2008 Formal Methods in Computer-Aided Design

Portland, Oregon, USA
17-20 November 2008

ISBN: 978-1-4244-2735-2

Library of Congress: 2008906093

Catalogue: CFP08FMC

Edited by
Alessandro Cimatti
Robert B. Jones

2008 Formal Methods in Computer Aided Design

IEEE Catalog Number	CFP08FMC
ISBN	978-1-4244-2735-2
Library of Congress	2008906093

Additional copies of this publication are available from

IEEE Operations Center
P. O. Box 1331
445 Hoes Lane
Piscataway, NJ 08855-1331 USA
+1 800 678 IEEE
+1 732 981 1393
+1 732 981 0600
+1 732 981 9667 (FAX)
email: customer.service@ieee.org

Produced by the IEEE eXpress Conference Publishing
For information on producing a conference proceedings and
quotations, contact conferencepublishing@ieee.org
http://www.ieee.org/conferencepublishing

Preface

This volume contains the proceedings of the International Conference on Formal Methods in Computer-Aided Design (FMCAD), held in Portland, Oregon, USA, November 17-20, 2008. FMCAD 2008 is the eighth in a series of conferences on the theory and applications of formal methods in hardware and system verification. FMCAD provides a leading forum to present and discussing ground-breaking methods, technologies, theoretical results, and tools for reasoning formally about computing systems. We are pleased to see wide spectrum of participation in the conference and in the organization of the conference. Participants include researchers and practitioners from academia, EDA vendors, and hardware and software companies.

FMCAD 2008 received 61 regular paper submissions and 9 short paper submissions. Each submission was reviewed by at least four members of the Program Committee. The Program Committee accepted 24 regular papers and 4 short papers.

In addition to the technical papers, the conference included two invited talks, a full day of invited tutorials, and two panel sessions.

The invited talks were:

- Ken McMillan (Cadence), *Interpolation: Theory and Applications*
- Carl Seger (Intel), *Formal Methods and Physical Design: Match Made in Heaven or Fools' Paradise?*

The tutorial day included four tutorials:

- Byron Cook (Microsoft), *Computing bounds on space and time for hardware compilation*
- David Hardin (Rockwell Collins), *Considerations in the Design and Verification of Microprocessors for Safety-Critical and Security-Critical Applications*
- Kevin Jones (Rambus), *Analog and Mixed Signal Verification: The State of the Art and some Open Problems*
- Moshe Levinger (IBM), *Building a Bridge: From Pre-Silicon Verification to Post-Silicon Validation*

The first panel, *High Level Design and ESL: Who Cares?*, was moderated by Alan Hu (Univ. of British Columbia). Panelists included: Masahiro Fujita (Univ. of Tokyo), Rajesh Gupta (Univ. of California, San Diego), Anmol Mathur (Calypto), and Scott Runner (Qualcomm).

The second panel, *The Future of Formal: Academic, IC, EDA, and Software Perspectives*, was moderated by Carl Pixley. Panelists included Tom Ball (Microsoft), Ziyad Hanna (Jasper), and Moshe Vardi (Rice University).

We are grateful to our sponsors for their support: FMCAD Inc., and the IEEE Council on Electronic Design Automation (CEDA). We are also grateful to the ACM Special Interest Group on Design Automation (SIGDA) for their support of FMCAD as an in-cooperation event. We gratefully acknowledge financial contributions from Cadence, Galois, IBM, Intel, NEC Labs America, and Synopsys.

We thank the Program Committee and the referees for their efforts in reviewing the papers. We also thank the Program Committee for their extensive discussion about the program. We thank the Organizing Committee for their hard work: Carl Pixley (Synopsys) as the Panels Chair, Anna Slobodova (Centaur Technologies) as the Tutorials Chair, and Lee Pike (Galois) as the Publicity

Chair. We extend special thanks to Annette Bunker (Intel), who served as the Local Arrangements Chair. She coordinated and arranged the venue, cruise, registration, audio/visual, and a myriad of other details. We also thank Anders Franzen (FBK-irst) and Alberto Griggio (University of Trento) for serving as Webmasters. We appreciate Andrei Voronkov's marvelous EasyChair conference management system. Finally, we thank the FMCAD Steering Committee for their help and guidance.

September 2008

Alessandro Cimatti (FBK-irst)
Robert Jones (Intel Corporation)

Conference Organization

Organizing Committee

Chairs:	Alessandro Cimatti (FBK-irst, Italy)
	Robert Jones (Intel Corp., USA)
Tutorials:	Anna Slobodova (Centaur Technology, USA)
Panels:	Carl Pixley (Freescale, USA)
Local Arrangements:	Annette Bunker (Intel, USA)
Publicity:	Lee Pike, (Galois Inc., USA)
Webmasters:	Anders Franzen, (FBK-irst, Italy)
	Alberto Griggio, (U. of Trento, Italy)

Program Committee

Mark Aagaard (U. of Waterloo, Canada)
Jason Baumgartner (IBM Corp., USA)
Valeria Bertacco (U. of Michigan, USA)
Armin Biere (Johannes Kepler U., Austria)
Per Bjesse (Synopsys, USA)
Roderick Bloem (TU Graz, Austria)
Dominique Borrione (Grenoble U., France)
Gianpiero Cabodi (Politecnico di Torino, Italy)
Alessandro Cimatti (FBK-irst, Italy)
Koen Claessen (Chalmers, Sweden)
Ganesh Gopalakrishnan (U. of Utah, USA)
Aarti Gupta (NEC Laboratories America, USA)
Alan Hu (U. of British Columbia, Canada)
Robert Jones (Intel Corp., USA)
Daniel Kroening (Oxford U., UK)
Andreas Kuehlmann (Cadence Labs, USA)
Wolfgang Kunz (U. of Kaiserslautern, Germany)
Shuvendu Lahiri (Microsoft, USA)
Jeremy Levitt (Mentor Graphics, USA)
Panagiotis Manolios (Northeastern U., USA)
Andy Martin (IBM Research Division, USA)
Tom Melham (Oxford U., UK)
Ken McMillan (Cadence Labs, USA)
John O'Leary (Intel Corp., USA)
Lee Pike (Galois Inc., USA)
Rajeev Ranjan (Jasper Design Automation, USA)
Sandip Ray (U. of Texas at Austin, USA)
Alper Sen (Freescale, USA)
Natasha Sharygina (U. of Lugano, Switzerland)
Eli Singerman (Intel Corp., Israel)
Fei Xie (Portland State U., USA)
Karen Yorav (IBM Haifa Research Laboratory, Israel)

Steering Committee

Jason Baumgartner (IBM Corp., USA)
Aarti Gupta (NEC Laboratories America, USA)
Warren Hunt (U. of Texas at Austin, USA)
Panagiotis Manolios (Northeastern U., USA)
Mary Sheeran (Chalmers, Sweden)

