, J. L. Pino, and E. A. Lee, “A Comparison of Synchronous and Cyclo-Static Dataflow,” in *Proceedings of the 29th Asilomar Conference on Signals, Systems and Computers (2-Volume Set)*, 1995.

A Theory of Abstraction for Arrays

Steven M. German
IBM T.J. Watson Research Center

***Abstract*— We develop a theory for reasoning about temporal safety properties of systems with arrays. The theory leads to an automatic algorithm for constructing sound and complete abstractions. Our approach has advantages over previous approaches for important classes of digital designs, including designs with clock gating. We define a function that gives, in a certain sense, the size of the smallest sound and complete array abstraction of a system. This function is difficult to compute. However, we present a static analysis algorithm that efficiently computes a safe size of a sound and complete abstraction by overapproximating the minimum size. Our algorithm can often construct abstractions with small arrays for complex industrial designs.**

I. INTRODUCTION

Because of their large state spaces, arrays create a special difficulty for formal verification of hardware designs. In this paper we develop a theory that gives conditions under which a system with large arrays can be replaced by a system having smaller arrays, such that safety properies hold in the larger system iff they hold in the smaller one. The size of the arrays in the smaller system is determined by an efficient static analysis algorithm. An advantage of our approach is that it transforms a sequential system into another sequential system, allowing any verification method for sequential systems to be used after running our algorithm.

Previous researchers have developed several methods for reasoning about hardware systems with arrays, especially in the context of model checking. One general approach is to verify systems by considering behavior over a small, bounded number of time steps. Over a small number of time steps, a system performs only a small number of array read and write operations, and accesses only a small number array elements. Thus, bounded time modeling leads naturally to abstractions for arrays [1]–[4]. Behavior over a bounded number of time steps can be used to prove bounded correctness properties [5], or to prove unbounded properties by induction [6].

Abstracting arrays over bounded time intervals has the disadvantage that as the length of the time interval increases, the size of the array abstraction must be increased as well. For example, when proving properties by induction based on k-step unwindings, it is necessary to increase the length of the bounded interval until the property becomes inductive, which may make it necessary to consider large arrays [7].

A second general approach is to transform a sequential system with arrays into another sequential system with smaller arrays. Bjesse [7] developed an approach in which a sequential model with small arrays is constructed by abstraction refinement. One difference between our approach and [7] is that we use a static analysis algorithm instead of abstraction refinement. In our approach, the static analysis algorithm produces a single abstract model that is sound and complete for safety properties. Thus our approach eliminates the need for iterative abstraction refinement.

Another advantage of our approach is that we can build sound and complete abstractions for designs where a value read from an array can take an unbounded length of time before affecting an output signal. In [7], every abstract model of an array is characterized by a set of clock-cycle delays. Specifically, a small model of an array represents the results of reading the array at a finite set of fixed times prior to the clock cycle at which the value read from the array propagates to the system output. However, in many systems, the results of reading an array are stored for an unbounded amount of time in hardware registers or other arrays before being used to produce a system output. Hardware clock-gating [8] is one important design technique that leads to unbounded delays between the time an array is read and the time the array's value affects a system output. Our approach builds abstract models with unbounded delays, while the approach in [7] cannot reduce such models.

The outline of this paper is as follows. In Section II we define the mathematical framework for our theory. We define an operational semantics for executing systems that is appropriate when arrays have been replaced with smaller abstract arrays. In Section III we prove theorems showing the existence of sound and complete abstractions of systems with arrays. In Section IV we present an algorithm for analyzing a system to determine the size of a sound and complete array abstraction. In Section V we show how to build an abstracted version of a system, using the sizes for arrays determined by our theory. Section VI presents initial results of using our algorithm on industrial examples. Due to space limitations most proofs are omitted. A version of the paper with proofs will be published as a technical report, and will be available from the author and the IBM Research Division Libraries before FMCAD 2011.

II. PRELIMINARIES

We begin by defining the syntax and semantics of a term-level logic with arrays. In our logic, there are two kinds of variables: signal variables, and array variables. Let X_s be the set of signal variables and X_a be the set of array variables. We define signal expressions and array expressions to be the smallest sets of expressions satisfying the following definitions. A variable (resp., expression) is either a signal variable (resp., expression) or an array variable (resp., expression).

1) A signal variable is a signal expression.
2) If op is a k-ary operator symbol and $e_1, \ldots, e_k$ are signal expressions, then $op(e_1, \ldots, e_k)$ is a signal expression.
3) If e_1, e_2, e_3 are signal expressions, then $mux(e_1, e_2, e_3)$ is a signal expression.

4) An array variable is an array expression.
5) If a is an array expression and e is a signal expression, then $a[e]$ is a signal expression.
6) If a is an array expression and e_1, e_2 are signal expressions, then $write(a, e_1, e_2)$ is an array expression.

For the semantics, we assume the existence of a set V of at least two *signal values*. We assume that V is finite, but much of the theory is true even if V is infinite. Let $0, 1$ be distinct signal values. We assume that for each domain value $i \in V$, there is a constant symbol c_i such that c_i evaluates to i. For $k \geq 0$, a k-ary operator *op* is a symbol whose interpretation is a function $OP : V^k \to V$.

An array will be abstracted by replacing it with an array having a smaller domain. We will need to give meaning to array access expressions $a[i]$, where the value of i is not in the domain of a. For this purpose, we introduce a bottom value $\bot \notin V$. Let $V^+ = V \cup \{\bot\}$. We will define a semantics that propagates the bottom value onward, starting from an array access that is outside the domain of the array. In the case of mux expressions, an expression will have a value in V provided the first argument has a value in $\{0, 1\}$ and the selected argument of the mux has a value in V.

In the semantics of our logic, a signal variable is a name that can be assigned signal values, and an array variable is a name that can be assigned array values. An array value is a function in $V \to V^+$. We explicitly allow array values to be partial functions whose domains are not equal to the entire set of signal values V. We will use array values that are partial functions on V to reason about systems containing an array without representing all of the array's elements. An array value v is said to be *pure* if $\forall x \in \mathrm{dom}(v) : v(x) \in V$.

A *state* assigns values to variables. A state assigns a signal value to each signal variable, and assigns a pure array value to each array variable. Note that states only contain the values in V, not the bottom value. In sequential systems, which will be defined shortly, we will use states only to represent the initial values of state variables and the values of input signals. The reason that states do not contain values with $\bot$ is that these values do not represent initial states or input values; the bottom value is only produced during evaluation of certain expressions.

The *domain* of an array expression a in a state σ, $\mathrm{D}(a, \sigma)$, is a set of index values for the array expression. For an array variable a, $\mathrm{D}(a, \sigma)$ is defined as $\mathrm{dom}(\sigma(a))$, the domain of the function value assigned to a. Array write operations do not change the domain of an array. The domain of $write(a_1, e_1, e_2)$ in state σ is inductively defined by $\mathrm{D}(write(a_1, e_1, e_2), \sigma) = \mathrm{D}(a_1, \sigma)$. We will need the notion of the *root* of an array expression. For an array variable a, $\mathrm{root}(a) = a$. The root of a write expression is $\mathrm{root}(write(b, e_1, e_2)) = \mathrm{root}(b)$.

We define the value of an expression exp with respect to a state σ, written $\sigma[\![exp]\!]$, as follows. In the following definition, $e, e_1, e_2, \ldots,$ are signal expressions and a is an array expression.

1) $\sigma[\![v]\!] = \sigma(v)$, where v is a signal variable.

2) $\sigma[\![op(e_1, \ldots, e_n)]\!] =$

$$\begin{cases} OP(\sigma[\![e_1]\!], \ldots, \sigma[\![e_n]\!]), & \text{if } \sigma[\![e_i]\!] \neq \bot, \text{ for } i = 1, \ldots, n, \text{ where } OP \text{ is the interpretation of the operator symbol } op \\ \bot & \text{if for some } i,\ \sigma[\![e_i]\!] = \bot \end{cases}$$

3) $$\sigma[\![mux(e_1, e_2, e_3)]\!] = \begin{cases} \sigma[\![e_2]\!] & \text{if } \sigma[\![e_1]\!] = 0 \\ \sigma[\![e_3]\!] & \text{if } \sigma[\![e_1]\!] = 1 \\ \bot & \text{if } \sigma[\![e_1]\!] \notin \{0, 1\} \end{cases}$$

4) $$\sigma[\![a[e]]\!] = \begin{cases} (\sigma[\![a]\!])(\sigma[\![e]\!]) & \text{if } \sigma[\![e]\!] \in \mathrm{D}(a, \sigma) \\ \bot & \text{if } \sigma[\![e]\!] \notin \mathrm{D}(a, \sigma) \end{cases}$$

5) $\sigma[\![a]\!] = \sigma(a)$, where a is an array variable.

6) $\sigma[\![write(a, e_1, e_2)]\!] =$

$$\begin{cases} (\sigma[\![a]\!])\,[\sigma[\![e_1]\!] \leftarrow \sigma[\![e_2]\!]] & \text{if } \sigma[\![e_1]\!] \in \mathrm{D}(a, \sigma) \\ \sigma[\![a]\!] & \text{if } \sigma[\![e_1]\!] \in V - \mathrm{D}(a, \sigma) \\ \mathrm{bottom}(a, \sigma) & \text{if } \sigma[\![e_1]\!] = \bot \end{cases}$$

Expressions of the form $op(e_1, \ldots, e_n)$ can be used to represent blocks of combinational logic containing many gates. In the semantics, an *op* expression has value $\bot$ whenever any of the input signals has value $\bot$. The advantage of this semantics is that it allows us to define a circuit that computes $\sigma[\![op(e_1, \ldots, e_n)]\!]$, without having to add signals to express whether the output of each gate in a large block has value $\bot$. We will use such circuits in building our abstract models. The abstract model can have fewer gates because only the inputs and output of a large block need to consider $\bot$.

In the array write expression $write(a, e_1, e_2)$, e_1 is the address and e_2 is the value written. There are three cases in the semantics of array writes. The first case updates a single element of an array when e_1 has a value in the domain of the array. In the second case, e_1 has a value in V, but the value is outside the domain of the array. In this case, the value of the array is not changed by the write operation. Note that the operation of writing to an array does not extend the domain of the array. In the third case, the index e_1 has the value $\bot$. For an array expression a and a state σ, we define $\mathrm{bottom}(a, \sigma)$ to be an array value, the function that maps all elements of $\mathrm{D}(a, \sigma)$ to $\bot$. The intuition is that if we write to an address e_1 that has value $\bot$, then it cannot be determined which element of the array is changed, so all elements are marked as having value $\bot$.

Electronic designs sometimes have arrays where writing is conditional. For example, writing can be controlled by an enable signal, with value 1 to indicate writing a new value. Conditional writing can be modelled by write expressions such as $write(a, address, mux(enable, a[address], new_value))$. This expression produces an unchanged array value when $enable = 0$ and $address$ is a value in the domain of the array, and writes a new value when $enable = 1$.

A signal expression e is said to be *satisfied* by a state σ if $\sigma[\![e]\!] = 1$. We write $\sigma \models e$ to denote that σ satisfies e. A signal expression e is said to be satisfiable if there exists a state that satisfies e, and e is said to be valid if it is satisfied by all states.

A *system* $\mathcal{M}$ has the form $(\mathcal{S}, \mathcal{I}, \mathcal{N}, \mathcal{O}, \mathcal{E})$. $\mathcal{S}$ is a set of signal and array variables forming the state variables of the system. $\mathcal{I}$ is a set of input signal variables. $\mathcal{N}$ defines the next-state function of the system. $\mathcal{N}$ is a function from the variable names in $\mathcal{S}$ to expressions, such that if e is a signal variable, then $\mathcal{N}(e)$ is a signal expression, and if a is an array variable, then $\mathcal{N}(a)$ is an array expression. For an array variable a, we require that $\mathrm{root}(\mathcal{N}(a)) = a$; that is, the next state expression for an array variable a must be formed by a sequence of writes to a. $\mathcal{O}$ is a set of signal variables that are the outputs of the system, and $\mathcal{E}$ is a function mapping variables in $\mathcal{O}$ to signal expressions. The sets $\mathcal{S}, \mathcal{I}, \mathcal{O}$ must be pairwise disjoint.

We define the executions of a system by giving an operational semantics based on expansions of the next-state functions for state variables. Given a system $\mathcal{M}$ and a state variable $s \in \mathcal{S}$, we define

$$\begin{array}{lcl} s^0 & = & s, \\ s^{k+1} & = & \mathcal{N}(s)[\mathcal{S}/\mathcal{S}^k,\ \mathcal{I}/\mathcal{I}^k],\ \text{for } k = 0, 1, \ldots, \end{array}$$

where $\mathcal{S}/\mathcal{S}^k$ replaces each variable s_j in $\mathcal{S}$ with the expression $(s_j)^k$, and $\mathcal{I}/\mathcal{I}^k$ replaces each input variable u in $\mathcal{I}$ with a fresh signal variable u^k. For an output variable $v \in \mathcal{O}$, the output value at step k depends on the state and input variables at step k,

$$v^k \ = \ \mathcal{E}(v)[\mathcal{S}/\mathcal{S}^k,\ \mathcal{I}/\mathcal{I}^k],\ \text{for } k = 0, 1, \ldots$$

For an arbitrary expression e over the variables of $\mathcal{M}$, we define $e^k = e[\mathcal{S}/\mathcal{S}^k, \mathcal{I}/\mathcal{I}^k, \mathcal{O}/\mathcal{O}^k]$.

The safety properties of a system are specified by its output signals. For an output signal v of a system $(\mathcal{S}, \mathcal{I}, \mathcal{N}, \mathcal{O}, \mathcal{E})$, v is said to be valid with respect to the system iff for all states σ, $\sigma \models v^k$, for all $k \geq 0$.

Our theory does not prescribe an approach to variable typing. Instead, we allow the set of initial states of a system to be a parameter of the theory. It will be useful to make two considerations about the set of initial states of a system. First, it is conventional to assume that an array variable has the same set of indices in any system state. We formalize this by saying that a *type* $\mathcal{T}$ is a set of states such that for any array variable a and $\sigma_1, \sigma_2 \in \mathcal{T}$, $\mathrm{dom}(\sigma_1(a)) = \mathrm{dom}(\sigma_2(a))$. We will evaluate correctness of systems over sets of initial states that are types. As mentioned previously, the dimension of an array does not change dynamically.

Second, because the value $\perp$ has a special meaning in our theory, care is needed when translating a design from a hardware language into our theory. We need to make sure that a $\perp$ value cannot be generated accidentally because the original design subscripts an array outside of the declared domain. One way is to define the design in a language where array subscripts can be checked statically, and then translate a checked design into our theory. Another approach is to translate an array read $a[i]$ in the hardware design into an expression $mux(lower \leq i \wedge i \leq upper, a[i], nondet)$, in our theory. Here, the value of $a[i]$ is used if i is in the declared domain, and otherwise a nondeterminsitic input value $nondet$ is used.

In our theory, we can express the notion that a design is well-typed by introducing an assumption that the set of initial states has the property that all the expressions generated by the operational semantics of a system produce values in V. We say that a state σ is *safe* with respect to an expression e iff $\sigma[\![e]\!] \in V$ and for every subexpression e' of e, $\sigma[\![e']\!] \in V$. For a state to be safe is a stronger condition than just saying the output has a value in V. We say that a type $\mathcal{T}$ is safe for a system $\mathcal{M}$ iff for all state or output variables v of $\mathcal{M}$, for all states $\sigma \in \mathcal{T}$, and for all $k \geq 0$, σ is safe with respect to v^k. We define the notion of correctness for safety properties by evaluating variables of a system in all states of a safe type. We say that a variable v is valid in a system $\mathcal{M}$ and initial state set $\mathcal{T}$, written $\mathcal{M}, \mathcal{T} \models v$, iff for all states $\sigma \in \mathcal{T}$, for all $k \geq 0$, $\sigma[\![v^k]\!] = 1$.

III. Existence of Array Abstractions

In this section we show that under certain conditions, there are small abstract models for systems with arrays. The abstract models are sound and complete for safety properties.

Because arrays and multiplexors propagate values only from the selected input, it is possible for an expression to have a value in V even if some array accesses have value $\perp$. To capture the notion that some, but not all, expressions must have values in V in order to compute the value of a larger expression, we define the set of *essential expressions* of an expression e in a state σ, written $\mathrm{eexp}(e, \sigma)$. See Figure 1. There are four cases for write expressions: Case 1 is when e_3 is not a valid index; Case 2 is when e_1 is not a valid index; Case 3 is when e_1, e_2 are valid indices with different values, so that $write(b, e_1, e_2)[e_3] = b[e_3]$; Case 4 is when $write(b, e_1, e_2)[e_3] = e_2$. Under the assumption that all of the essential expressions in $\mathrm{eexp}(e, \sigma)$, not including e itself, evaluate to values in V, then $\sigma[\![e]\!] \in V$. In reading the definition of $\mathrm{eexp}(e, \sigma)$, it is important to note that e is always an essential expression of itself. Also, the definition of eexp applies a case-splitting rule to array read operations with nested array writes. Because of the case-splitting rule, $\mathrm{eexp}(e, \sigma)$ can contain expressions that are not subexpressions of e.

Lemma 1. Let e be a signal expression and σ be a state. Then $\sigma[\![e]\!] \in V$, iff for all essential expressions f of e in σ, $\sigma[\![f]\!] \in V$. □

The set of *essential indices* of an array variable a with respect to an expression e and a state σ is the set of values in V of signal expressions f such that $b[f]$ is an essential expression, where b is an array expression with $\mathrm{root}(b) = a$.

Formally, we define

$$\begin{array}{l} \mathrm{eindx}(e, \sigma, a) = \\ \{\sigma[\![f]\!] \mid \sigma[\![f]\!] \in \mathrm{D}(a, \sigma) \wedge \exists\ b : b[f] \in \mathrm{eexp}(e, \sigma), \\ \ \text{where } b \text{ is an array expression and } \mathrm{root}(b) = a\}. \end{array}$$

As an example of essential expressions and indices, consider the expression $write(a, e, a[4])[5]$. In a state σ where $\sigma[\![e]\!] = 5$, the essential expressions are: $write(a, e, a[4])[5]$, e, and $a[4]$; the essential indices of the array a are: 4 and 5. Intuitively, the value of the expression is obtained by evaluating $a[4]$ when the value of $e = 5$. In a state σ where $\sigma[\![e]\!] \neq 5$, the essential expressions are: $write(a, e, a[4])[5]$,

$$
\begin{aligned}
&\text{eexp}(e,\sigma) = \\
&\quad \text{if } e \text{ is a signal variable} \Rightarrow \{e\} \\
&\quad \text{if } e \text{ is } op(e_1,\ldots,e_n) \Rightarrow \{e\} \cup \text{eexp}(e_1,\sigma) \cup \ldots \cup \text{eexp}(e_n,\sigma) \\
&\quad \text{if } e \text{ is } mux(e_1,e_2,e_3) \Rightarrow \\
&\quad \begin{cases} \text{if } \sigma[\![e_1]\!] \notin \{0,1\} \Rightarrow \{e\} \cup \text{eexp}(e_1,\sigma) \\ \text{if } \sigma[\![e_1]\!] = 0 \Rightarrow \{e\} \cup \text{eexp}(e_1,\sigma) \cup \text{eexp}(e_2,\sigma) \\ \text{if } \sigma[\![e_1]\!] = 1 \Rightarrow \{e\} \cup \text{eexp}(e_1,\sigma) \cup \text{eexp}(e_3,\sigma) \end{cases} \\
&\quad \text{if } e \text{ is } b[e_1] \text{, where } b \text{ is an array variable} \Rightarrow \{e\} \cup \text{eexp}(e_1,\sigma) \\
&\quad \text{if } e \text{ is } write(b,e_1,e_2)[e_3] \Rightarrow \\
&\quad \begin{cases} \text{if } \sigma[\![e_3]\!] \notin \text{D}(b,\sigma) \Rightarrow \{e\} \cup \text{eexp}(e_3,\sigma) \\ \text{if } \sigma[\![e_3]\!] \in \text{D}(b,\sigma) \wedge \sigma[\![e_1]\!] = \bot \Rightarrow \{e\} \cup \text{eexp}(e_1,\sigma) \cup \text{eexp}(e_3,\sigma) \\ \text{if } \sigma[\![e_3]\!] \in \text{D}(b,\sigma) \wedge \sigma[\![e_1]\!] \in V \wedge \sigma[\![e_1]\!] \neq \sigma[\![e_3]\!] \Rightarrow \{e\} \cup \text{eexp}(e_1,\sigma) \cup \text{eexp}(e_3,\sigma) \cup \text{eexp}(b[e_3],\sigma) \\ \text{if } \sigma[\![e_3]\!] \in \text{D}(b,\sigma) \wedge \sigma[\![e_1]\!] = \sigma[\![e_3]\!] \Rightarrow \{e\} \cup \text{eexp}(e_1,\sigma) \cup \text{eexp}(e_2,\sigma) \cup \text{eexp}(e_3,\sigma) \end{cases}
\end{aligned}
$$

Fig. 1. Definition of essential expressions eexp

e, and $a[5]$; the essential indices of a are: 5. In this case, the value of the expression is obtained by evaluating $a[5]$.

For states σ, σ', we say that σ' is a *substate* of σ, written $\sigma' \leq \sigma$ iff

1) For all signal variables v, $\sigma'(v) = \sigma(v)$, and
2) For all array variables a, $\text{dom}(\sigma'(a)) \subseteq \text{dom}(\sigma(a)) \wedge \forall i \in \text{dom}(\sigma'(a)) : \sigma'(a)(i) = \sigma(a)(i)$.

The following lemma says that if an expression e evaluates to a value in V in a state σ, then there is a substate $\sigma' \leq \sigma$, that gives e the same value and such that each array variable is only defined over the essential indices of the variable in σ. That is, for all array variables a, $\text{dom}(\sigma'(a)) = \text{eindx}(e,\sigma,a)$. This lemma will allow us to replace arrays by smaller abstractions.

Lemma 2. Let e be a signal expression and σ be a state such that $\sigma[\![e]\!] \in V$. Then there exists a state σ' such that $\sigma' \leq \sigma$, for all array variables a, $\text{dom}(\sigma'(a)) = \text{eindx}(e,\sigma,a)$, and $\sigma[\![e]\!] = \sigma'[\![e]\!]$. □

IV. Size of Array Abstractions

We define the size of a state σ, written $|\sigma|$, to be the function mapping each array variable a to the size of the domain of a: for all array variables a, $|\sigma|(a) = |\text{dom}(\sigma(a))|$. Similarly, we define the size of a type $\mathcal{T}$ to be $|\sigma|$, for any state $\sigma \in \mathcal{T}$.

From Lemma 2, we see that if $\sigma[\![e]\!] = v$, for $v \in V$, then there is a state σ' such that for each array variable a, the size of array a is $|\text{eindx}(e,\sigma,a)|$, and $\sigma'[\![e]\!] = v$. Now, suppose $\mathcal{U}$ is any set of states, and we want to evaluate an expression e over all states in $\mathcal{U}$. It is sufficient to evaluate e over all states where the size of each array a is the maximum size needed for any state. This value is given by the function $\Sigma_{\mathcal{U}} : \textit{expressions} \rightarrow (X_a \rightarrow \mathbb{N})$, where

$$\Sigma_{\mathcal{U}}(e)(a) = \max_{\sigma \in \mathcal{U}} |\text{eindx}(e,\sigma,a)|.$$

With this definition, $\Sigma_{\mathcal{U}}(e) : X_a \rightarrow \mathbb{N}$ is a function that encapsulates all of the sizes of arrays needed to evaluate the expression e. The function $\Sigma_{\mathcal{U}}(e)(a)$ always has a defined value when V is finite, because the value of eindx is a subset of V. When $\mathcal{U}$ is the set of all states over V, we drop the subscript and write $\Sigma(e)(a)$. $\Sigma(e)(a)$ gives an upper bound on the size of arrays needed to test if an expression is satisfiable in any state.

Proposition 1. Let σ', σ be states such that $\sigma' \leq \sigma$. Let e be a signal expression and let $i \in V$. Then the following three conditions hold:

1) $\sigma[\![e]\!] = i \Rightarrow (\sigma'[\![e]\!] = i \vee \sigma'[\![e]\!] = \bot)$
2) $\sigma[\![e]\!] = \bot \Rightarrow \sigma'[\![e]\!] = \bot$
3) $\sigma'[\![e]\!] = i \Rightarrow \sigma[\![e]\!] = i$ □

Proposition 2. Let e be an expression, σ be a state, and a be an array variable. Then $|\text{eindx}(e,\sigma,a)| \leq |\text{dom}(\sigma(a))|$. This is true because in the definition of eindx, each element of $\text{eindx}(e,\sigma,a)$ must be an element of $\text{dom}(\sigma(a))$. □

Theorem 1: Small Model Theorem. If a signal expression e is satisfiable, there is a state σ that satisfies e such that $|\sigma| = \Sigma(e)$.

Proof. Let σ be a state that satisfies e. By Lemma 2 there is a state σ' such that σ' satisfies e, and for all a, $|\text{dom}(\sigma'(a))| \leq \Sigma(e)(a)$. From Propositions 1.3 and 2, it follows that σ' can be expanded to a state σ'' such that $\sigma' \leq \sigma''$, $|\sigma''| = \Sigma(e)$, and σ'' satisfies e. □

We define an upper bound for a system $\mathcal{M}$ and a set of states $\mathcal{U}$ by a function $\Sigma^*_{\mathcal{M},\mathcal{U}} : X_s \rightarrow (X_a \rightarrow \mathbb{N})$,

$$\Sigma^*_{\mathcal{M},\mathcal{U}}(v)(a) = \max_{k=0,1,\ldots} \Sigma_{\mathcal{U}}(v^k)(a),$$

where v is a signal variable and a is an array variable. The value of $\Sigma^*_{\mathcal{M},\mathcal{U}}(v)(a)$ is an upper bound on the number of index values of the array a needed to evaluate all of the expansions of the signal variable v over all states in $\mathcal{U}$. Like $\Sigma_{\mathcal{U}}(e)(a)$, the function $\Sigma^*_{\mathcal{M},\mathcal{U}}(v)(a)$ has a defined value when V is finite. When $\mathcal{U}$ is the set of all states, we drop the second subscript and write $\Sigma^*_{\mathcal{M}}(v)(a)$.

The following theorem says that it is sound and complete to reason about an output variable v as a safety property of a system, by evaluating v in all states of size $\Sigma^*_{\mathcal{M},\mathcal{T}}(v)$. In the statement of the theorem, $\mathcal{T}$ is a set of states for the unabstracted model. We assume that $\mathcal{T}$ is safe, so that all expressions in the executions of the system can be evaluated. The theorem says that if we evaluate the truth of a safety property v over all states σ', such that σ' is a substate of some state in $\mathcal{T}$, and the size of σ' is $\Sigma^*_{\mathcal{M},\mathcal{T}}(v)$, then we can determine the result of evaluating v over all states in the unabstracted model over the set of states $\mathcal{T}$. The theorem provides a sound and complete method for reasoning about safety properties while reducing the size of arrays.

Theorem 2.1. Let $\mathcal{M}$ be a system with output variable v and let $\mathcal{T}$ be a safe type for $\mathcal{M}$. Let

$$\mathcal{T}' = \{\sigma' \mid \exists \sigma \in \mathcal{T} : \sigma' \leq \sigma \ \wedge \ |\sigma'| = \Sigma^*_{\mathcal{M},\mathcal{T}}(v)\}$$

Then $\mathcal{M}, \mathcal{T} \models v$ iff $\forall k \geq 0$, $\forall \sigma' \in \mathcal{T}' : \sigma'[\![v^k]\!] = 1 \vee \sigma'[\![v^k]\!] = \bot$.

Proof. $(\Rightarrow)$ $\mathcal{M}, \mathcal{T} \models v$ means $\forall k \geq 0$, $\forall \sigma \in \mathcal{T}$, $\sigma[\![v^k]\!] = 1$. If $k \geq 0$, $\sigma \in \mathcal{T}$, and $\sigma' \leq \sigma$, then by Proposition 1.1, $\sigma'[\![v^k]\!] = 1$ or $\sigma'[\![v^k]\!] = \bot$.

$(\Leftarrow)$ What we need to show is that if there is a counterexample for some state in $\mathcal{T}$ in the unabstracted system, then there is a counterexample in a state in $\mathcal{T}'$ in the abstracted system. Suppose there is a counterexample: let $k \geq 0$ and $\sigma \in \mathcal{T}$ be such that $i \in V$, $i \neq 1$ and $\sigma[\![v^k]\!] = i$. By Lemma 2, we know there is a state σ_1, such that $\sigma_1 \leq \sigma$, for all array variables a, $\mathrm{dom}(\sigma_1(a)) = \mathrm{eindx}(v^k, \sigma, a)$, and $\sigma_1[\![v^k]\!] = i$. By definition, $\Sigma^*_{\mathcal{M},\mathcal{T}}(v)(a)$ takes the maximum value of $|\mathrm{eindx}(v^k, \sigma, a)|$ over all $k \geq 0$ and $\sigma \in \mathcal{T}$, so that for all a, $|\mathrm{dom}(\sigma_1(a))| \leq \Sigma^*_{\mathcal{M},\mathcal{T}}(v)(a)$. By Proposition 2, for all a, $\Sigma^*_{\mathcal{M},\mathcal{T}}(v)(a) \leq |\mathrm{dom}(\sigma(a))|$, since $\sigma \in \mathcal{T}$. Therefore, there is a state σ_2, such that $\sigma_1 \leq \sigma_2 \leq \sigma$ and $|\sigma_2| = \Sigma^*_{\mathcal{M},\mathcal{T}}(v)$. Note that $\sigma_2 \in \mathcal{T}'$. By Proposition 1.3, $\sigma_2[\![v^k]\!] = i$. Therefore, σ_2 is a counterexample in $\mathcal{T}'$.

We are now ready to complete the proof. If it is the case that for all $k \geq 0$ and $\sigma' \in \mathcal{T}'$, $\sigma'[\![v^k]\!] = 1 \vee \sigma'[\![v^k]\!] = \bot$, then there cannot be a counterexample to the truth of v over all states in $\mathcal{T}$. Since $\mathcal{T}$ is a safe type, it must be the case that $\forall k \geq 0$, $\forall \sigma \in \mathcal{T}$, $\sigma[\![v^k]\!] = 1$. □

In practice, it is difficult to evaluate $\Sigma^*_{\mathcal{M},\mathcal{T}}(v)(a)$, since this involves finding a maximum value over all states in $\mathcal{T}$ and over all computation steps. However, we show later in the paper that there are ways to compute an upper bound on $\Sigma^*_{\mathcal{M}}(v)(a)$. Computing an upper bound is easier than computing the exact value, and also it is easier to compute an upper bound over all states instead of over a set $\mathcal{T}$. Since $\Sigma^*_{\mathcal{M}}(v)(a)$ is a maximum over all states, $\Sigma^*_{\mathcal{M},\mathcal{T}}(v)(a) \leq \Sigma^*_{\mathcal{M}}(v)(a)$ for all a. Note that it is possible for $\Sigma^*_{\mathcal{M}}(v)(a)$ to be larger than the size of a in $\mathcal{T}$. To prove properties over the set of states $\mathcal{T}$, we would not want to make the size each array variable a equal to $\Sigma^*_{\mathcal{M}}(v)(a)$. Instead we make the size of each array variable a equal to the minimum of $\Sigma^*_{\mathcal{M}}(v)(a)$ and the size of a in $\mathcal{T}$. Let v be a fixed variable name, and define a function $\mu : X_a \rightarrow \mathbb{N}$,

$$\mu(a) = \min(|\mathcal{T}|(a),\ \Sigma^*_{\mathcal{M}}(v)(a))$$

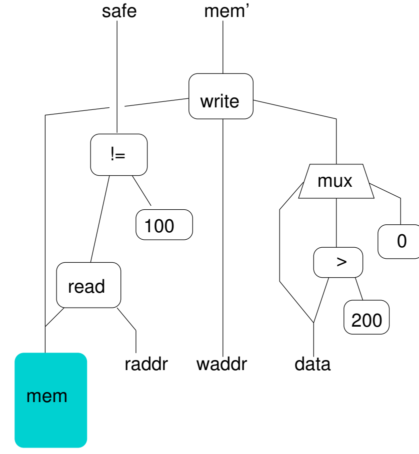


Fig. 2. Example 1

Let $\mathcal{T}''$ be the set of states that are smaller than states in $\mathcal{T}$ and of size μ,

$$\mathcal{T}'' = \{\sigma' \mid \exists \sigma \in \mathcal{T} : \sigma' \leq \sigma \ \wedge \ |\sigma'| = \mu\}$$

Since the size of arrays in $\mathcal{T}''$ are at least as large as in $\mathcal{T}'$ defined in Theorem 2.1, it follows from Proposition 1 that Theorem 2.1 holds if we use $\mathcal{T}''$ in place of $\mathcal{T}'$.

Theorem 2.2. Let $\mathcal{M}$ be a system with output variable v, let $\mathcal{T}$ be a safe type for $\mathcal{M}$ and let $\mathcal{T}''$ be as defined above. Then $\mathcal{M}, \mathcal{T} \models v$ iff $\forall k \geq 0$, $\forall \sigma' \in \mathcal{T}'' : \sigma'[\![v^k]\!] = 1 \vee \sigma'[\![v^k]\!] = \bot$. □

In an implementation of Theorem 2.2, a model would be constructed with each array variable a having a domain of size $\mu(a)$. As an example, if the original model has an array a with domain [1..100], and the size of the domain of a is 2 in the abstract model, then we need to evaluate the abstract model over all states where the domain of a is any two elements of [1..100]. As in [7], a value for the address of each row in the abstract array is chosen nondeterministically at the start of the run. The read and write operations in the abstract model use the address chosen for each of the rows, instead indexing into the array. The implementation then uses model checking to evaluate for all $\sigma \in \mathcal{T}''$, $\sigma[\![v^0]\!], \sigma[\![v^1]\!]$, and so on. If there is a state $\sigma \in \mathcal{T}$ such that for some k, $\sigma[\![v^k]\!]$ evaluates to a value $x \in V$ other than 1, then there is a state σ' such that $\sigma' \leq \sigma$, $|\sigma'| = \mu$, and $\sigma'[\![v^k]\!] = x$.

Example 1. This example appeared in [7]. On each clock cycle, the design inputs values for the signals `raddr`, `waddr`, and `data`. The array `mem` is an array state variable. A network with a multiplexor produces an output value that depends on the value of the input `data`. If `data` > 200, the multiplexor outputs the value of `data`; otherwise it outputs the value 0. On each clock cycle, the array `mem` is written at address `waddr` and with the value output from the multiplexor. On each clock cycle, the array `mem` is read at address `raddr`. The correctness property asserts that the value read from the array is never equal to 100.

Here, we render the example in our formalism. The system is $\mathcal{M} = (\mathcal{S}, \mathcal{I}, \mathcal{N}, \mathcal{O}, \mathcal{E})$, where the set of state variables is $\mathcal{S} = \{\mathtt{mem}\}$, the set of input variables is $\mathcal{I} = \{\mathtt{raddr}, \mathtt{waddr}, \mathtt{data}\}$, and the set of output variables is $\mathcal{O} = \{\mathtt{safe}\}$. The next state function $\mathcal{N}$ and output function $\mathcal{E}$ are

$$\mathcal{N}(\mathtt{mem}) = \mathit{write}(\mathtt{mem},\ \mathtt{waddr},\ \mathit{mux}(\mathtt{data} > 200,\ \mathtt{data},\ 0))$$
$$\mathcal{E}(\mathtt{safe}) = \mathtt{mem}[\mathtt{raddr}] \neq 100.$$

For this system, the first few expansions of `mem` and `safe` are given by

$$\begin{aligned}
&\mathtt{safe}^0 = \mathtt{mem}^0[\mathtt{raddr}^0] \neq 100\\
&\mathtt{mem}^1 = \mathit{write}(\mathtt{mem}^0, \mathtt{waddr}^0, \mathit{mux}(\mathtt{data}^0 > 200, \mathtt{data}^0, 0))\\
&\mathtt{safe}^1 = \mathit{write}(\mathtt{mem}^0, \mathtt{waddr}^0, \mathit{mux}(\mathtt{data}^0 > 200, \mathtt{data}^0, 0))\\
&\qquad\quad [\mathtt{raddr}^1] \neq 100\\
&\mathtt{mem}^2 = \mathit{write}(\mathit{write}(\mathtt{mem}^0, \mathtt{waddr}^0,\\
&\qquad\qquad\qquad \mathit{mux}(\mathtt{data}^0 > 200, \mathtt{data}^0, 0)),\\
&\qquad\qquad \mathtt{waddr}^1,\\
&\qquad\qquad \mathit{mux}(\mathtt{data}^1 > 200, \mathtt{data}^1, 0))\\
&\mathtt{safe}^2 = \mathtt{mem}^2[\mathtt{raddr}^2] \neq 100.
\end{aligned}$$

One can easily see by inspection that exactly one array index expression appears in $\mathtt{safe}^k$, for any value of k. Thus the maximum number of indexes into the array `mem` needed to evaluate `safe` is $\Sigma^*_{\mathcal{M}}(\mathtt{safe})(\mathtt{mem}) = 1$. It follows that it is sound and complete to verify the output signal `safe` as a safety property by evalating $\mathtt{safe}^k$, for all k, in states where `mem` is modelled as a single-element array.

If we change the example so that the result of reading `mem` is stored in a register for an unbounded number of cycles before the value is used to produce the output, then the method of [7] would not be able to reduce the size of the array. Our method does reduce the array to one element in this modified example. □

Example 2. This example is similar to Example 1, please see Figure 3. Here, the result of reading the memory is routed to a multiplexor. The multiplexor sends a value to the state variable `read1`. On each cycle, the variable `read1` either holds its previous value or stores the output of the array read, depending on the value of the input signal `hold`. Because the value read from the array can be held for an unbounded amount of time in `read1` beforefore reaching the output, the method of [7] cannot reduce the size of the array. In contrast, our method can reduce the array to one entry. □

The following definition of a function $\mathrm{ub}(e, a)$ computes a simple upper bound on $\Sigma(e)(a)$. For $b[e]$, if b is a write expression with root a, then e is counted as an array index of a. For *write* expressions, the definition says that the expression e_1 must always be evaluated, and there are two cases for b and $e2$. If the array is read at index e_1, then e_2 must be evaluated; otherwise b must be evaluated. To cover both cases, we take the maximum value.

$$\begin{aligned}
&\mathrm{ub}(\mathit{var}, a) = 0,\ \text{if } \mathit{var} \text{ is a signal variable or an array variable}\\
&\mathrm{ub}(c, a) = 0,\ \text{if } c \text{ is a constant}\\
&\mathrm{ub}(\mathit{op}(e_1, \ldots, e_n), a) = \mathrm{ub}(e_1, a) + \ldots + \mathrm{ub}(e_n, a)\\
&\mathrm{ub}(\mathit{mux}(e_1, e_2, e_3), a) = \mathrm{ub}(e_1, a) + \max(\mathrm{ub}(e_2, a), \mathrm{ub}(e_3, a))\\
&\mathrm{ub}(b[e], a) = \begin{cases} \mathrm{ub}(b, a) + \mathrm{ub}(e, a) + 1 & \text{if } \mathrm{root}(b) = a\\ \mathrm{ub}(b, a) + \mathrm{ub}(e, a) & \text{otherwise}\end{cases}\\
&\mathrm{ub}(\mathit{write}(b, e_1, e_2), a) = \mathrm{ub}(e_1, a) + \max(\mathrm{ub}(e_2, a), \mathrm{ub}(b, a))
\end{aligned}$$

Theorem 3. For any signal expression e, state σ, and array variable a, $|\mathrm{eindx}(e, \sigma, a)| \leq \mathrm{ub}(e, a)$. □

Proof. The theorem can be proved by induction on expressions. □

In the function $\mathrm{ub}(e, a)$, multiple instances of the same expression are added. A more accurate estimate of $\Sigma(e)(a)$ can be obtained by a function that computes the sets of possible index expressions.