Corporate Sponsors

Cadence
Galois
IBM
Intel
NEC Labs America
Synopsys

Referees

Perry Alexander
Himyanshu Anand
Zaher Andraus
Tamarah Arons
Domagoj Babic
Sharon Barner
Gerard Basler
Sergey Berezin
Nicolas Blanc
Peter Boehm
Angelo Brillout
Robert Brummayer
Roberto Bruttomesso
Michael Case
Donald Chai
Ben Chambers
Kai-hui Chang
Debapriya Chatterjee
Yury Chebiryak
Xiaofang Chen
Hana Chockler
Vijay D'Silva
Pallab Dasgupta
Jed Davis
Andrew DeOrio
David Dill
Peter Dillinger
Cindy Eisner
Ali El-Zein
Oded Fuhrmann
Eric Gascard
Amit Goel
Karin Greimel
Andreas Griesmayer
Anubhav Gupta
Christine Hang
Ziyad Hanna
Georg Hofferek
Warren Hunt
Joe Hurd
Geert Janssen
Barbara Jobstmann
Toni Jussila
Robert Kanzelman
Nathan Kitchen
Udo Krautz
Robert Krug
Bing Li
Jiang Long
Florian Lonsing
Alexander Malkis
John Matthews
Maher Mneimneh
In-Ho Moon
Hari Mony
Katell Morin-Allory
Ziv Nevo
Sergio Nocco
Vinit Ogale
Viresh Paruthi
Edgar Pek
Andrea Pellegrini
James Peterson
Laurence Pierre
Stephen Plaza
Mitra Purandare
Stefano Quer
Marco Roveri
Sitvanit Ruah
Roopsha Samanta
Jun Sawada
Viktor Schuppan
Roberto Sebastiani
Luciano Serafini
Sebastian Skalberg
Christian Stangier
Sol Swords
Stefano Tonetta
Aliaksei Tsitovich
Aaron Turon
Tatyana Veksler
Ilya Wagner
Thomas Wahl
Sean Weaver
Markus Wedler
Georg Weissenbacher
Tobias Welp
Peter White
Christoph Wintersteiger

Invited Tutorial: Considerations in the Design and Verification of Microprocessors for Safety-Critical and Security-Critical Applications

David S. Hardin
Advanced Technology Center
Rockwell Collins, Inc.
Cedar Rapids, IA USA
dshardin@rockwellcollins.com

Abstract—In this tutorial, we will examine issues in the design and verification of microprocessors for safety-critical and security-critical applications. We will consider architectural and design alternatives to support high-assurance applications, and will describe techniques to improve secure system evaluation -- measured in terms of completeness, human effort required, time, and cost -- through the use of highly automated formal methods. We will describe practical techniques for creating executable formal computing platform models that can both be proved correct, and also function as high-speed simulators. This allows us to both verify the correctness of the models, as well as validate that the formalizations accurately model what was actually designed and built. As a case study, we will examine the design and verification of the Rockwell Collins AAMP7G microprocessor. The AAMP7G, currently in use in Rockwell Collins high-assurance system products, supports strict time and space partitioning in hardware, and has received an NSA MILS (Multiple Independent Levels of Security) certificate based in part on proofs of correctness. We will discuss the AAMP7G verification effort, focusing on the proof architecture that enabled us to show that the AAMP7G separation kernel microcode implements a particular security specification, using the ACL2 theorem prover.

Keywords-Microprocessor verification, theorem proving, partitioning, ACL2.

I. INTRODUCTION

Security-critical applications at the highest Evaluation Assurance Level (EAL) require formal proofs of correctness in order to achieve certification [2]. At the highest EAL, EAL 7, the application must be formally specified, and the application must be proven to implement its specification. This can be a very expensive and time-consuming process. One of the main research goals of the Automated Analysis group at Rockwell Collins is to improve secure system evaluation -- measured in terms of completeness, human effort required, time, and cost -- through the use of highly automated formal methods. In support of this goal, we have developed practical techniques for creating executable formal computing platform models that can both be proved correct, and also function as high-speed simulators [3], [8]. This allows us to both verify the correctness of the models, as well as validate that the formalizations accurately model what was actually designed and built.

The AAMP7G microprocessor [17], currently in use in Rockwell Collins secure system products, supports strict time and space partitioning in hardware, and has received an NSA MILS certificate based in part of a formal proof of correctness of its separation kernel microcode [18]. In this paper, we will present an overview of the AAMP7G verification, built upon an executable formal model of the AAMP7G microcode written in ACL2 [10].

We will begin by describing the formally verified partitioning features of the AAMP7G.

II. THE AAMP7G

The AAMP7G is the latest in the line of Collins Adaptive Processing System (CAPS) processors and AAMP microprocessors developed by Rockwell Collins for use in military and civil avionics since the early 1970s [1]. Over the years, AAMP designs have been tailored to embedded avionics products requirements, accruing size, weight, power, cost, and specialized feature advantage over alternate solutions. Each new AAMP makes use of the same multi-tasking stack-based instruction set, while adding state of the art technology in the design of each new CPU and peripheral set. AAMP7G adds built-in partitioning technology among other improvements.

AAMP processors feature a stack-based architecture with 32-bit segmented, as well as linear, addressing. AAMP supports 16/32-bit integer and fractional, as well as 32/48-bit floating point operands. The lack of user-visible registers improves code density (many instructions are a single byte), which is significant in embedded applications where code typically executes directly from slow Read-Only Memory. AAMP provides a unified call and operand stack, and the architecture defines both user and executive modes, with separate stacks for each user "thread", as well as a separate stack for executive mode operation. The transition from user to executive mode occurs via traps; these traps may be

programmed, or may occur as the result of erroneous execution (illegal instruction, stack overflow, etc.). The AAMP architecture also provides for exception handlers that are automatically invoked in the context of the current stack for certain computational errors (divide by zero, arithmetic overflow). The AAMP instruction set is of the CISC variety, with over 200 instructions, supporting a rich set of memory data types and addressing modes.

A. AAMP7G Intrinsic Partitioning

The AAMP7G provides a feature called "Intrinsic Partitioning" that allows it to host several safety-critical or security-critical applications on the same CPU. The transition from multiple CPUs to a single multi-function CPU is shown in

Figure 1. On the left, three federated processors provide three separate functions, A, B, and C. It is straightforward to show that these three functions have no unintended interaction.

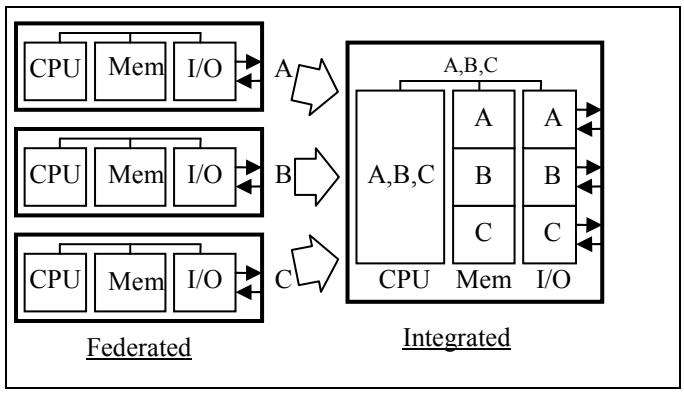


Figure 1. Transition to Multi-Function CPU.

On the right of

Figure 1, an integrated processor provides for all three functions. The processor performs code from A, B, and C; its memory contains all data and I/O for A, B, and C. A partition is a container for each function on a multi-function partitioned CPU like the AAMP7G. AAMP7G follows two rules to ensure partition independence:

1. TIME PARTITIONING. Each partition must be allowed sufficient time to execute the intended function.
2. SPACE PARTITIONING. Each partition must have exclusive-use space for storage.

1) Time Partitioning

Each partition must be allowed sufficient time to execute the intended function. The AAMP7G uses strict time partitioning to ensure this requirement. Each partition is allotted certain time-slices during which time the active function has exclusive use and control of the CPU and related hardware.

For the most secure systems, time slices are allocated at system design time and not allowed to change. For dynamic reconfiguration, a "privileged" partition may be allowed to set time slices. AAMP7G supports both of these cases determined by the system designer.

The asynchronous nature of interrupts poses interesting challenges for time partitioned systems. AAMP7G has partition-aware interrupt capture logic. Each interrupt is assigned to a partition; the interrupt is only recognized during its partition's time slice. Of course, multiple interrupts may be assigned to a partition. In addition, an interrupt may be shared by more than one partition if needed.

System-wide interrupts, like power loss eminent or tamper detect, also need to be addressed in a partitioned processor. In these cases, AAMP7G will suspend current execution, abandon the current list of partition control, and start up a list of partition interrupt handlers. Each partition's interrupt handler will run performing finalization or zeroization as required by the application.

2) Space Partitioning

Each partition must have exclusive-use space for storage. The AAMP7G uses memory management to enforce space partitioning. Each partition is assigned areas in memory that it may access. Each data and code transfer for that partition is checked to see if the address of the transfer is legal for the current partition. If the transfer is legal, it is allowed to complete posthaste. If the transfer is not legal, the AAMP7G Partition Management Unit (PMU) disallows the CPU from seeing read data or code fetch data; the PMU also preempts write control to the addressed memory device.

Memory address ranges may overlap, in order to enable inter-partition communication. In this case, interaction between partitions is allowed since it is intended by system design. For maximum partition independence, overlapping access ranges should be kept to a minimum.

As with time slices, memory ranges may be allocated at system design time and not allowed to change. Or, for dynamic reconfiguration, a "privileged" partition may be allowed to set memory ranges. AAMP7G supports both of these cases determined by the system designer.

B. Partition Control

Only a small amount of data space is needed for partition control structures. The data space is typically not intended to be included in any partition's memory access ranges. Each partition's control includes time allotment, memory space rights, initial state, and test access key stored in ROM. Each partition's saved state is stored in RAM. Partition control blocks are linked together defining a partition activation schedule. AAMP7G partition initialization and partition switching are defined entirely by these structures and performed entirely in microcode. Thus, no software access is needed for AAMP7G partitioning structures. This limits verification of AAMP7G partitioning to proving that the partitioning microcode performs the expected function and no other microcode accesses the partitioning structures.

III. AAMP7G Formal Processing Model

The AAMP7G formal processing model is shown in **Figure 2**. Actual AAMP7G processing layers are shown in non-italic text, while layers introduced for the sake of formal reasoning are shown in italics.

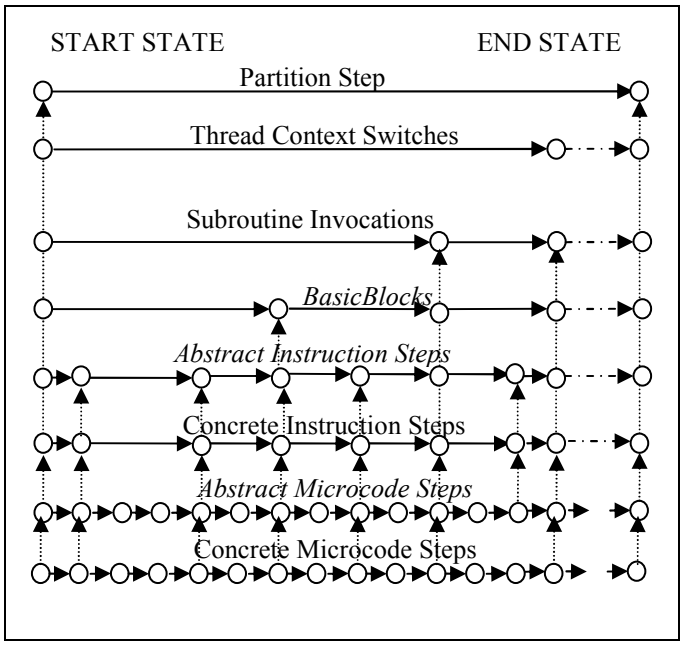


Figure 2. AAMP7G formal processing model.

We generally prove correspondence between a concrete model at a given level and a more abstract model. Sequences of microcode implement a given instruction; sequences of abstract instruction steps form basic blocks; and a machine code subroutine is made up of a collection of basic blocks. Subroutine invocations are performed in the context of an AAMP thread, and multiple user threads plus the executive mode constitute an AAMP7G partition. Our model supports the entire context switching machinery defined by the AAMP architecture, including traps, outer procedure returns, executive mode error handlers, and so on.

IV. AAMP7G Partitioning System Proofs

Rockwell Collins has performed a formal verification of the AAMP7G partitioning system using the ACL2 theorem prover [10]. This work was part of an evaluation effort which led the AAMP7G to receive a MILS Certificate from NSA, enabling a single AAMP7G to concurrently process Unclassified through Top Secret codeword information. We first established a formal security specification, as described in [6], and summarized in Section IV. We produced an abstract model of the AAMP7G's partitioning system, as well as a low-level model that directly corresponded to the AAMP7G microcode. We used ACL2 to automatically produce the following:

1. Proofs validating the security model
2. Proof that the abstract model enforces the security specification
3. Proof that the low-level model corresponds to the abstract model.

The use of ACL2 to meet high-assurance Common Criteria requirements is discussed in [14]. One interpretation of the requirement for a low-level design model is that the low-level design model be sufficiently detailed and concrete so that an implementation can be derived from it with no further design decisions. Because there are no design decisions remaining, one can easily validate the model against the implementation. Note that this low level of abstraction of a model can make a proof about it challenging.

V. A Formal Security Specification

High-assurance product evaluation requires precise, unambiguous specifications. For high-assurance products that are relied upon to process information containing military or commercial secrets, it is important to guarantee hat no unauthorized interference or eavesdropping can occur. A formal specification of what the system allows and guards against is called a formal security specification. The construction of a formal security specification that describes the needed behavior of a security-critical system under evaluation is now commonly required for high-level certification.

A computing system that supports multiple independent levels of security (MILS) provides protections to guarantee that information that is assigned different security levels is handled appropriately. The design of MILS systems guaranteed to perform correctly with respect to security considerations is a daunting challenge.