A better estimate of $\Sigma(e)(a)$ can be obtained by a function that computes the sets of possible index expressions. For any non-empty set X whose elements are a finite number of finite sets, let $\|X\|$ be the largest size of an element of X, $\max_{x \in X} |x|$. The function $\phi(a, e)$ defined below computes a set of sets of index expressions with the property that $\Sigma(e)(a) \leq \|\phi(e, a)\|$. Each element of $\phi(a, e)$ is a set of index expressions; the elements of $\phi(a, e)$ cover all the possible values of $\mathrm{eindx}(e, \sigma, a)$.

First, we define $X \uplus Y$ for sets X, Y.

$$X \uplus Y = \{x \cup y \mid x \in X, y \in Y\}$$

Then we define $\phi(e, a)$, where e is an expression and a is an array variable, as follows.

$$\phi(v, a) = \{\emptyset\},\ \text{if } v \text{ is a signal variable or an array variable}$$

$$\phi(c, a) = \{\emptyset\},\ \text{if } c \text{ is a constant}$$

$$\phi(\mathit{op}(e_1, \ldots, e_n), a) = \phi(e_1, a) \uplus \ldots \uplus \phi(e_n, a)$$

$$\phi(\mathit{mux}(e_1, e_2, e_3), a) = (\phi(e_1, a) \uplus \phi(e_2, a)) \cup (\phi(e_1, a) \uplus \phi(e_3, a))$$

$$\phi(b[e], a) = \begin{cases} \phi(b, a) \uplus \phi(e, a) \uplus \{\{e\}\} & \text{if } \mathrm{root}(b) = a\\ \phi(b, a) \uplus \phi(e, a) & \text{otherwise}\end{cases}$$

$$\phi(\mathit{write}(b, e_1, e_2), a) = (\phi(e_1, a) \uplus \phi(e_2, a)) \cup (\phi(e_1, a) \uplus \phi(b, a))$$

Theorem 4. For any signal expression e, state σ, and array variable a, $|\mathrm{eindx}(e, \sigma, a)| \leq \|\phi(e, a)\|$. □

In order to create sound abstractions for model checking, we want to compute an upper bound on the sequence $\phi(v^0, a), \phi(v^1, a), \ldots$, when an upper bound exists. One way of computing an upper bound uses a generalization of function $\phi(e, a)$ to a function $\Phi(e, a, \theta, \theta_A)$, where e is an expression, a is an array variable, and θ is a function mapping variables to sets of expressions. The idea is that the algorithm will iterate, at each step setting $\theta(v)$ to a set that is at least as general as each element of $\phi(v^k, a)$. The θ functions will provide a

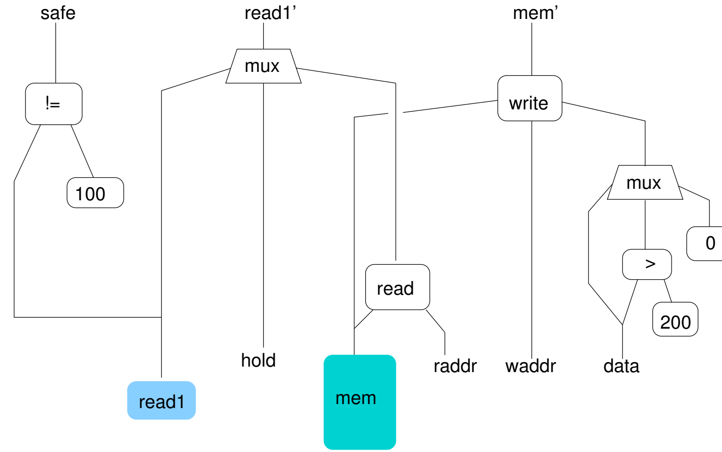


Fig. 3. Example 2

simple way to detect when a fixed point is reached in the iteration.

For a signal variable s, $\theta(s)$ will have the form $\{x_1, \ldots, x_n\}$, for some n, where the x_i are fresh distinct signal variables. Intuitively, we set $\theta(s)$ to a set of fresh variables larger than the set of essential indices needed to evaluate s. For array variables, we define $\theta_A(b, e)$ to be a function taking an array variable b and a signal expression e, and returning a set of signal variables, $\{x_{e,1}, \ldots, x_{e,n}\}$, for some n. The subscript on e simply indicates that $x_{e,i}$ is a fresh distinct signal variable related to e.

The definitions of $\phi(v, a)$ and $\Phi(v, a, \theta, \theta_A)$ differ in the case when v is a signal or array variable; in these cases Φ applies the function θ or θ_A. The functions θ, θ_A will be assigned increasingly large sets as the algorithm iterates. We define $\Phi(e, a, \theta, \theta_A)$ and $\Phi_A(b, e, a, \theta, \theta_A)$ by mutual recursion in the full paper .

Given a system $\mathcal{M} = (\mathcal{S}, \mathcal{I}, \mathcal{N}, \mathcal{O}, \mathcal{E})$, we define a sequence of approximations to $\phi(s^k, a)$, for each signal state variable s, and approximations to $\phi(b^k, a)$, for each array variable b. In the following equations, s is a signal variable, b is an array variable, e is a signal expression, and k is a natural number. The definitions use $\mathcal{N}(v)$, the next-state expression for the state variable v. See Figure 4.

For $k \geq 0$, let $\text{size}^k(v, a) = \|\text{approx}^k(v, a)\|$, be the size of the kth approximation. For an output variable v, we will define $\text{size}^k(v, a)$ using the approximations for the state variables:

$$\text{size}^k(v, a) = \|\Phi(\mathcal{E}(v), a, \theta^{a,k}, \theta_A^{a,k})\|,$$

The idea behind the definition of $\text{size}^k(s, a)$ is to make successive overapproximations to the value of $\Sigma(s^k)(a)$. We begin by setting the first approximation, $\text{size}^0(s, a)$, to 0. At each successive step, we make $\theta^k(s)$ be a set of n distinct signal variables if the value of $\text{size}^k(s, a)$ is n. This gives an overapproximation because, for example, there is no sharing of common expressions in $\theta^{a,k}(s_i), \theta^{a,k}(s_j)$, when s_i, s_j are different state variables. At each step we use the next-state expression $\mathcal{N}(s)$ to compute the maximal sets of indices needed for the array variable a in the next expansion of s.

$$\text{approx}^0(s, a) = \{\emptyset\}$$
$$\text{approx}^{k+1}(s, a) = \Phi(\mathcal{N}(s), a, \theta^{a,k}, \theta_A^{a,k})$$
$$\text{approx}_A^0(b, e, a) = \{\emptyset\}$$
$$\text{approx}_A^{k+1}(b, e, a) = \Phi_A(\mathcal{N}(b), e, a, \theta^{a,k}, \theta_A^{a,k})$$
$$\theta^{a,0}(s) = \emptyset$$
$$\theta^{a,k+1}(s) = \{s_1, \ldots, s_n\},$$
where $n = \|\text{approx}^{k+1}(s, a)\|$
and $s_1, \ldots, s_n$ are distinct fresh signal variables
$$\theta_A^{a,0}(b, e) = \emptyset$$
$$\theta_A^{a,k+1}(b, e) = \{b_{e,1}, \ldots, b_{e,n}\},$$
where $n = \|\text{approx}_A^{k+1}(b, 0, a)\|$
and $b_{e,1}, \ldots, b_{e,n}$ are distinct fresh signal variables

Fig. 4. Definitions of approx^k and θ

The following theorem says that $\text{size}^k(v, a)$ overapproximates $\Sigma(v^k)(a)$.

Theorem 6. For any array variable a, signal variable v, and $k \geq 0$, $\Sigma(v^k)(a) \leq \text{size}^k(v, a)$. □

We can now present an algorithm compute_size for computing a size for an array variable a that makes a sound and complete model for checking a property output variable v of a system. The inputs of compute_size are a system $\mathcal{M}$, a signal variable v of $\mathcal{M}$, an array variable a, and a natural number *OriginalSize*. The value of *OriginalSize* is the size of the array a in the original model. The algorithm computes $\text{size}^k(v, a)$ for increasing values of k, until either a fixed point is reached or $\text{size}^k(v, a) > \textit{OriginalSize}$.

We need to define a set $\text{dep}(v)$ of state variables on which a variable depends. For a state variable v, let $\text{dep}(v)$ be the smallest set of state variables such that 1) $v \in dep(v)$, and 2) for all state variables v', if $v' \in \text{dep}(v)$ and v'' is a state variable appearing in $\mathcal{N}(v')$, then $v'' \in \text{dep}(v)$. For an output variable v, we define $\text{dep}(v)$ to be the union of $\text{dep}(v')$ over

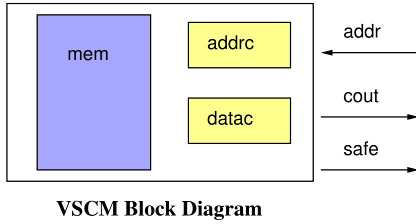


Fig. 5. Example 3

all state variables v' appearing in $\mathcal{E}(v)$.

Algorithm 1. The algorithm compute_size consists of the following steps.

1) $k \leftarrow 0$; $MaxSize \leftarrow 0$;
2) Until $((\forall v' \in \text{dep}(v) : \text{size}^k(v', a) = \text{size}^{k+1}(v', a)) \vee \text{size}^k(v, a) > OriginalSize)$ Do $\{MaxSize \leftarrow \max(\text{size}^k(v, a), MaxSize); k \leftarrow k+1\}$;
3) If $\text{size}^k(v, a) > OriginalSize$ then return $OriginalSize$;
4) If $\forall v' \in \text{dep}(v) : \text{size}^k(v', a) = \text{size}^{k+1}(v', a)$ then return $MaxSize$;

The algorithm always terminates, because for $k \geq 0$, $\text{size}^{k+1}(v, a) \geq \text{size}^k(v, a)$. If the algorithm exits with $\forall v' \in \text{dep}(v) : \text{size}^k(v', a) = \text{size}^{k+1}(v', a)$, then it is easy to see that the size computation has reached a fixed point at step k: The value of Φ for each state variable v depends on the previous values of $\theta(v')$ for the variables v' appearing in $\mathcal{N}(v)$. If $\text{size}^k(v', a) = \text{size}^{k+1}(v', a)$ for all v' in the transitive fan-in of v, then the size computation for v is at a fixed point. By Theorem 6, we know that $\forall k \geq 0 : \Sigma(v^k)(a) \leq \text{size}^k(v, a)$. The variable $MaxSize$ is now set to a value that is at least as large as any of the approximate values: $\forall k \geq 0 : \Sigma(v^k)(a) \leq MaxSize$. By Theorem 2 , we can construct a sound and complete abstraction for evaluating v in $\mathcal{M}$ by using compute_size to assign the size of each array variable.

In a more extended presentation, we would show how to improve the accuracy of the size computation by taking shared expressions into account in θ, θ_A.

Example 3. We illustrate Algorithm 1 by showing the analysis of a VSCM (very simple cache memory) unit. The block diagram of the VSCM is shown below. The state variables are $\texttt{mem}, \texttt{addrc}, \texttt{datac}$, the input variable is `addr`, and the output variables are $\texttt{cout}, \texttt{safe}$. The array `mem` represents a large main memory. Arrays `addrc` and `datac` are small arrays that form the cache. The array `addrc` stores the addresses that are cached, while `datac` stores the data for these addresses. The next state function $\mathcal{N}$ and output function $\mathcal{E}$ are shown in Figure 6. A unary operator `key` maps a full address into a value $\texttt{key}(\texttt{addr})$ that is an index into the arrays `addrc` and `datac`. On each clock cycle the cache inputs the signal `addr` and outputs the value of $\texttt{mem}[\texttt{addr}]$. If $\texttt{addrc}[k] = 0$, for some k, then the data at location k in the cache is considered invalid. Initially, $\texttt{addrc}[k] = 0$ for all k in the domain of `addrc`. If $\texttt{addrc}[\texttt{key}(\texttt{addr})] = \texttt{addr} \wedge \texttt{addr} \neq 0$, then the required memory data is provided from the cache by accessing $\texttt{datac}[\texttt{key}(\texttt{addr})]$. Otherwise, the data is fetched from main memory at $\texttt{mem}[\texttt{addr}]$, and $\texttt{addrc}, \texttt{datac}$ are updated. The output signal `cout` is the data output of the cache memory, and the signal `safe` asserts that $\texttt{cout} = \texttt{mem}[\texttt{addr}]$ holds.

The table summarizes the operation of Algorithm 1 for each of the three arrays. The numbers in the table show the values of $\text{size}^k(v, a)$, for each of the state variables. For the array $a = \texttt{mem}$, the algorithm reaches a fixed point at $k = 2$, with the array `datac` using one index value and the other arrays using no indices. For the output signals, `cout` uses one index value and `safe` uses two index values, because the expression for `safe` has subexpressions `cout` and $\texttt{mem}[\texttt{addr}]$. Therefore the abstract model for the output signal safe will reduce `mem` to two entries. For the array $a = \texttt{datac}$, the algorithm reaches a fixed point with one array index. For the array `addrc`, the algorithm is unable to reduce the size of the array, and the original size of the array will be used in the abstract model. At each iteration, $\text{size}^k(\texttt{datac}, \texttt{addrc})$ increases by one, because in the next-state expression for `datac`, the mux expression has a control expression (e_1) that uses one array index of `addrc`, and a second input (e_2) that uses k indices of `datac` on iteration k. The approximation computed by Φ therefore uses $k + 1$ array indices at step $k + 1$.

Note that data is read from the array `mem` and stored in `datac` for an unbounded length of time. Our algorithm abstracts `mem`, while the method of [7] cannot.

V. Construction of the Abstract Model

It is straightforward to build a model of a system that uses the semantics with the bottom value. We replace each signal of the original design with a composite signal having two elements: a `value`, which represents the value of the signal in the original design, and a `v` bit, which is true if the signal represents a value in V, and false if the signal represents $\perp$. Each state register is replaced with a register of the composite type. The signal operations of the original circuit are replaced with versions of the operations that recognize the value $\perp$.

The size of each array in the abstract model is determined by running Algorithm 1. An abstract array of size n is implemented using two arrays: an array of n addresses, and an array of n $(\texttt{value}, \texttt{v})$ pairs. The contents of the address array are set nondeterministically in the initial state of the system, and do not change over clock cycles. The array read and write operations are implemented according to the semantics of expressions.

If `p` is an output signal for a safety property $\texttt{p} = \texttt{true}$, then we construct a property expression of the form $\texttt{p.v} \rightarrow \texttt{p.value} = \texttt{true}$. Finally, we use model checking to verify that each of the new output property expressions is always true.

VI. Industrial Examples

In this section, we present initial results of using our abstraction algorithm on industrial hardware designs. The algorithm has been implemented in IBM's model checker.

Many of the arrays used in hardware designs have the property that at each time step, the output signal only depends on a small, bounded number of elements of the array. When an array has this property, our algorithm is often able construct

$\mathcal{N}(\texttt{mem}) = \texttt{mem}$
$\mathcal{N}(\texttt{addrc}) = \mathit{write}(\texttt{addrc},\ \texttt{key(addr)},\ \mathit{mux}(\texttt{addrc}[\texttt{key(addr)}] = \texttt{addr},\ \texttt{addrc}[\texttt{key(addr)}],\ \texttt{addr})$
$\mathcal{N}(\texttt{datac}) = \mathit{write}(\texttt{datac},\ \texttt{key(addr)},\ \mathit{mux}(\texttt{addrc}[\texttt{key(addr)}] = \texttt{addr},\ \texttt{datac}[\texttt{key(addr)}],\ \texttt{mem}[\texttt{addr}]))$

$\mathcal{E}(\texttt{cout}) = \mathit{mux}(\texttt{addr} \neq 0 \wedge \texttt{addrc}[\texttt{key(addr)}] = \texttt{addr},\ \texttt{datac}[\texttt{key(addr)}],\ \texttt{mem}[\texttt{addr}])$
$\mathcal{E}(\texttt{safe}) = (\texttt{cout} = \texttt{mem}[\texttt{addr}])$

Computation of $\text{size}^k(v, a)$

array $a = \texttt{mem}$				array $a = \texttt{datac}$				array $a = \texttt{addrc}$			
v	addrc	datac	mem	v	addrc	datac	mem	v	addrc	datac	mem
$k=0$	0	0	0	$k=0$	0	0	0	$k=0$	0	0	0
$k=1$	0	1	0	$k=1$	0	0	0	$k=1$	0	1	0
$k=2$	0	1	0					$k=2$	0	2	0
$\text{size}^2(\texttt{cout}, \texttt{mem}) = 1$				$\text{size}^1(\texttt{cout}, \texttt{datac}) = 1$				$k=3$	0	3	0
$\text{size}^2(\texttt{safe}, \texttt{mem}) = 2$				$\text{size}^1(\texttt{safe}, \texttt{datac}) = 1$				$k=n$	0	n	0

Fig. 6. Very Simple Cache Memory Definitions and Analysis

a small abstract model. On the other hand, there are common uses for arrays that do not have the necessary property. For instance, when an array is used as a content-addressable memory (CAM), each read operation accesses all elements of the array, and the array cannot be abstracted by our current approach.

The following results should be considered preliminary, because the implementation is in development. Data was collected for a set of 401 complex industrial examples. Many of the arrays in these designs have the property needed for our abstraction, but many of the arrays are used as CAMs, and hence are difficult to abstract. Individual designs in the set contain from one array up to several hundred arrays. Overall, our algorithm reduced the size of at least one array in 187 designs, or about 47 per cent of designs. The following table gives the total over all examples of the number of reduced arrays for each original and reduced size.

	Reduced Number of Rows						
Original Rows	1	2	3	4	6	8	> 8
2	144						
8	1	1					
16	14	13	55				
32	37	1	25				
39	24						
48	24						
64	46	29	20	18			
128	4	158	14	23	1	11	
256	3	40	10				
1024	3		10				2

The final version of the paper will compare the performance of other array abstraction algorithms.

One kind of verification problem where our techniques are valuable is proving sequential equivalence of two designs where an array has been reconfigured. In designing complex hardware systems, it is often necessary to reconfigure an array into two or more smaller arrays, due to physical circuit constraints. In simple cases, the reconfiguration consists of dividing an array into two arrays with the same number of index values (rows) as the original array, but narrower data values (fewer columns). For this kind of reconfiguration, it is often possible to prove the designs to be equivalent by automatically discovering a correspondence between the data columns of the original and reconfigured arrays [9].

When reconfiguration involves changing the number of rows in an array, it is harder to prove equivalence, because the two designs have differences in the addressing, data alignment and staging logic.

One real example that highlights the advantage of our techniques over previous approaches is an equivalence check where the original design has an array of 1024 rows by 16 columns, and the reconfigured design has two arrays, each with 128 rows by 64 columns. In this case, the logic near the arrays was substantially redesigned. Because the design uses clock gating, the method of [7] cannot reduce the size of the arrays. Our approach generates an abstract model and verifies equivalence, using four modeled rows from the large array of the original design and one row from each of the smaller arrays in the reconfigured design. The abstract model uses a total of 401 registers, including the three arrays and surrounding control logic. Without using our algorithm, we have found no way to verify this example other than to bit-blast the model into 32912 registers.

VII. Related Work

The work most closely related to our approach is by Bjesse [7]. Both our approach and [7] transform a register transfer level design into an abstract register transfer level design having smaller arrays, and allow any register transfer level verification method to be used on the abstract design. Both approaches use nondeterminism to choose which addresses are modeled in the abstract design. In our approach, the abstract model uses a semantics with a bottom value. The semantics limits evaluation of the correctness property to cases in which the nondeterministically chosen addresses are sufficient to determine the truth of the property. In [7], the correctness property at time t is made conditional on a formula saying that array read operations accessed only modeled addresses

at a list of previous time steps. Our approach is effective for reasoning about systems in which a value read from an array can affect the correctness property after an unbounded time delay. In [7], it is inherent in the construction of the abstract models, that reasoning is effective only in cases where there is a bound on the number of time steps after reading a value from an array that the value can affect the correctness property.

Several works [1]–[3] develop approaches for reasoning about systems with arrays by modeling the initial value and data forwarding properties of arrays operations over a bounded number of time steps. BAT [4] is another tool that builds abstractions for arrays over bounded time intervals. BAT uses several term-level techniques to reduce the size of abstract models of arrays before constructing a propositional model. These techniques include term-level uniqueness reductions and memory rewriting.

Model checkers in industry use a diversity of algorithms to analyze hardware designs. Baumgartner *et al* [9] describe enhancements to a number of algorithms in an industrial model checker, to provide more efficient processing by abstracting or simplifying arrays.

McMillan [10] developed a method of compositional model checking in which arrays can be abstracted to a small number of elements by temporal case splitting and symmetry reduction. In [10], the user proves complex designs by manually specifying a set of lemmas; the lemmas are checked automatically. In contrast, our method is directed towards fully automatic verification.

VIII. Discussion

We have introduced a logic of expressions for reasoning about arrays and developed some of its mathematical properties. The semantics of the logic permits reasoning about the value of an expression, when evaluated over states having arrays of different sizes. In Section III, we show that the truth of an expression can be evaluated over a state that may have smaller array sizes than the original model. The existence of adequate model sizes for expressions leads immediately to the existence of adequate model sizes for safety properties of systems. However, to compute the adequate model size directly from the results of Section III could be as difficult as verifying the original design. For this reason, we propose a method of safely overapproximating the minimum adequate size in Section IV. Our algorithm represents approximate sets of array indices using two-level sets of expressions. When using iteration on two-level sets, special care is needed to detect fixed points. Our algorithm constructs a most-general element from each two-level set at each step in the iteration as a way of detecting when a fixed point has been reached.

Although our main focus is on sequential systems, our results could also be useful for checking the satisfiability of formulas in the theory of arrays. Theorem 4 gives a way to overapproximate the number of array indices needed to check satisfiability. Our approximation could lead to improvements to model-based approaches to checking validity.

There exist many possible ways of overapproximating the minimum size of arrays defined in our theory. It should be possible to improve the approximation by using stronger methods to identify common subexpressions for array indices. A further improvement would be to identify expressions that are syntactically distinct but logically equivalent.

Acknowledgments The author gives special thanks to Jason Baumgartner for many helpful discussions, and for implementing the algorithm. Jessie Xu made helpful comments on the paper.

References

[1] M. Velev, R. E. Bryant, and A. Jain, "Efficient modeling of memory arrays in symbolic simulation," in *Proceedings of Computer Aided Verification 1977*, ser. LNCS, vol. 1254. Springer, pp. 388–399.

[2] M. K. Ganai, A. Gupta, and P. Ashar, "Efficient modeling of embedded memories in bounded model checking," in *Proc. of Computer Aided Verification, 2004*, ser. LNCS, vol. 3114. Springer, pp. 272–274.

[3] ——, "Verification of embedded memory systems using efficient memory modeling," in *Proceedings of DATE Europe '05*. IEEE Computer Society, 2005, pp. 1096–1101.

[4] P. Manolios, S. K. Srinivasan, and D. Vroon, "Automatic memory reductions for rtl model verification," in *Proceedings of ICCAD '06*. ACM, pp. 786–793.

[5] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu, "Symbolic model checking without bdds," in *TACAS*, 1999, pp. 193–207.

[6] M. Sheeran, S. Singh, and G. Stålmarck, "Checking safety properties using induction and a sat-solver," in *Formal Methods in Computer-Aided Design 2000*, pp. 108–125.

[7] P. Bjesse, "Word-level sequential memory abstraction for model checking," in *Formal Methods in Computer-Aided Design '08*, pp. 16:1–16:9.

[8] M. Keating, D. Flynn, R. Aitken, A. Gibbons, and K. Shi, *Low Power Methodology Manual*. Springer, 2007.

[9] J. Baumgartner, M. Case, and H. Mony, "Coping with Moore's law (and more): Supporting arrays in state-of-the-art model checkers," in *Formal Methods in Computer-Aided Design 2010*, pp. 61–69.

[10] K. L. McMillan, "Verification of infinite state systems by compositional model checking," in *CHARME*, 1999, pp. 219–234.

Parameterized Verification of Deadlock Freedom in Symmetric Cache Coherence Protocols

Brad Bingham and Mark Greenstreet
Department of Computer Science, University of British Columia
201-2366 Main Mall, Vancouver, B.C., Canada, V6T 1Z4
{binghamb, mrg}@cs.ubc.ca

Jesse Bingham
Intel, Oregon, U.S.A.
jesse.d.bingham@intel.com

***Abstract*—An important problem in the verification of hardware protocols is that of proving deadlock freedom. We view deadlock freedom as the property that for all reachable states, there exists some path to a quiescent state, i.e. one wherein all resources of interest are free and thus all prior requests have been resolved. We establish a framework for showing this property in a class of symmetric parameterized systems. Our approach is based on a mixed abstraction system than includes both an over-approximate and an under-approximate transition relation. Model checking is employed to compute all states reachable through overapproximate transitions, and from each of these states finds a path of underapproximate transitions to a quiescent state. When this fails because the under-approximation is too strong, we provide techniques to suggest additional transitions that can be introduced to soundly weaken the under-approximation. This approach can be viewed as an extension of the well-known approach of guard strengthening for verifying state invariants of parameterized systems. We present proof of deadlock freedom of the German and FLASH cache-coherence protocols as case studies using a semi-automated heuristic tool that mitigates the human effort.**

I. Introduction

Designing distributed protocols is known to be among the trickiest aspects of modern hardware design. A well-known problem that can arise in such systems is *deadlock*, which occurs when a state is reached that involves an unbreakable cyclic dependency between resources [1]. Here a *resource* might be an entry in a transaction table, a slot in a network, etc. We assume that one can easily characterize the *quiescent* states by a state predicate Q; the quiescent states are those wherein all resources are free and there are no in-flight transactions. A deadlock state, then, is simply a state from which there is no path to a quiescent state. Hence we can express deadlock freedom in Computation Tree Logic (CTL) by $\mathsf{AG}\,\mathsf{EF}\,Q$ (i.e. for all reachable states there exists a path to a Q-state).

Model checking [2] is a popular method for verifying that a system adheres to some specification. Classical model checking assumes that the system under consideration is finite-state. However, many researchers have explored techniques to generalize model checking to verify various classes of *parameterized* systems. For the purposes of this paper, a parameterized system $\mathcal{P}$ is a function that yields a finite-state system $\mathcal{P}(n)$ for all naturals $n \geq 1$. Here n indicates the number of values involved in some type P (called the *parametric type*) used by the system, for examples client IDs or addresses. A parameterized model checking problem asks if $\mathcal{P}(n)$ satisfies some given specification for all n[1]. Unless one puts severe restrictions on the class of systems, parameterized model checking is undecidable [3].

A promising approach to parameterized model checking is based on abstraction and compositional reasoning [4], [5], [6], [7], [8], [9], [10], [11] and is typically used to verify universally-quantified (over P) state assertions roughly as follows. An initial abstraction $\mathcal{A}_0$ is created from the syntax of $\mathcal{P}$. By construction, the transitions of $\mathcal{A}_0$ over-approximate those of $\mathcal{P}(n)$ for arbitrary $n > k$ (for some typically small k). If the state assertion holds of $\mathcal{A}_0$, then we can soundly conclude it holds too for $\mathcal{P}(n)$. However, often we are not so lucky and we must *strengthen* the transitions of $\mathcal{P}_0$ using a conjectured state invariant φ_1, yielding a tighter abstraction $\mathcal{A}_1$. This process iterates until we obtain a $\mathcal{A}_j$ wherein the original state assertion along with all of $\varphi_1, \ldots, \varphi_j$ hold. At this point our parametric verification goal has been achieved.

Our work extends this approach to handle deadlock freedom, i.e. properties of the form $\mathsf{AG}\,\mathsf{EF}\,Q$ for a state predicate Q. The key idea is to not only construct abstract transition relations that over-approximate those of $\mathcal{P}(n)$, but also transition relations that *under-approximate* $\mathcal{P}(n)$. The resulting verification framework is formalized in terms of *mixed-abstractions* [12] – systems with two transition relations O and U, which are respectively over-approximative and under-approximative. As in the traditional approaches, O is used to explore the reachable abstract states, which represent an over-approximation of the reachable states of $\mathcal{P}(n)$. However, during this exploration, we explore paths of U to check that the existential CTL formula $\mathsf{EF}\,Q$ holds for each reachable abstract state. If so, it is safe to conclude deadlock freedom of $\mathcal{P}(n)$; because the existence of a path in U implies that of a corresponding path in $\mathcal{P}(n)$. Initially, U can be constructed by checking syntactic properties of transitions of O. Then, analogous to how O might be too *weak*, the initial U might be too *strong*. A key contribution we present are a set of heuristic methods that allow the user to soundly weaken U in these cases. Another contribution is a theorem

[1] Many approaches, including ours, only verify $\mathcal{P}(n)$ for all $n \geq n_0$, where n_0 is a small constant. This is not a shortcoming since the $\mathcal{P}(n)$ where $n < n_0$ are either "uninteresting" or can be dispatched by finite-state model checking.

that supports deadlock freedom verification for Q involving universal quantification over the parametric type.

Ideally, one would seek to establish Linear Time Logic (LTL) *response* properties [13] of the form $\mathsf{G}\,(req \rightarrow \mathsf{F}\,resp)$, where req and $resp$ respectively mean a some request is sent and the corresponding response is received. However, response properties are difficult to model check for parameterized systems and require both computationally and conceptually complex approaches (we review some in Sect. II). We see our deadlock freedom verification as a lighter-weight alternative that is also practically relevant. Indeed, experience working with real hardware protocols in industry indicates that response failures are almost always caused by deadlocks (i.e. violations of $\mathsf{AG}\,\mathsf{EF}\,Q$) rather than more subtle "live-lock" style failures.

II. Related Work

There have been many previous efforts to extend compositional techniques to parameterized safety property verification [4], [5], [6], [7], [8], [9], [10], [11]. As for liveness-like properties, there are several notable works.

McMillan's work on using compositional methods for LTL liveness properties [14] was applied to parametric liveness verification of the FLASH coherence protocol [5]. Although this paper focuses on a proof of safety, the same framework was used to show that whenever the directory is in the pending state, it is eventually not pending [15]. This proof relies on a handful of lemmas and fairness assumptions, designed and proven within SMV.

Fang *et al.* proposed an interesting technique called *invisible ranking* [16] which attempts to automatically guess ranking functions to prove response properties. The associated proof obligations (from [13]) are decided using some small-model theorems and BDDs. The authors have previously used counter abstraction for parameterized liveness verification [17].

Baukus *et al.* employ *WS1S* (a decidable second-order logic) to perform liveness verification of parameterized systems [18], and verify response properties for the German protocol as a case study [19]. Like our approach, human effort is required, to select both abstract predicates and ranking predicates needed to create an appropriate abstraction. The complexity of deciding WS1S is well-known to be super-exponential, hence scalability of this approach seems unlikely.

The earliest example we could find where both over-approximative and under-approximative abstractions of a transition system are employed for verification is the work of Larsen and Thomsen [20]. They distinguish between *necessary* and *admissible* transitions; for a process to refine another it must over-approximate the former and under-approximate the latter. Of course our interests are in *abstraction* rather than *refinement*, which are in a sense inverses of each other. Later the work of Dams *et al.* [12] and independently Cleaveland *et al.* [21] used *mixed transition systems* which are defined with *two* transition relations to formulate abstractions that preserve both universal and existential properties of the modal μ-calculus; our mixed-abstractions are very similar.

III. Preliminaries

This section presents the formal framework that we use to verify quiescence properties of parameterized systems. Section III-A introduces *mixed abstractions* that have two transition relations: one that under-approximates the behaviors of the concrete systems and another that provides an over-approximation. Section III-B presents the idea of *insufficiency* – a mixed-abstraction may have an under-approximation that is too strong to verify the desired quiescence property or an over-approximation that is too weak. Section III-C describes parameterized systems.

A. Systems and Mixed Abstractions

A *system* $\mathcal{S}$ is a tuple (S, I, T) where S is a set of *states*, $I \subseteq S$ is the set of the initial states, and $T \subseteq S \times S$ is the *transition relation*. We write $s_1 \rightsquigarrow_T s_2$ to denote that $(s_1, s_2) \in T^*$. A state s is said to be *$\mathcal{S}$-reachable* (or simply *reachable* if $\mathcal{S}$ is understood) if $s_0 \rightsquigarrow_T s$ for some $s_0 \in I$. A *state predicate* p is simply a subset of S; if $s \in p$ we call s a *p-state*. Following standard CTL syntax, for state predicates p and q, we write: $\mathcal{S} \models \mathsf{AG}\,p$ if all reachable states are p-states; $\mathsf{AG}\,(p \rightarrow \mathsf{EF}\,q)$, if for all reachable p-states s there exists a q-state s' such that $s \rightsquigarrow_T s'$; and $\mathsf{AG}\,\mathsf{EF}\,q$ to mean $\mathsf{AG}\,(true \rightarrow \mathsf{EF}\,q)$.

To show that $\mathsf{AG}\,(p \rightarrow \mathsf{EF}\,q)$ can be inferred for a concrete system, $\mathcal{S}$, by establishing properties of an abstraction, $\mathcal{A}$, we employ Lynch and Vaandrager's notion of *forward simulation* [22]. Let $\mathcal{S}_1 = (S_1, I_1, T_1)$ and $\mathcal{S}_2 = (S_2, I_2, T_2)$ be two systems and $\theta \in S_1 \times S_2$ be an abstraction relation. We say that T_2 forward simulates (or "simulates" for short) T_1 if for every $(s_1, s_1') \in T_1$ and for all s_2 such that $(s_1, s_2) \in \theta$, there is a $s_2' \in S_2$ such that $s_2 \rightsquigarrow_{T_2} s_2'$ and $(s_1', s_2') \in \theta$. This allows system $\mathcal{S}_2$ to take multiple steps that may be invisible in $\mathcal{S}_1$ including possibly steps that have no "explanation" in $\mathcal{S}_1$. This general sense of simulation is motivated by our goal of showing the existence of trajectories. Now let $s_1, \ldots, s_\ell$ be a T_1-path, and suppose T_2 simulates T_1 with respect to θ. By induction on ℓ, there exists an T_2-path $s_1', \ldots, s_k'$ and a non-decreasing surjection $f : \{1, \ldots, \ell\} \rightarrow \{1, \ldots, k\}$ such that $(s_i, s'_{f(i)}) \in \theta$ for all $1 \leq i \leq \ell$. In this case we say that $s_1', \ldots, s_k'$ is a *θ-simulation* of $s_1, \ldots, s_\ell$.

To show $\mathsf{AG}\,(p \rightarrow \mathsf{EF}\,q)$ using abstraction, the abstract system must, for soundness, *over-approximate* the set of reachable p-states, and under-approximate the set of paths from p-states to q-states. Thus we introduce a *mixed abstraction* as defined below.

Definition 1: Let $\mathcal{S} = (S, I, T)$ be a system and let $Reach$ be the $\mathcal{S}$-reachable states. A *mixed abstraction* of $\mathcal{S}$ (relative to $\theta : S \rightarrow S_{\mathcal{A}}$) is a quadruple $\mathcal{A} = (S_{\mathcal{A}}, I_{\mathcal{A}}, U, O)$ such that

- $S_{\mathcal{A}}$ is a set of abstract states,
- $I_{\mathcal{A}} \subseteq S_{\mathcal{A}}$ are the initial abstract states and satisfy $\theta(I) \subseteq I_{\mathcal{A}}$,
- $O \subseteq S_{\mathcal{A}} \times S_{\mathcal{A}}$ simulates T with respect to $\theta \cap (Reach \times S_{\mathcal{A}})$, and
- T simulates $U \subseteq S_{\mathcal{A}} \times S_{\mathcal{A}}$ with respect to $(\theta \cap (Reach \times S_{\mathcal{A}}))^{-1}$.

Note that we require θ to be a function. Here U and O are respectively called the *under-approximative* (UA) and *over-approximative* (OA) transition relations of the mixed abstraction. When discussing mixed-abstractions, we will often refer to $\mathcal{S} = (S, I, T)$ as the *concrete system*, to S as the *concrete states*, etc.

The following serves as the basis for our approach to proving deadlock freedom for a class of parameterized systems. Let p and q be state predicates over $S_{\mathcal{A}}$, and suppose for all $(S_{\mathcal{A}}, I_{\mathcal{A}}, O)$-reachable p-states s there exists a q-state r such that $s \rightsquigarrow_U r$. We assert this by writing

$$\mathcal{A} \vDash \mathsf{AG}\,(p \rightarrow \mathsf{EF}\,q) \tag{1}$$

Hence, (1) holds of a mixed abstraction if for all p-states reachable using the over-approximative transition relation, there exists a path through the under-approximative transition relation to a q state.

Lemma 1: Let $\mathcal{A}$ be a mixed abstraction of $\mathcal{S}$ relative to θ and let p and q be state predicates on $S_{\mathcal{A}}$. If $\mathcal{A} \vDash \mathsf{AG}\,(p \rightarrow \mathsf{EF}\,q)$ then $\mathcal{S} \vDash \mathsf{AG}\,(\theta^{-1}(p) \rightarrow \mathsf{EF}\,\theta^{-1}(q))$.