The goal in building a partitioning mechanism is to limit what must be evaluated in a verification or certification context. For example, secure systems can be developed that use partitions to enforce separation between processes at different security levels. A small, trusted "separation kernel" mediates all communication between partitions thereby assuring that nonauthorized communication does not occur. Assuming that the partitioning system is implemented properly and that the communication policy between partitions is loaded correctly, there is no need to evaluate the applications running in different partitions to show that the communication policy is enforced. Safety-critical applications can also exploit intrinsic partitioning: by hosting different applications in separate partitions it is possible to architect a system so that applications need be evaluated at only the needed level of rigor. This system architecting philosophy is described by John Rushby in [15], [16].

The correct implementation of the partitioning mechanism is of course vital to assure the correctness of a larger system that depends upon it. Furthermore, some of the initial applications of the AAMP7G are security applications that are architected to exploit intrinsic partitioning and require stringent evaluation of all mechanisms being relied upon to separate data at different classification levels.

Architecting a MILS system using a separation kernel breaks the security challenge into two smaller challenges:

1. Building and verifying a dependable separation kernel; and
2. Building applications that, relying upon protections afforded by the separation kernel, enforce sensible system security policies.

Broadly speaking, a good specification of a system component has two characteristics. First, it can be mapped to concrete component implementations using convenient and reliable methods. That is, the specification can be proved about a particular system component. Second, a good specification encapsulates needed behavior so that the larger system can benefit from an assurance that the specification holds of the component. That is, the specification can be used in the larger system that contains the component about which the specification has been proved.

The sections that follow describe a security property that has both of these desired properties, and which has been proved as part of the AAMP7G MILS certification.

We have choosen the ACL2 logic, an enhancement of the Common Lisp programming language [10], to describe our security specification. ACL2 is a good choice for this work because of its usefulness in modeling and reasoning about computing systems [3], [8] and the support afforded by the ACL2 theorem proving system.

A. The Formal Security Specification in ACL2

The formal security specification describes abstractly what a separation kernel does. The machine being modeled supports a number of partitions whose names are provided by the constant function allparts. We use the notation of ACL2's encapsulate command to indicate a function of no arguments that returns a single value.

```
((allparts) => *)
```

One of the partitions is designated the “current” partition. The function current calculates the current partition given a machine state.

```
((current *) => *)
```

We use the notation of ACL2's defthm command, which presents a theorem expressed in Common Lisp notation, to indicate a property about the functions current and allparts.

```
(defthm current-is-partition
  (member (current st) (allparts)))
```

Associated with partitions are memory segments. Memory segments have names and are intended to model portions of the machine state. The names of the memory segments associated with a particular partition are available from the function segs, which takes as an argument the name of the partition. (Note that since segs is a function only of partition name, not, for example, a function of machine state, the assignment of segments to partitions is implicitly invariant.)

```
((segs *) => *)
```

The values in a machine state that are associated with a memory segment are extracted by the function select. select takes two arguments: the name of the memory segment and the machine state.

```
((select * *) => *)
```

The separation kernel enforces a communication policy on the memory segments. This policy is modeled with the function dia, which represents the pairs of memory segments for which direct interaction is allowed. The function takes as an argument a memory segment name and returns a list of memory segments that are allowed to affect it. (Note that since dia is a function only of the memory segment name, the formalization here implicitly requires that the communication policy is invariant.)

```
((dia *) => *)
```

The last function constrained in the security specification is next, which models one step of computation of the machine state. The function next takes as an argument a machine state and returns a machine state that represents the effect of the single step.

```
((next *) => *)
```

The aforementioned constrained functions are used to construct several additional functions. selectlist takes a list of segments and returns a list of segment values; segslist takes a list of partition names and returns the list of memory segment associated with the partitions; and run takes an initial machine state and number of steps and returns an initial machine state updated by executing the number of steps indicated.

```
(defun selectlist (segs st)
  (if (consp segs)
    (cons
      (select (car segs) st)
      (selectlist (cdr segs) st)) nil))

(defun segslist (partnamelist)
  (if (consp partnamelist)
    (append
      (segs (car partnamelist))
      (segslist (cdr partnamelist))) nil))

(defun run (st n)
  (if (zp n)
    st
    (run (next st) (1- n))))
```

The formal security specification theorem, now referred to as "GWV" after its authors, is shown in **Figure 3**. The GWV theorem utilizes a "two worlds" formulation, in which two arbitrary states of the system (in this case, st1 and st2) are hypothesized to both satisfy some predicate(s). The two states are both stepped, and some predicate is shown to hold on the two resulting states. In particular, the GWV theorem says that the effect of a single step on the system state on an arbitrary segment of the state, seg, is a function only of the segments associated with the current partition that are allowed to interact with seg.

```
(let ((segs (intersection-equal (dia seg)
                                (segs (current st1)))))
  (implies
    (and
      (equal (selectlist segs st1) (selectlist segs st2))
      (equal (current st1) (current st2))
      (equal (select seg st1) (select seg st2)))
    (equal
      (select seg (next st1))
      (select seg (next st2)))))
```

Figure 3. The formal security specification.

This is the entirety of the separation kernel security specification.

B. Utility of the Formal Security Specification

We have established the utility of this formal security specification by stating the policy as an axiom, and then attempting to prove several well-known security-related theorems using the axiom. In this way, we have proved exfiltration, infiltration, and mediation theorems, as well as proved the functional correctness of a simple firewall, all using the ACL2 system. Subsequently, we have shown that GWV is an equivalent formulation to classical noninterference [7].

C. Proof that the AAMP7G implements the Formal Security Specification

We constructed a low-level design model of the partitioning-relevant operation of AAMP7G. That model consists of approximately 3000 lines of ACL2 code. A crucial consideration is how to validate this hand-written model against the actual AAMP7G. The AAMP7G is a microcoded microprocessor, and much of the functionality of the machine is encoded in its microcode.

"Trusted" microcode is microcode that operates with memory protection turned off, thereby providing access to the data structures maintained by the AAMP7G to support intrinsic partitioning.

All the partitioning-relevant microcode runs in this trusted mode, and the low-level design model of the AAMP7G models all the microcode that implements this functionality.

The next step in the process was to conduct a code-to-spec review with a National Security Agency evaluation team. This review validated the formal model against the actual AAMP7G. The documentation package that was created for this review included:

- material explaining the semantics of ACL2 and AAMP7G microcode;
- listings of the AAMP7G microcode and the ACL2 low-level model;
- the source code listing of a tool that identifies trusted-mode microcode sequences, and a listing of such sequences in the AAMP7G microcode;
- cross-references between microcode line numbers, addresses, and formal model line numbers; and
- the ACL2-checkable proofs in electronic form.

The low-level design model was written specifically to make this code-to-spec review relatively straightforward. An ACL2 macro allows an imperative-style description that eases comparison with microcode. Also, very importantly, the model is written with the model of memory that the microcode programmer uses when writing microcode. That is, memory has only two operations: read and write. The simplicity of the memory model makes the code-to-spec review easier but adds a great deal of complexity to the proof. Since the proof is machine-checked while the model validation process requires evaluation, this is a good tradeoff. It provides a high level of

assurance with a reasonable level of evaluation. Nearly all the time on the project was spent constructing the proofs, but they were evaluated relatively easily by the evaluators because they could be replayed using ACL2. Most of the evaluation time was spent on the code-to-spec review.

Figure 4 presents a fragment of the low-level design model.

```
;=== ADDR: 052F
(st. ie = nil)
(Tx = (read32 (vce_reg st) (VCE.VM_Number)))
;=== ADDR: 0530
(st. Partition = Tx)
;=== ADDR: 0531
(TimeCount = (read32 (vce_reg st) (VCE.TimeCount)))
;=== ADDR: 0532
(PSL[0]= TimeCount st)
```

Figure 4. A fragment of the AAMP7G formal microcode-level model.

Much as large software implementations require an architecture, so too do large proof efforts.

Figure 5 shows the final theorem proved about the AAMP7G. Note that this is an instance of the separation theorem described in **Figure 3**, with a few differences. First, functions that describe the communication policy ("dia") and the segments associated with a particular partition ("segs") are functions of segment name and processor state rather than just segment name. This allows them to "pull" the configuration information out of the processor state. We prove this is appropriate by proving that these functions are invariant with respect to a step of the low-level design model ("next"). Second, we add some assumptions about a secure initial state of the AAMP7G. These assumptions guarantee that the AAMP7G's state is reasonable – that the data structures have a reasonable shape, that different data structures do not overlap, etc.

The proof architecture breaks the proof into three main pieces:

1. Proofs validating the correctness theorem (as described in [4])
2. Proof that the abstract model meets the security specification
3. Proof that the low-level model corresponds with the abstract model

In addition to libraries provided in the standard ACL2 release, several libraries of ACL2 lemmas were developed for this project. As previously indicated, an important challenge of this project was developing a method for reasoning about read and write operations on a linear address space. The approach, termed GACC for Generalized Accessor, is outlined in [5]. GACC provides a systematic approach for describing data structures and a template for proving a few helpful facts about each operation.

```
(implies
 (and
  (secure-configuration spex)
  (spex-hyp :any :trusted :raw spex fun::st1)
  (spex-hyp :any :trusted :raw spex fun::st2))
 (implies
   (let ((abs::st1 (lift-raw spex fun::st1))
         (abs::st2 (lift-raw spex fun::st2)))
     (and
      (let ((segs (intersection-equal
                    (dia-fs seg abs::st1)
                    (segs-fs (current abs::st1) abs::st1))))
        (equal (raw-selectlist segs abs::st1)
               (raw-selectlist segs abs::st2)))
        (equal (current abs::st1)(current abs::st2))
        (equal (raw-select seg abs::st1)
               (raw-select seg abs::st2))))
   (equal
    (raw-select seg (lift-raw spex (fun::next spex fun::st1)))
    (raw-select seg (lift-raw spex (fun::next spex fun::st2))))))
```

Figure 5. Theorem that the AAMP7G implements the security specification.

VI. AAMP7G INSTRUCTION SET MODEL

Having established the correctness of the AAMP7G's partitioning system, we next wished to provide a formal model of the instruction set processing that occurs within a partition's time slice. Having such a model would enable us to perform machine code proofs of correctness that could be used in high-assurance evaluations. The instruction set model and all the necessary support books consists of some 100,000 lines of ACL2 code [9]. The AAMP7G instruction set model is shown in the architectural context in **Figure 2**. We begin with a concrete instruction model, written in a sequential manner that is reflective of how the machine actually operates. The AAMP memory model is based on the linear address space book previously used in the AAMP7G partitioning proofs [5]. The

AAMP7G machine state, including the architecturally-defined registers, is represented as an ACL2 single-threaded object (stobj) [11] for performance reasons.

A. Model Validation

Since we model the AAMP7G instruction set in its entirety, we can analyze AAMP7G machine code from any source, including compilers and assemblers. Additionally, since we directly model memory, we merely translate the binary file for a given AAMP7G machine code program into a list of (address, data) pairs that can be loaded into ACL2. We load the code, reset the model, and the execution of the machine code then proceeds, under the control of an Eclipse-based user interface that was originally written to control the actual AAMP7G.

We then validate the AAMP7G instruction set model by executing instruction set diagnostics on the model that are used for AAMP processor acceptance testing. A typical diagnostic exercises each instruction, plus context switching, exception handling, etc.

B. Abstract Instruction Set Modeling and Symbolic Simulation

For a given AAMP instruction X, The "abstract" function OP-X-PRECONDITIONS collects those conditions that need to hold for "normal" execution of the instruction. VM-X-EXPECTED-RESULT gives the expected output state as a modification of the input state. As this function is an abstraction of the concrete instruction set execution, it does not characterize the *steps* of the computation, but rather the *result* in a very compact and readable form.

Finally, proving the theorem VM-X-REWRITE establishes that stepping the AAMP instruction set model on this instruction will yield exactly the expected result as the abstract formulation, assuming the preconditions are satisfied. If the instruction semantics involves interesting exceptions, such as overflow or divide-by-zero, these are characterized by additional expected result functions and additional branches in the right hand side of this rewrite rule.

We have provided similar treatment for each of the AAMP7G instructions. Assuming that we can relieve the preconditions at each step, this allows us efficiently to symbolically step through even very long sequences of AAMP7G instructions. After each step, the rewriter effectively canonicalizes the result into a very compact and readable form. This is practical within the context of a theorem prover in as much as the ACL2 rewriter can be constrained to be quite fast.

C. Compositional Code Proof

Our verification of AAMP7G programs is done compositionally. That is, we verify programs one subroutine at a time. We try to ensure that, after we verify a subroutine, we never have to analyze it again. Thus, the correctness theorem (or theorems) for a routine R must be strong enough to support the verification of any routine that calls R, without the need to analyze R again.

Before we verify a subroutine R, we must verify all of the routines that call R. Thus, the order in which we verify a program's subroutines is a topological order on its call graph.

To prove the correctness theorem for a subroutine we use a proof methodology called "compositional cutpoints". Our method borrows parts of the method put forth in [13]; both methods are inspired by observations first made by Moore [12].

Cutpoint proofs require annotating the subroutine to be verified by placing assertions at some of its program locations; those locations are called "cutpoints". Every cutpoint has a corresponding assertion which is taken to apply to those states that arise just before the instruction at program location k is executed.

The resulting full set of cutpoints is sufficient if it "cuts every loop," that is, if every cycle in the routine's control flow graph contains a cutpoint. We need not consider cycles in the code of called subroutines; any subroutine call should either be a call to an already-verified routine, or be a recursive call (which we handle specially).

For code emitted by a compiler for an imperative language with standard looping constructs like "for" and "while," it is usually sufficient to put a cutpoint at the continuation test of each loop, even in the presence of break and continue statements. Sometimes we can do even better. For example, a single cutpoint sometimes suffices to verify a program with two nested loops.