Proof: Let *Reach* be the set of $\mathcal{S}$-reachable states. Let w be any $\theta^{-1}(p)$-state in *Reach*. Because O simulates T with respect to $\theta \cap (\mathit{Reach} \times S_{\mathcal{A}})$, $\theta(w)$ is $(S_{\mathcal{A}}, I_{\mathcal{A}}, O)$-reachable, and furthermore $\theta(w)$ is clearly a p-state. Let $a_0, \ldots, a_m$ be a U-path from $\theta(w) = a_0$ to a q-state a_m. Because T simulates U with respect to $(\theta \cap (\mathit{Reach} \times S_{\mathcal{A}}))^{-1}$, for all $0 \le i < m$ and all $w_i \in \theta^{-1}(a_i)$ there exists $w_{i+1} \in \theta^{-1}(a_{i+1})$ such that $w_i \rightsquigarrow_T w_{i+1}$. Therefore, taking $w_0 = w$, there is a path $w \rightsquigarrow_T w_m$ where $w_m \in \theta^{-1}(q)$. ■

Note that the definition of mixed-abstraction explicitly mentions the reachable states of $\mathcal{S}$ (in the involved simulation relations), this is just our means of formalizing the minimal requirements a mixed-abstraction must satisfy in order to prove Lemma 1. In other words, any methodology that aims to construct mixed-abstractions must guarantee *at least* these simulations. We emphasize that this is different than asking the user of such a methodology to precisely characterize *Reach* (indeed our methodology does not make such a demand).

When performing reasoning that allows us to add (or remove) transitions from U and O in a mixed abstraction, we often will employ the following sufficient conditions. We conclude this section by stating a connection between $\mathcal{S}$ and the transitions of U and O.

Lemma 2: Suppose $\mathcal{S} = (S, I, T)$, $S_{\mathcal{A}}$, $I_{\mathcal{A}}$, and $\theta : S \rightarrow S_{\mathcal{A}}$ are as in Def. 1. If $U, O \subseteq \mathcal{A} \times \mathcal{A}$ satisfy

1) for all $(w, w') \in T$ such that w is $\mathcal{S}$-reachable we have $(\theta(w), \theta(w')) \in O$, and
2) $(s, s') \in U$ implies for all $\mathcal{S}$-reachable $w \in \theta^{-1}(s)$ there exists $w' \in \theta^{-1}(s')$ such that $w \rightsquigarrow_T w'$.

then $(S_{\mathcal{A}}, I_{\mathcal{A}}, O, U)$ is a mixed-abstraction for $\mathcal{S}$.
Proof: Follows trivially from Def. 1.

B. Insufficiency

Lemma 1 allows us to infer $\mathcal{S} \vDash \mathsf{AG}\,(\theta^{-1}(p) \rightarrow \mathsf{EF}\,\theta^{-1}(q))$ if model checking (or any other means) verifies $\mathcal{A} \vDash \mathsf{AG}\,(p \rightarrow \mathsf{EF}\,q)$. However, the converse of the lemma does not hold in general (or even in common cases). Let us call the mixed-abstraction $\mathcal{A}$ *insufficient* if $\mathcal{S} \vDash \mathsf{AG}\,(\theta^{-1}(p) \rightarrow \mathsf{EF}\,\theta^{-1}(q))$ holds but $\mathcal{A} \nvDash \mathsf{AG}\,(p \rightarrow \mathsf{EF}\,q)$. If $\mathcal{A}$ is insufficient, it follows that there exists a p-state a_p such that $a_0 \rightsquigarrow_O a_p$ for some $a_0 \in I_{\mathcal{A}}$ but there is no q-state a_q such that $a_p \rightsquigarrow_U a_q$. There are two common causes for insufficiency:

- **OA insufficiency.** There is no (S, I, T)-reachable $s_p \in S$ such that $(s_p, a_p) \in \theta$. Hence a_p does not abstract any reachable state of $\mathcal{S}$. This is often caused by O being too *weak*, i.e. there exists a proper subset $O' \subset O$ such that $(S_{\mathcal{A}}, I_{\mathcal{A}}, O', U)$ is a mixed abstraction of $\mathcal{S}$ wherein a_p becomes unreachable.
- **UA insufficiency.** There *is* (S, I, T)-reachable $s \in S$ such that $(s, a) \in \theta$, however none of the T-paths $s = s_0, \ldots, s_\ell$ where $s_\ell \in \theta^{-1}(q)$ (at least one such T-path must exist), are simulations of any U-paths. This is often caused by U being too *strong*, i.e. there exists a proper superset $U' \supset U$ such that $(S_{\mathcal{A}}, I_{\mathcal{A}}, O, U')$ is a mixed abstraction of $\mathcal{S}$. Here the transitions introduced in U' would be sufficient to ensure the existence of a U'-path that $s_0, \ldots, s_\ell$ is a simulation of.

For the mixed abstractions we use to verify our parameterized systems, we will observe that OA insufficiency is solved in the previous literature, however UA insufficiency is not. Our basic approach is to identify UA insufficiency from a counter-example trace. In practice, the transitions from O that are needed in U are apparent from this counter-example. The basic idea is to show that for each such transition of O, there is a corresponding *path* in the concrete system. Sections IV and V present how this can be done by syntactic pattern matching and model checking of the *abstract* system for properties with the form shown by formula (1).

There is also a third flavor of insufficiency,

- **Abstract quiescence insufficiency.** Note that in Lemma 1, the quiescent predicate actually verified is of the form $\theta^{-1}(q)$, where q is a predicate on the abstract states. Suppose, however that there does not exist a q such that $\theta^{-1}(q)$ characterizes the desired set of quiescent concrete states. We experience this for our case studies; the desired quiescence predicate involves a universal quantification over the parametric type that the underlying simulation relation cannot precisely characterize. That is, if the concrete quiescent states are characterized by a predicate of the form $\forall i.\ \phi(i)$, then there is no abstract predicate q such that $\theta^{-1}(q) = \forall i.\ \phi(i)$. We deal with this form of insufficiency via Theorem 1.

C. Parameterized Systems

For the purposes of this paper, a parameterized system $\mathcal{P}$ is a function mapping natural numbers to systems. We write $\mathcal{P}(n) = (S(n), I(n), T(n))$ to denote the components of $\mathcal{P}(n)$ for an arbitrary n. The states $S(n)$ are the type-consistent assignments to a set of state variables. For a state w of a parameterized system and a state variable v, we write $w.v$ to denote the value w assigns to v. We allow four types of variables:

- finite types that are independent of n, such as booleans and enumerations; for simplicity, we denote all such types as B
- a type which has cardinality n, denoted P_n
- arrays indexed by P_n with elements in B, denoted $\mathsf{array}\,[\mathsf{P}_n]$ of B
- arrays indexed by P_n with elements in P_n, denoted $\mathsf{array}\,[\mathsf{P}_n]$ of P_n

In this paper we identify the set P_n with the numbers $\{1, \ldots, n\}$ called *nodes*; the only operations supported on nodes are equality comparison, assignment, and nondeterministic choice.[2] This ensures that for all n, $\mathcal{P}(n)$ is fully symmetric [23] in P_n; here we give a brief review of this notion. Let us write $\lambda i.e$ to denote the array a indexed by P_n where $a[i] = e$ and e is an expression of the appropriate type. Let π be a permutation on P_n. We overload π to act on $w \in S(n)$ by defining $\pi(w) \in S(n)$ to be the state such that for each state variable v, $\pi(w).v$ is equal to:

- $w.v$, if v has type B
- $\pi(w.v)$, if v has type P_n
- $\lambda i.(w.v)[\pi^{-1}(i)]$, if v has type $\mathsf{array}\,[\mathsf{P}_n]$ of B
- $\lambda i.\pi((w.v)[\pi^{-1}(i)])$, if v has type $\mathsf{array}\,[\mathsf{P}_n]$ of P_n

Then $(S(n), I(n), T(n))$ is called *fully symmetric* if for all $w, w' \in S(n)$ and all permutations π on P_n we have both that $w \in I(n)$ iff $\pi(w) \in I(n)$, and $(w, w') \in T(n)$ iff $(\pi(w), \pi(w')) \in T(n)$. The following lemma has a simple inductive proof using the latter.

Lemma 3 (Path Symmetry): For $w, w' \in S(n)$ we have $s \rightsquigarrow_{T(n)} w'$ if and only if $\pi(w) \rightsquigarrow_{T(n)} \pi(w')$.

In Section IV-A, we impose restrictions on parameterized systems in order to be admissible for our method.

IV. Syntactical Abstraction

We assume that the parameterized system is modeled by Murφ [24] or a similar guarded-command notation. Given a program P to describe the parameterized system, we use well-established techniques [4], [5], [6], [7], [8], [9], [10], [11] to obtain an abstraction of P. Our formulation is inspired by the Krstic's "syntactic" approach [7]; Section IV-A states restrictions that we assume on the form of P, and Section IV-B summarizes the abstraction technique. In Section V, we show how the abstraction can be generalized to produce an under-approximate transition relation, U, and how U can be soundly weakened to prove quiescence properties.

A. Syntax and Restrictions

We assume that the guarded-command program that models the parameterized system satisfies certain syntactic restrictions described in this section. These restrictions ease the syntactical abstraction process and simplify reasoning about the program because many useful properties are guaranteed by construction. From the case studies reported in Section VI, we've found that these restrictions are not problematic in practice.

[2]If i and j are nodes, a parameterized system is not allowed to perform a comparison like $i < j$ or perform an incrementation $i := i + 1$.

We say that such a program is *admissible*, and we write AP as a shorthand for an admissible program. An AP has set of variables of the types indicated in Table I. A state of the AP is a type-consistent assignment of values to these variables. If e is a term, we write $s(e)$ to denote the value of e in state s. In Murφ, a guarded command is called a *rule* and has the form: *guard* $\twoheadrightarrow$ *action*, where the *guard* is a boolean-valued expression, and the *action* is a sequence of one or more assignments. We write $r : \rho \twoheadrightarrow \boldsymbol{a}$ to denote rule r with guard ρ and action $\boldsymbol{a}$.

The denotation $[\![r]\!]$ of r is the set of tuples $(s, s') \in S^2$ such that $s(\rho)$, and s' is the state reached by performing action $\boldsymbol{a}$ from state s. Murφ has *rulesets* of the form:

```
ruleset i in Pn do r(i) end;
```

where $r(i)$ is a rule (or a ruleset, as they may be nested). Here, $i \in \mathsf{P}_n$ is called the *ruleset parameter*. If rs is the ruleset indicated above, then

$$[\![rs]\!] \quad = \quad \{(s, s') \mid \exists i \in \mathsf{P}_n.\ (s, s') \in [\![r(i)]\!]\} \tag{2}$$

A *local boolean predicate* L is a propositional formula over the variables of type $\mathsf{array}\,[\mathsf{P}_n]$ of B. For node i, we say $L[i]$ holds of a state if L evaluates to true when its variables are assigned according to the ith array entries of the state. An *admissible program* (AP) must satisfy the following syntactic restrictions. Rulesets have guards that are a conjunct of:

- Boolean terms, composed of variables of type B or $\mathsf{array}\,[\mathsf{P}_n]$ of B indexed by a ruleset parameter, and the logical connectives `AND`, `OR` and `NOT`.
- At most one *forall condition*, of the form $\forall i \in \mathsf{P}_n.\ C[i]$ where C is a local boolean predicate.
- Any number of P-comparisons, of the form $v_1 = v_2$ or $v_1 \neq v_2$, where v_1 and v_2 are variables of type P or $\mathsf{array}\,[\mathsf{P}]$ of P indexed by a ruleset parameter, or a ruleset parameter. Without loss of generality, we restrict each ruleset parameter to appear in at most one P-comparison of equality.

The initial states and ruleset commands given by a sequence assignments of the following forms:

- Assignments of the form $b_1 := b_2$, $a^1_{\mathsf{B}}[i] := b_2$, $a^1_{\mathsf{B}}[i] := a^2_{\mathsf{B}}[i]$, where b_1 and b_2 are variables of type B, a^1_{B} and a^2_{B} are variables of type $\mathsf{array}\,[\mathsf{P}_n]$ of B, and i is a ruleset parameter. RHS values may also be the constants $true$ and $false$.
- Assignments of the form $p_1 := p_2$, $a^1_{\mathsf{P}}[i] := p_2$, $a^1_{\mathsf{P}}[i] := a^2_{\mathsf{P}}[i]$, where p_1 and p_2 are variables of type P_n, a^1_{P} and a^2_{P} are variables of type $\mathsf{array}\,[\mathsf{P}_n]$ of P_n, and i is a ruleset parameter.
- *Forall updates* of the form $\forall i \in \mathsf{P}_n.\ a_{\mathsf{B}}[i] := \ell(i)$, where ℓ is a boolean function depending on variables of type B, P_n and on the ith index of array variables.

A BNF grammar for this restriction of Murφ is given in the Appendix.

These restrictions ensure that guards in APs do not contain disjunctions of comparisons between variables of type P and have no existentially quantified terms; updates in APs do not contain if-then-else clauses. These constructs can be handled

concrete type	abstract type	abstraction ψ
B	B	$\psi(s).v = s.v$
P_n	$\widehat{\mathsf{P}}_k$	$\psi(s).v = \widehat{\psi}_k(s.v)$
array $[\mathsf{P}_n]$ of B	array $[\mathsf{P}_k]$ of B	$\forall i \in \mathsf{P}_k \ : \ \psi(s).v[i] = s.v[i]$
array $[\mathsf{P}_n]$ of P_n	array $[\mathsf{P}_k]$ of $\widehat{\mathsf{P}}_k$	$\forall i \in \mathsf{P}_k \ : \ \psi(s).v[i] = \widehat{\psi}_k(s.v[i])$

TABLE I

The abstract state space $S_{\mathcal{A}}$ and the abstraction function $\psi : S(n) \to S_{\mathcal{A}}$. For a system variable v and $s \in S(n)$, the leftmost column gives the type of v in the concrete domain, the second column gives the type of v in $S_{\mathcal{A}}$, and the third column specifies the value v is assigned by $\psi(s)$ in terms of $s.v$.

by a straightforward splitting into multiple rulesets. The Murφ systems for German and FLASH are admissible, and from this experience, we believe that the systems for many other symmetric protocols will be admissible or easily modified to produce an admissible equivalent.

B. Abstraction

Let $\mathcal{P}(n) = (S(n), I(n), T(n))$ be a the denotation of an AP, P. We want to construct a mixed-abstraction, $\mathcal{A} = (S_{\mathcal{A}}, I_{\mathcal{A}}, O, U)$. In this section, we show how $S_{\mathcal{A}}$, $I_{\mathcal{A}}$, and O can be readily by syntactic transformations of the source-code of P. Section V extends this approach to the construction of U. To create these abstractions, we introduce a new type to represent type P_n from the concrete system; this type requires the user to choose a constant k. It is assumed throughout that k is *at least* the greatest number of ruleset parameters for any ruleset in P (typically, $k \leq 3$). Let $\widehat{\mathsf{P}}_k = \mathsf{P}_k \cup \{Other\}$, and given $x \in \mathsf{P}_n$, let $\widehat{\psi}_k(x) = x$ if $x \leq k$; otherwise, $\widehat{\psi}_k(x) = Other$. Table I specifies how each variable of $\mathcal{P}$ is typed in $\mathcal{A}$ and how the abstraction function ψ acts on v. Intuitively, $\psi(s)$ preserves B variable values, replaces values of type P_n greater than k with $Other$, and restricts arrays to the indices P_k (hence all array entries $v[i]$ for $i > k$ are deleted by ψ). Although ψ is a function, we will treat it as a relation, $\psi \subseteq S(n) \times S_{\mathcal{A}}$, and freely employ its inverse $\psi^{-1} \subseteq S_{\mathcal{A}} \times S(n)$. We call elements of P_k *non-abstracted* and elements of $\mathsf{P}_n \backslash \mathsf{P}_k$ *abstracted*. For every ruleset parameter i interpreted as abstracted, all updates with $a_{\mathsf{B}}[i]$ or $a_{\mathsf{P}}[i]$ appearing on the LHS are deleted. All instances $a_{\mathsf{B}}[i]$ or comparisons depending on $a_{\mathsf{P}}[i]$ appearing positively in the guard that depend replaced with $true$; those appearing negatively are replaced with $false$. Instances of i appearing on the RHS of assignments are replaced with $Other$. Finally, equality comparisons with i appearing positively in the guard are replaced with $true$. The state variables of $\mathcal{A}$ have the same names as those of $\mathcal{P}$, with the types changed as shown in Table I.

We now overload ψ to map rules of $\mathcal{P}$ system to rules that generate the state transitions of O. Rules of $\mathcal{P}$ that are not in rulesets are copied without change of syntax (therefore, with the implied change of types), to O. If ruleset $r : \rho \twoheadrightarrow \boldsymbol{a}$ depends on m ruleset parameters, consider the set of *rule instantiations*, obtained from assigning each ruleset parameter a value in P_n. This set is partitioned as $R_1, ..., R_{2^m}$, where all rule instantiations of R_j have the same partitioning of ruleset parameters into F and NF, where $i \in F \Leftrightarrow i \in \mathsf{P}_k$ and $i \in NF \Leftrightarrow i \in \mathsf{P}_n \setminus \mathsf{P}_k$ (since there are m ruleset parameters, there are 2^m possible partitions). Each set R_j abstracts to an abstract ruleset $\widehat{r}_j$ according to the described syntactic transformation. We denote the set of corresponding abstract rulesets to concrete ruleset r by $\psi(r) = \{\widehat{r}_1, ..., \widehat{r}_{2^m}\}$. Let $\widehat{r}_v : \widehat{\rho}_v \twoheadrightarrow \widehat{\boldsymbol{a}}_v$ denote the unique element of $\psi(r)$ such that all ruleset parameters are non-abstracted. Note that although the set of rule instantiations differ depending on the value of n, the set $\psi(r)$ does not, for any $n > k$, hence we can fix $n = k + 1$ to perform this abstraction.

Example: Consider the concrete rule from German `SendGntE`.

```
ruleset i : NODE do rule "SendGntE"
CurCmd = ReqE ∧ CurPtr = i ∧ Chan2[i].Cmd = Empty
∧ ¬ExGntd ∧ forall j : NODE do ¬ShrSet[j] end ==>
Chan2[i].Cmd := GntE; ShrSet[i] := true;
ExGntd := true; CurCmd := Empty;
```

The abstraction contains two corresponding rulesets, one where i is non-abstracted and one where i is abstracted, with the former corresponding to $\widehat{r}_v$:

```
ruleset i : NODE do rule "ABS_SendGntE1"
CurCmd = ReqE ∧ CurPtr = i ∧ Chan2[i].Cmd = Empty
∧ ¬ExGntd ∧ forall j : NODE do ¬ShrSet[j] end ==>
Chan2[i].Cmd := GntE; ShrSet[i] := true;
ExGntd := true; CurCmd := Empty;
rule "ABS_SendGntE2"
CurCmd = ReqE ∧ CurPtr = Other ∧ ¬ExGntd
∧ forall j : NODE do ¬ShrSet[j] end ==>
ExGntd := true; CurCmd := Empty;
```

V. VERIFYING UNIVERSAL QUIESCENCE

We want to verify properties of the form

$$\mathcal{P}(n) \models \mathsf{AG}\,\mathsf{EF}\, Q_n \ , \tag{3}$$

where $\mathcal{P}(n)$ is a parameterized system and

$$Q_n \quad = \quad G \wedge \bigwedge_{i \in \mathsf{P}_n} L[i] \tag{4}$$

is the quiescence property to be verified. Here, G is a *boolean predicate*, meaning G only depends on variables of type B, while L is a local boolean predicate (defined in Sect. IV-A). To verify (3), we construct a mixed abstraction, $\mathcal{A} = (S_{\mathcal{A}}, I_{\mathcal{A}}, O, U)$, and that for all O-reachable states, there exists a U-path to a state that satisfies Q_n. To do so, we must address two key issues. First, Q_n cannot be established directly from $\mathcal{A}$, as Q_n refers to variables of the concrete system that do not appear in the abstraction. This is the "abstract quiescence insufficiency" defined in Section III-B, and Section V-A shows how it can be addressed. Second, U may omit transitions that are required to reach states that satisfy Q_n. This is the UA insufficiency from Section III-B, and we address it in Sections V-B and V-C.

A. Universally Quantified Quiescence

To show (3), we need to show that $L[i]$ holds for all i, not just non-abstracted i. Intuitively, we show that $\mathcal{A}$ can reach a state where $L[i]$ holds for all non-abstracted i, then use

Lemma 3 to exchange any abstracted j for which $L[j]$ might not hold with a non-abstracted i, find a path that establishes $L[i]$, and we can establish (3) by induction. For this approach to work, we must show that if G holds, then for each non-abstracted node i, there is a U-path to a state that satisfies $L[i]$ whose concretization in $\mathcal{P}(n)$ does not falsify $L[j]$ for any abstracted node j. To do this, we introduce the notion of L-preserving transitions.

For local boolean predicate L, abstract transition (s, s') is *L-preserving* if

$$\begin{aligned} & \forall w \in \psi^{-1}(s).\ \exists w' \in \psi^{-1}(s').\ w \rightsquigarrow_T w' \\ \wedge \quad & \forall i \in \mathsf{P}_n \setminus \mathsf{P}_k.\ w \in L[i] \Rightarrow w' \in L[i]\ . \end{aligned}$$

Abstract ruleset $\widehat{r}$ is L-preserving if all transitions in $[\![\widehat{r}]\!]$ are L-preserving. A mixed abstraction is called L-preserving if its UA transitions contain only L-preserving rules.

We can now state our main theorem for showing universally quantified quiescence; for the proof of Theorem 1 see the appendix.

Theorem 1 (Universally Quantified Quiescence): Let G denote a boolean predicate, L a local boolean predicate, and let $\mathcal{A}$ and $\mathcal{B}$ be mixed abstractions of $\mathcal{P}(n)$, and assume that $\mathcal{B}$ is L-preserving. If

1) $\mathcal{A} \models \mathsf{AG}\,\mathsf{EF}\,(G)$, and
2) $\mathcal{B} \models \mathsf{AG}\,(G \rightarrow \mathsf{EF}\,(G \wedge \bigwedge_{i \in \mathsf{P}_k} L[i]))$

then $\mathcal{P}(n) \models \mathsf{AG}\,\mathsf{EF}\,(G \wedge \bigwedge_{i \in \mathsf{P}_n} L[i])$.

B. Abstract Rule Tags

Given a program P, we use the syntactical abstraction technique to produce an abstract program $\widehat{P}$. In the remainder of this paper, we use the term "ruleset" to refer to Murφ-style rulesets with any degree of ruleset nesting including no such quantification – i.e. a "ruleset" could be a simple rule. We want to identify which rulesets of $\widehat{P}$ have denotations that are UA, and which are L-preserving, for a given local boolean predicate L. Throughout the rest of the paper we use $r : \rho \rightharpoonup \boldsymbol{a}$ and $\widehat{r}_v$ as defined in Section IV-B, and use $\widehat{r}_j$ to denote an arbitrary element of $\psi(r)$.

We will tag abstract rulesets tags from the following set $\{\mathsf{AUG}, \mathsf{AEG}, \mathsf{AUC}, \mathsf{AEC}\}$; the first two elements are called *guard tags*, and the last two are called *command tags*. These indicate reasons (in the guard and command, respectively) why the abstract ruleset is not trivially UA or L-preserving. An abstract ruleset can be tagged with *any* of the 16 subsets of these tags. AUG and AUC indicate that a universal quantifier has been abstracted; similarly AEG and AEC indicate that existential information has been abstracted.[3]

We call ρ and $\widehat{\rho}_j$ *syntactically equivalent* (SE) if they are expressed with identical syntax, and ρ contains no forall conditions. In this case, we attach no guard tags to $\widehat{r}_j$. Likewise, if $\boldsymbol{a}$ and $\widehat{\boldsymbol{a}}_j$ have identical syntax and contain no forall updates, then we attach no command tags. If a ruleset r has no guard or command tags, then it is simple to show that r is both UA and L-preserving for any local predicate L. Typically, the set of all such rulesets is insufficient to establish the desired quiescence property.

[3] Note that AEG and AEC don't indicate explicit existential quantifiers in the concrete system syntax; *existential* refers to the quantifier in the ruleset denotation, which ranges over the ruleset parameter.

Tag \Property	UA	L-preserving
AEG	Heuristic 1	Heuristic 2
AUG	Heuristic 3	Heuristic 4
AEC	None	Heuristic 2
AUC	None	Heuristic 4

TABLE II
Obligations associated with each ruleset tag and property pair. "None" means there is no obligation to show. A ruleset with no *tags* is L-preserving, while one with no *guard tags* is UA.

The elements of $\psi(r) \setminus \widehat{r}_v$ may not have SE guards because some ruleset parameter i is abstracted, so the abstraction will syntactically change the guard (except for degenerate cases). These rulesets have guards that optimistically abstract away references to abstracted i; this is safe when constructing the OA but not the UA. Such rulesets are tagged with AEG (abstract existential in guard).[3] When ρ contains a forall condition, it is necessarily weakened in every rule of $\psi(r)$. In this case, every ruleset of $\psi(r)$ including $\widehat{r}_v$ is tagged with AUG (abstract universal in guard).

Similarly, existential or universal updates may be missing from the command of an abstract ruleset, relative to the concrete version. If local update $a_b[i] := e_b$ appears in $\boldsymbol{a}$ for ruleset parameter i, then any ruleset of $\psi(r)$ where i is abstracted (that is, where the update $a_b[i] := e_b$ vanishes), is tagged with AEC (abstract existential command).[3] If $\boldsymbol{a}$ contains a forall update, then every ruleset of $\psi(r)$ is tagged with AUC (abstract universal command).

Example: Referring to the example in Section IV-B, abstract ruleset `ABS_SendGntE1` is tagged with AUG and no command tags, and abstract ruleset `ABS_SendGntE2` is tagged with AUG, AEG and AEC.

C. Heuristics

Each tag assigned to a ruleset corresponds to a set of proof obligations for showing it is UA or L-preserving (for some local boolean predicate L). For either of these properties, each tag must be separately *discharged* through the corresponding heuristic according to Table II. Once a tag is discharged we may safely ignore it as a potential reason why the desired property does not hold. Each of the heuristics involves model checking a mixed abstraction. In this Section, the various heuristics are stated; see the supplementary material [25] for proofs.

An abstract ruleset is called *local to i* (as a special case of having no tags) when the guard only depends on variables of type B, P, and the ith index array variables a_{B}, and the command only updates the local state of non-abstracted i. Here, given an abstract or concrete state, the *local state* of i is simply the values of all array variables at index i. The transitions that compose such rules are called *local transitions*. A mixed abstraction with UA set U composed only of rulesets local to i is denoted $\mathcal{A}_{\ell(i)}$. Assuming ruleset $\widehat{r}$ is UA, we write

$\mathcal{A}_{\ell(i)} \models_{\widehat{r}} \mathsf{AG}\,(A \rightarrow \mathsf{EF}\,B)$ when every O-reachable A-state has a path to some B-state consisting of transitions of rules local to i and *necessarily a single transition* of ruleset $\widehat{r}$.

When showing rulesets are UA (Heuristics 1 and 3), note that the tags AEG or AUG indicate guards that are OA because they have abstracted away information about abstracted nodes. Our heuristics compute O-reachable states and exploit the path symmetry of Lemma 3 to find the possible local state of abstracted nodes under some boolean predicate. Then, if the local state of node i does not have a required property, we find "hidden paths" composed entirely of rulesets local to i that reach a state that does have the property. This assures that although some states in the concretization of abstract guard $\widehat{\rho}_j$ *do not* satisfy the corresponding concrete guard ρ, there is a guaranteed path that is not observable in the abstract system from every $\psi^{-1}(\widehat{\rho}_j)$ to a ρ-state. For simplicity, we present our heuristics for rulesets with at most one abstracted ruleset parameter, however generalizing is straightforward.

When showing rulesets are L-preserving it must be checked that aspects of the guard and update that have been abstracted away do not affect L-preservation in the abstracted nodes; Heuristics 2 and 4 pertain to this check. The obligations for these heuristics require that a certain transition must fire on each path that justifies the deadlock freedom property. Intuitively, when the heuristic obligation holds, the concrete paths that justify the tagged ruleset in question $\widehat{r}$ to be UA must have a certain form. For abstracted node i, each path is

- a (possibly empty) path composed of transitions of rules local to i, followed by
- a transition of concrete rule r (possibly changing non-local variables), followed by
- a (possibly empty) path composed of transitions of rules local to i.

Furthermore, we only seek a path when the starting state is an $L[i]$-state, and the final state must also be a $L[i]$-state. For which i this is shown depends on the heuristic. Heuristic 2 reasons about those abstracted nodes that are abstracted ruleset parameters in $\widehat{r}$. Heuristic 4 reasons about those abstracted nodes that are *not* abstracted ruleset parameters in $\widehat{r}$. Note that we assume a ruleset has been proven UA before it is proven L-preserving.

A few definitions are needed for the heuristic statements. If $\widehat{r}$ is an abstract ruleset with ruleset parameter i, let $\widehat{r}|_{i=1} : \widehat{\rho}|_{i=1} \rightarrow \widehat{\boldsymbol{a}}|_{i=1}$ be the ruleset where all instances of i are replaced with the constant value 1. Also, let $relax(i, \widehat{r})$ be the rule $\widehat{r}$ but with the values of variables $a_{\mathsf{B}}[i]$ and $a_{\mathsf{P}}[i]$ unconstrained in the guard. If $A \subseteq S_{\mathcal{A}}$, let $\Gamma(A)$ denote the strongest boolean predicate implied by A.

Heuristic 1: For ruleset $\widehat{r}_j$ tagged with AEG and with abstracted ruleset parameter i, suppose that $\widehat{r}_v$ is UA. If $\mathcal{A}_{\ell(1)} \models \mathsf{AG}\,(relax(\widehat{\rho}_v, i)|_{i=1} \rightarrow \mathsf{EF}\,(\widehat{\rho}_v|_{i=1}))$ then tag AEG is discharged for showing $\widehat{r}_j$ to be UA.

Heuristic 2: For ruleset $\widehat{r}_j$ tagged with AEG and/or AEC with abstracted ruleset parameter i, suppose that $\widehat{r}_v$ is UA. If $\mathcal{A}_{\ell(1)} \models_{\widehat{r}_v} \mathsf{AG}\,((relax(\widehat{\rho}_v, i)|_{i=1} \wedge L[1]) \rightarrow \mathsf{EF}\,(L[1]))$ then AEG and AEC are discharged for showing $\widehat{r}_j$ to be L-preserving.

Heuristic 3: For ruleset $\widehat{r}_j$ tagged with AUG, and let $\forall i \in \mathsf{P}_n.\, C[i]$ be the forall condition of ρ. If $\mathcal{A}_{\ell(1)} \models \mathsf{AG}\,(\Gamma(\widehat{\rho}_j) \rightarrow \mathsf{EF}\,(C[1]))$, then tag AUG is discharged for showing $\widehat{r}_j$ to be UA.

Heuristic 4: For ruleset $\widehat{r}_j$ tagged with AUG and/or AUC, let $\widehat{r}* \in \psi(r)$ be the abstract ruleset where all ruleset parameters of r are abstracted. If $\mathcal{A}_{\ell(1)} \models_{\widehat{r}*} \mathsf{AG}\,((\Gamma(\widehat{\rho}_j) \wedge L[1]) \rightarrow \mathsf{EF}\,(L[1]))$, then tags AUG and AUC are discharged for showing $\widehat{r}_j$ to be L-preserving.

We apply each of these heuristics by performing model checking using a mixed abstraction that uses *only* local rules for U. As local rules are identified entirely by syntax, they are known *a priori*; therefore, we could take a brute force approach that attempts to use our heuristics to prove every abstract rule is UA and L-preserving. However, we prefer to take a counter-example driven approach, as there are two distinct situations in which our heuristics may not suffice that arose in our case studies. Firstly, additional auxiliary variables may be needed to capture the system state with a slightly finer-grained abstraction. Secondly, if the ruleset is not underapproximate, manual guard strengthening or splitting into multiple rulesets may help. These are illustrated with examples in Section VI.

VI. Case Studies

Mixed abstractions are expressed as Murφ models. The OA rulesets are borrowed from Chou *et al.* [6] and the (initial) UA transitions are derived manually according to tags – those rules with no guard tags. Thus, the UA rulesets are maintained as a subset of the OA rulesets. Rulesets with no tags at all are identified as L-preserving, and the relevant subset of these are identified as local.

We use a distributed explicit state model checker for Murφ called PReach [26] for the mixed abstraction checks. Initially designed to check state-invariants, we have added a feature to check CTL properties of the form $\mathsf{AG}\,(p \rightarrow \mathsf{EF}\,q)$. The search algorithm is simple: for every $(p \wedge \neg q)$-state s visited during the forward reachability computation, choose an enabled rule of U and fire it to reach a new state. Firing rules of U continues until one of the following occurs. 1) a q-state is found, 2) a U-dead-end state is found, or 3) a cycle is detected. In the first case, a path from s to a q-state exists and we proceed with the forward reachability computation. In the second case, there *may* not exist such a path (although we believe that in practice this is strong evidence that no path exists). If a cycle is found, this is usually an indication that U contains rules that do not help us reach q-states, so we might as well exclude them and try again[4]. For example, there are several easily identifiable rules in both German and FLASH that initiate requests by injecting messages, and are not useful transitions in finding a quiescent state where all messages are consumed. Notice that deadlock freedom properties can be verified by a CTL model

[4]Removing transitions from U trivially preserves mixed-abstractions.

checker, but for our case studies we chose PREACH because it was straightforward to implement the notion of UA rulesets and counterexample generation.

Due to space constraints, this section contains a brief overview of the case studies. For a more detailed report, the reader may refer to supplementary material [25] including the Murφ sources.

A. Automatic Deadlock Freedom Predicates

As mentioned above, it is common when checking antecedent 1 of Theorem 1 to reach a U-dead-end state $\check{s}$ where no further progress can be made toward the goal. When this occurs, the model checker reports a failure and prints the rules of O that are enabled in $\check{s}$, as a guide to the user of which rules could be useful to prove UA and add to U. These enabled rulesets necessarily have tags AEG or AUG or both. We have written a simple tool that, given a particular rule/ruleset name, will determine the tags and generate the model checking obligation to prove it is UA through Heuristics 1 and 3.

Example: Suppose we seek to show ruleset $\widehat{r}_2$ = `ABS_SendGntE2` is UA, and suppose it is already known by Heuristic 3 that associated $\widehat{r}_1 = \widehat{r}_v$ = `ABS_SendGntE1` is UA. Ruleset `ABS_SendGntE2` is tagged with AEG because ruleset parameter i is abstracted. Then, $relax(\widehat{\rho}_v)|_{i=1}$ is

```
CurCmd = ReqE ∧ CurPtr = 1 ∧ ¬ExGntd
∧ forall j : NODE do ¬ShrSet[j] end
```

and $\widehat{\rho}_v|_{i=1}$ is

```
CurCmd = ReqE ∧ CurPtr = 1 ∧ Chan2[1].Cmd = Empty
∧ ¬ExGntd ∧ forall j : NODE do ¬ShrSet[j] end
```

As implemented, our tool does not support automatic generation of the properties to check for Heuristics 2 and 4. However, this is generally straightforward to do by hand, and could be automated as well. In cases when the deadlock freedom property for some Heuristic when applied to ruleset $\widehat{r}_j$ fails to verify, the user may use the counterexample trace as a guide for strengthening $\widehat{\rho}_j$ manually. Any ruleset of O may be duplicated and strengthened with some predicate, which is trivially sound because the O transitions are not changed. The resulting strengthened ruleset might satisfy the Heuristic deadlock freedom property and be proven UA or L-preserving. Such manual strengthening is required in the verification of both German and FLASH.

B. The German Protocol

The system used for O is the abstract Murφ model for German of Chou *et al.*, instantiated with a single non-abstracted node ($k = 1$). The initial set of UA transitions U_0 includes all rulesets with no guard tags and the local subset of these are also identified.

The property we verify is (4), where G states that that the directory is not currently processing a transaction (`CurCmd = Empty`) and $L[i]$ states that all communication channels associated with the ith cache are empty:

```
Chan1[i].Cmd = Empty ∧ Chan2[i].Cmd = Empty
∧ Chan3[i].Cmd = Empty.
```

Antecedent 1 of Theorem 1 requires $\mathcal{A} \models \mathsf{AG\,EF}\,(G)$ for a mixed-abstraction $\mathcal{A}$. Checking this property, the model-checker gets stuck at a U-dead-end state where the rule `ABS_SendGntE1` is enabled (see Section IV-B). Our tool recognizes this as a AUG-tagged rule and generates the obligations to according to Heuristic 1 so the rule can be soundly added to U. The model checker discharges the obligation, and `ABS_SendGntE1` is added to U.

Checking Antecedent 1 is repeated with the weakened U and gets stuck three more times: once where `ABS_SendGntE2` is enabled (tagged with AEG *and* AUG), and twice where other AEG-tagged rulesets are enabled. The Heuristic 3 obligation for `ABS_SendGntE2` is identical to the one previously shown for `ABS_SendGntE1`, so there is no need to repeat its verification. The tool generates the Heuristic 1 obligation and it is discharged by model checking. In the other two, AEG cases, the corresponding rulesets $\widehat{r}_v$ are already known to be in U, so we proceed directly with the tool and obligations for Heuristic 1 are generated. One is discharged automatically; the other requires human guidance because the generated deadlock freedom property fails to verify. An examination of the counterexample reveals that when exclusive access has been granted to an abstracted node, there is no pointer indicating which node has been granted (only a flag to indicate that it has indeed been granted, `ExGntd`). Without this pointer, the permutation of Heuristic 1 is not applied to the proper abstract node actually holding exclusive access. Although manual, the solution is straightforward: add a new system variable `EPtr` of type P that points to the node holding exclusive access, and strengthening the guard of the ruleset. This is done in a sound manner where only the ruleset version we prove is UA is strengthened in this way; the original ruleset belonging to O is not modified. After this modification, the relevant property is verified.

Having added these four rules to U of mixed abstraction $\mathcal{A}$, Antecedent 1 of Theorem 1 is established by model checking. We now describe the procedure to show of Antecedent 2. Initially, every ruleset with no tags are known to be L-preserving are added to U for mixed abstraction $\mathcal{B}$. Model checking then reveals that two additional rules are needed to establish the Antecedent: `ABS_SendGntE1` (tagged AUG) and `ABS_RecvInvAck2` (tagged AEG and AEC). These tags are discharged by automatically generating and checking the obligations of Heuristics 4 and 2, respectively. Adding these two rules to U for mixed abstraction $\mathcal{B}$ allows Antecedent 2 to hold and completes the verification of the German protocol.

C. The FLASH Protocol

The quiescence property verified of FLASH is of the same form as (4), and states that all channels are clear, and the directory is not waiting to perform a write-back[5]. Antecedent 1 of Theorem 1 holds immediately using the initial set of UA rulesets having no guard tags.

[5]Although the Murφ system for the mixed abstraction of FLASH contains rules where two index variables have been instantiated as *Other*, none of these must be shown UA/L-preserving to prove our example property. Some such rules are needed to be shown UA if the conjunct `¬Pending` is added to the quiescent property. We omit these from this paper for ease of presentation, but note that similar reasoning to Heuristic 1, which assumes only one such index variable, is sufficient.

To show Antecedent 2 of Theorem 1, we start with the set U of L-preserving states provided by tag examination and use model checking as with the German protocol. Four rules, each tagged with AEG and AEC, must be shown L-preserving. We first show that they are UA, by applying Heuristic 1. For two of these rulesets, model checking the obligations for Heuristic 1 succeeds. For the other two, model checking fails upon reaching a dead-end state $\check{s}'$ where no local rules to 1 are enabled. The manual strengthening needed in for these two ruleset is identical; without loss of generality let the ruleset be $\widehat{r}_j$. Inspecting the counter example reveals that the state $s \in relax(\widehat{r}_v, i)|_{i=1}$ that led to $\check{s}'$ has different values for some B-type variables that those in $\check{s}$, the original dead-end state revealed when checking Antecedent 2, where $\widehat{r}_j$ is enabled. This indicates that the guard $\widehat{r}_j$ is too weak and must be strengthened with a predicate on these variables. We duplicated the ruleset for the aforementioned reasons of soundness, and strengthened the guard with a predicate requiring these variables to match their value in $\check{s}$. Then, the automated procedure completed successfully and the four rules are established as UA. To show they are L-preserving, Heuristic 2 is applied to each ruleset and the obligations are discharged automatically; this establishes the quiescence property by Theorem 1. With regard to the manual strengthening step, we note that in principle the model checker could classify the reachable states of $relax(\widehat{r}_v, i)|_{i=1}$ for which a path to $\widehat{r}_v|_{i=1}$ is found versus those where no such path is found. Thus, the strengthening predicate could be generated automatically.

VII. Discussion and Future Work

We presented a practical method for proving deadlock freedom in parameterized cache coherence protocols. Our approach uses a mixed abstraction of over (OA) and under (UA) approximate transitions to parametrically verify properties of the form $\mathcal{P}(n) \models \mathsf{AG}\,\mathsf{EF}\, G \wedge \bigwedge_{i \in \mathsf{P}_n} L[i]$, where n is the number of cache *nodes*, G is a predicate depending on boolean variables, and L is a predicate depending on boolean variables local to each node i. We infer this parameterized property by model checking a pair of antecedent *deadlock freedom* properties in an automatically generated mixed abstraction.

First, the model checker explores all states s reachable through OA transitions, and for each s a UA-path to some G-state s' constructed via a forward search. When no such s' is found, the user determines if s is only reachable due to the overapproximation of OA, or if s' is unreachable from s due to UA being too strong. For the former, we use existing methods to strengthen OA. For the latter, we have presented heuristic methods to soundly weaken the UA transitions by proving some transitions of OA are in fact UA. These heuristics involve checking specific deadlock freedom properties in the mixed abstraction. Second, it is verified that all G-states reachable through OA transitions have an *L-preserving* UA-path to a state where G holds *and* $L[i]$ holds for all k nodes maintained by the abstraction. Abstract transitions that are L-preserving have the property that the set of corresponding paths in the concrete system will *preserve* $L[i]$ for all nodes i that are abstracted away. Once again, we provide heuristic methods for showing that OA transitions are in fact L-preserving UA transitions. With each of these antecedents established, a simple induction proof is used to show the parameterized deadlock freedom property holds.

For the German and FLASH protocols, the strengthening of OA that was required to prove safety [6] was sufficient to establish liveness as well. Furthermore, most weakenings needed for UA were identified and verified without need for human insight. The only places where human reasoning was needed was to identify one auxiliary variable needed in the German model, and three guard strengthenings for FLASH. We believe that these steps are candidates for automation as well. Such automation may be desirable when these techniques are applied to more complicated protocols.

We described an enhancement to the PReach model checker [26] to support the "EF" part of our model checking obligations using a forward search to find the existential paths. Though the reachable state computation is fully distributed, the EF searches currently are not; one area of future work is to distribute this aspect of the model checking. However, our current implementation running with a single thread was sufficient to handle all the obligations for our two case studies. The largest model was an abstraction of FLASH that has about 2.4 M reachable states and, for the properties of our case study, each was checked on a modern desktop machine in less than 10 minutes.

As another direction of future work, we are in the process of writing a Murφ model of the L2 cache controller in the OpenSPARC multiprocessor design. Here the parameter of interest is memory addresses, rather than cache IDs. This is interesting since different addresses share resources in non-trivial ways that can lead to deadlock in our experience with real designs. Investigating parameterized deadlock freedom of this cache controller will test the applicability of our approach of a vastly different parameterized verification problem.

Acknowledgment

We thank John O'Leary for providing an Ocaml Murφ front-end that is the basis for our tag/heuristics tool, and anonymous reviewers for constructive feedback. This research was funded in part by NSERC RGPIN 138501-07, NSERC graduate fellowships, and a grant by Oracle Corporation.

References

[1] R. C. Holt, "Some deadlock properties of computer systems," *ACM Computing Surveys*, vol. 4, no. 3, pp. 179–196, 1972.

[2] E. M. Clarke, O. Grumberg, and D. Peled, *Model Checking*. MIT Press, 1999.

[3] K. Apt and D. Kozen, "Limits for automatic verification of finite-state concurrent systems," *Information Processing Letters*, vol. 15, pp. 307–309, 1986.

[4] K. L. Mcmillan, "Verification of infinite state systems by compositional model checking," in *in CHARME*. Springer, 1999, pp. 219–233.

[5] K. L. McMillan, "Parameterized verification of the FLASH cache coherence protocol by compositional model checking," in *Correct Hardware Design and Verification Methods (CHARME)*, 2001, pp. 179–195.

[6] C.-T. Chou, P. K. Mannava, and S. Park, "A simple method for parameterized verification of cache coherence protocols," in *FMCAD*, 2004, pp. 382–398.

[7] S. Krstic, "Parameterized system verification with guard strengthening and parameter abstraction." in *Automated Verification of Infinite-State Systems*, 2005.

[8] Y. Lv, H. Lin, and H. Pan, "Computing invariants for parameter abstraction," in *MEMOCODE '07: Proceedings of the 5th IEEE/ACM International Conference on Formal Methods and Models for Codesign*, 2007, pp. 29–38.

[9] J. Bingham, "Automatic non-interference lemmas for parameterized model checking," in *Formal Methods in Computer Aided Design (FMCAD)*, 2008.

[10] Y. Li, "Mechanized proofs for the parameter abstraction and guard strengthening principle in parameterized verification of cache coherence protocols," in *Proceedings of the 2007 ACM symposium on Applied computing*, 2007, pp. 1534–1535.

[11] J. O'Leary, M. Talupur, and M. R. Tuttle, "Parameterized verification using message flows: An industrial experience," in *International Conference on Formal Methods in Computer Aided Design (FMCAD)*, 2009.

[12] D. Dams, R. Gerth, and O. Grumberg, "Abstract interpretation of reactive systems," *ACM Trans. Program. Lang. Syst.*, vol. 19, no. 2, pp. 253–291, 1997.

[13] Z. Manna and A. Pneuli, "Completing the temporal picture," *Theoretical Computer Science*, vol. 83, no. 1, pp. 97–130, 1991.

[14] K. L. McMillan, "Circular compositional reasoning about liveness," in *Correct Hardware Design and Verification Methods (CHARME)*, 1999, pp. 342–345, an extended version appeared as a Cadence technical report.

[15] K. McMillan, "Personal correspondence," 2011.

[16] Y. Fang, N. Piterman, A. Pnueli, and L. Zuck, "Liveness with invisible ranking," *Int. J. Software Tools for Technology Transfer*, vol. 8, no. 3, pp. 261–279, June 2006.

[17] A. Pnueli, J. Xu, and L. D. Zuck, "Liveness with (0, 1, infinity)-counter abstraction," in *Proceedings of the 14th International Conference on Computer Aided Verification (CAV)*, 2002, pp. 107–122.

[18] K. Baukus, S. Bensalem, Y. Lakhnech, and K. Stahl, "Abstracting WS1S systems to verify parameterized networks," in *International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, 2000, pp. 188–203.

[19] K. Baukus, Y. Lakhnech, and K. Stahl, "Parameterized verification of a cache coherence protocol: Safety and liveness," in *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, 2002, pp. 317–330.

[20] K. G. Larsen and B. Thomsen, "A modal process logic," in *Third Annual Symposium on Logic in Computer Science (LICS)*, 1988, pp. 203–210.

[21] R. Cleaveland, S. P. Iyer, and D. Yankelevich, "Abstractions for preserving all CTL* formulae," Tech. Rep., 1994, tech. Rep. 9403, Dept. of Comp. Sc., North Carolina State University, Raleigh, NC.

[22] N. Lynch and F. Vaandrager, "Forward and backward simulations – part I: Untimed systems," *Information and Computation*, vol. 121, no. 2, pp. 214–233, 1995.

[23] C. N. Ip and D. L. Dill, "Better verification through symmetry," in *11th IFIP WG10.2 International Conference on Computer Hardware Description Languages and their Applications*, 1993, pp. 97–111.

[24] D. L. Dill, A. J. Drexler, A. J. Hu, and C. H. Yang, "Protocol verification as a hardware design aid," in *IEEE International Conference on Computer Design: VLSI in Computers and Processors*, 1992, pp. 522–525.

[25] B. Bingham, J. Bingham, and M. Greenstreet, "Supplementary material," `http://www.cs.ubc.ca/~binghamb/fmcad2011.html`, 2011.

[26] B. Bingham, J. Bingham, F. M. de Paula, J. Erickson, G. Singh, and M. Reitblatt, "Industrial strength distributed explicit state model checking," in *Parallel and Distributed Model Checking*, 2010.

APPENDIX

BNF for Admissible Parameterized Systems

AdmissibleSystem : r | ps ';' ar ;
r : rule | ruleset ;
rule : guard '==>' action ;
guard : '(' gTerm ')' | guard 'AND' '(' gTerm ')' ;
gTerm : '(' bTerm ')' | '(' eTerm ')' | '(' fTerm ')' ;
bTerm : bConst | bVar | bArray '[' pVar ']' | bTerm 'AND' bTerm
| bTerm 'OR' bTerm | | 'NOT' bTerm | '(' bTerm ')' ;
pComp : pTerm '=' pTerm ;
pTerm : pVar | pArray '[' pVar ']' ;
action : assignment | action ';' assignment ;
assignment : simpleAssignment | forAllAssign ;
simpleAssignment : bVar ':=' bTerm | bArray '[' pVar ']' ':=' bTerm
| pVar ':=' pTerm | pArray '[' pVar ']' ':=' pTerm ;
forAllAssign : 'forall' pVar ':' pType 'do' assignment 'end' ;
ruleset : 'ruleset' pVar ':' pType 'do' r 'end' ;

Where *bVar* is an identifier for a boolean-scalar variable, *bArray* is an identifier for a boolean-array variable, *pVar* is an identifier for a scalar variable of the parameter type, and *pArray* is an identifier for an array variable of the parameter type.

Restrictions: Any *pVar* declared as a ruleset index may appear in at most one *pComp* conjunct of any guard.

Proof of Theorem 1

Let us fix a mixed-abstraction $\mathcal{A} = (S_\mathcal{A}, I_\mathcal{A}, U, O)$ for $\mathcal{P}(n)$, where $n > k$. We also use L to denote a local boolean predicate, and $\mathcal{B} = (S_\mathcal{A}, I_\mathcal{A}, U_\mathcal{B}, O)$ to denote a mixed abstraction with only L-preserving transitions for $U_\mathcal{B}$. For $i, j \in \mathsf{P}_n$, define $\mathsf{P}_i^j \subseteq \mathsf{P}_n$ as $\{\ell : i \leq \ell \leq j\}$.
Let permutation $\pi_{j \leftrightarrow h}$ map elements of P_n according to

$$\pi_{j \leftrightarrow h}(i) = \begin{cases} j & \text{for } i = h, \\ h & \text{for } i = j, \\ i & \text{otherwise.} \end{cases}$$

Let T be shorthand for $T(n)$ and let $Reach$ denote the reachable states of $\mathcal{S}(n)$.

Theorem 1 (Universally Quantified Quiescence): Let G be a boolean predicate. If

1) $\mathcal{A} \models \mathsf{AG}\,\mathsf{EF}\,(G)$, and
2) $\mathcal{B} \models \mathsf{AG}\,(G \rightarrow \mathsf{EF}\,(G \wedge \bigwedge_{i \in \mathsf{P}_k} L[i]))$

then $\mathcal{P}(n) \models \mathsf{AG}\,\mathsf{EF}\,(G \wedge \bigwedge_{i \in \mathsf{P}_n} L[i])$.

Proof: For $1 \leq h \leq n$, let J_h denote the property $\forall w \in G \wedge Reach.\ \exists w' \in (G \wedge \bigwedge_{i \in \mathsf{P}_h} L[i])$ where $w \rightsquigarrow_T w'$, and $\forall i \in \mathsf{P}_{k+1}^n.\ w \in L[i] \rightarrow w' \in L[i]$.
By definition, Antecedent 2 implies J_k. Assume J_h holds for $k \leq h < n$. Applying permutation $\pi_{1 \leftrightarrow h+1}$ to J_k gives $\forall w \in G \wedge Reach.\ \exists w' \in (G \wedge \bigwedge_{i \in \mathsf{P}_2^{h+1}} L[i])$ where $w \rightsquigarrow_T w'$, and $\forall i \in \mathsf{P}_{k+1}^n.\ w \in L[i] \rightarrow w' \in L[i]$. This property with Antecedent 2 implies J_{h+1} by transitivity. Thus, property J_n follows by induction. The paths implied by Antecedent 1 composed with those of J_n complete the proof by transitivity. ■

Scaling Probabilistic Timing Verification of Hardware Using Abstractions in Design Source Code

Jayanand Asok Kumar, Lingyi Liu and Shobha Vasudevan
Department of Electrical and Computer Engineering
University of Illinois at Urbana-Champaign
{jasokku2, liu187, shobhav}@illinois.edu

***Abstract*—Sources of randomness such as physical process variations and input pattern variations make hardware timing a statistical measure. It is desirable to verify statistical timing properties at the higher levels of design such as the Register Transfer Level (RTL). The RTL design can be modeled as a Discrete Time Markov Chain (DTMC) and probabilistic model checking then applied to verify that the DTMC satisfies a desired timing specification. However, we find that such an approach does not scale beyond 10^{10} states. In this paper, we introduce an abstraction methodology to scale this approach to large designs. Instead of considering the entire space of data values that can be assigned to the design input variables, we perform a *value-based interval* abstraction by considering only those intervals of input values that are relevant to a given timing property. We employ *symbolic execution* on the RTL source code to automatically derive such intervals for the design inputs, with respect to a given timing property. We use these intervals to construct smaller abstract DTMCs and thereby make the corresponding probabilistic model checking problems more tractable. We show that our abstraction is sound since we do not remove any probabilistic behavior that is relevant to the property of interest. We demonstrate the effectiveness of our technique using multiple designs used in communication systems such as FFT, filters and several modules of a real world H.264 decoder. We use our technique to successfully verify timing of an H.264 module, for which the concrete model contains more that 10^{80} states, by constructing an abstract model with approximately only 10^{10} states.**

I. Introduction

Adaptive techniques like voltage and frequency scaling, process variations that affect physical device parameters [1], aging effects [2] and physical faults [3] contribute significantly to the stochastic nature of contemporary hardware. As a consequence, the timing associated with hardware computations is also statistical in nature. Recent high performance designs [4] allow long computations to make timing errors that can eventually be corrected. Therefore, contemporary semiconductor environments are increasingly interested in the question: *"What is the probability that the correct hardware output is available with a delay less than a timing specification T?"* . Such information, if available at the higher levels of design such as Register Transfer Level (RTL), would facilitate informed choices early in the hardware design cycle and avoid oversights that may prove costly in the later stages.

Traditionally, RTL verification checks functional correctness and adherence to timing specification is considered at the lower, circuit level in the design cycle. Due to the growing sources of variations in lower level hardware, it is desirable to incorporate statistical timing into the definitions of correctness at the higher, RTL. Viewing RTL designs as probabilistic entities, with non-deterministic notions of correctness opens the door to using formal verification for new sources of uncertainty like process variation and aging.

Probabilistic model checking based techniques [5] [6] [7] can be used for verifying timing properties of hardware designs in the presence of statistical variations. However, from our experience [8], we find that such an approach is limited by the capacity of the probabilistic model checking engine to less than 10^{10} states.

In this work, we present a *value-based interval abstraction* technique to mitigate the state space explosion during probabilistic model checking of RTL designs. We perform our abstraction with respect to the timing property $P[Delay < T]$. We treat RTL source code descriptions as "programs" [9]. As in the case of non-probabilistic RTL verification [10] [11], we perform our property-specific abstraction by using static analysis at the RTL design source code level (Figure 1). Abstractions performed in non-probabilistic verification produce smaller Kripke structures. As an analogue, our abstraction produces smaller DTMCs that make probabilistic model checking feasible for large RTL designs. In the abstract DTMC that we obtain, all the states of the original DTMC that are not relevant to the specified timing property are lumped together.

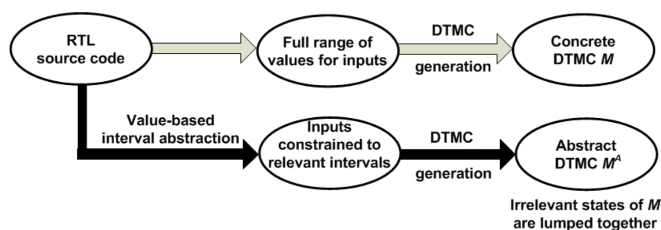


Fig. 1. Our value-based interval abstractions are applied at design source code level, leading to smaller DTMCs.

We demonstrate the application of our technique on multiple

practically useful designs that could not otherwise be verified due to space limitations. Timing cannot be compromised in microprocessor control logic. It is only in the datapath that probabilistic timing is acceptable. Datapath verification is notoriously hard [12] to get guarantees for. Our abstractions are intended to verify probabilistic timing on the datapath of RTL designs. We consider filters and FFT blocks that are widely used in communication/DSP systems, as well as an H.264 decoder. We show consistent and significant reductions in state space which make probabilistic model checking of these RTL designs feasible. For example, we are able to model check a module in the H.264 decoder for which the concrete model contains over 10^{80} states. The abstract DTMC contains approximately 10^{10} states.

In previous work [8] [13], we introduced a methodology for formally verifying the statistical timing properties of an RTL design. We statically analyzed Verilog [14] RTL source code to determine statistical correlation among the RTL signals (*i.e*, variables in RTL). We combined this statistical information with gate level delay models and represented RTL designs as finite DTMCs. We then used the probabilistic model checking tool, PRISM [15], to verify that an RTL DTMC M satisfies a statistical timing property ϕ, denoted by $M \models \phi$.

We use this framework to describe our value-based interval abstractions. We are interested in properties ϕ of the form $P[exp(V) < T]$, where exp is a real valued function that is defined over the set of RTL variables V and T is a user-specified value. $exp(V) < T$ is a *predicate* that evaluates to TRUE or FALSE in a DTMC state depending on the numeric values assigned to V in that state (for example, $Delay < T$). When we verify $M \models \phi$, we are actually computing the probability of being in a DTMC state where the predicate is TRUE. Therefore, among all possible concrete states of M, only those states where the predicate $exp(V) < T$ evaluates to TRUE are relevant. Each state of the DTMC M corresponds to a unique assignment of values to the input variables in the RTL design [8]. We restrict inputs to intervals of values (*value-based intervals*) such that only the relevant states of M are generated during DTMC construction. All the irrelevant states are lumped together to a single representative state. Lumping is a well-known abstraction approach for DTMCs [16] [17]. We show that, using this elegant abstraction, we are able to handle complex RTL designs. The complexity of our technique is not as much in the abstractions as in the process of obtaining them automatically from hardware descriptions.

Value-based intervals for RTL inputs can be used to construct an abstract DTMC M^A by lumping the irrelevant states together even at the model construction stage in the probabilistic model checker. In order to derive these intervals, we first consider the predicate $exp(V) < T$ as a symbolic constraint on the values of variables in V. We rewrite such symbolic constraints as constraints that are expressed over the input variables. We achieve this by performing *symbolic execution* [18] on the RTL source code. We use an integer constraint solver to obtain lower and upper bound values of the intervals for these inputs by maximizing (or minimizing) the value of the input for which the predicate $exp(V) < T$ is satisfied. We use these intervals while describing the model in the probabilistic model checker which then constructs the abstract DTMC M^A and checks if $M^A \models \phi$. We show that the abstract DTMC M^A is an exact reduction of the concrete DTMC M, *i.e.* $M^A \models \phi$ iff $M \models \phi$.

The value of our work is twofold. Firstly, we scale probabilistic model checking, by adapting symbolic execution techniques to hardware, and integrating them with other techniques known in software. To the best of our knowledge, we are the first to use these set of techniques in the context of probabilistic model checking for RTL designs. Secondly, we demonstrate that, using our technique, it is feasible to reason reliably about very low level physical variations.

II. Preliminaries: Probabilistic model checking of RTL designs

We now describe the framework that we use for formally representing RTL designs in order to employ probabilistic model checking. We reuse some of the model definitions from our previous work [13].

We shall use the following example RTL source code in order to illustrate the steps of our abstraction technique (Section IV).

```
always @(posedge clk)
if (sel)
  O1 <= I1 + I2;
else
  O1 <= 4*I2 + I3;
end
```

where $I1$, $I2$, $I3$ are the inputs, sel is a Boolean control variable and $O1$ is the output. All input and output variables are of 10 bits and can therefore be assigned 1024 different numeric values.

The *always @(posedge clk)* blocks can be thought of as processes that are executed in parallel at every rising edge of the hardware clock signal which is considered as a time step. At any time step t, the <= operator evaluates the *right-hand side* (RHS) value and assigns it to the *left-hand side* (LHS) variable in the next step $t + 1$. Typical data intensive RTL designs that are used in communication/DSP systems mostly perform arithmetic operations.

A. Variables in RTL designs

In RTL source code, variables are used to represent the data that is processed in hardware. A variable v can be assigned integer values in the range $[l\ u]$ where l and u denote the lower and upper bound, respectively. For an N-bit variable, we assume the default range of values to be [0 2^N-1]. We refer to the probability distribution of v as the PMF of v. We define a set of variables V to be independent if all the variables in V are mutually independent. The joint PMF of an independent set V is simply a product of the individual PMFs of the variables $v \in V$.

We assume knowledge of the distribution of primary input variables and that they are independently distributed. However, our approach is not limited to designs with independent primary inputs. We also assume stationary* probability distributions for our inputs, and therefore for all variables in the system. Such an assumption is reasonable since statistical timing/aging analysis of hardware datapaths is often performed by considering time-invariant statistical distributions for input data [2] [20]. The PMFs of stationary variables are independent of time. Therefore, the values assigned to stationary variables in two different time steps are statistically independent of each other.

Let V be the set of all variables in a system and $I \subset V$ be the set of input variables. In order to find the PMFs of a variable $v \in V$ from the PMFs of I, we need to find the function f such that $v = f(I)$. We call f a *system function*. For a variable v, $f(I)$ is the symbolic expression that includes inputs, or the "formula" that corresponds to its evaluation. $f(I)$ may comprise Boolean or arithmetic operators that are allowed in the source code. The *support* of v, denoted by $Sup(v)$, is the set of all input variables in the expression corresponding to $f(I)$. $Sup(v)$ is a subset of I, *i.e.* $Sup(v) \subseteq I$.

The values assigned to the control variables activates/selects one among several possible paths in the RTL design. Each path may result in a different assignment to a variable. Therefore, for each path i, a system function f_i needs to be defined for each variable v that is of interest. However, $Sup(v)$ is computed by considering all possible paths. In the RTL example, there are two possible paths (sel=0 and sel=1) and $Sup(O1) = \{I1, I2, I3\}$.

B. Modeling RTL designs as DTMCs

In an extension of [9], we model both input and process variations. Since our abstraction technique is not dependent on the type of variation, we consider only input variations in this work. Therefore, the probabilistic behavior of a variable of interest, v, can be completely represented by the inputs $Sup(v)$, along with their joint PMF. We now describe the process of representing an RTL design by using a finite-state probabilistic system, namely a finite DTMC.

A DTMC can be completely specified by using a triple (S, $Trans$, μ_0) where S is the set of state variables, $Trans$ is the probabilistic state transition relation and μ_0 is the initial state. Each state μ of the DTMC corresponds to a unique assignment of values to the variables in S.

We construct the DTMC model M for a variable v, with the support of v being the state variables ($S = Sup(v)$). We define the initial state by setting the value of all state variables to 0. Each hardware clock cycle corresponds to a time step in which new values are assigned to the variables. Therefore, each such time step corresponds to a DTMC transition from one state μ to another state μ' that corresponds to the new assignment of values to $Sup(v)$.

The probability of a transition to a new state μ' is equal to the probability with which the corresponding new values of the state variables are assigned to $Sup(v)$. Since all the variables in the $Sup(v)$ are assumed to be independent, we obtain the state transition probabilities by taking the product of the individual probabilities of all variables (Section II.A). If we do not assume independence for the inputs, we would use the specified joint PMF of $Sup(v)$. All such possible state transitions labeled with the corresponding probabilities constitute $Trans$ for the DTMC M.

If a set of variables Π are of interest, we construct the corresponding DTMC M such that

$$\begin{aligned} S &= Sup(\Pi) \\ &= \bigcup_{v \in \Pi} Sup(v) \end{aligned} \quad (1)$$

In the RTL example, $O1$ is the variable of interest. Therefore, we construct the corresponding DTMC with $Sup(O1) = \{I1, I2, I3\}$ as state variables.

C. Model checking a statistical timing property in RTL

We now describe the notion of delay in RTL presented in [8] and how we formally represent a statistical timing property.

1) Modeling delay in RTL: We consider delay in terms of RTL assignment statements. The delay of an RTL assignment depends on the operator and the values of the operands in the RHS. We consider real-valued analytical functions exp, which we call *macromodels* [8], that estimate the delay of an operator based on the value of the operands.

For each RTL operator, we derive the macromodel exp by performing extensive simulations of a gate-level implementation of the operator. We repeat this for several possible implementations of each RTL operator and construct a library of macromodels. We perform this whole macromodeling process offline for a given technology library.

In the RTL example, the delay of $I1 + I2$ can be computed by using the macromodel $exp(I1, I2)$ corresponding to the specified adder implementation. With a Ripple Carry Adder implementation, exp is a polynomial function of the number of carry bits in the addition of $I1$ and $I2$. Further details of the macromodeling process can be found in [8].

Each state in the RTL DTMC is associated with an RTL delay which can be computed based on the values of the RTL inputs (*i.e.*, state variables) in that state. We "tag" each state with the associated RTL delay which we compute by using the appropriate macromodel. Each DTMC transition represents a change in value of the RTL inputs and does not contain any information regarding the RTL delay.

2) Specifying an RTL timing property: We wish to compute the probability that the RTL delay meets a timing requirement T. The delay of an RTL block can be expressed as a combination of the macromodels of all the operators in the block [8]. Let this RTL delay be denoted by $exp(\Pi)$, which is an expression defined over a set of variables, $\Pi \subseteq V$. We define probabilistic invariants Γ [21] based on the timing requirement of the design, given by

$$\Gamma \triangleq P[exp(\Pi) < T] \quad ^{\dagger} \quad (2)$$

*A function of stationary variables is also stationary [19].

†In place of $<$, we allow for the use of other relational operators as well.

where T is a real-valued constant and $exp(\Pi) < T$ is the *predicate* that is of interest to us. P[Predicate = TRUE] denotes the probability that the predicate is satisfied (*i.e.* TRUE) by an assignment of concrete numeric values to Π. Γ is the probability of being in a state (*i.e.*, an input pattern) where the tagged delay is less than T.

We formally define probabilistic timing properties ϕ of the form,

$$\phi \triangleq \Gamma \leq p \tag{3}$$

where $p \in$[0,1] is a specification of the design. We allow logical comparison operators other than $\leq$ to be used for comparing the probabilistic invariant with p.

We are interested in computing Γ for values of Π at some time step t. For all the variables in Π, we consider the values assigned to them in the same time step. Since we assume the probabilities to be stationary, the value of t does not affect the correctness of our approach. In this paper, we omit the index t in order to simplify our notation. In this regard, our properties can be thought of stationary/steady-state properties that are not dependent on time.

We employ probabilistic model checking and verify that a DTMC M satisfies a property ϕ, denoted by $M \models \phi$. The model checking procedure for properties described in Equation 3 involves the computation of the invariant Γ and comparing it with p. If p is not specified, verifying $M \models \phi$ is equivalent to the computation of Γ. In this paper, we use PRISM [15] as the probabilistic model checking engine.

Probabilistic model checking explores all possible behaviors of the DTMC (*i.e.* all values of RTL inputs) and therefore, computes the exact probability with which the timing requirement is met.

D. Describing DTMC models in PRISM

In PRISM, we describe a DTMC M by defining the assignments to each state variable $s_v \in Sup(\Pi)$, independently. Let s_v correspond to an N-bit input variable in RTL. s_v can be assigned a value $j \in 0, 1, ..2^N - 1$ with probability p_j. We model this in PRISM by the statement,

$p_j : (s'_v = j);$

Therefore, there are 2^N statements corresponding to the description of s_v. If there are K such N-bit variables, $2^N * K$ assignment statements are required. PRISM supports assignment statements for multiple state variables. However, this approach would require 2^{NK} statements, which is inefficient.

III. Our abstraction using value-based intervals

In this section, we define and establish criteria for performing value-based interval abstractions on probabilistic systems of our interest, namely RTL designs. We perform our abstraction by statically analyzing the RTL source code. In our approach, we directly generate the abstract DTMC M^A without generating the concrete DTMC M first.

Let Λ be the predicate that is specified in the property ϕ. Let Π be the set of RTL variables over which Λ is expressed. We construct the DTMC M using $Sup(\Pi)$ as state variables.

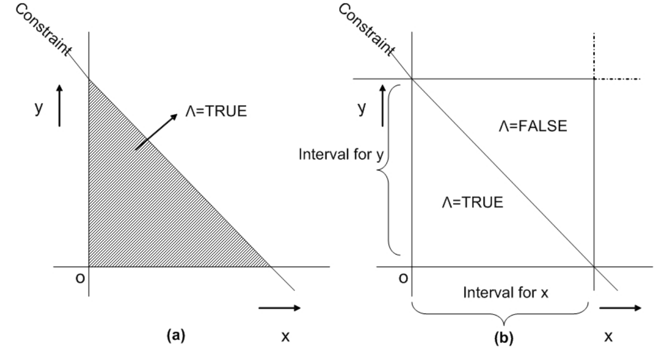


Fig. 2. a) Exact constraint for values of inputs x,y b) Conservative value-based intervals for inputs x,y

We wish to verify whether M satisfies ϕ, denoted by $M \models \phi$. In other words, we wish to compute the probability of being in a DTMC state where Λ is TRUE. This can be achieved by considering a smaller DTMC M^A that contains all the states of M where Λ = TRUE. M^A is the abstract DTMC model corresponding to the concrete DTMC M. Since each state of M corresponds to a unique assignment of concrete numeric values to the input variables $Sup(\Pi)$, the construction of M^A corresponds to retaining only those values of inputs for which Λ = TRUE. All other values of the inputs are inconsequential and can be lumped together by using a single representative value. This forms the basis of our data abstraction technique.

Λ = TRUE imposes a constraint on the values that can be assigned to the variables in Π. In order to construct an abstract DTMC M^A, we wish to use this constraint to determine the concrete values of the input variables $Sup(\Pi)$ that need to be considered. We achieve this by using RTL symbolic execution [18] to rewrite the constraint on variables Π as a constraint on inputs $Sup(\Pi)$. Symbolic execution statically explores all possible paths through the RTL design and determines a constraint C_i on the values of $Sup(\Pi)$, for each path i.

Each constraint C_i specifies an exact bound on the values of $Sup(\Pi)$ for which Λ =TRUE on path i. However, in general, C_i is specified jointly over multiple input variables in $Sup(\Pi)$. C_i cannot be included in the PRISM model description since we define assignments to input variables independently (Section II.D). Therefore, we use a constraint solver (ILP) with C_i to derive *value-based intervals* for each input variable in $Sup(\Pi)$. Since we wish to compute the probability of Λ = TRUE for all paths through the design, we construct an abstract interval ψ_{abs} for v that includes all the values from the intervals computed using each C_i.

Finally, we use the abstract intervals for each $v \in Sup(\Pi)$ in order to construct the abstract DTMC M^A. We then verify $M \models \phi$ by checking $M^A \models \phi$

For each $v \in Sup(\Pi)$, we consider all values of v such that there is a possible assignment of values to the other input variables $\in Sup(\Pi) \setminus \{v\}$ that would satisfy Λ =TRUE. Therefore, the value-based intervals that we construct are conservative (Figure 2). M^A may contain states from M in

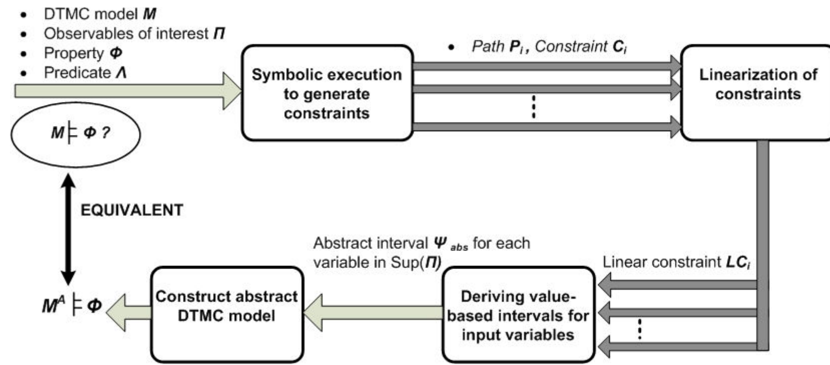


Fig. 3. Block diagram showing the stages of our predicate-based data abstraction technique. The labels on the arrows show the outputs of each stage.

which Λ=FALSE. However, we do not discard any state from M in which Λ= TRUE. We show that our abstraction is sound with respect to the probabilistic property of interest.

Most existing abstractions for probabilistic model checking, excepting a few recent ones such as [22] [23], operate on the concrete DTMC. Instead, we perform our abstraction entirely at the RTL source code level prior to DTMC construction. Since we specify the abstracted intervals in the PRISM model description, we directly generate the abstract DTMC and circumvent the capacity issues of PRISM that are associated with generating the larger concrete DTMC.

IV. Algorithm for value-based interval abstraction

We wish to construct an abstract DTMC M^A in order to determine $P[(exp(\Pi) < T)]$, where $exp(\Pi) < T$ is the predicate of interest, Λ. Figure 3 shows the different steps in our abstraction technique. We now describe each of these steps in detail.

We shall illustrate our technique using the RTL example in Section II. Let $P[O1 < 100]$ be the invariant that we wish to compute.

A. Symbolic execution to generate constraints

We use symbolic execution to explore each possible path i in the RTL design and generate a corresponding constraint C_i on the input variables. For each path i, let v=$f_i(Sup(v))$ where f_i is the system function for variable v. Therefore, a predicate $exp(v) < T$ can be written as $exp(f_i(Sup(v))) < T$ which is a constraint C_i on the values of the input variables $Sup(v)$.

Symbolic execution refers to the execution of a single path with symbolic inputs. Symbolic execution of a path generates symbolic expressions that are a logical conjunction of the guards (conditional expression of branches) and assignments to the variables used in guards along that path. Symbolic execution is well known in software [24]. In recent work [18], symbolic execution has been introduced for RTL source code. The RTL symbolic execution engine works on the CFG and expression tree structure of each RTL "program" statement. For each single statement or conditional expression in the design, the expression tree structure exactly records the corresponding assignment or expressions and is linked to corresponding CFG node.

The RTL symbolic execution engine [18] considers exactly one path i of the CFG at any given time. At each CFG node in path i, the corresponding expression tree is traversed and output as symbolic expression. When a variable $v \in \Pi$ is encountered, the corresponding system function $f_i(Sup(v))$ is output by the engine. Every occurrence of $v \in \Pi$ in $exp(V) < T$ is substituted with the corresponding system function $f_i(Sup(v))$. We thus obtain the constraint C_i on $Sup(v)$ corresponding to the path i. We repeat this for all possible paths i in the RTL design and obtain the corresponding constraints on the input variables. Further details of the RTL symbolic execution engine, along with an optimization strategy for path exploration, can be found in [18].

In the RTL example (Section II), we obtain the linear constraints $I1$+$I2$ <100 and $4 * I2$+$I3$ <100 corresponding to the paths sel=1 and sel=0, respectively.

B. Linearizing the constraints

A linear constraint will have *terms* on the left hand side that are separated by +/- signs. Each term can be a variable multiplied by a constant numeric value. Since datapaths of RTL designs comprise mainly of arithmetic operators, each constraint C_i is typically a linear constraint that is defined over the input variables. However, if the constraints are not linear, we transform them into a set of linear constraints.

In Figure 4, we outline a set of rules for transforming non-linear operations into linear constraints. All the rules that we have defined can be extended easily for relational operators other than <. The terms $(X >> m)$ and $(X << m)$ represent shifting the variable X by m bits towards the right and left, respectively. These operations are equivalent to division and multiplication by 2^m, respectively.

If there is a term corresponding to multiplication of non-constants, we split the term into a set of linear constraints. Let $X1 * X2$ be the non-linear term in the constraint C_i. We treat $X1 * X2$ as an input variable and compute its upper bound T_i (Section IV.D). We then rewrite this term as two linear constraints $LC1$ and $LC2$, as in Figure 4.

Operation	Constraint C_i	Linear constraints
Left shift	$(X << m) < T_i$	$(X * 2^m) < T_i$
Right shift	$(X >> m) < T_i$	$(X / 2^m) < T_i$
Multiplication	$(X1 * X2) < T_i$	$X1 < T_i$ $X2 < T_i$
Power functions	$X^q < T_i$	$X < \sqrt[1/q]{T_i}$

(a)

Objective Function: max v

Set of linear constraints:
LC_i

$v_j \geq 0$, v_j is an integer
(for all v_j that appears in LC_i)

(b)

Fig. 4. a) Rules for linearizing constraints, b) ILP instance for computing upper bound of v.

Concatenation of variables is supported in RTL designs. Let $X1$, $X2$ be $n1$-bit and $n2$-bit variables, respectively. The variables can be treated as strings of bits and concatenated to get a string of $n1+n2$ bits, represented by the term $\{X1, X2\}$. This algebraic operation corresponding to this term can be rewritten as the linear expression $X1 * 2^{n2} + X2$.

We apply the above rules recursively to each non-linear constraint and derive a set of linear constraints LC_1 to LC_{NumLC}, where $NumLC$ is the total number of linear constraints. The rules that we have defined in Figure 4 are not complete, since RTL designs support several other operators. However, our rules are sufficient for the large class of datapath RTL designs that are used in DSP systems.

In general, the predicate can be expressed as a polynomial function over variables Π. In such cases, we can define rules to convert non-linear terms such as $(X^q < T_i)$ into corresponding linear terms $(X < \sqrt[1/q]{T_i})$ (for $q > 1$ and non-negative X). However, in this paper, we only consider predicates that are linear functions over Π.