A routine with no loops typically requires no user cutpoints; that is, the starting program location usually sufficies as the only cutpoint. This is true even if the routine contains branches or subroutine calls, including recursive calls.

If the set of cutpoints for a routine is insufficient to cut all the loops in its control flow graph, the symbolic simulation described below may loop forever -- in which case the proof will fail.

The "cutpoint to cutpoint" proof for a routine involves symbolic simulation of the machine model. The simulation starts at a cutpoint and assumes that the assertion for that cutpoint holds. We simulate the machine until it either reaches another cutpoint or exits by executing a return instruction. At the resulting state, we must show that the corresponding assertion holds. Recall that each cutpoint has a corresponding assertion. The corresponding assertion for a state in which the routine has just exited is the routine's postcondition. Thus, functional proofs of correctness of AAMP7G machine code can proceed largely automatically once the necessary cutpoint assertions have been introduced.

VII. Conclusion

We have examined issues in the design and verification of microprocessors for safety-critical and security-critical applications by way of a case study of the Rockwell Collins AAMP7G microprocessor. We described the formal model architecture of the AAMP7G at several levels, including the microcode and instruction set levels. We introduced a formal security specification theorem, and described an effort that resulted in a mathematical proof that the AAMP7G trusted

microcode implemented that security specification. This proof was part of a MILS certification of the processor, enabling a single AAMP7G to concurrently process Unclassified through Top Secret codeword information. Finally, we discussed a technique for compositional reasoning at the instruction set level, using a technique called "compositional cutpoints" based on symbolic simulation.

ACKNOWLEDGMENT

Many thanks to the AAMP7G development team at Rockwell Collins, and especially to Matt Wilding, Dave Greve, and Ray Richards for their pioneering work on the AAMP7G partitioning microcode proofs. Thanks also go to the ACL2 development team at the University of Texas at Austin.

REFERENCES

[1] D. Best, C. Kress, N. Mykris, J. Russell, and W. Smith, An advanced-architecture CMOS/SOS microprocessor. *IEEE Micro*, Aug. 1982, 11-26.

[2] Common Criteria for Information Technology Security Evaluation (CCITSE), March 1999. Available at http://www.radium.ncsc.mil/tpep/library/ccitse/ccitse.html.

[3] D. Greve, M. Wilding, and D. Hardin: High-speed, analyzable simulators. In M. Kaufmann, P. Manolios, and J S. Moore, eds.: *Computer-Aided Reasoning: ACL2 Case Studies*, Kluwer Academic Publishers, 2000, 89–106.

[4] D. Greve, M. Wilding, and M. Vanfleet, A separation kernel formal security policy. In Proceedings of ACL2'03, 2003.

[5] D. Greve: Address enumeration and reasoning over linear address spaces. In Proceedings of ACL2'04, Austin, TX, Nov. 2004.

[6] D. Greve, R. Richards, and M. Wilding: A summary of intrinsic partitioning verification. In Proceedings of ACL2'04, Austin, TX, Nov. 2004.

[7] D. A. Greve: personal communication. April 2008.

[8] D. Hardin, M. Wilding, and D. Greve, Transforming the theorem prover into a digital design tool: from concept car to off-road vehicle. In Hu, A. and Vardi, M., eds.: CAV'98. Volume 1427 of LNCS, Springer-Verlag, 1998, 39-44.

[9] D. Hardin, E. Smith, and W. Young: A robust machine code proof framework for highly secure applications. In Proceedings of ACL2'06, Seattle, WA, Aug. 2006.

[10] M. Kaufmann, P. Manolios, and J S. Moore, *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, 2000.

[11] J S. Moore and R. Boyer: Single-threaded objects in ACL2. In Proceedings of PADL 2002, Volume 2257 of LNCS, Springer-Verlag, 2002, 9-27.

[12] J S. Moore, Inductive assertions and operational semantics. In D. Geist, ed.: CHARME 2003. Volume 2860 of LNCS, Springer-Verlag, 2003, 289–303.

[13] J. Matthews, J S. Moore, S. Ray, and D. Vroon: Verification Condition Generation via Theorem Proving. In Proceedings of LPAR'06, volume 4246 of LNCS, 2006, 362-376.

[14] R. Richards, D. Greve, M. Wilding, and M. Vanfleet, The Common Criteria, formal methods, and ACL2. In Proceedings of ACL2'04, Austin, TX, Nov. 2004.

[15] J. Rushby: Design and verification of secure systems. In Proceedings of the Eighth Symposium on Operating Systems Principles, volume 15, December 1981.

[16] J. Rushby: Partitioning for safety and security: requirements, mechanisms, and assurance, NASA contractor report CR-1999-209347, 1999.

[17] Rockwell Collins, Inc.: *AAMP7r1 Reference Manual*, 2003.

[18] Rockwell Collins, Inc.: Rockwell Collins receives MILS certification from NSA on microprocessor, Rockwell Collins press release, 24 August 2005, http://www.rockwellcollins.com/news/page6237.html.

Invariant-Strengthened Elimination of Dependent State Elements

Michael L. Case[1,2] Alan Mishchenko[1] Robert K. Brayton[1] Jason Baumgartner[2] Hari Mony[2]
[1] Department of EECS, University of California at Berkeley, CA
[2] IBM Systems and Technology Group, Austin, TX

***Abstract*— This work presents a technology-independent synthesis optimization that is effective in reducing the total number of state elements of a design. It works by identifying and eliminating *dependent state elements* which may be expressed as functions of other registers. For scalability, we rely exclusively on SAT-based analysis in this process. To enable optimal identification of all dependent state elements, we integrate an inductive invariant generation framework. We introduce numerous techniques to heuristically enhance the reduction potential of our method, and experiments confirm that our approach is scalable and is able to reduce state element count by 12% on average in large industrial designs, even after other aggressive optimizations such as min-register retiming have been applied. The method is effective in simplifying later verification efforts.**

I. Introduction

Logic synthesis and formal verification are closely related fields. Verification tools often rely on technology-independent synthesis optimizations to reduce the size of the design being verified, thereby enhancing the scalability of the verification process. In this work we present a technology-independent synthesis optimization that can reduce the number of state elements in a design, thereby enhancing the scalability of verification tools. The focus of this paper is on synthesis for verification.

A *dependent state element* is one which may be expressed as a function over other state elements in the design. Once identified, the overall state element count of the design may be reduced by replacing the dependent elements by the corresponding functions over the remaining elements. Many verification algorithms are highly sensitive to the number of state elements present in the design. For example, BDD-based reachability analysis generally requires exponential resources with respect to state element count, hence it may dramatically benefit from the elimination of dependent state elements [1]. The effectiveness of induction is also generally sensitive to the implementation of the logic, particularly in the presence of unreachable states [2]. Dependent state element elimination inherently reduces the fraction of unreachable states of a design, thereby enhancing inductiveness.