C. Deriving value-based intervals for input variables

We consider a linear constraint LC_i. For each input variable v that appears in the expression for LC_i, we wish to compute the interval $\psi^{(i)} = [l^{(i)}\ u^{(i)}]$ of values that can be assigned to it. We achieve this by formulating an instance of the ILP problem.

Figure 4 shows the ILP instance for computing the upper bound of v. Each ILP instance comprises one linear constraint LC_i, and a set of constraints that force all variables v_j (including v) that appear in LC_i to be non-negative integers. The objective of the the ILP problem is to maximize the integer value of v such that all the constraints are satisfied.

If the ILP instance has an optimal solution, we set $u^{(i)}$ to be equal to that solution. If the ILP instance is "unbounded", it implies that all non-negative integer values for v will satisfy the given set of constraints. In this case, we set $u^{(i)}$ to the default upper bound (*i.e.* 2^N -1, as in Section II.A) and mark v as a *free variable*. Similarly, we compute $l^{(i)}$ by changing the objective function to $min\ v$.

We perform this interval computation for all linear constraints. If a variable is marked to be free, we do not compute its intervals for any of the subsequent linear constraints.

Finally, we compute the most conservative interval for each input variable v, by computing the union of the intervals $\psi^{(i)}$ that are obtained using the linear constraints LC_i. We call this the *abstraction interval* ψ_{abs} for the variable v.

$$\psi_{abs} = \bigcup_{i=1}^{NumLC} \psi^{(i)} \tag{4}$$

In the RTL example (Section II), we compute the intervals [0 99] for both $I1$ and $I2$ based on the sel=1 path. Based on the constraint in the sel=0 path, we compute the intervals [0 24] and [0 99] for the variables $I2$ and $I3$, respectively. After computing the union of the two intervals for $I2$, we observe that ψ_{abs} for all $I1$, $I2$ and $I3$ is equal to [0 99].

If there is a "-" sign in the left hand side of the constraint, all the variables that appear in the constraint will be unbounded. For example, it is possible for each value of the variable $X1$ (and $X2$) to satisfy the constraint $X1 - X2 < 100$. However, it is possible to compute a lower bound for $X1$ if the $>$ operator is used in the constraint instead of $<$.

In this work, we have considered *unsigned* arithmetic where RTL variables are interpreted to have non-negative integer values. However, the rules in Figure 4 can be easily extended to the case of *signed* arithmetic, where negative integer values are also allowed.

D. Describing the abstract DTMC model

We use the abstraction intervals in order to describe an abstract DTMC model M^A in PRISM. For each variable $\{v : v \in Sup(\Pi)$ and v is not free$\}$, we update the assignment statements (Section II.D) in the corresponding module in PRISM. We select a non-negative integer $z \notin \psi_{abs}$. For each statement that assigns a numeric value $j \notin \psi_{abs}$ to v, we replace j with z. Therefore, z is a single value that we use to represent all numeric values of v that lie outside the abstraction interval. With the updated description, the DTMC constructed by PRISM is the abstract model M^A.

In the RTL example (Section II), we use the numeric value 100 to represent all values of $I2$ outside the interval [0 99]. Therefore, all DTMC states corresponding to values of $I2$ outside [0 99] are lumped to a single state in which $I2$ = 100.

V. Mapping from concrete to abstract DTMCs

We now describe how our abstraction maps the states and transitions of the concrete DTMC to those of the abstract DTMC. We will denote abstractions with respect to Λ by $\alpha(\cdot, \Lambda)$.

We first define the abstraction for states. For each state μ in the concrete DTMC M, $\alpha(\mu, \Lambda) = \mu^A$, where μ^A is a state in the abstract DTMC M^A. There are two possible outcomes under the abstraction:

1) μ^A=μ
   This necessarily happens if Λ = TRUE in state μ of DTMC M. This can also happen if Λ = FALSE in μ but there exists at least one state variable whose valuation in μ lies within its corresponding abstraction interval ψ_{abs}.
2) μ^A=$\mu^\ulcorner$
   This implies that Λ = FALSE in state μ of DTMC M and the values assigned to all state variables lie outside their corresponding abstraction intervals. All such states in M are mapped to a single representative state $\mu^\ulcorner$ in M^A

We now define the effect of α on the state transition probabilities of M. Let $p(\mu_1^A \rightarrow \mu_2^A)$ denote the probability of transition from state μ_1^A to state μ_2^A in M^A. We consider the following cases:

Case 1: $\mu_1^A \neq \mu^\ulcorner$ and $\mu_2^A \neq \mu^\ulcorner$

$$p(\mu_1^A \rightarrow \mu_2^A) = p(\mu_1 \rightarrow \mu_2) \quad (5)$$

where $\alpha(\mu_1, \Lambda) = \mu_1^A$, $\alpha(\mu_2, \Lambda) = \mu_2^A$

Case 2: $\mu_1^A \neq \mu^\ulcorner$ and $\mu_2^A = \mu^\ulcorner$

$$p(\mu_1^A \rightarrow \mu_2^A) = \sum_{\mu_i \in M : \alpha(\mu_i, \Lambda) = \mu^\ulcorner} p(\mu_1 \rightarrow \mu_i) \quad (6)$$

where $\alpha(\mu_1, \Lambda) = \mu_1^A$.

Case 3: μ_1^A=$\mu^\ulcorner$ and $\mu_2^A \neq \mu^\ulcorner$

$$p(\mu_1^A \rightarrow \mu_2^A) = p(\mu_i \rightarrow \mu_2) \quad (7)$$

where $\alpha(\mu_2, \Lambda) = \mu_2^A$ and μ_i is any state in M. Since the probability distributions for all input variables are stationary (Section II.A), the probability of reaching any state $\mu_2 \in M$ is independent of the previous state. Therefore, the exact identity of state μ_i in Equation 7 is irrelevant and our abstraction does not remove any relevant probabilistic behavior. The same applies for Equation 5 as well.

Proof of correctness:
We now present a brief proof intuition for the soundness of our technique. We wish to prove that $M \models \phi$ is equivalent to $M^A \models \phi$. We achieve this by using the Strong Lumping Theorem [16] [17] to show that M^A is a probabilistic bisimulation [25] of M with respect to ϕ.

α can be thought of as an equivalence relation between the states in M and M^A. α relates a state μ in M to the state $\mu^A = \alpha(\mu, \Lambda)$ in M^A. By construction, α also preserves the valuation of Λ. Therefore, μ^A is locally equivalent to μ with regard to Λ. In fact, all states μ^A in M^A can be viewed as equivalence classes of M under the relation α. With the exception of $\mu^\ulcorner$, all equivalence classes μ^A contain only one state.

The abstract model M^A can be thought of as a quotient DTMC that comprises of equivalence classes defined by α. Equations 5, 6 and 7 can then be used to invoke the Strong Lumping Theorem and prove that M^A is a probabilistic bisimulation of M.

VI. Experimental Results

We implement the RTL symbolic execution algorithm using C++. We perform all our experiments on an Intel i5 2.67GHz quad-core machine with 16GB of memory. We use *lpsolve* [26], an open source ILP solver, in order to solve the set of integer linear constraints and derive the value-based intervals for the inputs.

We demonstrate the effectiveness of our methodology by applying it on two sets of data-intensive RTL designs. The first set of designs comprise `fir`, `elliptic` and `fft8` all of which are high-level synthesis benchmarks [27] that are commonly used in communication/DSP systems. Filter coefficients are fixed and stored in a ROM table. We consider constant values for these coefficients. `Inter_pred_LPE`, `Inter_pred_pipeline` and `Inter_pred_sliding_window` are different modules from a real-world H.264 decoder[‡] and constitute our second set of designs. In this work, we analyze each of the H.264 modules independently.

In Table I, *Number of paths* represents the total number of paths that needs to be explored during symbolic execution. This number is with regard to the variables which appear in the predicate (described in Table II) of the specified property. Since our designs do not contain multiplication of variables with each other, there should be exactly one linear constraint per path (Section IV.B). However, in some paths, all variables are assigned a constant value and therefore, the predicate is vacuously TRUE or FALSE. We discard these paths and consider only the linear constraints (*Number of constraints*) corresponding to the remaining paths while formulating the ILP instances.

In Table I, *Number of inputs* represents the total number of input variables (and their bitwidths) on which the variables in the predicate depend, *i.e.* $Sup(\Pi)$. Therefore, each of these input variables appear in at least one of the linear constraints. However, in each linear constraint, at most a small subset of these input variables are present. Therefore, each ILP instance is small and the corresponding runtime of *lpsolve* is also negligibly small. The total abstraction time, which includes the time for both the generation of linear constraints and the ILP solver, is less than 10 seconds in all our experiments.

For each of the designs that we consider, we specify a property that is defined over some internal data variables. Table II provides a description of all the predicates that we define in order to specify the properties of interest. `fir`

[‡] www.opencores.org

TABLE I
SIZES OF THE ILP INSTANCES THAT WE USE TO DERIVE THE INTERVALS FOR INPUT VARIABLES.

Design name	Predicate name	Number of inputs	Number of paths	Number of constraints	Abstraction time
fir	p8	6 (8-bit)	1	1	<10s
elliptic	p9	12 (8-bit)	1	1	<10s
fft8	p10	8 (16-bit)	4	4	<10s
Inter_pred_LPE	p1	5 (8-bit)	180	131	<10s
Inter_pred_LPE	p2	5 (8-bit)	180	131	<10s
Inter_pred_LPE	p3	5 (8-bit)	180	131	<10s
Inter_pred_pipeline	p4	32 (8-bit)	1936	1932	<10s
Inter_pred_pipeline	p5	3 (8-bit)	8	5	<10s
Inter_pred_sliding_window	p6	19 (8-bit)	29	21	<10s
Inter_pred_sliding_window	p7	16 (8-bit)	29	21	<10s

TABLE II
DESCRIPTION OF THE PREDICATES THAT WE USE TO SPECIFY PROPERTIES OF OUR INTEREST. TO VERIFY THESE PROPERTIES, WE COMPUTE P[PREDICATE = TRUE].

Predicate name	Predicate description
p1	bilinear0_A + bilinear0_B $<$ 8
p2	bilinear0_A + bilinear0_B $<$ 6
p3	bilinear0_A + bilinear0_B $<$ 4
p4	8*Inter_blk_mvx + Inter_blk_mvy $<$ 2
p5	Inter_pred_out0 $<$ 200
p6	Inter_pix_copy0 $<$ 2
p7	Inter_H_window_0_0 $<$ 3
p8	y<30
p9	outp $<$ 30
p10	s3r $<$ 127

and elliptic are filter designs in which it is common to check whether the output is less than a user-defined threshold. Therefore, we define the predicates p8 and p9 over the output variables y and $outp$, respectively. Although not exact models, these predicates can be viewed as being representative of certain timing properties of the design. For example, y can be an input to an adder block (Section II.C.1) for which the timing constraint requires that $y < 30$, as in p8. For our experiments, we consider predicates that are linear functions over a set of RTL variables and we use the "$<$" relational operator. For each of the H.264 modules, we consider multiple predicates.

Table III demonstrates the reduction in state-space provided by our abstraction method. PRISM runs out of memory while trying to construct any of the concrete DTMC models and therefore, these designs can not be model checked. We estimate the number of states in the concrete model based on the total number of combinations of values that can be assigned to the corresponding input variables. There is no reason to believe that the RTL inputs, which are data variables, are restricted and we use their full range of values to estimate the concrete state-space. In all the designs, with the exception of fft8, we are able to obtain significant reductions in state space by using our abstraction technique and PRISM successfully constructs the corresponding abstract DTMCs. We approximately represent the number of states in the abstract DTMC model as powers of 2, in order to facilitate comparison with the concrete number of states. Model checking of the smaller abstract DTMCs by PRISM requires only a few seconds.

p1, p2 and p3 are all the same predicate that differ only in the constraint values that are specified in the RHS. We observe that as the constraint values get smaller, the number of relevant data values (and hence states) also decrease. Our technique is extremely effective when the predicate is TRUE for only a small fraction of the possible data values. Although our technique would still be sound for larger constraint values, the reduction that we achieve may be far more modest.

Since model checking could not be completed for the concrete DTMCs in Table III, we do not present a comparison of the model checking results for these designs. Instead, as a proof of concept, we construct smaller versions (smaller bitwidth for inputs) of fir and elliptic and verify that the results computed using the concrete DTMC and the abstract DTMC are exactly the same (Table IV). We choose the smaller bitwidths such that PRISM model checks the concrete DTMC. For example, we consider 3-bit data for fir(small) and the runtime is $<$10s. We do not consider a smaller fft8 since our abstraction does not provide any reductions for it (Table III).

In fft8, we are not able to demonstrate any reduction in state-space using our abstraction. This is due to "-" operator in the RTL design. As described in Section IV.B, a "-" sign on the left hand side of a "less than" constraint will result in unconstrained values for all the input variables. JPEG encoder is another design for which we cannot obtain reductions. The module of the encoder design that we consider is control-intensive and therefore, the number of paths that need to be explored by the symbolic execution algorithm is huge (Section IV.A). For this design, we stopped the symbolic execution engine after 1hr of exploring paths and generating the corresponding constraints. We could not use this incomplete set of constraints since all possible paths in the design need to be considered in order to guarantee correctness of our abstraction.

In all the designs mentioned above, we find that the control paths are independent of the values of data. This is fairly common for a large class of data-intensive designs that are

TABLE III
REDUCTIONS IN NUMBER OF STATES THAT WE ACHIEVE BY USING OUR ABSTRACTION

		Concrete DTMC		Abstract DTMC	
Design name	Predicate name	Number of states	Model checking time	Number of states	Model checking time
`fir`	p8	2^{56}	*Out of memory*	2^{28}	<2s
`elliptic`	p9	2^{96}	*Out of memory*	$\approx 2^{29.73}$	<2s
`fft8`	p10	2^{16}	*Out of memory*	2^{16}	*Out of memory*
`Inter_pred_LPE`	p1	2^{40}	*Out of memory*	$\approx 2^{15.85}$	<2s
`Inter_pred_LPE`	p2	2^{40}	*Out of memory*	$\approx 2^{14.04}$	<2s
`Inter_pred_LPE`	p3	2^{40}	*Out of memory*	$\approx 2^{11.61}$	<2s
`Inter_pred_pipeline`	p4	2^{256}	*Out of memory*	2^{32}	<2s
`Inter_pred_pipeline`	p5	2^{24}	*Out of memory*	$\approx 2^{22.95}$	<2s
`Inter_pred_sliding_window`	p6	2^{152}	*Out of memory*	$\approx 2^{30.11}$	<2s
`Inter_pred_sliding_window`	p7	2^{128}	*Out of memory*	2^{32}	<2s

TABLE IV
DEMONSTRATING CORRECTNESS OF OUR ABSTRACTIONS USING SMALLER, CONTRIVED VERSIONS OF BENCHMARKS DESIGNS SINCE THE CONCRETE DTMCS CANNOT BE CONSTRUCTED FOR THE ACTUAL SIZES.

		Concrete DTMC		Abstract DTMC	
Design name	Predicate	Number of states	P[Predicate = TRUE] (PRISM result)	Number of states	P[Predicate = TRUE] (PRISM result)
`fir` (small)	$(y < 12)$	2^{24}	4.6539×10^{-4}	$\approx 2^{15.57}$	4.6539×10^{-4}
`elliptic` (small)	$(outp < 30)$	2^{30}	9.6485×10^{-7}	$\approx 2^{14.39}$	9.6485×10^{-7}

commonly used in DSP systems. For example, typical control variables that we observe are *counters* that are not data-dependent. Since control variables control the selection of paths and since we wish to consider all possible paths, we cannot constrain the values of such variables. In non-DSP designs, the control variables may depend on input data variables and therefore, all such input variables must also be unconstrained. In these cases, the overall reduction achieved by our abstraction technique may not be very large.

In RTL designs, it is possible that arithmetic operations can result in an overflow (or underflow) due to insufficient number of bits that are assigned to store the results. Ideally, such incorrect computations should not be allowed. Our technique cannot detect such overflow errors. Typically, overflows are prevented in DSP designs by assigning sufficient number of bits to the different variables in the design. Our abstraction techniques can be applied on such designs.

VII. RELATED WORK AND CONCLUSION

In the realm of software verification, there exist several techniques [28] [29] for predicate abstraction. Properties regarding program correctness/safety can be expressed using a set of predicates, that are either specified or automatically inferred. These predicates can be used to abstract a program and convert it into a Boolean program on which the properties can be easily verified. More generally, *abstract interpretation* [30] is the theory of reasoning with the approximate semantics of a large program rather than the set of all possible concrete behaviors. However, unlike predicate abstraction, all such abstractions are not necessarily property-specific. In all these abstractions, the concrete numeric values of data can either be completely abstracted out of the program or can be restricted to finite intervals [31].

Data abstraction techniques have been applied even in the context of hardware verification [10]. These techniques employ predicate abstraction in order to focus on the verification of Boolean control logic for which the exact numeric values of datapath variables are inconsequential. In [11], RTL designs are verified by restricting data values to intervals that are imposed by the execution of the RTL program. Therefore, these intervals are not property-specific.

Abstraction techniques have been employed in the context of probabilistic systems as well [32] [33] [22] [23]. In [23], the abstraction is performed on the source code itself. However, this technique is intended for probabilistic software and cannot be extended to RTL designs. In [22], the authors present a predicate-based abstraction for Markov Decision Processes (MDPs). They employ an SMT solver in order to implement this abstraction at the level of the PRISM language itself. However, this implementation is very inefficient for the bulky PRISM descriptions that are used for RTL designs (Section II.D).

In conclusion, we have presented a property-specific value-based interval abstraction technique that is applied at the source code level. We intend our abstraction for scaling probabilistic model checking of hardware designs. Widespread adoption of formal verification is feasible only if it remains relevant and practicable in critical, emerging areas of need like variation-aware timing verification. Our work represents a strategic step in this direction.

References

[1] K. A. Bowman, M. Orshansky, and S. S. Sapatnekar, "Tutorial ii: Variability and its impact on design," in *Proc. of ISQED'06*, 2006, p. 5.

[2] S. V. Kumar, C. H. Kim, and S. S. Sapatnekar, "NBTI-aware synthesis of digital circuits," in *Proc. of DAC'07*, 2007, pp. 370–375.

[3] N. J. Wang, J. Quek, T. M. Rafacz, and S. J. patel, "Characterizing the Effects of Transient Faults on a High-Performance Processor Pipeline," in *Proc. of DSN'04*, 2004, p. 61.

[4] D. Ernst, N. S. Kim, S. Das, S. Pant, T. Pham, R. Rao, C. Ziesler, D. Blaauw, T. Austin, and T. Mudge, "Razor: A low-power pipeline based on circuit-level timing speculation," in *Proc. of MICRO'03*, Dec. 2003.

[5] M. Z. Kwiatkowska, G. Norman, and R. Segala, "Automated Verification of a Randomized Distributed Consensus Protocol Using Cadence SMV and PRISM," in *Proc. of CAV'01*, 2001, pp. 194–206.

[6] M. Kwiatkowska, G. Norman, and D. Parker, "Symmetry Reduction for Probabilistic Model Checking," in *Proc. of CAV'06*, 2006, pp. 234–248.

[7] J.-P. Katoen, "Advances in Probabilistic Model Checking," in *Proc. of VMCAI'10*, 2010, p. 25.

[8] J. A. Kumar and S. Vasudevan, "Variation-Conscious Formal Timing Verification in RTL," in *Proc. of VLSI Design'11*, 2011, extended version available at http://users.crhc.illinois.edu/jasokku2/docs/TCAD_final.pdf.

[9] E. M. Clarke, M. Fujita, S. P. Rajan, T. W. Reps, S. Shankar, and T. Teitelbaum, "Program slicing of hardware description languages," in *Proc. of CHARME'99*, 1999, pp. 298–312.

[10] E. Clarke, O. Grumberg, and et al., "High level verification of control intensive systems using predicate abstraction," *ACM Transactions on Programming Languages and Systems*, vol. 16, pp. 1512–1542, 2003.

[11] V. P. Nazanin, N. Mansouri, and R. Vemuri, "Automatic Data Path Abstraction for Verification of Large Scale Designs," in *Proc. of ICCD'98*, 1998, pp. 192–194.

[12] P. Johannsen, "BOOSTER: Speeding Up RTL Property Checking of Digital Designs by Word-Level Abstraction," in *Proc. of CAV'01*, 2001, pp. 373–377.

[13] J. A. Kumar and S. Vasudevan, "Automatic compositional reasoning for probabilistic model checking of hardware designs," in *Proc. of QEST'10*, 2010, pp. 143–152.

[14] "Verilog Reference Manual," http://eesun.free.fr/DOC/VERILOG/verilog_manual1.html.

[15] M. Kwiatkowska, G. Norman, and D. Parker, "PRISM 2.0: A tool for probabilistic model checking," in *Proc. of QEST'04*, 2004, pp. 322–323.

[16] J. Kemeny and J. Snell, *Finite Markov chains*, repr ed., ser. University series in undergraduate mathematics. New York: VanNostrand, 1969.

[17] S. Derisavi, H. Hermanns, and W. H. Sanders, "Optimal state-space lumping in Markov chains," *Information Processing Letters*, vol. 87, no. 6, pp. 309 – 315, 2003.

[18] L. Liu and S. Vasudevan, "Efficient Validation Input Generation in RTL by Hybridized Source Code Analysis," in *Proc. of DATE'11*.

[19] M. Mitzenmacher and E. Upfal, *Probability and Computing: Randomized Algorithms and Probabilistic Analysis*, 2005.

[20] L. Wan and D. Chen, "Dynatune: circuit-level optimization for timing speculation considering dynamic path behavior," in *Proc. of ICCAD'09*, 2009.

[21] T. S. Hoang, Z. Jin, K. Robinson, A. McIver, and C. Morgan, "Probabilistic invariants for probabilistic machines," in *Proc. of ZB'03*. Springer, 2003, pp. 240–259.

[22] M. Kattenbelt, M. Kwiatkowska, G. Norman, and D. Parker, "Game-Based Probabilistic Predicate Abstraction in PRISM," *ENTCS*, vol. 220, no. 3, pp. 5–21, 2008.

[23] M. Kattenbelt, M. Kwiatkowska, G. Norman, and D. Parker, "Abstraction Refinement for Probabilistic Software," in *Proc. of VMCAI'09*, 2009, pp. 182–197.

[24] J. C. King, "Symbolic execution and program testing," *Commun. ACM*, vol. 19, pp. 385–394, July 1976.

[25] K. G. Larsen and A. Skou, "Bisimulation through probabilistic testing," *Information and Computation*, vol. 94, no. 1, pp. 1–28, 1991.

[26] M. Berkelaar, K. Eikland, and P. Notebaert, "lp_solve 5.5, open source (mixed-integer) linear programming system," Software, May 1 2004, available at http://lpsolve.sourceforge.net/5.5/. Last accessed Dec, 18 2009. [Online]. Available: http://lpsolve.sourceforge.net/5.5/

[27] P. R. Panda and N. D. Dutt, "1995 high level synthesis design repository," in *ISSS*, 1995, pp. 170–174, available at http://ftp.ics.uci.edu/pub/hlsynth/.

[28] T. Ball, B. Cook, V. Levin, and S. K. Rajamani, "SLAM and static driver verifier: Technology transfer of formal methods inside microsoft," in *Proc. of IFM'04*, 2004, pp. 1–20.

[29] T. A. Henzinger, R. Jhala, R. Majumdar, and K. L. McMillan, "Abstractions from proofs," *SIGPLAN Not.*, vol. 39, no. 1, pp. 232–244, 2004.

[30] P. Cousot and R. Cousot, "Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixed points," in *Proc. of POPL'77*. ACM, 1977, pp. 238–252.

[31] D. Monniaux, "A minimalistic look at widening operators," *Higher Order Symbol. Comput.*, vol. 22, no. 2, pp. 145–154, 2009.

[32] L. D. Alfaro and P. Roy, "Magnifying-Lens Abstraction for Markov Decision Processes," in *Proc. of CAV'07*, 2007, pp. 325–338.

[33] P. R. D'argenio, H. E. Jensen, and K. G. Larsen, "Reachability analysis of probabilistic systems by successive refinements," in *Proc. of PAPM/PROBMIV'01*, 2001, pp. 39–56.

Pervasive Formal Verification in Control System Design

Lee Pike
Galois, Inc.
Email: leepike@galois.com

I. Motivation

Control systems design is a multifaceted field, drawing not only on control theory, but on results from computer science, electrical engineering, mechanical engineering, and physics. A controller often must satisfy regimented size, weight, power, and timing constraints, integrate with the overall system, and perform properly in a variety of harsh environments. Furthermore, control systems are arguably the lynchpin of safety in critical embedded systems, ranging from nuclear reactors to avionics to medical devices.

Progress has been made in the formal verification of aspects of control system design. Advances in hybrid system verification show promise in automating the verification of abstract models of dynamical systems. Advances in software and hardware formal verification may contribute to ensuring the correctness of implementations. Nevertheless, industrial uptake of these advances is still in its infancy, particularly as compared to disciplines such as digitial hardware design.

This panel will address the impediments to the adoption of formal verification techniques in industrial control system design. Furthermore, the panel will address what research topics would most benefit the adoption of formal verification in industry.

II. Panel Organization

The panelists will primarily be drawn from industry, having first-hand knowledge of the state-of-the-art in control system design practices.

This panel discussion will address the following questions:

- How can formal verification compliment current simulation and testing procedures?
- What will control system design look like in 10 years? 20 years?
- Can formal verification help build safer "intelligent" control systems?
- Where can the greatest impact be made in improving control system quality and reducing design costs? Better hybrid system verification tools? Better languages? More compiler assurance? Easier timing analysis? Automated power analysis?
- Could more aggressive control systems (i.e., that save energy, reduce operational wear, reduce the need for redundancy) be pursued if better design assurance could be provided?
- What social and educational impediments are there to having control systems engineers use formal verification tools?

Hybrid Verification of a Hardware Modular Reduction Engine

Jun Sawada*, Peter Sandon†, Viresh Paruthi†, Jason Baumgartner†, Michael Case† Hari Mony†,
*IBM Austin Research Laboratory
†IBM Systems & Technology Group

***Abstract*—Wide-operand modular math functions pose an enormous challenge for verification. We present a novel method to verify a modular reduction engine implemented as a finite state machine (FSM), leveraging a combination of model checking and theorem proving. As a first step of the verification, pre-conditions and post-conditions for each state transition of the FSM are identified. Next the implications from the pre-conditions to the post-conditions are verified using a model checker. The last step entails combining all the implications in a theorem prover to derive the overall correctness proof. We carried out this verification using a hybrid formal verification platform comprising the *ACL2* theorem prover and IBM's model checker *SixthSense*, along with numerous techniques to cope with the complexities of this verification task. To our knowledge, this is the first published method for the exhaustive verification of an RTL-implementation of a wide-operand industrial modular reduction engine.**

I. Introduction

A. Modular Reduction

Cryptography is becoming a central feature of our networked world. Increasing performance demands on modern microprocessors have mandated native hardware support for encryption and decryption algorithms in the form of an on-board cryptographic accelerator co-processor.

Classes of cryptography asymmetric algorithms such as Rivest-Shamir-Adleman (RSA) and Elliptical Curve Cryptography (ECC) are realized using lower-level functions, such as Modular Reduction or Modular Exponentiation. These are implemented with finite state machines (FSM) which operate on wide-operands (e.g., on the order of 4096 bits), and may require a large number of clock cycles to complete the computation – even hundreds of thousands of clock cycles for large operand bit-widths.

Verification of such complex hardware is of critical importance, though poses formidable challenges. Traditional *informal* verification methods offer insufficient coverage given the wide operand widths, sequential depth of the computation and the inherently difficult nature of the logic. Even hardware-accelerated simulation and post-Silicon debug, offering dramatically greater explicit-state coverage, are rendered insufficient given the sheer size of the state-space. Additionally, the reference result needs to be computed in software which can prove to be the bottleneck. Traditional bit-level model checking approaches are unscalable even for small bit-widths of such arithmetic functions, and traditional higher-level techniques such as theorem proving become extremely tedious due to the need to reason about the intricate sequentially-deep state machines at the RTL level.

In this paper we present a method to verify modular reduction implemented as an FSM by leveraging a combination of model checking and theorem proving. Our approach decomposes the verification task into two parts: 1) verification of invariants associated with the FSM and 2) Combining the verified invariants to form a proof of correctness. The set of invariants describing the behavior of the FSM are verified using the model checker. These invariants are then combined by the theorem prover to form a proof that the FSM correctly implements the target algorithm. The presented technique allows us to overcome the limitations of traditional verification disciplines as outlined above. We can scale our technique to verify the correctness of modular reduction for a number of operand widths, leveraging the strengths of theorem proving to reason about parametrized computations, and leveraging the model checker to verify invariants which require precise characterization of temporally-deep RTL-implemented state machines.

B. ACL2SIX

There are two predominant formal verification techniques that have been successfully used to verify the correctness of bit-accurate sequential machines: model checking and theorem proving. Model checking is automated, though often fails to scale for designs containing complex arithmetic datapaths. On the other hand, interactive theorem proving techniques do scale, though often only with significant human effort – which may become formidable if requiring reachable-state characterization of complex bit-level state machines, or reasoning about bit-optimized arithmetic designs.

The combination of these two techniques is sometimes called *hybrid verification*, and may provide an ideal formal verification environment. The main motivation for the combination is to use the model checking for verifying the low-level details of bit-level hardware, and use theorem proving to focus the high-level mathematical and algorithmic correctness. A number of different hybrid tools have been developed [1], [2] and used for a variety of verification tasks [3], [4] in the past. However, it is our thesis that such hybrid tools have not been leveraged fully, due to either the weakness of the underlying model checker or limiting the theorem proving to a rather simplistic analysis.

Relating to verification of hardware encryption, Slobodová [5] verified microprocessor *instructions* to implement Advanced Encryption Standard (AES) algorithms against a reference model derived from its specification. Erkök et. al. [6] verified cryptographic hardware by checking equivalence between different stages of implementation. Smith et. al. [7] verified a Java block cipher implementations using a hybrid tool to symbolically simulate Java bytecode and equivalence-check the resulting expressions. This kind of equivalence checking relies on the fact that the operation takes a fixed delay and/or a fixed number of (micro-)instructions. A similar equivalence checking approach did not work for the FSM we will discuss in this paper, because the behavior of the machine significantly changes depending on input data, rendering typical equivalence algorithms such as BDD-sweeping ineffective. To our knowledge, encryption procedures implemented as intricate sequentially-deep state machines have not been fully verified.

From a perspective of practicality, it is also important to directly verify designs written in a Hardware Description Language (HDL) such as Verilog or VHDL. Our goal is to accept industrial designs written in the HDL without any modification. Some tools and past work have attempted to translate HDL into the theorem proving language [8], [9], [10]; however, full formalization of HDL is very tedious and difficult, or creates semantic gaps. Another problem in the translation of HDL to a formal language is that the theorem prover has to deal with low-level details of hardware. Industrial HDL often includes bit-level optimizations and peripheral circuit artifacts such as *scan* and *power-optimization* logic, overall hindering the effectiveness of the approach.

In our hybrid verification tool called ACL2SIX, we use a small subset of the theorem prover's language to specify properties of target hardware. The tool reads the unmodified hardware design written in an HDL and directly verifies properties on it. We use a powerful bit-level model checking tool in order to automatically prove sizable verification sub-problems, thus reducing the burden on interactive theorem proving while making the proof script robust and reusable. We use a fully featured, general-purpose theorem prover to allow verification of sometimes-difficult higher level mathematical problems.

The rest of the paper is organized as follows. We start by outlining a typical modular reduction engine implemented in hardware as an FSM. We next describe the ACL2SIX hybrid formal verification platform which combines the ACL2 theorem prover with IBM's formal toolset SixthSense. We then describe our verification approach including the pre-conditions and post-conditions used in the context of modular reduction, as well as enhancements to the underlying model checker SixthSense to enable application of the hybrid platform to a large design. Finally we provide some results and conclusions from the novel application.

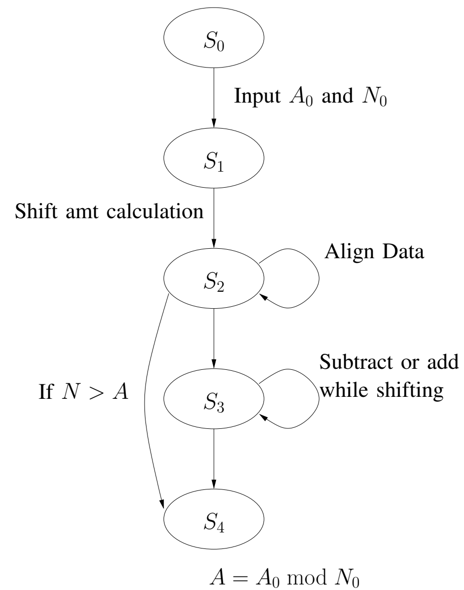


Fig. 1. Modular Reduction State Diagram

II. A Modular Reduction Engine

A modular reduction engine computes the remainder of one integer divided by another. That is, the engine computes $R = A_0 \bmod N_0$ for positive integers A_0 and N_0, such that $A_0 = N_0X + R$, and $0 \leq R < N_0$ for some integer X. For cryptographic applications, the size of these integers varies according to the strength of the cryptographic algorithm, with current applications requiring several thousand bits of precision.

The following is the modular reduction algorithm we wish to verify:

1. Set $A := A_0$ and $N := N_0$. Ensure that $A \geq 0, N > 0$.
2. If $A < N$, set $R := A$ and exit.
3. Left shift N to align its most significant '1' with that of A.
4. Divide Loop:
5. If $A \geq 0$, $A := A - N$, otherwise $A := A + N$.
6. If $N = N_0$, exit loop.
7. Right shift N by one bit.
8. Go to Divide Loop.
9. If $A \geq 0$, set $R := A$, otherwise $R := A + N$.

To understand how the algorithm works, note that $(A \bmod N_0) = (A_0 \bmod N_0)$ is an invariant, and $0 \leq R < N_0$ at the end of the algorithm.

Our goal is to verify a hardware which imlements this algorithm as an FSM. Figure 1 presents a state transition diagram from the design document of the hardware, and Table I provides an action table for the FSM.

S_0 **:** The FSM reads two input operands, A_0 and N_0, and stores them in the registers A and N.

S_1 **:** The FSM counts leading zero bits of N and A and stores their difference $lz(N) - lz(A)$ in the registers D and C. This corresponds to the number of bits to left-shift N in order to align the most significant bit of one in A and N.

TABLE I
ACTIONS OF MODULAR REDUCTION FINITE STATE MACHINE

State	Actions
$S_0(S=0)$	$S := 1; A := A_0; N := N_0$
$S_1(S=1)$	$S := 2; C := lz(N) - lz(A); D := C$
$S_2(S=2)$	if $(D < 0)$ $\{S := 4\}$ if $(D > 0)$ $\{N := N \ll 1; D := D - 1\}$ if $(D = 0)$ $\{S := 3\}$
$S_3(S=3)$	if $(C \geq 0 \wedge A \geq 0)$ $\{A := A - N; N := N \gg 1; C := C - 1\}$ if $(C \geq 0 \wedge A < 0)$ $\{A := A + N; N := N \gg 1; C := C - 1\}$ if $(C = 0 \wedge A \geq 0)$ $\{A := A - N; C := C - 1\}$ if $(C = 0 \wedge A < 0)$ $\{A := A + N; C := C - 1\}$ if $(C < 0 \wedge A \geq 0)$ $\{S := 4\}$ if $(C < 0 \wedge A < 0)$ $\{S := 4; A := A + N\}$

S_2 **:** If $D < 0$, N is larger than A and the FSM directly goes to the final state S_4. Otherwise, the FSM left-shifts N by one bit and remains in the same state. The FSM makes self-loop transitions $D = lz(N) - lz(A)$ times, and then it goes to the state S_3.

S_3 **:** The FSM remains in this state for $C + 1 = lz(N) - lz(A) + 1$ iterations. It subtracts or adds N to A depending on the sign of A. It also shifts N to the right by one bit, except for the last iteration. Finally, it adds N if A is negative, and moves to the final state S_4.

S_4 **:** The register A stores the final answer of $A_0 \bmod N_0$.

Although this description is considerably simpler than the optimized hardware implementation, it is sufficient to explain our verification approach in later sections. The modular reduction engine is implemented to accept input operands of different data widths. All the arithmetic operations on A and N, such as bit vector addition, subtraction, shifting and leading zero counting, are performed as per the size of input operands. The FSM implements the variable-size operations by iterating fixed-size arithmetic operations. For example, 65-bit adders are used to implement variable-size bit-vector addition up to 4096-bits. As a result, what appears to be a simple state transition is in fact iterative operations on fixed data width over many clock cycles.

Current guidelines for the use of the RSA algorithm for public key encryption in commercial applications call for the use of 1024 bit or 2048 bit keys [11]. The modular reduction operation used in this algorithm takes a number of clock cycles proportional to the square of the input data width divided by the size of the fixed-width processing and storage elements in the implementation. Even a single 512-bit computation will, therefore, take several thousand clock cycles to execute, while the number of possible input operand pairs is 2^{1024}. This makes it very difficult to verify the entire range of interesting cases using simulation.

III. ACL2SIX HYBRID VERIFICATION SYSTEM

The ACL2SIX verification system is a combination of the open-source theorem prover ACL2 [12], [13] and the IBM verification tool SixthSense [14]. An early version of the system has been reported in [15], and its application in [16]. As this system has been significantly modified since it was first reported, we outline the salient features of the enhanced hybrid environment.

The main philosophy of this hybrid tool is a divide-and-conquer approach for the verification problem. When we want to verify a property which cannot be verified by an automated model checker, we decompose it into a number of easier sub-problems, solve them one-by-one, and combine the results together. Each sub-problem is thus solved by a model checker, while the results are combined by a theorem prover. However, when the verification problem is decomposed into too many small problems, the burden of recombination via the theorem proving becomes rather high, and the proof may become labor intensive. Thus, it is critical to contain the degree of decomposition using a powerful model checker to scale to as large of sub-problems as possible.

An overview of the ACL2SIX system is shown in Figure 2. Suppose a user attempts to verify certain properties on a *design under test* (*DUT*). A DUT is usually a complex RTL hardware design written in VHDL or Verilog. A verification driver defines the environment in which the DUT operates, e.g. clocking conditions and other input constraints. In a typical setting, the verification driver may assert the reset signal at the beginning of the test, and then initiate the operation of the machine with non-deterministic data inputs. A verification driver is usually written in VHDL or some *synthesizable* language. As we discuss later, the verification driver is also used to help writing invariant conditions succinctly in the ACL2 language.