Dependency is traditionally identified using BDD-based algorithms (e.g., [1]), which practically limits its application to smaller designs or requires approximate compositional analysis, resulting in suboptimalities. Recently, it has been demonstrated that *combinational dependency* of next-state functions may be identified using purely SAT-based analysis [3], enabling the analysis to scale to much larger designs. However, the previous research lacks several elements that make this method effective in practice. In this paper, we address the topic of sequential dependent state variable elimination using SAT-based techniques. Our contributions are as follows.

- To enhance state element reduction capability, we formulate the dependency check in a way that allows our formulation to directly reduce the total number of state elements, discussed in Section IV-A.
- Because the dependent state element elimination process provides network simplifications that may not be *compatible*, we introduce a technique to heuristically arrive at a legal netlist of minimal size by finding a high-quality compatible subset of simplifications in Section IV-C.
- SAT-based dependent state element elimination can introduce logic bloat in the design, and to mitigate this we introduce a way to leverage flexibilities in the dependency check to enable heuristically greater combinational logic reductions (Section IV-D).
- To enable the identification of state elements which are dependent in the reachable states though not necessarily in arbitrary states, we integrate an unreachability invariant framework to approximate unreachable state *don't cares*. To enable a purely SAT-based method, our invariant generation framework uses random simulation and k-step induction to derive *inductive invariants* in a format suitable for efficient SAT-based dependency computation. This will be discussed in Sections IV-B and V. While the invariants here are derived to benefit dependent state state element elimination, they have application in many other verification contexts as well.

The algorithms described in this work can be partitioned into two main components: 1) dependent state element identification and 2) invariant generation. A synthesis loop alternates between these two components along with combinational synthesis until no further design reductions are possible. The technique has been shown to provide significant reductions even after a design has first been heavily synthesized. The number of registers in a design can be reduced by 12% on average, even after optimizations such as min-register retiming have been applied. The dependent state elements in our experience are re-expressed as complex functions over the remaining elements, indicating that these dependencies cannot be found by simpler techniques such as register correspondence.

II. Background and Related Work

In this section we review fundamental synthesis concepts and terminology used throughout this paper. We also provide

978-1-4244-2735-2/08/$25.00 ©2008 IEEE

a review of prior SAT-based resubstitution research.

A. Boolean Networks and And-Inverter Graphs

A Boolean Network (BN) is a directed acyclic graph (DAG) composed of a set of input signals and Boolean functions [4]. Each Boolean function in the BN has an *on-set* (*off-set*) which is the set of valuations of the functions inputs that cause the function to evaluate to 1 (0). By the definition of a function, the on and off-sets must be disjoint, but there may be valuations of the inputs for which the value of the function is undefined. These input valuations are not contained in either the on- or off-sets and represent *don't care* conditions against which an implementation of the function may be optimized. In synthesis, such don't cares arise for several reasons, and two such reasons that are important in this work are: 1) There may be *sequential don't cares* in the form of *unreachable states* against which combinational logic functions may be minimized, and 2) There may be *satisfiability don't cares* where the sub-BN's that drive the function inputs may be incapable of generating certain input patterns.

In this work we operate on a particular type of BN referred to as an *And/Inverter Graph (AIG)*. An AIG is a DAG where each node is either a 2-input AND gate or an input to the AIG. Inverters may be present and are indicated with a complementation attribute on the edges in the DAG. A *sequential AIG* is an AIG that also contains state elements, hereafter referred to as *registers*. The register r is the only state-holding AIG primitive, which has an associated *initial state* $init(r)$, defining its time-0 behavior, as well as a *next-state function* $next(r)$ which defines the value of r at the following time-frame.

B. SAT-Based Resubstitution

In logic synthesis, *resubstitution* refers to a process that recasts a Boolean expression as a function over other pre-existing Boolean expressions [4]. For example, suppose there are Boolean signals X, g_1, g_2, ..., g_n. Resubstitution may be used to build a function $F(\cdot)$ such that $X = F(g_1, g_2, \ldots, g_n)$, or to prove that no such function exists. The functions g_1, g_2, ..., g_n are referred to as the *basis* and $F(\cdot)$ as the *dependency function*. Upon finding F, the old implementation of X can be removed and replaced with with the new implementation of F. Often resubstitution yields a reduction in the size of the AIG, and for this reason has been the focus of much synthesis research.

Traditionally, resubstitution is performed using Binary Decision Diagrams (BDDs) [1]. While efficient for small functions, the scalability of BDD-based techniques is often limited to modestly sized functions and bases, preventing optimal dependency identification in larger netlists. Recently it has been demonstrated that resubstitution may be cast as a Boolean Satisfiability (SAT) problem [3]. The formulation builds a combinational test circuit and uses SAT to determine if the circuit's single output is satisfiable. If the answer is `unsat` then the dependency function $F(\cdot)$ exists and can be extracted from the proof of unsatisfiability through interpolation. This method offers substantially greater scalability than possible with BDD-based analysis.

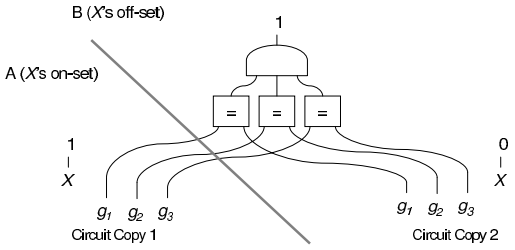


Fig. 1. SAT-based dependency formulation [3]

Suppose that we wish to express X as a function over signals $g_1, \ldots, g_3$. This is possible if and only if for each valuation to signals $g_1, \ldots, g_3$ there is a single possible X valuation. We can test if such a resubstitution exists using the circuit shown in Figure 1. Two separate copies of X, g_1, g_2, and g_3 are instantiated. Each pair of g's is constrained to have the same value, and the pair of X's is constrained to have differing values. A resubstitution exists if and only if this test circuit is unsatisfiable. Many SAT solvers can be configured to record a proof of unsatisfiability [5], and interpolation on this proof provides the dependency function.

Given Boolean formulas $A(x, y)$ and $B(y, z)$, if $A(x, y) \cdot B(y, z) = 0$ then there exists a *Craig Interpolant* [6] I such that I refers to only the common variables y of A and B, and $A \implies I \implies \overline{B}$. [7] provides an algorithm to extract the interpolant I from the proof of unsatisfiability of $A \cdot B$. This technique was first introduced to the verification community in a SAT-based unbounded verification algorithm [8] where the interpolant represents an overapproximate image computation.

In the resubstitution context, interpolation will be used to derive a dependency function. We may partition the resubstitution test circuit in Figure 1 into two halves A and B. A represents the set of g's where $X = 1$, the on-set of X. Similarly, B represents the off-set of X. Because $A \implies I \implies \overline{B}$, I is a function that lies in between the on and off-sets of X, and we can replace X with I. Furthermore, I only refers to the common variables between A and B, namely the g's, hence I provides the dependency function.