When a user attempts to check if a certain property holds using the ACL2SIX system, he/she writes the property in a small subset of the ACL2 theorem prover language. When invoked, ACL2 first compiles the property to a *property checker*. A property checker is a synthesized automata for the desired property, effectively a small state machine which asserts a particular gate to a logical ‘1’ when the property holds. SixthSense then composes the DUT, the verification driver, and the property checker, and checks whether the property checker always evaluates to ‘1’ for all input sequences. When the verification is successful, the property is saved in ACL2 as a theorem and may be used for future proofs. If the check fails, SixthSense produces an counterexample trace to assist the user in determining why the property does not hold.

Since ACL2 is a general-purpose theorem prover, its language is too expressive to be translated into HDL. Instead, the ACL2SIX system allows only a subset of the ACL2 language for specifying properties to be verified. The subset is rich enough to write various properties to prove the correctness of the DUT, and the translation of the properties does not cause any semantic inconsistency between this ACL2 language

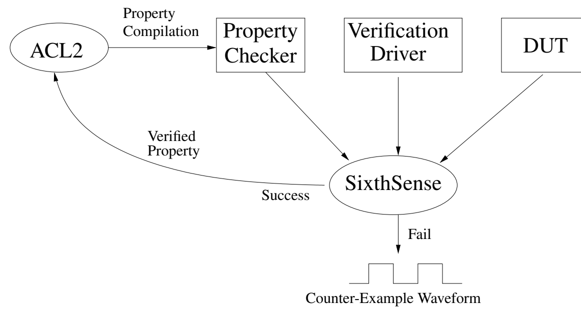


Fig. 2. Overall Data Flow for Property Check in ACL2SIX

TABLE II
EXAMPLE ACL2SIX PRE-DEFINED FUNCTIONS

Function Name	Brief Description
`(bv i j)`	Bit vector of value i and length j
`(b1p b)`	True if $b = 1$, false if $b = 0$
`(bv+ v1 v2)`	Sum of two bit vectors
`(bv- v1 v2)`	Difference of two bit vectors
`(bv-sll v n)`	Logical left shift of vector v by n bits
`(bv-srl v n)`	Logical right shift of vector v by n bits
`(bv-lz v i)`	Vector of length i counting leading zeros of v

subset and VHDL.

The language for ACL2SIX has four data types: bits, bit vectors, Boolean values and natural numbers. All properties must be written in terms of ACL2SIX pre-defined functions, under which those types are closed. The user may also specify user-defined non-recursive functions. However, these functions must also be defined in terms of those pre-defined functions. Additionally, the user-defined functions must carry *type* information using the ACL2 guard mechanism [17], so that the translation process can infer types of expressions. Table II lists some of the ACL2SIX pre-defined functions.

The signal values in the DUT and the driver can be referenced by the following terms:

```
(vhdl-sigbit m sig n)
(vhdl-sigvec m sig (i j) n)
```

`(vhdl-sigbit` m sig n`)` is used to reference the value of bit sig in hardware model m at clock cycle n. Similarly, `(vhdl-sigvec` m sig `(`i j`)` n`)` refers to the bit range i to j of bit vector sig at cycle n. In these terms, sig is the name of a signal, and i, j and n are natural numbers. Model m is a list structure, from which we can infer the DUT, the verification driver, and other parameters needed to set up verification of the DUT. For example, the model for an adder may be simply defined as:

```
(defun adder ()
  '("adder"
    :driver "adder_dr.vhdl"))
```

where `"adder"` is the VHDL entity name of the adder and `"adder_dr.vhdl"` is a verification driver name. Other information such as the path to the VHDL file, or how the DUT is initialized may be added to the model definition.

The main idea behind the use of `vhdl-sigbit` and `vhdl-sigvec` is that they logically reference the signal values of the DUT, but they do not actually compute the values. In a system that fully embeds an HDL, a hardware model would be a translated HDL and signal values would be defined by its interpreter. In ACL2SIX, the model is just a stub to access the DUT written in HDL, and signal values are only defined using *constraint functions*. Specifically, both `vhdl-sigbit` and `vhdl-sigvec` are ACL2 macros defined in terms of ACL2 encapsulated functions `sigbit` and `sigvec`. An ACL2 encapsulated function is a mechanism to define an uninterpreted function with some constraints. We can infer types of the value returned by `sigbit` and `sigvec`, but its value is uninterpreted, and can be inferred only by calling SixthSense through the ACL2SIX system.

A typical ACL2 theorem definition to invoke SixthSense property checking has the following syntax:

```
(defthm name
  (implies type-info expr)
  :hints (("goal" :clause-processor
                  ( :function acl2six
                    :hints acl2six-args)))).
```

In this definition, *name* is the name of the theorem, *type-info* is the type information for the free variables in *expr*, and *expr* is a property expression which is defined in terms of the ACL2SIX pre-defined functions. The ACL2 hint provided after keyword `:hints` usually tells the theorem prover how to prove a theorem, and in this case, it invokes a clause processor function `acl2six`. A clause processor is an ACL2 mechanism to implement an extension of the prover. It allows a user-defined function to simplify or even prove a logical expression. It may also work as an interface with other verification tools. When invoked through the clause processor mechanism, function `acl2six` translates *expr* to a property checker implemented in VHDL, runs SixthSense, and records successfully verified properties as theorems. A call to `acl2six` can be accompanied by additional arguments *acl2six-args*, with which the user can control SixthSense and specify types of algorithms to verify the property.

One important limitation of the ACL2SIX property compilation is that the verified property should be defined in terms of signals with fixed timing delays. For example, in order to check the output `"SUM"` of a two-stage 32-bit adder , we can evaluate the following ACL2 term:

```
(defthm adder-output
 (implies (natp n)
  (equal (vhdl-sigvec (addr) "SUM" (0 31) (+ n 2))
         (bv+ (vhdl-sigvec (addr) "A" (0 31) n)
              (vhdl-sigvec (addr) "B" (0 31) n))))
  :hints (("goal" :clause-processor
                  (:function acl2six
                   :hint '((:cycle-var n))))))
```

This property check compares the value of vector `"SUM"` at cycle $n+2$ with the summation of two vectors `"A"` and `"B"` at cycle n, where n is an arbitrary natural number. The cycle delay `2` in cycle expression `(+ n 2)` should be a constant,

and cannot be replaced with a variable or a complex expression. In terms of LTL, [18], we can only check a formula of the form $G(expr)$ where $expr$ is a formula written only with X operators. While we are generally interested in *unbounded* model checking of the underlying design where reasoning about specific clock cycles may seem to contradict this goal, we use this style of reasoning in an *inductive* framework where state machines are evaluated relative to arbitrary states which adhere to established invariants, vs. evaluating only relative to initial states.

IV. Verification of a Modular Reduction Engine

A. General Approach to Verifying a Finite State Machine

An FSM (such as that described in Section II) can be verified by a hybrid system such as ACL2SIX by first verifying every state transition using a model checker, and then combining the results using a theorem prover.

For example, let us consider an FSM that makes a state transition sequence of $S_0, S_1, S_2, \ldots, S_n$. Each state transition may take a number of clock cycles, and we assume that the transition from state S_i to state S_{i+1} takes Δ_i cycles. Each state S_i has a corresponding property P_i that must hold. For the state transition from S_i to S_{i+1}, P_i is the pre-condition and P_{i+1} is the post-condition. Let us write $P_i\{S_i\}$ to indicate that property P_i holds for state S_i. If we can verify $P_0\{S_0\}$ and $P_i\{S_i\} \Rightarrow P_{i+1}\{S_{i+1}\}$ for all $i < n$, it is straightforward to prove $P_n\{S_n\}$ using a theorem prover. In this way, we can verify the machine correctness specified by P_n. We may define P_n to specify, for example, that the final answer of the machine is correct.

Thus the verification problem is reduced to the verification of $P_i\{S_i\} \Rightarrow P_{i+1}\{S_{i+1}\}$ for each i. Let us write $P_i(n)$ to indicate that P_i holds at clock cycle n. Since the transition from S_i to S_{i+1} takes Δ_i cycles, proving

$$P_i(n) \Rightarrow P_{i+1}(n + \Delta_i) \quad (1)$$

for all n will be sufficient. Let us further define $Q_i(n) = \big(P_i(n) \Rightarrow P_{i+1}(n + \Delta_i)\big)$. The ACL2SIX system and SixthSense use the following steps to verify $\forall n.Q_i(n)$

1) Convert $Q_i(n)$ to a circuit using logical gates and latches. Since $P_i(n)$ can be represented as a combinational circuit, we can latch the value of $P_i(n)$ for Δ_i-cycles, and then check that the latched value of P_i implies P_{i+1}.
2) Simplify the circuit representation of $Q(n)$ using a number of circuit reduction techniques, such as constant propagation, combinational and sequential simplifications [19], retiming [20], phase abstraction [21] and transient logic elimination [22]. This reduction itself may reduce $Q_i(n)$ to a tautology, in which case $Q_i(n)$ is proven and we stop. Otherwise, we go to the next step.
3) Prove $Q_i(n)$ by k-induction. This is done by proving the base cases $Q_i(0), Q_i(1), \ldots, Q_i(k-1)$ and the induction step $Q_i(n) \wedge Q_i(n+1) \wedge \cdots \wedge Q_i(n+k-1) \Rightarrow Q_i(n+k)$. This is attempted for ever-increasing values of k until either $Q_i(n)$ is proved or computational resources are exhausted.

The ACL2SIX system and SixthSense are highly configurable, and so we could use any other model checking algorithms to verify $Q_i(n)$. However, we found that logic reductions followed by k-induction work well for the verification of many properties of our modular reduction engine.

TABLE III
Pre and post-conditions of State Transitions of the Modular Reduction Engine

Transition	Pre-condition	Post-Condition
S_0 to S_1	$S = 0$	$S' = 1 \wedge A' = A_0 \wedge N' = N_0$
S_1 to S_2	$S = 1$	$S' = 2 \wedge C' = lz(N) - lz(A) \wedge$ $D' = C' \wedge A' = A \wedge N' = N$
S_2 to S_2	$S = 2 \wedge D > 0$	$S' = 2 \wedge N' = N \ll 1 \wedge$ $D' = D - 1 \wedge C' = C \wedge$ $A' = A$
S_2 to S_3	$S = 2 \wedge D = 0$	$S' = 3 \wedge N' = N \wedge A' = A$
S_2 to S_4	$S = 2 \wedge D < 0$	$S' = 4 \wedge A' = A$
S_3 to S_3	$S = 3 \wedge C > 0 \wedge$ $A \geq 0$	$S' = 3 \wedge A' = A - N \wedge$ $N' = N \gg 1 \wedge C = C - 1$
S_3 to S_3	$S = 3 \wedge C > 0 \wedge$ $A < 0$	$S' = 3 \wedge A' = A + N \wedge$ $N' = N \gg 1 \wedge C = C - 1$
S_3 to S_3	$S = 3 \wedge C = 0 \wedge$ $A \geq 0$	$S' = 3 \wedge A' = A - N \wedge$ $N' = N \wedge C = C - 1$
S_3 to S_3	$S = 3 \wedge C = 0 \wedge$ $A < 0$	$S' = 3 \wedge A' = A + N$ $N' = N \wedge C = C - 1$
S_3 to S_4	$S = 3 \wedge C < 0 \wedge$ $A \geq 0$	$S' = 4 \wedge A' = A$
S_3 to S_4	$S = 3 \wedge C < 0 \wedge$ $A < 0$	$S' = 4 \wedge A' = A + N$

B. Verification of a Modular Reduction Engine

Here we discuss the use of ACL2SIX to verify the modular reduction engine. Table III shows the list of pre-conditions and post-conditions for each state transition, as per Figure 1. For any symbol X, let X' represent its value after the state transition. The table therefore shows how symbol values change when state transitions occur. If we can verify that the pre-condition implies the post-condition for all possible state transitions, we can use a theorem prover to show that the value of register A is $A_0 \bmod N_0$ when state S_4 is reached. In other words, the FSM correctness is the logical consequence of this set of pre-condition and post-condition pairs.

We can represent the pre-condition and post-condition relation using the supported language of ACL2SIX. While the number of clock cycles between FSM state transitions is generally a function of the data width, for a given data width of input A_0 and N_0 each state transition requires a fixed number of clock cycles. Our approach is to verify the operational correctness for each input data width separately. Then, the

relation of the pre and post-conditions can be written using the ACL2SIX language, which requires that each delay be a fixed constant. We define the delay of the state transition parametrically, so that we can rerun the same proof script to re-verify the modular reduction engine for different input data widths by just changing parameters.

When actually writing an ACL2 theorem representing the conditions in Table III, additions $+$, subtractions $-$, shifting $\ll$, $\gg$, and leading zero counting lz are specified using the pre-defined functions given in Table II. As briefly discussed in Section II, the hardware implements the arithmetic operations of long bit vectors by repeatedly applying 65-bit arithmetic operations. For example, 512-bit addition is performed by repeating 65-bit additions 8 times over tens of clock cycles. However, such hardware implementation details should be automatically verified and hidden from the ACL2 proof level. In fact, our proof script simply specifies such an addition as the sum of two long and continuous bit vectors. In this way we simplify the to-be-proven theorems as much as possible. This requires the underlying model checker such as SixthSense to do the heavy lifting of verifying high-level specifications against intricate implementation artifacts.

The abstraction of bit-level details allows the pre-conditions and post-conditions (Table III) to be described concisely at a high-level. However, simply attempting to verify "pre-condition implies post-condition" frequently fails because the hardware often requires additional conditions to operate properly. For example, the hardware goes through an initialization phase that sets up the clock buffers, the hardware control logic and other components for proper operations. The hardware is designed to operates properly only *after* such initialization, relying upon post-initialization reachable state invariants. Let us define such a global invariant as $\text{inv}(n)$. Additionally, there might be other reachability invariants that holds when the machine is at state S_i but not captured in the the conditions described in Table III. Let such a state invariant be denoted as $\text{cond}_i(n)$. Then it is sufficient to verify:

$$(\text{inv}(n) \wedge \text{cond}_i(n) \wedge P_i(n)) \Rightarrow P_{i+1}(n + \Delta_i) \qquad (2)$$

for all the state transitions, instead of Equation 1. Separately, we need to verify that the global invariant condition is in fact an invariant by:

$$\text{inv}(n) \Rightarrow \text{inv}(n+1) \qquad (3)$$

and the state invariant condition is satisfied at each state by:

$$(\text{inv}(n) \wedge \text{cond}_i(n)) \Rightarrow \text{cond}_{i+1}(n + \Delta_i). \qquad (4)$$

In our approach, we define the global and state invariants in the verification driver in Figure 2. For example, the global invariant inv may be defined as a VHDL signal `"DRIVER.INV"` in the verification driver which represents the conjunction of numerous invariant conditions. In ACL2SIX, we can refer to this global invariant at any time n as `(vhdl-sigbit (modred) "DRIVER.INV" n)`. In this way, we keep the hardware-dependent and sometimes tedious definition of invariant conditions out of the proof script.

Finding the proper global invariant inv and state invariant cond_i is the most critical task for the entire verification methodology. This is usually done by repeated attempts to verify formula 2, 3 and 4, analyzing failed verification results by viewing generated counterexample waveforms, and iteratively tightening the invariants until the proof is successfully completed.

Some simple invariant conditions are automatically deduced during the 3-step verification algorithm discussed in Subsection IV-A. For example, circuit reduction algorithms in step 2) may simplify the design by merging redundant gates, or performing other property-preserving temporal abstractions. Such transformations are critical to simplify the manual effort of deriving invariants; in a sense, such transformations automate the derivation of a subset of design invariants. For example, if two latches are merged since they always evaluate to the same value, this rules out a possible induction counterexample where they exhibit differing values. Similarly, k-induction with a larger value of k tends to prove more properties without manually specifying some invariants. Thus, the more powerful the underlying bit-level model checker is, the less the verification engineer must manually specify the invariant conditions.

Once all the post-conditions are verified from pre-conditions, the theorem-prover is used to deduce the correctness proof of the hardware operation as a logical consequence of all the verified properties. During theorem proving, it is critical to analyze state loops. This is usually carried out by specifying loop invariants, verifying them by induction, and using them to deduce the termination condition. For example, in the i'th iteration of S_2 of our modular reduction finite state machine, the following loop invariant should hold.

$$(A = A_0) \wedge (N = N_0 \ll i)$$
$$\wedge \big(C = lz(N_0) - lz(A_0)\big) \wedge (D = C - i)$$

The state loop at S_3 satisfies a slightly more complicated loop invariant, with inequality $-2N \leq A < 2N$ being true except during the last iteration of S_3. This condition is critical for proving the correctness of the final answer.

At this high-level analysis of loop invariants, the theorem proving task is no different from a pure theorem proving verification approach. However, a pure theorem proving approach typically requires significant effort in verifying the low-level implementation of hardware. We can instead accelerate the process using the automated model checker to reason about intricate implementation details, and let the theorem prover focus on the algorithmic level. This leverages the orthogonal strengths of theorem proving and model checking: theorem proving becomes more robust as details of the hardware implementation are abstracted away, and model checking becomes more robust as it focuses on a specific small function of a large sequential machine.

C. Counter-Example Generation

ACL2SIX relies upon SixthSense to unboundedly verify a set of properties, inasmuch as those properties represent

temporally-bounded pre-condition to post-condition checks. SixthSense will produce one of the following three answers: 1) the property fails relative to specified initial states; 2) the property passes; 3) the property is unsolved given the specified set of algorithms.

When using induction as the core proof technique, properties may often be reported as unsolved even if they truly hold in all reachable states. This is a byproduct of the weakness of induction: an induction counterexample due to a transition from a passing to a failing state render the inductive check inconclusive, yet it is not known whether the inductive starting state is reachable or not. If relying upon induction as a proof technique, it is necessary for a verification engineer to analyze the induction counterexample to derive invariants which rule out that counterexample.

In the course of this verification effort, SixthSense was enhanced to produce *induction counterexample traces* for analysis by the verification engineer. SixthSense is based on the concept of transformation-based verification [20], where synergistic algorithms are applied to simplify large problems into smaller problems before applying a core proof technique. These simplifications include logic rewriting techniques [19], phase abstraction [21], redundancy removal [23], and transient logic elimination [22]. Such simplifications often considerably reduce verification resources for the core proof technique, and often considerably improve the effectiveness of induction since they rule out possible induction counterexamples where the reduced behavior does not hold. Without such reductions, the manual effort to derive such invariants often becomes infeasible given a significant amount of design artifacts.

When SixthSense generates a counterexample trace after such simplifications, that trace must be "lifted" to undo the effects of those transformations before it can be presented to the user. For traditional counterexamples, this process is straightforward as only *input* valuations need to be accounted for, allowing a top-level simulation to be used relative to this *test case* to derive values to all signals. When lifting an induction counterexample, the set of valuations to be accounted for include those of the state elements in the inductive starting state. It is further noteworthy that such counterexamples should be minimally-assigned, to improve the identification of the root cause of the induction failure.

SixthSense required several customizations to support induction trace generation. For transformation engines which may merge redundant gates, bookkeeping was added reflecting such transformations so that it may be back-annotated in a lifted trace, without which the induction trace may not truly reflect an induction counterexample. Additionally, some transformations performed by SixthSense are not themselves inductively provable, requiring more intricate unreachable-state invariants. When leveraging such reductions, we found it necessary to pass the automatically-derived unreachable-state invariants to the induction process along with the reduced design, to avoid it from rendering induction counterexamples which had no counterpart in the pre-reduced design.

TABLE IV
TIME AND MEMORY REQUIRED FOR VERIFYING MODULAR REDUCTION

Data Width	56-bit	256-bit	384-bit	512-bit
Total Time	10442s	20646s	37607s	98199s
Theorem Prover Time	257s	289s	474s	1690s
Property Check Time	10188s	20261s	37139s	97012s
Avg. Time per Prop.	118s	151s	223s	489s
Max Time per Prop.	138s	368s	1232s	3456s
Avg. Mem. per Prop.	1195MB	1459MB	1967MB	2719MB
Max Mem. per Prop.	1393MB	4201MB	5680MB	8571MB

D. Verification Results

With the approach discussed in the previous subsections, we have verified the mathematical correctness of the modular reduction engine for input data widths of 56-bits, 192-bits, 256-bit, 384-bits and 512-bits. In addition to verifying simple modular reduction, we also verified modular addition, modular subtraction and modular negation. Table IV shows the time and memory required to verify all four of these operations using a 2.27GHz Intel Xeon X7560 processor running Linux 2.6.18. The number of properties verified by invoking SixthSense varied from 86 for the 56-bit operation to 198 for the 512-bit operation. For the 1024-bit and larger input data widths, some properties could not be proven by SixthSense in 24 hours, and we did not complete the verification.

The verification process requires several iterations to attempt to inductively prove the properties. An initial property check almost always fails, causing SixthSense to produce induction counterexamples. The examination of the counterexample often reveals that the state invariants are not strong enough to constrain the hardware to behave correctly. This leads to manual strengthening of the invariants to help the verification process converge. The proof scripts are written parametrically, so that the verification for different bit widths goes through automatically, or with little human guidance.

The total labor time is difficult to measure scientifically, as it depends on numerous factors. Roughly speaking, one engineer finished the verification of 56-bit modular reduction in a few weeks. Then, two engineers spent several months to extend the results to various operations including modular add, subtract and negation operations of various data widths, while working on this part-time. Roughly equal amount of time was spent on invariant property checking and theorem proving. However, this could change significantly depending on how the verification problem is decomposed.

During the course of this effort, an engineer with a background in the VHDL and LISP languages was readily able to learn the ACL2SIX system to specify and debug invariants. However, the use of theorem proving beyond trivial proofs, such as case splitting, has a steeper learning curve.

A similar approach has been applied to the modular inverse operation implemented in the same modular reduction engine. Given operands A and N, it obtains a number X such that $(A \times X) \bmod N = 1$. The hardware uses a binary extended

Euclid algorithm [24] to calculate the number. We quickly identified that the operation may overflow out of fixed-size registers. Even the original algorithm description in [24] failed to warn that there is a danger of overflow. The DUT had a 4-bit head-room for 256-bit modular inverse calculation and 6-bits for 384-bit operation. In other words, the intermediate value can be 16 times or 64 times the maximal input values, respectively, and designers believed this to be sufficient to avoid an overflow. Using bounded model checking, we identified combinations of A and N which overflow the registers. This event is extremely rare, and neither a random simulation nor post-silicon testing could have identified such A and N. Designers added an hardware overflow check and software support to correct the problem.

V. Conclusion

In this paper, we have verified an industrial modular reduction engine implemented in the cryptographic function accelerator. We have successfully verified the mathematical correctness of the modular reduction engine upto 512-bit input data width. This is beyond what can be formally verified by either a stand-alone model checker or a theorem prover. We also applied this approach to the modular addition, modular subtraction, and modular negation operations, and verified their correctness.

We have found the hybrid verification technique using ACL2SIX an extremely powerful tool to analyze hardware accelerators implementing finite state machines. Our formal verification approach should not be viewed as an alternative to random simulation. Rather we provide an additional capability to verify systems that typical random simulation approaches fail to verify due to the sheer size of the input domain and the length of simulation cycles. We are currently working on Montgomery multiplication and exponentiation, which are yet another important subroutine of encryption accelerators.

An important trick to successful hybrid verification is to decompose the correctness problem into sub-problems of the right size. If the problem is decomposed into too many sub-problems, the theorem proving becomes time-consuming. If the problem is decomposed into larger sub-problems, the model checking fails to discharge them.

Our hybrid verification tool is already quite powerful, but it still has room for improvement. The verification of wider modular calculations, such as with data width of 4092-bits, have not been completed yet because certain state transition property checks are beyond the tool's capability. Also the theorem proving using ACL2 takes some expertise and time, even if the underlying automatic property check by SixthSense has significantly removed the burden. Further improvements are in progress to alleviate these issues.

Acknowledgment

We thank Bart Blaner and Ross Leavens at IBM for their constant feedback for the direction of our research, and on the design of the AMF engine.

References

[1] S. Rajan, N. Shankar, and M. Srivas, "An integration of model-checking with automated proof checking," in *CAV '95*, ser. Lecture Notes in Computer Science, P. Wolper, Ed., vol. 939. Liege, Belgium: Springer-Verlag, jun 1995, pp. 84–97.

[2] J. O'Leary, X. Zhao, R. Gerth, and C.-J. H. Seger, "Formally verifying IEEE compliance of floating-point hardware," *Intel Technology Journal*, vol. Q1, Feb. 1999.

[3] R. Kaivola and M. Aagaard, "Divider circuit verification with model checking and theorem proving," in *Proceedings of the 13th International Conference on Theorem Proving in Higher Order Logics*, ser. TPHOLs '00. London, UK: Springer-Verlag, 2000, pp. 338–355.

[4] A. Slobodová, *Challenges for Formal Verification in Industrial Setting*, ser. Lecture Notes in Computer Science. Springer, 2007, vol. 4346, pp. 1–22.

[5] ——, "Formal verification of hardware support for advanced encryption standard," in *FMCAD*. IEEE Press, 2008, pp. 1–4.

[6] L. Erkök, M. Carlsson, and A. Wick, "Hardware/software co-verification of cryptographic algorithms using cryptol," in *FMCAD*, 2009, pp. 188–191.

[7] E. W. Smith and D. L. Dill, "Automatic formal verification of block cipher implementations," in *FMCAD*. Piscataway, NJ, USA: IEEE Press, 2008, pp. 6:1–6:7.

[8] D. Borrione and P. Georgelin, "Formal verification of vhdl using VHDL-like ACL2 models," in *In Forum on Design Languages (FDL)*, 1999.

[9] D. Russinoff, "A mechanically checked proof of IEEE compliance of a register-transfer-level specification of the AMD-K7 floating-point multiplication, division, and square root instructions," *London Mathematical Society Journal of Computation and Mathematics*, vol. 1, pp. 148–200, December 1998.

[10] W. A. Hunt and E. Reeber, "Formalization of the DE2 language," in *Proceedings of the 13th Working Conference on Correct Hardware Design and Verification Methods (CHARME 2005), LNCS*. Springer-Verlag, 2005, pp. 20–34.

[11] E. Barker, W. Barker, W. Burr, W. Polk, and M. Smid, "Recommendation for key management part 1: General," in *NIST Special Publication 800-57, August 2005, National Institute of Standards and Technology. Available at http://csrc.nist.gov/publications/nistpubs/800-57/SP800-57-Part1.pdf*, 2005.

[12] M. Kaufmann and J. S. Moore, "An industrial strength theorem prover for a logic based on common lisp," *IEEE Transactions on Software Engineering*, vol. 23, no. 4, pp. 203–213, apr 1997.

[13] M. Kaufmann, J. S. Moore, and P. Manolios, *Computer-Aided Reasoning: An Approach*. Norwell, MA, USA: Kluwer Academic Publishers, 2000.

[14] H. Mony, J. Baumgartner, V. Paruthi, R. Kanzelman, and A. Kuehlmann, "Scalable automated verification via expert-system guided transformations," in *FMCAD*, 2004, pp. 159–173.

[15] J. Sawada and E. Reeber, "ACL2SIX: A hint used to integrate a theorem prover and an automated verification tool," in *FMCAD*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 161–170.

[16] J. Sawada, "Automatic verification of estimate functions with polynomials of bounded functions," in *FMCAD*, 2010, pp. 151–158.

[17] M. Kaufmann and J. S. Moore, "ACL2 user's manual," See URL http://www.cs.utexas.edu/users/moore/acl2/acl2-doc.html#User's-Manual.

[18] A. Pnueli, "The temporal logic of programs," in *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*. Washington, DC, USA: IEEE Computer Society, 1977, pp. 46–57.

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

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

[21] P. Bjesse and J. Kukula, "Automatic generalized phase abstraction for formal verification," in *ICCAD*, Nov. 2005.

[22] M. Case, H. Mony, J. Baumgartner, and R. Kanzelman, "Enhanced verification through temporal decomposition," in *FMCAD*, Nov. 2009.

[23] C. A. J. van Eijk, "Sequential equivalence checking without state space traversal," in *DATE*, Feb. 1998.

[24] A. J. Menezes, S. A. Vanstone, and P. C. V. Oorschot, *Handbook of Applied Cryptography*, 1st ed. Boca Raton, FL, USA: CRC Press, Inc., 1996.

Desynchronization: Design For Verification

Sudarshan K. Srinivasan and Raj S. Katti

***Abstract*—Desynchronization is used to synthesize asynchronous circuits from synchronous specifications. Controller networks used for desynchronization are highly nondeterministic and are not easily amenable for verification. We adapt the desynchronization controllers for verifiability by imposing additional sequential dependencies among controller events that reduces nondeterminism. We deduce properties of the adapted controllers, which we use to develop methods for reachability analysis and verification of desynchronized circuits. The methods are demonstrated using seven desynchronized processor models.**

I. Introduction

The impact of persistent technology scaling results in a previously ignored set of design challenges such as manufacturing and process variability, and increased significance of wire delays. The challenges threaten to invalidate the effectiveness of synchronous design paradigms at the system-level. Asynchronous circuits provide several alluring properties—over their synchronous counterparts—that pose solutions to many of these challenges. Such properties include locally generated timing signals in the place of global clocks, potential performance speedups, robustness towards variability in the manufacturing process and operating conditions, etc. [7], [5], [6], [27], [28], [29], [32], [30], [36], [39]. However, design of asynchronous circuits has been a challenge and currently lacks support of Computer-Aided Design tools. Desynchronization [7], [10] is proposed as a design solution, where pipelined circuits and systems with a high degree of asynchronocity are synthesized from synchronous parents in a manner that exploits existing CAD tool support for synchronous designs. Desynchronization methods have in fact been successfully used to design and fabricate circuits that implement the DLX architecture and the DES encryption/decryption algorithm [7].

For desynchronization to be a feasible design solution, one of the critical challenges however is verification. Verification becomes a challenge when the desynchronized circuits are pipelined. For example, the controller of a desynchronized 5-stage pipeline can have more than $16*15^{10}$ states [7]. Thus, our approach is focused on verifying desynchronized pipelines against their non-pipelined synchronous specifications.

One of the more effective formal methods to verify pipelined circuits and systems is refinement-based verification. Refinement is a formal correctness notion that can be used to check the equivalence of an implementation system and a specification system, even if the implementation and specification are at very disparate levels of abstraction. In the context of pipelines, refinement is used to verify the pipelined system (such as a pipelined processor model) against a high-level non-pipelined specification (such as an instruction set architecture machine). A number of refinement-based verification solutions have been developed for synchronous pipelines [23][25] [26][16] [13] [12] [11] [15] [33] [34] [35] [19] [41] [18] [1] [2]. These verification solutions have been developed in the context of pipelined microprocessor models. While there have been some efforts toward the verification of asynchronous pipelines [20], this area has not been explored as much.

The desynchronized pipeline controller network is highly non-deterministic in nature and also exhibits a large state space. These two factors make it hard to track the states of the controller network that are reachable from the initial or reset states (reachable states). Identifying reachable states is important as unreachable states are often inconsistent and can cause spurious counter examples. More specifically, there are two approaches to compute reachable states of the desynchronized controller network.

1) The first approach is based on symbolic simulation. The idea is to start from the set of reset states and perform symbolic simulation until no new states are discovered, *i.e.*, until a fixed point is reached. Or, start from the set of all states and perform symbolic simulation until a fixed point is reached where no new states are eliminated [25]. Approaches based on symbolic simulation of the implementation model cannot be used because the complexity of the desynchronized controller network requires a prohibitively large number of symbolic simulations of the model.
2) The second approach is to compute invariant properties of the desynchronized controller network that characterize the set of reachable states. We explored this approach and found that because of the large state space and the high degree of nondeterminism of the controller networks, we could not find a systematic approach to generate invariants to characterize the reachable states of the controller networks. The primary problem is that the behavior of a desynchronized controller depends on the state of controllers on its output side, but, does not have any dependencies on the input side *i.e.*, the source controllers. This lack of dependency on the source side results in a high degree of nondeterminism of the resulting controller networks.

Since verifiability is an important consideration for design, we propose changes to the desynchronized controllers introduced by Cortadella et al. [7]. These changes add additional sequential dependencies between controller events so that the state space of each controller and resulting controller networks are simplified and reduced. We refer to the modified con-

Sudarshan K. Srinivasan and Raj S. Katti are with the Department of Electrical & Computer Engineering, North Dakota State University, Fargo, North Dakota 58105. Email: sudarshan.srinivasan@ndsu.edu and rajendra.katti@ndsu.edu. This research was funded in part by NSF grant CCF-1117164.

trollers as Design For Verification Desynchronization (DFVD) controllers. We also develop a refinement-based verification method for desynchronized pipelines. We show that when the DFVD controllers are used, the resulting desynchronized pipelines can be verified using our approach. The specific contributions of our work are:

1) The controller used for desynchronization can hold zero, one or two tokens. The controller is initialized with one token and can transition to states with zero tokens or two tokens. Our contribution is the DFVD controller that satisfies the following property. If the controller currently holds one token, then the controller will hold on to that token until a new token is accepted on its input. This is not a property satisfied by the original controller used for desynchronization. Therefore, when the DFVD controller is initialized with one token, it will always remain in states where it has one or two tokens. This property of the DFVD controller makes it possible to compute reachable states of pipeline controller networks, which is a requirement for refinement-based verification. However, the verifiability of the controller is achieved by trading with performance. We estimate that in the worst case, pipeline throughput is degraded by the delay of four transitions of a muller-C element.
2) We analyzed and deduced 15 properties of the DFVD controller. These properties (an important contribution of our work) can be used as rules and applied to systematically generate invariants and characterize the reachable states of any DFVD controller network.
3) We have also developed a refinement-based verification procedure for desynchronized pipelined systems. Proving refinement requires a refinement map, which (in this context) is a function that maps states of the implementation (desynchronized pipelined system) to states of the specification (non-pipelined synchronous machine). Defining the refinement map in this context requires identifying duplicate information in the pipeline (which is possible in desynchronized pipelines). Our specific contribution here is to identify conditions of the controller network state that correspond to duplicate information in the data path. These duplicate conditions are generic and can be applied to any DFVD controller network. The key here is that these conditions are applicable only for reachable states (which we have been able to characterize using the DFVD properties).
4) We developed 7 desynchronized processor models of varying pipeline length (between 5 and 7 stages) and controller complexity. The models used the DFVD controllers. Our design for verification and verification approaches are demonstrated by checking the correctness of these models.

The rest of the paper is organized as follows. Related work is given in Section II. Desynchronization and DFVD controllers are described in Section III. The properties of the DFVD controllers and reachability analysis of desynchronization controller networks are described in Section IV. The desynchronized processor models used for experiments are described in Section V. The refinement-based verification procedure is detailed in Section VI. Experimental results are given in Section VII and we conclude in Section VIII. The benchmarks and tools required to reproduce our results are available in [9].

II. Related Work

Current verification technology for asynchronous circuits can be classified as property checking approaches [3][43] or methods based on trace theory [29]. The trace theory approaches target the verification of gate-level asynchronous circuits. Our focus is on the verification of desynchronized pipelined circuits and systems. Verifying pipelines has been a challenge and warrants specialized techniques. Approaches based on property checking can be used for desynchronized pipelined circuits, but, are cumbersome because a large number of properties are required and also the properties themselves can be hard to write leading to erroneous specifications [29].

Loewenstein [20] verified some properties of a counter-flow pipeline using the HOL theorem prover. Counter-flow pipelines are asynchronous in nature with results flowing in the pipeline in a direction opposite to that of instruction flow. The desynchronized pipelines we verify do not use the counter-flow mechanism. Also, our correctness proofs are based on the use of decision procedures and are highly automated.

Cortadella et al. [7] have used flow equivalence (FE) to prove the correctness of their desynchronization method and FE is well suited for this purpose. However, they have not demonstrated verification based on FE. Why do we use refinement instead of FE? Refinement is a more general notion. For example, one requirement of FE is that the specification and implementation should have the same set of latches. This requirement is not satisfied when comparing pipelined systems with non-pipelined specifications. Also, if after desynchronization, optimizations such as retiming or pipelining is applied, then the design cannot be related back to its synchronous specification using FE.

In previous work, we have developed a refinement-based verification method for desynchronized pipelines [38]. The original desynchronization controllers are used. The approach for reachability is based on performing symbolic simulation of the implementation model, starting from reset states, until no new states are discovered, *i.e.*, until a fixed point is reached. However, this approach is not viable because the complexity of the desynchronized controller network requires a prohibitively large number of symbolic simulations of the model. As a result, we were only able to verify a small subset of the reachable states and therefore, the verification method is only partial. In the current work, our approach is to generate constraints (also known as inductive invariants) on the state variables that characterize the reachable states of the system. This approach is also not viable for the original desynchronized controllers. Hence we use the design for verification approach to develop the DFVD controller. For the DFVD controller network, the latter approach for reachability is viable as shown in Section IV.

We proposed the idea of using completion functions to define the refinement map for desynchronized pipelines in [38]. We adopt this approach in our current work. However, the key difference is how duplicate data is identified in the pipeline. In [38], we relied on observing the flow of tokens in the controller network, as symbolic simulation was used for reachability analysis. However, as stated earlier, using symbolic simulation for reachability analysis is not viable. In this work, we have deduced generic conditions of the state of controller network that identify duplication in the data path, which can be used for complete safety verification and is described in Section VI.

III. Desynchronization and DFVD Controllers

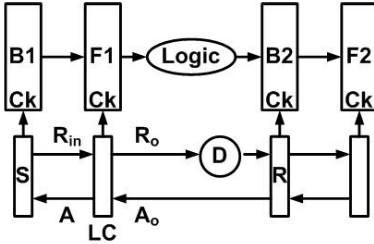


Fig. 1. A Pipeline Stage with A Latch Controller Network.

Desynchronization is the process of converting a synchronous circuit into an asynchronous one by replacing the clock network with a network of handshaking latch controllers. The edge-triggered D-flipflops of the synchronous circuit are replaced by two D-latches which are transparent when their clock input is a 1 and are in the hold mode otherwise. The clock signals or triggers (Ck) for the latches are obtained by latch controllers with two inputs (R_{in}, A_o), and two outputs ($R_o, A = \neg Ck$). $R's$ denote a request signal and $A's$ denote an acknowledge signal. Consider a synchronous pipeline stage with a logic block whose inputs are provided by a flipflop and whose outputs are input to another flipflop. A desynchronized version of such a stage is shown in Figure 1. Each flipflop is converted into two latches shown on either side of the logic block. In a pair of consecutive latches, the left latch is the back latch and the right is the front latch, indicated by subscripts "b" (or "B") and "f" (or "F"), respectively. Each latch has a controller associated with it. The latch controller used by us is the semi-decoupled controller in [10]. If G is a signal then G^+ corresponds to a rising edge on G and G^- corresponds to a falling edge on G. We now describe the operation of the latch controller labeled, LC, in Fig 1 with the help of the two latch controllers , S (for sender), and R (for receiver). LC receives R_{in}^+ from S indicating the availability of data at the input of the LC latch (F1). LC sends A^+ to S indicating that the data has been captured by F1. LC then sends R_o^+ to R to indicate that its output is valid and will be stable until A_o^+ is received. S sees A^+ and puts out R_{in}^-. LC sees R_{in}^- and A_o^+ and puts out A^- and R_o^-. R puts out A_o^- when it receives R_o^-. The above description of the latch controller (called the 4-phase controller) can be converted to the following five logic equations.

1. $A^+ = R_{in} \wedge \neg R_o$
2. $A^- = \neg R_{in} \wedge R_o \wedge A_o$
3. $R_o^+ = A \wedge \neg A_o$
4. $R_o^- = \neg A$
5. $Ck = \neg A$

Note that the clock input to the latch is Ck and the delay element D in Fig. 1 mimics the delay of the logic block. This kind of desynchronization is similar to that performed in [7] and has been proven to work well.

One of the important aspects of desynchronization is that it leads to duplicated tokens in the data path. When a token is passed from a source latch to a destination latch, the source latch holds on to a copy of the token until it receives an acknowledge signal indicating that the token has reached its destination. Thus, for a period of time, the source and destination latches both have copies of the same token. Duplication leads to some issues with refinement-based verification, which we discuss in Section VI.

A. DFVD Controller

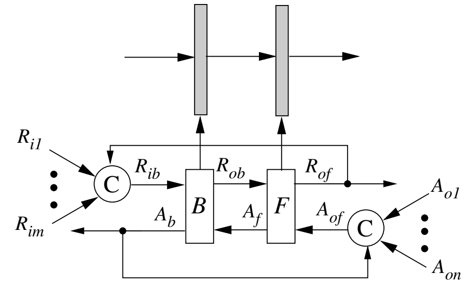


Fig. 2. DFVD Controller

The proposed Design For Verification Desynchronization (DFVD) controllers for a pipeline latch pair is shown in Figure 2. The difference between the proposed controller and the controller in the desynchronized circuit of [7] is the feedback of R_{of} to the muller-C element that generates R_{ib} and the feed forward of A_b to the muller-C element that generates A_{of}. The connections are not part of the original controller. The A_b connection enforces the property that if the controller currently holds only one token in the F latch, then the controller will hold on to that token until it has received a new token in the B latch. The F latch will drop its token when it receives an acknowledge (A_{of}^+). Since A_b is connected to the muller-C element that generates A_{of}, unless latch B acknowledges the receipt of a new token by asserting A_b, the F latch will not drop its token.

An additional dependency is enforced by the R_{of} connection that allows the B latch to receive a new token only when the F latch has signaled a request on the output side. The new controller results in only minor delays but satisfies properties that allow for reachability analysis (see Section IV).

We now estimate the *worst case* increase in delay for the new controller by estimating the maximum delay between consecutive R_{ib}^+ transitions in the controller of latch B. This

TABLE I
WORST CASE DELAY ANALYSIS OF DFVD CONTROLLER

State Label	State $\langle A_b R_{ib} R_{ob} A_f R_{of} A_{of} \rangle$	Event	Delay
S1	$\langle$ 000100 $\rangle$	R_{ob}^-	N
S2	$\langle$ 000110 $\rangle$	R_{of}^+	N
S3	$\langle$ 010110 $\rangle$	R_{ib}^+	Y
S4	$\langle$ 110110 $\rangle$	A_b^+	N
S5	$\langle$ 100110 $\rangle$	R_{ib}^-	Y
S6	$\langle$ 100111 $\rangle$	A_{of}^+	Y
S7	$\langle$ 100011 $\rangle$	A_f^-	N
S8	$\langle$ 101011 $\rangle$	R_{ob}^+	N
S9	$\langle$ 101001 $\rangle$	R_{of}^-	N
S10	$\langle$ 101101 $\rangle$	A_f^+	N
S11	$\langle$ 001101 $\rangle$	A_b^-	N
S12	$\langle$ 001100 $\rangle$	A_{of}^-	Y

delay gives us the minimum time between consecutive sets of data getting stored into a latch. In the new controller circuit R_{ib} cannot change to 1 (or 0) unless R_i's (R_{i1}–R_{im}) and R_{of} are all 1 (or 0). Similarly A_{of} cannot change to 1 (or 0) unless A_o's (A_{o1}–A_{on}) and A_b are all 1 (or 0). The set of transitions that lead to *worst case* delay for the proposed controller circuit is shown in Table I. This has been derived from the state diagram of an individual semi-decoupled 4-phase controller of [10] (see Figure 8 in [10]). The controller transitions from state S_i to S_{i+1}, starting at state S_1 and until it reaches S_{12}. From S_{12}, it transitions back to S_1. The events that causes the state transition is also shown in Table I. Delays occur when a transition of R_{ib} or A_{of} has to occur.

From the state diagram it is clear that it takes 12 state transitions for two consecutive R_{ib}^+ transitions to occur. However without the new connections to R_{ib} and A_{of} it takes 8 state transitions for two consecutive R_{ib}^+ transitions to occur. Thus we obtain a *worst case* delay of 4 state transitions for the new controller to have two consecutive R_{ib}^+ compared to the existing semi-decoupled 4-phase controller of [10], which is usually negligible compared to the delay of pipeline processing logic in a stage. Also, note that many of the transitions of the additional muller-C elements can take place simultaneously with other events in the circuit and on average the performance degradation could be much lower.

IV. REACHABILITY ANALYSIS OF DFVD CONTROLLER NETWORK

To perform verification, we need to compute the reachable states of the desynchronized pipeline controller network. Computing reachable states has two ends. First, unreachable states can be inconsistent w.r.t. the correctness property and flag spurious counter examples that hinder the verification process. Identifying reachable states of the implementation solves this problem as verification properties can now be checked only on the reachable states ensuring that spurious counter examples are eliminated. Second, our procedure for computing refinement maps for desynchronized pipelined machines is based on reachability analysis. ***Note that the reachability method eliminates unreachable states that hinder verification, which is what is required. The reachable states of the controller network may in fact only be a subset of the set of states computed by the reachability method.***

We now describe the general invariant generation rules. The first 8 rules (P1-P8) apply to the DFVD controller shown in Figure 2. Note that these rules are properties of the DFVD controller, and should be applied to each of the DFVD controllers in a DFVD pipeline controller network.

The acknowledge signal for the front and back latches A_f and A_b, respectively, also act as the clock for the front and back latches. When A_f or A_b are asserted, the corresponding latches are in a hold state, and when A_f or A_b are de-asserted, the corresponding latches are transparent (not holding any data tokens). Thus Property P_1 is a significant property as it implies that the DFVD controller will always be in a state where one or both of the latches is in a hold state. In other words, the DFVD controller will never reach a state where both latches are empty/transparent. This is not a property satisfied by the desynchronization controller proposed by [7], which allows the state where both latches are transparent. Property P_1 makes it possible for us to compute reachable states and define refinement maps in a systematic manner for desynchronized pipelines.

P_1: $A_b \vee A_f$

Property P_1 is not an invariant by itself, because there are states of the DFVD controller, which satisfy the property, but which can transition to states that do not satisfy P_1. Therefore, we need low-level properties P_2–P_8 that eliminate all such states.

P_2: $\langle A_b \wedge A_f \wedge R_{of} \rangle \rightarrow (\neg R_{ob})$

Properties P_3–P_5 identify the conditions under which the muller-C element corresponding to A_{of} should hold values of 0 and 1.

P_3: $\langle A_b \wedge A_f \wedge (\neg R_{of}) \rangle \rightarrow (\neg A_{of})$
P_4: $\langle (\neg A_b) \wedge A_f \wedge R_{of} \rangle \rightarrow (\neg A_{of})$
P_5: $\langle A_b \wedge (\neg A_f) \rangle \rightarrow A_{of}$

Properties P_6–P_8 identify the conditions under which the muller-C element corresponding to R_{ib} should hold values of 0 and 1.

P_6: $\langle A_b \wedge (\neg A_f) \wedge (\neg R_{of}) \rangle \rightarrow R_{ib}$
P_7: $\langle (\neg A_b) \wedge A_f \wedge (\neg R_{of}) \rangle \rightarrow (\neg R_{ib})$
P_8: $\langle A_b \wedge A_f \wedge R_{of} \rangle \rightarrow R_{ib}$

The conjunction of properties P_1–P_8 form an inductive invariant, which we have verified using the ACL2-SMT verification system [37] by proving that for every state of the DFVD controller that satisfies the conjunction of properties P_1–P_8, its successor also satisfies the conjunction of P_1–P_8.

Properties P_9–P_{15} apply to the circuit shown in Figure 3 that occurs in the desynchronized pipeline controller network when data is passing from one stage of the pipeline to another. In this situation, the front latch of the source stage is connected to the back latch of the destination stage. Hence the front controller of the source stage (labeled F_1 in the figure) is connected to

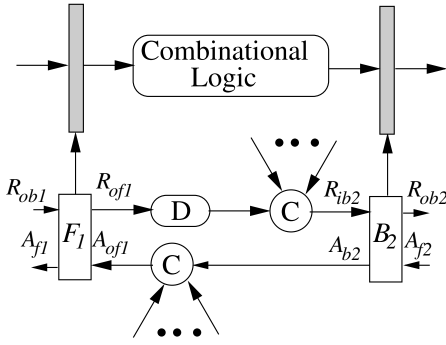


Fig. 3. DFVD Controller Circuit for Data Transfer

the back controller of the destination stage (labeled B_2 in the figure).

Properties/rules P_9–P_{15} should be applied to every source to destination connection in the pipeline including feedback connections as well. The properties P_9–P_{15} are used to eliminate inconsistent states by identifying the conditions in which the muller-C elements corresponding to R_{ib2} and A_{of1} hold values of 1 and 0.

P_9: $\langle(\neg A_{f1}) \wedge (\neg A_{b2})\rangle \rightarrow (\neg R_{ib2})$
P_{10}: $\langle(\neg A_{f1}) \wedge R_{of1}\rangle \rightarrow R_{ib2}$
P_{11}: $\langle A_{f1} \wedge (\neg A_{b2}) \wedge (\neg R_{of1})\rangle \rightarrow (\neg R_{ib2})$
P_{12}: $\langle A_{f1} \wedge A_{b2} \wedge R_{of1}\rangle \rightarrow R_{ib2}$
P_{13}: $\langle A_{f1} \wedge A_{b2} \wedge (\neg R_{of1})\rangle \rightarrow A_{of1}$
P_{14}: $\langle A_{f1} \wedge (\neg A_{b2}) \wedge R_{of1}\rangle \rightarrow (\neg A_{of1})$
P_{15}: $\langle(\neg A_{f1}) \wedge A_{b2}\rangle \rightarrow A_{of1}$

The conjunction of properties P_9–P_{15} also form an inductive invariant, which we have verified using the ACL2-SMT system by proving that for every state of the circuit shown in Figure 3 that satisfies the conjunction of properties P_9–P_{15}, its successor also satisfies the conjunction of P_9–P_{15}.

V. Desynchronized Pipelined Models

Five desynchronized pipelined processor models were developed and used as benchmarks to demonstrate the applicability and efficiency of the proposed verification solution for desynchronized systems. The models are specified using the ACL2 programming language [17]. First, a 5-stage synchronous pipelined processor model based on the DLX pipeline [31] was constructed. Three desynchronized versions of the synchronous pipeline were developed, including DPM5-1, DPM5-2, and DPM5-5. In DPM5-1, one desynchronization controller is used to control all the stages of the pipeline using the idea of clustering [8]. Clustering is also used in DPM5-2, where two desynchronization controllers are employed (one controller for the fetch and decode stages, and the second controller for the execute, memory, and write back stages). DPM5-5 is a fully desynchronized model, where 5 controllers are used (one for each stage of the pipeline). The fetch stage in DPM5-5 is further pipelined (resulting in a short instruction queue) to create DPM6-6 and DMP7-7, both of which are fully desynchronized models employing one controller for each pipeline stage. The high-level organization of DPM6-6 is shown in Figure 4.

The models are specified at the term-level [4], [40], an abstraction level in which the bit-vector data path is abstracted using integers (also called terms in this context). Also, functions that operate on data are abstracted using Uninterpreted Functions (black box functions that only satisfy the property that equal inputs produce equal outputs). Term-level abstraction is used as it drastically improves the efficiency of verification.

VI. Refinement-Based Verification

The goal of our verification procedure is to show equivalence between a pipelined desynchronized circuit/system and its non-pipelined synchronous specification. The notion of equivalence that we use is Well Founded Equivalence Bisimulation (WEB) refinement [22] and is based on stuttering bisimulation. Proving refinement guarantees that every behavior of the implementation is matched by behavior of the specification and vice versa. A detailed description of the theory of refinement can be found in [22]. It is enough to check the following correctness formula [21] to establish refinement (thereby establish equivalence) between an implementation and its specification.

Definition 1: (Core WEB Refinement Correctness Formula)

$$\begin{aligned} \langle \forall w \in \mathtt{IMPL} :: \quad & s = r(w) \quad \wedge \quad u = Sstep(s) \quad \wedge \\ & v = Istep(w) \quad \wedge \quad u \neq r(v) \\ \rightarrow \quad & s = r(v) \quad \wedge \quad rank(v) < rank(w) \rangle \end{aligned}$$

In the formula above, `IMPL` denotes the set of implementation states, *Istep* is a step of the implementation machine, and *Sstep* is a step of the specification machine. The refinement map *r* (a mechanism not found in stuttering bisimulation) is a function that maps implementation states to specification states thereby making it easy to compare systems at different abstraction levels. *rank*, used for deadlock detection, is a witness function from implementation states to natural numbers whose value decreases when there is stutter. The proof obligation that $s = r(v)$ is the safety component and guarantees that if the implementation makes progress, then the result of that progress is correct as given by the specification. The proof obligation that $rank(next\text{-}impl) < rank(impl)$ is the liveness component and guarantees that the machine will not deadlock, *i.e.*, will always make forward progress. In this work, we solve the problem of safety verification for desynchronized pipelines and reserve liveness verification for future work.

The specific steps involved in a refinement-based verification methodology for checking safety are: (a) Compute the states of the implementation model that are reachable from reset (known as reachable states). ***We use the rules given in Section IV to generate invariant properties that characterize the reachable states of any desynchronized pipeline controller network.*** (b) Construct a refinement map. (c) The models and the refinement map can now be used to state the safety component of the refinement-based correctness formula for the implementation model, which can then be automatically checked for the set of all reachable states using

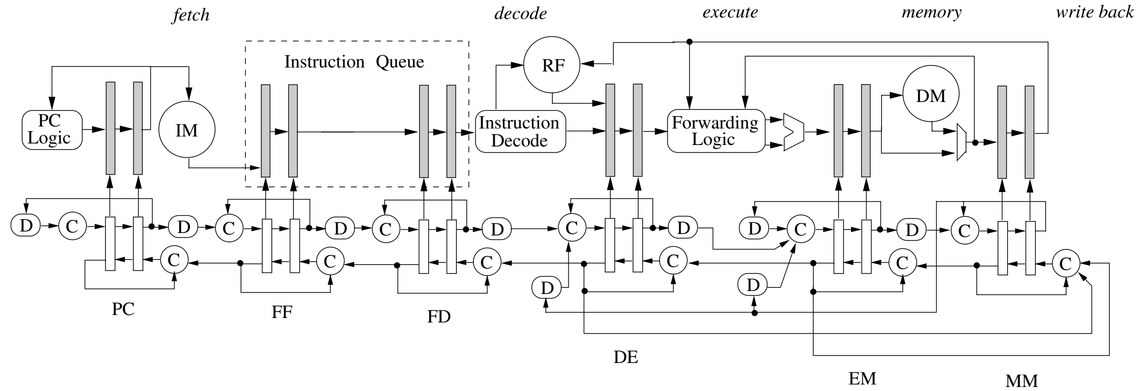


Fig. 4. High-Level Organization of Desynchronized 6-Stage Pipelined Processor Model

a decision procedure. Verification is performed using ACL2-SMT [37], a system developed by combining the ACL2 theorem prover [17] with the Yices Satisfiability Modulo Theories (SMT) solver [42].

Therefore, to perform verification, in addition to the implementation and specification models, and reachability analysis, we also require a refinement map. Next, we provide a procedure for computing refinement maps for desynchronized pipelined systems.

A. *Refinement Maps*

Our approach for defining refinement maps for desynchronized pipelines is based on the technique of completion functions proposed by Hosabettu et al. [14]. The idea with the completion functions approach is as follows. We use the DLX pipeline as an example, but the approach can be applied to any pipelined system. For a DLX pipeline, the non-pipelined specification is its instruction set architecture (ISA) machine. The state elements of this ISA machine includes its program counter, register file, data memory, and instruction memory. In addition to these state elements, a pipelined machine state also includes pipeline latches that have inflight instructions. The completion functions approach constructs a refinement map by completing the partially executed instructions in a pipelined machine state without fetching any new instructions. In the resulting, pipelined machine state, the pipeline latches are empty. Therefore, projecting out the ISA state elements from such a state would give the corresponding ISA state.

The partially executed instructions are completed by defining one function for each latch in the pipeline that observes the contents of that latch and computes how that instruction will update the ISA state elements. As instructions in the pipeline can depend on older instructions, the older instructions (instructions towards the end of the pipeline) are completed first. The values of the state elements obtained from completing older instructions are then used to complete younger instructions. Therefore, this approach allows younger instructions access to results of older instructions. Note that refinement maps based on symbolic simulation of the implementation model such as commitment [21] [23] or flushing [4] are not viable because a prohibitively large number of symblolic simulations are required for desynchronized pipelines.

The completion functions approach is efficient in terms of computational complexity and works well for synchronous pipelines. However, for desynchronized pipelines, the problem is that desynchronization allows for duplication of data in the data path. This is an issue with the completion functions approach. Completing the same instruction twice (from two different latches) will lead to erroneous results. Therefore, the duplicate instructions/data in the pipeline latches need to be identified and omitted from being completed.

Since duplication is a result of desynchronization, the latches that have duplicate data in a desynchronized pipelined machine state can be determined by observing the controller state. There are two conditions of the pipeline latch controller that identify duplicate data in the pipeline, which are given below.

D_1: $A_b \wedge R_{ob} \wedge A_f$

The first duplication condition (D_1) occurs between the latch pair used to separate two stages of a pipeline and is depicted in Figure 2. The condition occurs when the B latch is holding its data (indicated by A_b), which has also been transmitted to the F latch (indicated by $R_{ob} \wedge A_f$).

D_2: $A_{f1} \wedge R_{of1} \wedge R_{ib2} \wedge A_{b2}$

The second duplication condition (D_2) occurs between the F latch of a source latch pair (controller 1) and B latch of a destination latch pair (controller 2). The corresponding circuit is shown in Figure 3. The condition occurs when the F latch of controller 1 is holding its data (indicated by A_{f1}), which has also been transmitted to the B latch of controller 2 (indicated by $R_{of1} \wedge R_{ib2} \wedge A_{b2}$).

VII. Results

Table II reports the results for safety verification of the five desynchronized processor models described in Section V. One indicator of the complexity of the processor models is the number of lines of term-level ACL2 code required to specify the models, which is reported in the table. Note that the models are quite complex. For example, the size of the model DPM7-7 is 949 lines of term-level ACL2 code (obtained after abstracting combinational circuits blocks such as the ALU).

TABLE II
VERIFICATION TIMES AND SMT STATISTICS

Processor Model	No. Of Lines ACL2 Code	ACL2-SMT Verification Times (sec)	SMT Statistics			
			Decisions	Conflicts	Bool Vars	Memory Used (MB)
DPM5-1	687	1.19	6,292	1,202	2,723	9.41
DPM5-2	708	1.98	5,558	1,548	2,798	11.56
DPM5-5	783	4.37	70,662	16,950	3,456	13.09
DPM6-6	866	53.21	834,187	219,160	4,542	17.97
DPM7-7	949	2417.74	25,231,948	7,304,751	5,940	32.49
DPM-B1-5-2	708	1.91	5,665	1,058	2,940	11.62
DPM-B2-5-2	708	1.74	358	49	1,529	11.01

Table II reports the verification time for checking safety for the desynchronized processor models. The experiments were conducted using an Intel(R) Core(TM)2 CPU 6400, with a cache size of 2048 KB. Verification was performed using the ACL2-SMT system [37], obtained by combining ACL2 (version 3.3) and the Yices decision procedure (version 1.0.10).

The table also provides SMT statistics that are indicative of the complexity of the verification problem. Note that overall, verification is efficient as safety is verified for all the models with a maximum running time of 2417.74 seconds required by the DPM7-7 model. A well-known trend in verification of pipelined machines is the exponential increase in verification times with increase in the number of pipeline stages [24]. The results exhibit this trend as well. Compositional approaches have been successfully demonstrated to improve the scalability of refinement-based verification of synchronous pipelines [24]. For future work, we plan to explore compositional approaches for desynchronized pipelines.

Buggy Models: Two buggy variations of the DPM5-2 model were also checked using the verification procedure and resulted in the ACL2-SMT tool flagging counter examples that pointed to the source of the bug. The buggy models are DPM-B1-5-2 and DPM-B2-5-2. In DPM-B1-5-2, we injected a bug in the data path. In the forwarding path for source operand 2 from memory stage to execute stage, the destination operand address is compared with the source address of operand 1 instead of operand 2. In DPM-B2-5-2, we injected a bug in the desynchronized pipeline controller network. The DPM5-2 model has 2 DFVD controllers. The R_{ob} signal instead of the R_{of} signal of controller 1 is connected to the R_{ib} muller-C element of controller 2. The verification statistics are also reported for the buggy models in Table II.

VIII. CONCLUSIONS

Formal verification methods have become an integral part of the design cycle to ensure reliable IC designs. Therefore, verifiability has become an important consideration for any design paradigm. In this work, we propose improved verifiability for desynchronization, which is achieved with a worst case performance penalty of 4 muller-C element delay in pipeline throughput.

For future work, we plan to address liveness verification of desynchronized pipelines and explore compositional methods to improve scalability. We also plan to explore design for verification solutions for desynchronization with lower performance degradation.

REFERENCES

[1] T. Arons and A. Pnueli. Verifying tomasulo's algorithm by refinement. In *Proc. 12th International Conference on VLSI Design*, 1999.

[2] T. Arons and A. Pnueli. A comparison of two verification methods for speculative instruction execution. In *TACAS00: Tools and Algorithms for the Construction and Analysis of Systems*, pages 487–502, 2000.

[3] J. R. Burch. Combining ctl, trace theory and timing models. In J. Sifakis, editor, *Automatic Verification Methods for Finite State Systems*, volume 407 of *Lecture Notes in Computer Science*, pages 334–348. Springer, 1989.

[4] J. R. Burch and D. L. Dill. Automatic verification of pipelined microprocessor control. In *CAV '94*, pages 68–80.

[5] J. Cortadella. Petrify: A tool for manipulating concurrent specifications and synthesis of asynchronous controllers. *IEICE Trans. Inf. Syst.*, E80-D(2):315–325, 1997.

[6] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev. *Logic Synthesis of Asynchronous Controllers and Interfaces*. New York: Springer-Verlag, 2002.

[7] J. Cortadella, A. Kondratyev, L. Lavagno, and C. P. Sotiriou. Desynchronization: Synthesis of asynchronous circuits from synchronous specifications. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 25(10):1904–1921, 2006.

[8] A. Davare, K. Lwin, A. Kondratyev, and A. L. Sangiovanni-Vincentelli. The best of both worlds: the efficient asynchronous implementation of synchronous specifications. In S. Malik, L. Fix, and A. B. Kahng, editors, *(DAC'04)*, pages 588–591. ACM, 2004.

[9] Benchmarks and tools for: Desynchronization: Design for verification, 2011. See URL `http://-venus.ece.ndsu.nodak.edu/~s.srinivasan/-fmcad11.tar.gz`.

[10] S. B. Furber and P. Day. Four-phase micropipeline latch control circuits. *IEEE Trans. VLSI Syst.*, 4(2):247–253, 1996.

[11] R. Hosabettu, G. Gopalakrishnan, and M. Srivas. A proof of correctness of a processor implementing Tomasulo's algorithm without a reorder buffer. In L. Pierre and T. Kropf, editors, *Correct Hardware Design and Verification Methods, 10th IFIP WG10.5 Advanced Research Working Conference, (CHARME '99)*, volume 1703 of *LNCS*, pages 8–22. Springer-Verlag, 1999.

[12] R. Hosabettu, M. Srivas, and G. Gopalakrishnan. Decomposing the proof of correctness of a pileplined microprocessors. In *CAV '99*.

[13] R. Hosabettu, M. Srivas, and G. Gopalakrishnan. Proof of correctness of a processor with reorder buffer using the completion functions approach. In N. Halbwachs and D. Peled, editors, *Computer-Aided Verification–CAV '99*, volume 1633 of *LNCS*. Springer-Verlag, 1999.

[14] R. Hosabettu, M. K. Srivas, and G. Gopalakrishnan. Decomposing the proof of correctness of pipelined microprocessors. In A. J. Hu and M. Y. Vardi, editors, *Computer Aided Verification*, volume 1427 of *Lecture Notes in Computer Science*, pages 122–134. Springer, 1998.

[15] W. A. Hunt, Jr. Microprocessor design verification. *Journal of Automated Reasoning*, 5(4):429–460, 1989.

[16] R. Kane, P. Manolios, and S. K. Srinivasan. Monolithic verification of deep pipelines with collapsed flushing. In G. G. E. Gielen, editor, *Design, Automation and Test in Europe, (DATE'06)*, pages 1234–1239. European Design and Automation Association, Leuven, Belgium, 2006.

[17] M. Kaufmann, P. Manolios, and J. S. Moore. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, July 2000.

[18] D. Kroning. *Formal Verification of Pipelined Microprocessors*. PhD thesis, Universität des Saarlandes, 2001.

[19] S. Lahiri, S. Seshia, and R. Bryant. Modeling and verification of out-of-order microprocessors using UCLID. In *Formal Methods in Computer-*

Aided Design (FMCAD'02), volume 2517 of *LNCS*, pages 142–159. Springer-Verlag, 2002.

[20] P. Loewenstein. Formal verification of counterflow pipeline architecture. In *Proceedings of the 8th International Workshop on Higher Order Logic Theorem Proving and Its Applications*, 1995.

[21] P. Manolios. Correctness of pipelined machines. In W. A. Hunt, Jr. and S. D. Johnson, editors, *FMCAD 2000*, volume 1954 of *LNCS*, pages 161–178. Springer-Verlag, 2000.

[22] P. Manolios. *Mechanical Verification of Reactive Systems*. PhD thesis, University of Texas at Austin, August 2001. See URL `http://-www.cc.gatech.edu/~manolios/publications.html`.

[23] P. Manolios and S. K. Srinivasan. Automatic verification of safety and liveness for xscale-like processor models using web refinements. In *Design, Automation and Test in Europe (DATE'04)*, pages 168–175. IEEE Computer Society, 2004.

[24] P. Manolios and S. K. Srinivasan. A complete compositional reasoning framework for the efficient verification of pipelined machines. In *ICCAD'05*, pages 863–870, 2005.

[25] P. Manolios and S. K. Srinivasan. A computationally efficient method based on commitment refinement maps for verifying pipelined machines. In *Formal Methods and Models for Co-Design (MEMOCODE'05)*, pages 188–197. IEEE, 2005.

[26] P. Manolios and S. K. Srinivasan. Refinement maps for efficient verification of processor models. In *Design, Automation and Test in Europe (DATE'05)*, pages 1304–1309. IEEE Computer Society, 2005.

[27] A. Martin and M. Nystrom. Asynchronous techniques for system-on-chip design. *Proc. IEEE*, 94(6):1089–1120, 2006.

[28] A. J. Martin, A. Lines, R. Manohar, M. Nyström, P. I. Pénzes, R. Southworth, and U. Cummings. The design of an asynchronous mips r3000 microprocessor. In *ARVLSI*, pages 164–181. IEEE Computer Society, 1997.

[29] C. J. Myers. *Asynchronous Circuit Design*. New York: Wiley, 2001.

[30] S. M. Nowick, M. B. Josephs, and C. H. V. Berkel. Special issue: Asynchronous circuits and systems. *Proc. IEEE*, 87(2):217–396, 1999.

[31] D. A. Patterson and J. L. Hennessy. *Computer Organization and Design: The Hardware/Software Interface*. Elsevier Morgan Kaufmann, 2009.

[32] J. Rabaey, A. Chandrakasan, and B. Nikolic. *Digital Integrated Circuits: A Design Perspective*. Prentice-Hall, 2003.

[33] J. Sawada. *Formal Verification of an Advanced Pipelined Machine*. PhD thesis, University of Texas at Austin, Dec. 1999. See URL `http://-www.cs.utexas.edu/users/sawada/dissertation/`.

[34] J. Sawada and W. A. Hunt, Jr. Trace table based approach for pipelined microprocessor verification. In *Computer Aided Verification (CAV '97)*, volume 1254 of *LNCS*, pages 364–375. Springer-Verlag, 1997.

[35] J. Sawada and W. A. Hunt, Jr. Processor verification with precise exceptions and speculative execution. In A. J. Hu and M. Y. Vardi, editors, *Computer Aided Verification (CAV '98)*, volume 1427 of *LNCS*, pages 135–146. Springer-Verlag, 1998.

[36] J. Sparso and E. S. Furber. *Principles of Asynchronous Circuit Design: A Systems Perspective*. Boston, MA: Kluwer, 2001.

[37] S. K. Srinivasan. *Efficient Verification of Bit-Level Pipelined Machines Using Refinement*. PhD thesis, Georgia Institute of Technology, December 2007. See URL `http://etd.gatech.edu/theses/-available/etd-08242007-111625/`.

[38] S. K. Srinivasan and R. S. Katti. Verification of desynchronized circuits. In *ISCAS'09*, 2009.

[39] C. H. van Berkel and R. Saejis. Compilation of communicating processes into delay insensitive circuits. In *Proc. Int. Conf. Computer Design (ICCD)*, pages 157–162. IEEE Computer Society, 1988.

[40] M. N. Velev and R. E. Bryant. Bit-level abstraction in the verfication of pipelined microprocessors by correspondence checking. In *FMCAD'98*, pages 18–35, 1998.

[41] M. N. Velev and R. E. Bryant. Superscalar processor verification using efficient reductions of the logic of equality with uninterpreted functions to propositional logic. In L. Pierre and T. Kropf, editors, *Correct Hardware Design and Verification Methods, 10th IFIP WG10.5 Advanced Research Working Conference, (CHARME '99)*, volume 1703 of *LNCS*, pages 37–53. Springer-Verlag, 1999.

[42] Yices homepage, 2007. See URL `http://fm.csl.sri.com/-yices`.

[43] T. Yoneda and B.-H. Schlingloff. Efficient verification of parallel real-time systems. *Formal Methods in System Design*, 11(2):187–215, 1997.

Hunting deadlocks efficiently in microarchitectural models of communication fabrics

Freek Verbeek
Radboud University Nijmegen
Institute for Computing and Information Sciences
The Netherlands
Email: f.verbeek@cs.ru.nl

Julien Schmaltz
Open University of the Netherlands
School of Computer Science
The Netherlands
Email: julien.schmaltz@ou.nl

***Abstract*—Communication fabrics constitute an important challenge for the design and verification of multi-core architectures. To enable their formal analysis, microarchitectural models have been proposed as an efficient abstraction capturing the high-level structure of designs. We propose a novel algorithm to deadlock verification of microarchitectural designs. The basic idea of our algorithm is to capture the structure of the wait-for relations of a microarchitectural model in a *labelled waiting-graph* and to express a deadlock as a feasible *closed subgraph* of the waiting-graph. We apply our algorithm to academic and industrial Networks-on-Chip (NoC) designs. With examples we show that our tool is fast, scalable, and capable of detecting intricate message-dependent deadlocks. Deadlocks in networks with thousands of components are detected within a few seconds.**

I. Introduction

In modern architectures, performance is gained by increasing parallelism [1]. Multi-Processor Systems-on-Chips (MPSoCs) integrate on a single die several processing, memory, and I/O devices. As bus performance degrades when the number of cores increases, complex Networks-on-Chips (NoCs) constitute an alternative solution for scalable interconnect infrastructures [2], [3]. Formal verification of NoCs is a challenge. In particular, deadlock freedom is a crucial property that also is difficult to automatically verify. A solution is to analyze abstract microarchitectural models of communication fabrics. A well-defined set of primitives – named xMAS for eXecutable MicroArchitectural Specifications – has been proposed by Intel to precisely describe these models [4]. Chatterjee and Kishinevsky developed techniques to generate inductive invariants and use these invariants to improve the performance of hardware model-checking of Verilog descriptions [5]. Recently, Gotmanov *et al.* proposed a Boolean encoding of deadlock equations [6]. Using these equations and automatically generated invariants, the authors were able to verify Verilog designs for deadlocks. Their techniques scale up to networks with hundreds of components and tens of queues. Actual designs typically consist of hundreds or even thousands of queues. We report results[1] on networks with thousands of components and hundreds of queues. A direct comparison with Intel's algorithms is not possible as their tools and benchmarks are not publicly available. We exhibit one example that is out-of-reach for Intel's techniques but is verified instantaneously by our algorithm.

Our novel deadlock detection algorithm is based on the following two key concepts. The wait-for relations of xMAS models are captured in a *labelled waiting-graph*. A deadlock is defined as a *feasible closed subgraph* of the waiting-graph. Our algorithm analyses each queue of a network and either stops if a blocking queue has been found or returns "no deadlock" when all queues have been visited. For each queue, a labelled waiting-graph is built. A deadlock is found when a feasible logically closed subgraph is found in the waiting-graph of a queue. Building the waiting-graph and searching for a feasible logically closed subgraph happen on-the-fly.

The next section briefly introduces the xMAS language and illustrates the difficulty of finding deadlocks in xMAS models. Section 3 presents the theoretical foundations of our algorithm which is detailed in Section 4. Section 5 demonstrates the applicability and the efficiency of our algorithm on several and distinct examples extracted from academic and industrial NoC designs. Both routing and message dependent deadlocks are detected within seconds in designs with thousands of components. Finally, Section 6 relates our work to Intel's approach and Section 7 concludes.

II. xMAS Models

We briefly introduce the xMAS language. Our presentation is inspired by the original xMAS paper where more details can be found [4].

An xMAS model is a network of primitives connected via typed data *channels*. A channel is connected to an *initiator* and a *target*. A channel is composed of three signals. Channel signal $x.irdy$ indicates whether the initiator is ready to write to channel x. Channel signal $x.trdy$ indicates whether the target is ready to read channel x. Channel signal $x.data$ contains data that are transferred from the initiator output to the target input if and only if both signals $x.irdy$ and $x.trdy$ are set to true. Figure 1 shows the eight primitives of the xMAS language. A *function* primitive manipulates data. Its parameter is a function that produces an outgoing packet from an incoming packet. Typically, functions are used to convert message types and represent message dependencies inside the

[1]The source code for the algorithm presented in the paper are available at http://www.cs.ru.nl/~freekver/fmcad11/

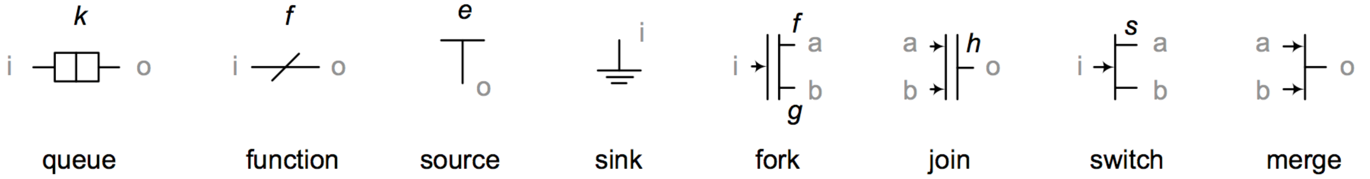


Fig. 1: Eight primitives of the xMAS language. Italicized letters indicate parameters. Gray letters indicate ports.

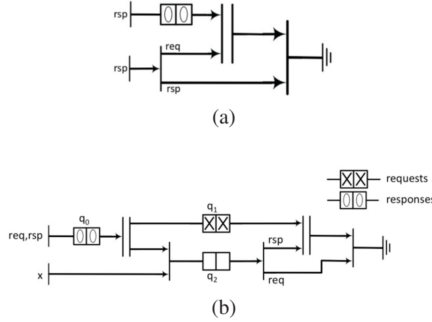


Fig. 2: Microarchitectural models

fabric or in the model of the environment. A *fork* duplicates an incoming packet to its two outputs. Such a transfer takes place if and only if the input is ready to send and the two outputs are both ready to read. As in previous publications [5], [6], we assume forks with identity functions. A *join* is the dual of a fork. The function parameter determines how the two incoming packets are merged. A transfer takes place if and only if the two inputs are ready to send and the output is ready to read. The function parameter must be total, i.e., a join is always able to produce a packet if both inputs are ready. A *switch* uses its function parameter to determine to which output an incoming packet must be routed. A *merge* is an arbiter. It grants its output to one of its inputs. A merge is fair, i.e., all inputs are served eventually. A *queue* stores data. As we assume fair arbiters, we abstract away from their internal state and a queue is the only state holding element. Messages are non-deterministically produced and consumed at *sources* and *sinks*. Sources and sinks are fair, i.e., messages are eventually created or consumed. A source or sink may process multiple message types. A *configuration* σ represents the current occupation of queues, i.e., the current state. The semantics of an xMAS network is specified using synchronous equations for each primitive [4]. Configurations are updated when messages are produced, consumed, or moved to a next queue. A *legal* configuration is a configuration where the buffer sizes of the queues are not exceeded. A configuration is *reachable* if it is possible to reach it starting from the empty network. A channel c *has type* p if and only if there exists a reachable configuration such that a packet p is located in channel c. The set of all types of channel c is noted $\tau(c)$.

Deadlocks are difficult to find in xMAS models as the traditional association between cycles and deadlocks is neither sufficient nor necessary. Consider the microarchitectural model in Figure 2b. One source emits both response and request packets. The type of packets of the other source is left uninterpreted for now. The first source feeds into queue q_0 which then enters a fork. The lower output of the fork is merged with the other source into queue q_2. From q_2, request packets are routed to a sink while response packets are joined with packets stored in q_1. The configuration in Figure 2b has a request packet in q_1 and a response packet in q_0. The join waits for response packets in q_2. Response packets wait for the fork. This fork waits for space in q_1 which in turn waits for the join. This completes a circular wait, but this circular wait is not necessarily a deadlock. If $x = \{rsp\}$, i.e., the second source generates response packets, the network is deadlock-free. If $x = \{req\}$, the configuration is a deadlock. Consider the microarchitectural model in Figure 2a. The queue waits for the join. The join waits for a request packet. As the source never produces a request packet, the configuration is a deadlock without circular waits.

III. Theoretical Foundations

Let Q be the set of queues in the network and let $q.out$ denote the output channel connected to queue q. A configuration is *stuck* if and only if the packets in all queues are blocked, i.e.:
$\mathbf{stuck}(\sigma) \stackrel{\text{def}}{=} \forall q \in Q \cdot q.out.irdy \implies \neg q.out.trdy.$

Definition 1: A configuration σ is a *deadlock configuration*, notation $\mathbf{dl}(\sigma)$, if and only if it is a non-empty configuration such that:

$$\mathbf{dl}(\sigma) \stackrel{\text{def}}{=} \mathbf{legal}(\sigma) \wedge \mathbf{reachable}(\sigma) \wedge \mathbf{stuck}(\sigma)$$

In a deadlock, the output channel of each queue that contains packets is blocked. None of the packets can proceed.

We formulate a set of *blocking equations*, notation $\mathbf{Block}(c, p)$, representing whether a packet p can be permanently blocked in channel c (Figure 4). We also define the *idle equations*, notation $\mathbf{Idle}(c, p)$, representing whether channel c can be permanently empty for packet p. We define a blocked queue, notation $\mathbf{BlockQ}(q)$, as a queue q containing a blocked packet.

$$\mathbf{BlockQ}(q) \equiv \exists p \in \tau(q.out) \cdot \#q.p \geq 1 \wedge \mathbf{Block}(q.out, p)$$

The equations in Figure 4 capture the reason why components are permanently blocking or idle. A queue is blocking if it is full and the component connected to its output channel is blocking ($\mathbf{full}(q)$ denotes "$\#q = q.size$"). A queue is idle for packet p either if it is empty and its input is connected to an idle component or if messages with packet p are blocked

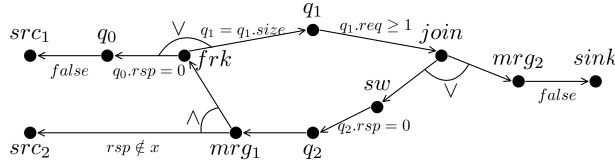


Fig. 3: Labelled waiting graph for the model in Figure 2b

Definition 2: Let c be a channel, let x (y) be the target (initiator) component of c, and let $x.in'$ ($x.out'$) be the other in- (out-) port of component x (y).

$$
\begin{array}{lcll}
\mathbf{Block}(c,p) & \equiv & \mathbf{full}(x) \wedge \mathbf{BlockQ}(x) & \text{iff } x = \text{queue} \\
 & \equiv & \mathbf{Block}(x.out, f(p)) & \text{iff } x = \text{function} \\
 & \equiv & \text{false} & \text{iff } x = \text{sink} \\
 & \equiv & \mathbf{Block}(x.out_1, p) \vee \mathbf{Block}(x.out_2, p) & \text{iff } x = \text{fork} \\
 & \equiv & \mathbf{Block}(x.out, p) \vee \forall p' \in \tau(x.in') \cdot \mathbf{Idle}(x.in', p') & \text{iff } x = \text{join} \\
 & \equiv & \mathbf{Block}(x.out_1, p) & \text{iff } x = \text{switch} \wedge b(p) \\
 & \equiv & \mathbf{Block}(x.out_2, p) & \text{iff } x = \text{switch} \wedge \neg b(p) \\
 & \equiv & \mathbf{Block}(x.out, p) & \text{iff } x = \text{merge} \\
\mathbf{Idle}(c,p) & \equiv & \left\{ \begin{array}{l} y.p = 0 \wedge \mathbf{Idle}(y.in, p) \vee \\ \exists p' \in \tau(y.out) \cdot p \neq p' \wedge y.p' \geq 1 \wedge \mathbf{Block}(y.out, p') \end{array} \right. & \text{iff } y = \text{queue} \\
 & \equiv & \forall p' \in \tau(y.in) \cdot f(p') = p \implies \mathbf{Idle}(y.in, p') & \text{iff } y = \text{function} \\
 & \equiv & p \notin \tau(y) & \text{iff } y = \text{source} \\
 & \equiv & \mathbf{Idle}(y.in, p) \vee \exists p' \in \tau(y.out') \cdot \mathbf{Block}(y.out', p') & \text{iff } y = \text{fork} \\
 & \equiv & \mathbf{Idle}(y.in_1, p) \vee \mathbf{Idle}(y.in_2, p) & \text{iff } y = \text{join} \\
 & \equiv & \mathbf{Idle}(y.in, p) & \text{iff } \left\{ \begin{array}{l} y = \text{switch} \wedge \\ (b(p) \iff c = y.out_1) \end{array} \right. \\
 & \equiv & \mathbf{Idle}(y.in_1, p) \wedge \mathbf{Idle}(y.in_2, p) & \text{iff } y = \text{merge}
\end{array}
$$

Fig. 4: Blocking equations

by other packets and cannot leave the queue. Formally, the latter means that the channel written by the queue never receives packet p. A function is blocking if its output channel is blocking after application of the function. A function is idle for packet p if its input channel is idle for all packets for which the application results in p. A sink is never blocked. A source can be idle for a particular message type. A fork is blocked if one of its outputs is blocked. A fork is idle if its input is idle. A fork can also be blocked if an output channel is blocking, since a fork can only produce two packets if all its output channels are ready to receive. A join is blocked if its output is blocked or one of its inputs is idle for any packet. A join is idle if one of its inputs is idle. A switch has one blocking equation for each possible output. The first (second) output channel of a switch is idle for p if the condition (i.e., function s applied to packet p) does not (does) hold for p or its input is idle. A merge is blocked if its output is blocked. Note that a merge may also be blocking if the other input channel is selected. However, since we assume fair merges, this cannot permanently block the input channel. As our equations capture the reason why a component is *permanently* blocking an input channel, this blocking scenario need not be reflected in the deadlock equations of the merge. A merge is idle if *both* its inputs are idle.

We now prove[2] correctness of the deadlock equations, i.e., a configuration is stuck if and only if there is a blocked queue.

Lemma 1: There exists a non-empty stuck configuration if and only if for some queue q the blocking equations are feasible:

$$\exists q \in Q \cdot \mathbf{BlockQ}(q) \iff \exists \sigma \cdot \mathbf{stuck}(\sigma)$$

Configuration σ in Lemma 1 is a configuration in which all packets are blocked. The configuration is not necessarily legal or reachable. *Legality equations* (noted $\mathbf{Legal}$) are added to bound the number of packets stored in queues. They have the following form: $\{"\#q \leq q.size" \mid q \text{ is a queue}\}$. To rule-out unreachable configurations, a *reachability invariant* (noted $\mathbf{Inv}$) is automatically generated. We have made a quick re-implementation of the invariant generation technique used in [5]. In all examples presented in this paper, the invariants generated by our quick re-implementation were enough.

The next Lemma shows that if there is a deadlock then our algorithm will find it. Note that because we may output a deadlock that is not reachable, the other direction does not hold.

Lemma 2: For any set of invariants $\mathbf{Inv}$, if there exists a deadlock configuration, then there exists a blocked queue q.

$$\exists \sigma \cdot \mathbf{dl}(\sigma) \implies \exists q \in Q \cdot \mathbf{BlockQ}(q) \wedge \mathbf{Legal} \wedge \mathbf{Inv}$$

[2]All proofs are available in an appendix at the end of this paper.

Given a queue q, the *labelled waiting graph* is a graph with as vertices the components of the network. Figure 3 shows this graph for queue q_1 in Figure 2b. The next Section details the efficient construction of this graph. Function $E_{\texttt{wait}}$ represents the edge function, i.e., $E_{\texttt{wait}}(x)$ returns the set of neighbors of component x. We let $\bigwedge(x)$ and $\bigvee(x)$ return true if and only if the edges going out of component x are conjunctive or disjunctive. An edge (x_0, x_1) between components x_0 and x_1 is labelled according to the deadlock equations of channel $x_0.out$ connecting these components. Starting from queue q the labels directly correspond to the set of equations $\mathbf{BlockQ}(q)$.

Definition 3: A waiting subgraph S is *logically closed*, notation $\mathbf{closed}(S)$, iff:

$$\mathbf{closed}(S) \stackrel{\text{def}}{=} \forall x \in S \cdot \begin{cases} \forall n \in E_{\texttt{wait}}(x) \cdot n \in S \text{ iff } \bigwedge(x) \\ \exists n \in E_{\texttt{wait}}(x) \cdot n \in S \text{ iff } \bigvee(x) \end{cases}$$

A subgraph S is *feasible* if and only if the conjunction of the constraints on all edges in S, the set of legality constraints, and the set of invariants, is feasible. For instance, subgraph $\{q_1, join, mrg_2, sink\}$ in Figure 3 is logically closed but not feasible. The next lemma shows that a deadlock is a feasible logically closed subgraph.

Lemma 3: For queue q, the deadlock equations are feasible if and only if the waiting graph of q contains a feasible and closed subgraph.

$$\forall q \in Q \cdot (\mathbf{BlockQ}(q) \wedge \mathbf{Legal} \wedge \mathbf{Inv} \iff \exists S \cdot \mathbf{feasible}(S) \wedge \mathbf{closed}(S))$$

Our final theorem is a corollary from Lemmas 2 and 3.

Theorem 1: For any set of invariants $\mathbf{Inv}$, if there is a deadlock, then there exists a waiting subgraph that is feasible and closed.

IV. Algorithm

The algorithm detects closed subgraphs and determines their feasibility. It starts a search in some queue q_0 with some packet p. The current subgraph S under consideration is $\{q_0\}$. The search expands waiting neighbors, adding them to S, as long as the subgraph is open and feasible. The search starts with forward expansion. Each forward edge requires the next component to be permanently blocked. When encountering a join, the search proceeds both forwards to determine whether the output channel can be permanently blocked *and* backwards to determine whether the input channel can be permanently idle. The result is that a tree – spanning over the waiting graph – with as root q_0 is created on-the-fly. In case of a conjunctive component, unexplored edges are marked as 'open', since they must still be explored. The algorithm proceeds its search until S is closed. The algorithm keeps track of the set of equations $\mathcal{E}_{\texttt{curr}}$ of the path leading from the initial queue q_0 to the current component x.

If a cycle, a sink, or a source is encountered, the algorithm ends its recursion. If there are no open edges and if the current subgraph S is feasible, a deadlock has been found and the algorithm terminates. Otherwise, the algorithm backtracks to the latest disjunctive point. To prevent an exponential graph exploration, we implement a memoization technique. After each recursive call, the equations – named *closing equations* – of each path leading to a cycle or source are stored. If a component is encountered that has already been visited, a deadlock has been found if the conjunction of $\mathcal{E}_{\texttt{curr}}$ and the closing equations is feasible. This ensures that each component of the waiting graph has to be visited at most once.

Consider the network in Figure 2b. We let the algorithm start in queue q_1 with packet req. It will create the graph in Figure 3 on-the-fly. The algorithm starts with expanding the join, adding "$\#q_1.req \geq 1$" to $\mathcal{E}_{\texttt{curr}}$. There are two ways to proceed: forwards to mrg_2 or backwards to the switch. The algorithm proceeds forwards. As this leads to a sink, no deadlock is found. The algorithm associates the closing equation "false" to the sink. The algorithm then proceeds backwards to determine whether the switch can be permanently idle for packet rsp. Queue q_2 is expanded, adding "$\#q_2.rsp = 0$" to $\mathcal{E}_{\texttt{curr}}$. The algorithm expands mrg_1. There are two ways to proceed: backwards to the fork or backwards to the source. The algorithm first expands the fork, but keeps track of the open edge to the source. Again, there are two ways to proceed: one forwards leading to queue q_1 and one backwards leading to queue q_0. The algorithm first proceeds forwards, adding "$\#q_1 = q_1.size$" to $\mathcal{E}_{\texttt{curr}}$. Queue q_1 has already been explored. Since there is one open edge, the algorithm starts propagating information upwards to the fork by associating "$\#q_1 = q_1.size$" as a closing equation for the fork. Consequently, it is removed from $\mathcal{E}_{\texttt{curr}}$. It proceeds by exploring queue q_0. Since this is connected to a source that injects rsp messages, queue q_0 cannot be idle for rsp. The algorithm associates closing equation "false" to the source and to queue q_0. This is propagated upwards. The closing equations of the fork become: "$\#q_1 = q_1.size \vee false$". The open edge from the merge to the source is explored. If we assume that src_2 injects rsp-messages, the algorithm associates closing equation "true" to src_2. This is propagated upwards, and the closing equation associated to the merge becomes $(\#q_1 = q_1.size \vee false) \wedge true$. As there are no more open edges, the algorithm checks the feasibility of the conjunction of $\mathcal{E}_{\texttt{curr}}$ and the closing equation of the merge, i.e., feasibility of $\{\#q_1.req \geq 1, \#q_2.rsp = 0, \#q_1 = q_1.size\}$. The solution to these equations corresponds to any deadlock configuration where q_1 is full with a request at its head, no responses are in q_2.

Algorithm 1 shows the pseudo code of our algorithm. This is one half of the algorithm, as function IdleDetect is needed to determine deadlocks for joins. Function IdleDetect is the exact dual of DeadDetect. The complete algorithm is a mutual recursion between these two dual functions. The algorithm takes four parameters: a component x that is to be explored, the current packet p, the number of open edges $open$ and the set of equations $\mathcal{E}_{\texttt{curr}}$. For each queue q, the closing equations are stored in $\mathcal{E}_{\texttt{closing}}[q]$.

Algorithm 1 $\textsc{DeadDetect}(x, p, open, \mathcal{E}_{\texttt{curr}})$

1: **if** $x ==$ queue **then**
2: $\quad\mathcal{E}_{\texttt{curr}} \wedge\!\!= \text{“}\#x = x.size\text{''}$
3: $\quad$**if** $\mathcal{E}_{\texttt{closing}}[x] == \emptyset$ **then**
4: $\quad\quad\mathcal{E}_{\texttt{closing}}[x] = \text{“}false\text{''}$
5: $\quad\quad$**for all** $p' \in \tau(x.out)$ **do**
6: $\quad\quad\quad\mathcal{E}_{\texttt{curr}} \wedge\!\!= \text{“}\#x.p' \geq 1\text{''}$
7: $\quad\quad\quad\textsc{DeadDetect}(x.out, p', open, \mathcal{E}_{\texttt{curr}})$
8: $\quad\quad\quad\mathcal{E}_{\texttt{curr}} \not\wedge\!\!= \text{“}\#x.p' \geq 1\text{''}$
9: $\quad\quad\quad\mathcal{E}_{\texttt{closing}}[x] \vee\!\!= \mathcal{E}_{\texttt{closing}}[x.out]$
10: $\quad\quad$**end for**
11: $\quad$**else if** $open == 0$ **then**
12: $\quad\quad$* Determine feasibility of equations *\
13: $\quad\quad$* Report deadlock if feasible *\
14: $\quad$**end if**
15: $\quad\mathcal{E}_{\texttt{curr}} \not\wedge\!\!= \text{“}\#x = x.size\text{''}$
16: **else if** $x ==$ function **then**
17: $\quad\textsc{DeadDetect}(x.out, f(p), open, \mathcal{E}_{\texttt{curr}})$
18: $\quad\mathcal{E}_{\texttt{closing}}[x] = \mathcal{E}_{\texttt{closing}}[x.out]$
19: **else if** $x ==$ sink **then**
20: $\quad\mathcal{E}_{\texttt{closing}}[x] = \text{“}false\text{''}$
21: **else if** $x ==$ fork **then**
22: $\quad\textsc{DeadDetect}(x.out_1, p, open, \mathcal{E}_{\texttt{curr}})$
23: $\quad\textsc{DeadDetect}(x.out_2, p, open, \mathcal{E}_{\texttt{curr}})$
24: $\quad\mathcal{E}_{\texttt{closing}}[x] = \mathcal{E}_{\texttt{closing}}[x.out_1] \vee \mathcal{E}_{\texttt{closing}}[x.out_2]$
25: **else if** $x ==$ join **then**
26: $\quad\textsc{DeadDetect}(x.out, p, open, \mathcal{E}_{\texttt{curr}})$
27: $\quad$**for all** $p' \in \tau(x.in')$ **do**
28: $\quad\quad\textsc{IdleDetect}(x.in', p', open, \mathcal{E}_{\texttt{curr}})$
29: $\quad$**end for**
30: $\quad\mathcal{E}_{\texttt{closing}}[x] = \mathcal{E}_{\texttt{closing}}[x.out] \vee \mathcal{E}_{\texttt{closing}}[x.in']$
31: **else if** $x ==$ switch **then**
32: $\quad$**if** cond(p) **then**
33: $\quad\quad\textsc{DeadDetect}(x.out_1, p, open, \mathcal{E}_{\texttt{curr}})$
34: $\quad\quad\mathcal{E}_{\texttt{closing}}[x] = \mathcal{E}_{\texttt{closing}}[x.out_1]$
35: $\quad$**else**
36: $\quad\quad\textsc{DeadDetect}(x.out_2, p, open, \mathcal{E}_{\texttt{curr}})$
37: $\quad\quad\mathcal{E}_{\texttt{closing}}[x] = \mathcal{E}_{\texttt{closing}}[x.out_2]$
38: $\quad$**end if**
39: **else if** $x ==$ merge **then**
40: $\quad\textsc{DeadDetect}(x.out, p, open, \mathcal{E}_{\texttt{curr}})$
41: $\quad\mathcal{E}_{\texttt{closing}}[x] = \mathcal{E}_{\texttt{closing}}[x.out]$
42: **end if**

We detail the case where x is a queue. Other cases are processed similarly. In the case of a queue, x must be full in order to be blocking. Equation ”$\#x = x.size$” is conjunctively added to the current set of equations (line 2). For each new packet p', the algorithm adds equation ”$\#x.p' \geq 1$” and recursively determines whether the next component can be permanently blocking (lines 5–7). After the recursive call, equation ”$\#x.p' \geq 1$” is retracted (line 8). After all recursive calls, equation ”$\#x = x.size$” is retracted (line 15).

The number of open edges can increase only in IDLEDETECT. Open edges occur with functions, switches, and merges. Only if the number of open edges is equal to zero and if some cycle has occurred, the sets of equations are fed to a linear programming solver. We use lp_solve, an off-the-shelve linear programming solver [7]. We have equations stored in efficient data structures in such a way that, e.g., $(\#q_1 = q_1.size \vee false) \wedge true$ is stored simply as $\#q_1 = q_1.size$. Adding equations to this data structure is only possible if the resulting set of equations is still internally feasible, i.e., feasible without further invariants. This prevents unnecessary exploration of infeasible paths.

Correctness of the algorithm means that function DEADDETECT returns true if and only if there is a feasible closed subgraph.

Lemma 4:

$$\exists x, p \cdot \textsc{DeadDetect}(x, p, 0, \emptyset) \iff \exists S \cdot \mathbf{feasible}(S) \wedge \mathbf{closed}(S))$$

Remarks:

Counterexamples: If our algorithm finds a feasible and closed subgraph, it has given the set of constraints corresponding to this subgraph to a linear programming solver. This solver not only returns a boolean value indicating that the set of constraints is feasible, but also a solution. This solution assign integers to queues and headers. It is a detailed representation of a counterexample, i.e., a deadlock configuration.

Running time: Each separate run of the algorithm visits each component at most once. As per component deadlock equations are memoized there is no need to re-explore a visited component. The algorithm is executed once for each queue. The number of recursive calls is therefore $\mathrm{O}(Q \cdot C)$ with Q the number of queues and C the number of components.

Before running the algorithm, the typing information needs to be computed, i.e., we need to compute $\tau(c)$ for all channels c. To obtain this information we perform exhaustive simulations. For each source and for each possible packet p injected at the source, we simulate the injection in an empty network until it is consumed. During this simulation p is added to $\tau(c)$ for each visited channel c. During this simulation, queues may need to be visited more than once.

Consider the network in Figure 6. The network is deadlock-free. To establish this, it must be established that always eventually a packet “5” arrives at queue q_1. During the simulation of packet “0” in source src_0, queue q_0 is visited 6 times. This establishes that $\tau(c) = \{0, 1, \ldots, 5\}$. Using this information, our algorithm needs to visit queue q_0 just once to establish deadlock freedom of the network.

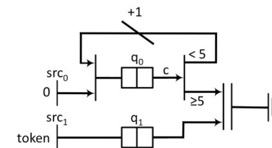


Fig. 6: xMAS model

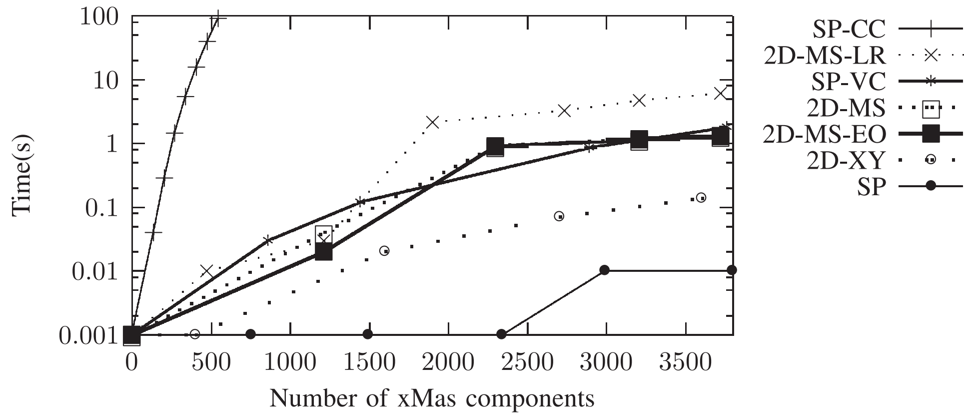


Fig. 5: Experimental results

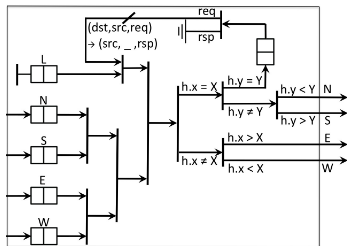
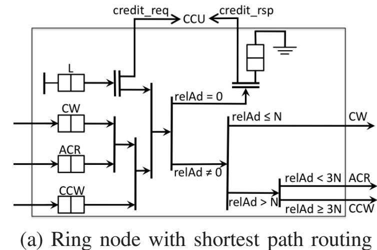
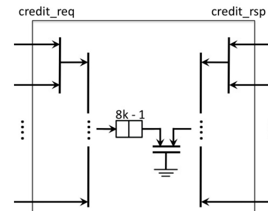


Fig. 7: xMAS model of an HERMES node



(a) Ring node with shortest path routing



(b) Credit control unit

Fig. 8: Spidergon with flow control

V. Experimental results

We consider two Network-on-Chip (NoCs): The HERMES NoC [8] from the University of Rio Grande in Brazil and the Spidergon NoC from STMicroelectronics [9]. All experiments have been performed on a Ubuntu machine with a 2.93 GHz Intel Core 2 Duo processor and 2 GB memory. Figure 5 gives an overview of experimental results on the benchmarks described hereafter.

HERMES is a two-dimensional mesh using XY routing [10]. Figure 7 shows an xMAS specification of a processing node with coordinates (X, Y). This node is a "slave". It introduces message dependencies as responses are generated upon reception of requests. A master node would only inject requests in its local queue and consume responses. We experimented with different layouts of masters and slaves: no master and no slave (curve 2D-XY), all nodes are both master and slave (curve 2D-MS), masters on the left part of the mesh and slaves on the right part (curve 2D-MS-LR), or masters on even columns and slaves on odd columns (curve 2D-MS-EO). Two layouts only are deadlock-free (2D-XY and 2D-MS-LR). The results show good performance for detecting deadlocks and proving their absence.

Spidergon is a ring where each processing node can send messages clockwise, counter clockwise, or across. Shortest path routing is used. At each node, the routing decision is based on the relative address $relAd = (d - s) \mod N$. Here d is the destination, s is the current node, and N is the total number of nodes. Because of the ring, this architecture has a deadlock (curve SP). In this case, performance is linear in the size of the ring.

To resolve this deadlock, virtual channels [11] are inserted to the right upper quarter of the ring only (curve SP-VC). The routing function is modified such that virtual channels are only used for each destination inside the quarter, other messages still use the regular channels. This case is slightly more difficult because there is no deadlock. If virtual channels are wrongly designed, deadlock detection is as in curve SP.

Another approach is to add a credit control unit (CCU, Figure 8) limiting the number of packets in the ring to $N \cdot k - 1$, where N is the size of the ring and k the size of the queues. When injecting messages in local queues, these messages are duplicated and sent to the CCU. When messages are sunk, they are also duplicated and sent to the CCU to free space. This unit may look unrealistic but its main purpose is to illustrate a difficult case for our algorithm. Indeed, the merges force our algorithm to branch on all the inputs of these merges. As it can be seen in curve SP-CC, this case is much harder. Still, networks with tens of agents and hundreds of components can be proven deadlock-free within a few minutes. If the counter

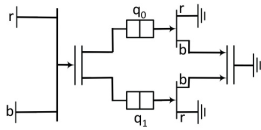


Fig. 9: Example from Intel

is wrongly sized, e.g., `cc_queue.size` $= Nk$, a deadlock is found as in curve SP.

Figure 9 shows a network abstracted from a real design from Intel [12]. The sources in the network emit red and blue tokens respectively. These tokens are duplicated into two queues. Red tokens are sunk, blue tokens are joined and then sunk. The network is deadlock-free, as queues q_0 and q_1 are fed with tokens in the same order. Given invariants automatically generated by [5], the approach of [6] cannot handle this example while our algorithm returns "no deadlock" instantaneously.

VI. Related work

We define a deadlock configuration while Gotmanov *et al.* [6] define a dead channel, i.e., a channel that is never idle but always blocked in some execution. Assuming fair merges and that a dead channel coincides with a blocked queue, the two definitions are logically equivalent. We can prove that there exists a dead channel if and only if there exists a deadlock configuration. Our approach covers a similar property as [6]. An important difference is that we directly tackle xMAS models and not their Verilog implementation. The two techniques are complementary. Our tool can be used to quickly remove all deadlocks in xMAS models before proving the Verilog deadlock-free.

VII. Conclusion

We have shown that based on the notions of a labelled waiting graph and a logically closed subgraph it is possible to efficiently detect deadlocks in microarchitectural models of communication fabrics. We demonstrated the applicability and efficiency of our solution on several deadlock avoidance mechanisms used in academic and industrial NoCs designs. Deadlocks are found within seconds in networks with thousands of components. We exhibited an example that can be proven deadlock-free using our technique but could not be handled by Intel's recent related solution. Our technique uses less and simpler invariants showing that using the labelled waiting graph we capture more information about the structure of xMAS models.

Acknowledgments

This research is supported by NWO/EW project Formal Validation of Deadlock Avoidance Mechanisms (FVDAM) under grant no. 612.064.811. This research received a gift from Intel Corporation.

References

[1] W. Dally, "The end of denial architecture," Keynote at Design Automation Conference (DAC'09), 2009.

[2] W. J. Dally and B. Towles, "Route packets, not wires: on-chip interconnection networks," in *Proceedings of the Design Automation Conference*, Las Vegas, NV, 2001, pp. 684–689.

[3] L. Benini and G. D. Micheli, "Networks on Chips: A New SoC Paradigm," *Computer*, vol. 35, no. 1, pp. 70–78, 2002.

[4] S. Chatterjee, M. Kishinevsky, and U. Ogras, "Quick formal modeling of communication fabrics to enable verification," in *Proc. of High Level Design Validation and Test Workshop (HLDVT'10)*, 2010, pp. 42–49.

[5] S. Chatterjee and M. Kishinevsky, "Automatic generation of inductive invariants from high-level microarchitectural models of communication fabrics," in *Proc. of Computer Aided Verification (CAV'10)*, 2010, pp. 321–338.

[6] A. Gotmanov, S. Chatterjee, and M. Kishinevsky, "Verifying deadlock-freedom of communication fabrics," 2011, to appear in Proceedings of VMCAI '11.

[7] M. Berkelaar, K. Eiklan, and P. Notebaert, *lp_solve (version 5.5.2.0)*, available under GNU LGPL. [Online]. Available: http://lpsolve.sourceforge.net/5.5/

[8] F. Moraes, N. Calazans, A. Mello, L. Möller, and L. Ost, "Hermes: an infrastructure for low area overhead packet-switching networks on chip," *Integration, the VLSI Journal*, vol. 38, no. 1, pp. 69 – 93, 2004.

[9] F. K. A. Karim and S. Dey, "An interconnect architecture for networking systems on chips," *IEEE Micro*, pp. 36–45, 2002.

[10] L. Ni and P. Mckinley, "A survey of wormhole routing techniques in direct networks," *IEEE Computer*, vol. 26, pp. 62–76, Februari 1993.

[11] M. Coppola, M. D. Grammatikakis, R. Locatelli, G. Maruccia, and L. Pieralisi, *Design of Cost-Efficient Interconnect Processing Units: Spidergon STNoC*, 1st ed. CRC Press, Inc., 2008.

[12] S. Chatterjee and M. Kishinevsky, personal communication.

Appendix
Proofs of Lemmas

Lemmas in Section 3

Lemma 1: There exists a non-empty stuck configuration if and only if for some queue q the blocking equations are feasible:

$$\exists q \in Q \cdot \mathbf{BlockQ}(q) \iff \exists \sigma \cdot \mathbf{stuck}(\sigma)$$

Proof:
($\Longleftarrow$) Assume queue q is non-empty and blocked in configuration σ. We prove that σ satisfies the set of blocking equations by structural induction on the definition of $\mathbf{Block}$. Thus σ is a solution of these equations, implying they are feasible.

The base case is trivial. For the inductive case, we proceed by case distinction and only detail the case where x is a queue. The other cases are similar. Channel $x.in$ is permanently blocked by assumption. This happens only when x is full. Thus σ satisfies the equation $\#x = x.size$. Furthermore, the packet at the head of x must be permanently blocked. Let p' denote the header of this packet. Channel $x.out$ must be permanently blocked for packet p', since otherwise the packet at the head of x eventually is removed from the queue and channel $x.in$ becomes alive, contradicting the assumption that $x.in$ is permanently blocked. By induction hypothesis, if channel $x.out$ is permanently blocked for p', σ satisfies $\mathbf{Block}(x.out, p')$.
($\Longrightarrow$) Assume that for some queue q the equations are feasible. This means there exists a solution, which is an assignment of integers to queues and packets. This solution is thus a configuration. We prove that in this configuration, each

channel c involved in the set of equations is permanently blocked by its target component x. The proof is again by induction on $\mathbf{Block}$ and then by case distinction. We provide details on the case of the join. By the induction hypothesis we know that either the output is permanently blocking join x, or the other input channel is permanently idle. In both cases, channel c is permanently blocked by the join. This concludes the proof that the target of the channel is permanently blocked. ■

Lemma 2: For any set of invariants $\mathbf{Inv}$, if there exists a deadlock configuration, then there exists a blocked queue q.

$$\exists \sigma \cdot \mathbf{dl}(\sigma) \implies \exists q \in Q \cdot \mathbf{BlockQ}(q) \wedge \mathbf{Legal} \wedge \mathbf{Inv}$$

Proof: From Lemma 1, we know that a configuration that is stuck implies a blocked queue. By definition, a legal configuration implies the legality constraints. A reachable configuration implies the reachability invariant. ■

Lemma 3: For queue q, the deadlock equations are feasible if and only if the waiting graph of q contains a feasible and closed subgraph.

$$\forall q \in Q \cdot (\mathbf{BlockQ}(q) \wedge \mathbf{Legal} \wedge \mathbf{Inv} \iff \exists S \cdot \mathbf{feasible}(S) \wedge \mathbf{closed}(S))$$

Proof:
($\Longrightarrow$) By assumption, the set of blocking equations is feasible for queue q and packet p. Consider the set of equations $\mathcal{E}$ obtained by replacing each disjunction in $\mathbf{Block}(q.out, p)$ by one feasible operand of the disjunction. Let S be the waiting graph corresponding to $\mathcal{E}$. S is a subgraph from the waiting graph of q. Since $\mathbf{Block}(q.out, p)$ is feasible, S is feasible as well. Finally, by construction S contains all its conjunctive neighbors and exactly one disjunctive neighbor for each disjunctive component. Thus S is a feasible and closed waiting subgraph.
($\Longleftarrow$) Assume a feasible and closed subgraph S in the waiting graph of queue q. Let $\mathcal{E}$ be the set of equations corresponding to S. The set of equations $\mathcal{E}$ is a subset of $\mathbf{Block}(q.out, p)$ for some p. We prove that the feasibility of $\mathcal{E}$ implies feasibility of $\mathbf{Block}(q.out, p)$. Adding disjunctive operands to a disjunction somewhere in $\mathcal{E}$ can make it infeasible only if the number of operands is equal to zero. This is not possible since – as S is closed – $\mathcal{E}$ contains at least one disjunctive neighbor for each component. Adding conjunctive operands can make a conjunction infeasible, but since S is closed it already contains all its conjunctive neighbors. The feasibility of $\mathcal{E}$ implies the feasibility of the deadlock equations of q. ■

Lemmas in Section 4

Lemma 4:

$$\exists x, p \cdot \textsc{DeadDetect}(x, p, 0, \emptyset) \iff \exists S \cdot \mathbf{feasible}(S) \wedge \mathbf{closed}(S))$$

Proof: The algorithm reports a deadlock only at line 13. It keeps at all time track of the number of open edges in parameter $open$. As line 13 is only reached when the current subgraph is closed, i.e., $open == 0$, and when the linear program solver has determined feasibility, partial correctness is trivial to prove. What remains to be proven is termination. As the algorithm keeps track of visited components, each component is visited at most twice (once in DEADDETECT and once in IDLEDETECT). Thus the algorithm terminates. ■

Relation to Intel's approach

We have the following assumptions: 1) The network is livelock free, 2) the network is starvation free, and 3) a blocked channel implies a blocked packet in some queue.

We first prove a Lemma on *draining* a configuration. Let σ be a configuration. Draining σ is defined as:

- Canceling all further injections at the sources;
- Having the sinks consume all packets deterministically;
- Let the network execute until no packet in the network can be moved, i.e., until $\neg c.trdy$ for all channels c.

Lemma 5: Let σ be a legal and reachable configuration. Draining σ yields a unique legal and reachable configuration, denoted with $\mathbf{drain}(\sigma)$.

$$\mathbf{legal}(\sigma) \wedge \mathbf{reachable}(\sigma) \implies \mathbf{legal}(\mathbf{drain}(\sigma)) \wedge \mathbf{reachable}(\mathbf{drain}(\sigma))$$

Proof: Any configuration obtained from an execution starting in a legal and reachable configuration is legal and reachable. Non-determinism occurs at sources and sinks only. Since sources do not inject any further packets, and since sinks are deterministic while draining, no non-determinism occurs. Draining is a deterministic process and thus it suffices to show termination to show that it yields a unique configuration. By Assumption 1 no packet can move around infinitely in the network. Eventually, all packets will either be permanently blocked or arrive at a sink. Thus draining terminates. ■

Lemma 6: There exists a dead channel if and only if there exists a deadlock configuration:

$$\exists c \cdot \mathbf{dead}(c) \iff \exists \sigma \cdot \mathbf{dl}(\sigma)$$

Proof:
($\Longrightarrow$) Let c be a dead channel in some execution S. We know that $S \models \Diamond(c.irdy \wedge \Box\neg c.trdy)$. From the semantics of $\Diamond$, we can split S in to execution S_1 and S_2 such that $S = S_1 S_2$, S_1 is a finite execution, and $S_2 \models c.irdy \wedge \Box\neg c.trdy$. Let σ' be the configuration obtained after execution of S_1. Let $\sigma = \mathbf{drain}(\sigma')$. In other words, replace execution S_2 by draining. By Lemma 5, σ is legal and reachable. By definition of draining, σ is stuck: either there are no more packets in the network in which case the network is stuck trivially, or all packets are blocked. What remains to be proven is that σ is non-empty. In execution S_2, channel c is permanently blocked. Execution S_2 is replaced by draining. This preserves the permanent blocking of channel c. Channel c can either be permanently blocked by a starvation scenario or because of a local deadlock. By Assumption 2, only the second can occur. This local deadlock is not resolved by draining, as the packets participating in this local deadlock are permanently blocked. Therefore, there is at least one channel that is blocked in σ. By Assumption 3 there is at least one queue blocked, meaning

that σ is non-empty. As σ is non-empty, legal, reachable, and stuck, σ is a deadlock configuration.
($\Longleftarrow$) As there exists a deadlock configuration, there exists a non-empty queue q which is permanently blocked. Channel $q.out$ is dead: as queue q is non-empty, the initiator of $q.out$ is not idle. As the packet in queue q cannot move, the target of $q.out$ is permanently blocked. ■

Author Index

Aggarwal, Prashant 9
Ahmad, Tariq 67
Asok Kumar, Jayanand 196

Basith, Mohamed 67
Baumgartner, Jason 101, 109, 207
Belov, Anton 37
Bingham, Brad 186
Bingham, Jesse 186
Bloem, Roderick 91
Bradley, Aaron 3, 144
Brady, Bryan 116
Brayton, Robert 125
Bryant, Randal 116

Case, Michael 109, 207
Case, Mike 101
Chaki, Sagar 72
Chamarthi, Harsh Raju 46
Chockler, Hana 135
Chu, Darrow 9
Ciesielski, Maciej 67
Cimatti, Alessandro 54

Een, Niklas 125

German, Steven 176
Greenstreet, Mark 186
Griggio, Alberto 28
Gupta, Aarti 1
Gurfinkel, Arie 72

Hasan, Osman 163
Hassan, Zyad 144
Havlicek, John 155
Hughes, John 17

Ivrii, Alexander 135

Kadamby, Vijay 9
Kanzelman, Bob 101
Kanzelman, Robert 109
Katti, Raj 215

Koenighofer, Robert 91
Kotker, Jonathan 81

Little, Scott 155
Liu, Lingyi 196

Malik, Sharad 63
Mandel, Louis 171
Manolios, Panagiotis 41, 46
Marques-Silva, Joao 37
Matsliah, Arie 135
Mcmillan, Kenneth 19
Mishchenko, Alan 125
Mony, Hari 101, 109, 207
Moore, J Strother 18
Moran, Shiri 135
Mover, Sergio 54

Nevo, Ziv 135

Papavasileiou, Vasilis 41
Paruthi, Viresh 207
Pike, Lee 206
Plateau, Florence 171
Pouzet, Marc 171

Rossi, Andre 67

Sadigh, Dorsa 81
Sandon, Peter 207
Sawada, Jun 207
Schmaltz, Julien 223
Seshia, Sanjit A. 81, 116
Siddique, Umair 163
Singhal, Vigyan 9
Somenzi, Fabio 3, 144
Srinivasan, Sudarshan 215
Strichman, Ofer 72
Sutherland, Ivan 2

Talupur, Muralidhar 154
Tonetta, Stefano 54

Vasudevan, Shobha 196
Verbeek, Freek 223

Weissenbacher, Georg 63

Zhang, Yan 144
Zhu, Charlie Shucheng 63

FMCAD 2011 SPONSORS