While this SAT-based formulation of [3] enables substantially greater scalability than BDD-based techniques, this formulation is limited in four key ways:

1) The prior research can only re-express combinational logic and cannot directly eliminate registers. In this work we simplify verification problems, and reducing the number of registers is important to increasing the scalability of many verification algorithms. Section IV-A will discuss how we directly eliminate registers.
2) The prior research cannot identify dependencies which hold in all reachable states but not in arbitrary unreachable states. In our experience, the reduction potential of this combinational analysis is often a subset of that possible using min-register retiming with integrated resynthesis [9]. We use invariants to overcome this limitation, as discussed in Sections IV-B and V.
3) The prior research does not address incompatibilities

present in the set of found dependencies. Often, dependencies must be discarded to avoid creating combinational cycles in the logic. We discuss an efficient way to compute a compatible set of dependencies in Section IV-C.

4) The prior research does not address the logic bloat that may result from interpolation. In general, logic generated by interpolation is highly redundant. We discuss a method to mitigate this logic bloat in Section IV-D.

III. Dependent Register Elimination Algorithm

Our overall dependent register elimination routine is illustrated in Algorithm 1. The method used to eliminate dependent registers will be detailed in Section IV, and the method used to compute invariants will be discussed in Section V. At the top level, these two methods are iterated along with combinational synthesis (e.g., [10]) until design size is no longer reduced.

Algorithm 1 Dependent register elimination

```
 1: function sequentialResynthesize(design)
 2:     invariants := ∅
 3:     repeat
 4:         invariants += gatherInvariants(design, invariants)
 5:         design := eliminateRegisters(design, invariants)
 6:         design := combinationalSynthesis(design)
 7:     until (No change in design size)
 8:     return design
 9: end function
10:
11: function gatherInvariants(design, invariants)
12:     while (Config file calls for more invariants) do
13:         family, parameters := readConfigFile()
14:         candidates := getCandidates(family, parameters)
15:         candidates := reduceToBest(candidates, invariants)
16:         candidates := testBaseCase(candidates)
17:         repeat
18:             candidates := testIndStep(candidates, invariants)
19:         until (No change in candidate set)
20:         invariants += candidates
21:     end while
22:     return invariants
23: end function
24:
25: function eliminateRegisters(design, invariants)
26:     depends := ∅
27:     for all (registers R in design) do
28:         test := buildResubTest(next(R), other next-states)
29:         if (satSolve(test) == unsat) then
30:             proof := getResolutionProof()
31:             curr := getDependencyFunc(R, proof)
32:             notCurr := getDependencyFunc(¬R, proof)
33:             depends += pickBest(curr, notCurr)
34:         end if
35:     end for
36:     depends := makeCompatible(design, depends)
37:     return simplifyDesign(design, depends)
38: end function
```

IV. Resubstituting for Optimal Logic Reduction

In this section, we discuss our enhanced resubstitution procedure (function `eliminateRegisters` in Algorithm 1). Our resubstitution setup is illustrated in Figure 2, which

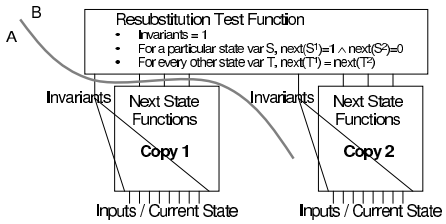


Fig. 2. Our enhanced resubstitution framework

is similar to Figure 1 but modified in several ways. This section will discuss how we target the formulation to find dependent registers as well as enhancements that make SAT-based resubstitution effective in practice. We iteratively call this procedure for every next-state function in the design in order to find the set of all dependent registers.

A. Register Elimination

If the resubstitution formulation illustrated in Figure 2 is unsatisfiable, then a dependency function will be obtained that may be used as a replacement for $next(S)$. Trading the existing implementation of $next(S)$ with the dependency function may yield a savings in the number of ANDs in the AIG. This paper is targeted to aiding verification where the number of registers often is more important than the number of ANDs. Here we present a formulation by which a dependency function can be used to directly eliminate a register in the design.

Consider Figure 3a where the logic needed to implement $next(S)$ is highlighted. If a dependency function exists, it will express $next(S)$ as a function of the other two next-state signals $next(T_1)$ and $next(T_2)$. The implementation of $next(S)$ may be replaced with this dependency function, as illustrated in Figure 3b. We can further simplify the design by expressing this dependency function over the current states instead of the next-states, thereby eliminating register S.

Definition 1: Let an *orphan state* be any state σ_1 for which there does not exist a state σ_2 such that σ_1 is reachable from σ_2 in one transition.

Note that every reachable state, with the possible exception of the design's initial state(s), is not an orphan state.

Theorem 1: For registers S, T_1, ..., T_n, if there exists an $F(\cdot)$ such that $next(S) = F(next(T_1), \ldots, next(T_n))$ then for every state that is not an orphan state $S = F(T_1, \ldots, T_n)$.

Proof: Let σ_1 be a state of the design and a concrete valuation of the registers S, T_1, ..., T_n. If σ_1 is not an orphan state then there exists a state σ_2 such that σ_1 can be reached in one transition from σ_2. Let $X(\sigma_j)$ denote the

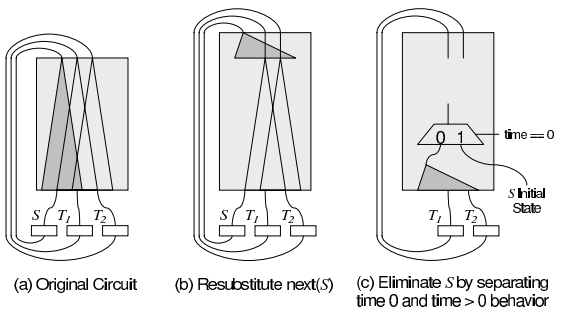


Fig. 3. Register elimination process

valuation of register X in state σ_j and note that there exists inputs such that $next(X(\sigma_2)) = X(\sigma_1)$. From the hypothesis we have $next(S(\sigma_2)) = F(next(T_1(\sigma_2)), \ldots, next(T_n(\sigma_2)))$. Rewriting we see that $S(\sigma_1) = F(T_1(\sigma_1), \ldots, T_n(\sigma_1))$. ■

Theorem 1 allows for the dependency function computed over next-state functions to be expressed over current states, provided the initial state(s) are accounted for. The result of this process is illustrated in Figure 3c. The dependency function between Figure 3b and 3c is identical; only the logic driving its inputs has changed. The register S has been completely eliminated at the cost of initial state correction logic. To correct the initial state, we introduce a multiplexor which at time 0 will drive the initial value of the register being eliminated, and thereafter will drive the dependency function for the next-state function of the eliminated register. To enable selection of these two values, a register may need to be introduced to the design which initializes to 1 and thereafter drives 0. This register is reused across all resubstitutions.

While Theorem 1 is not guaranteed to hold in the initial state, in some cases the initial value may be preserved by that dependency function. That is, the value produced by the dependency function at time 0 may be identical to the initial value of the register being removed, and the initial state correction logic can be omitted. On a benchmark suite for which 3,390 resubstitutions were performed, the initial state had to be corrected 67% of the time. Our designs had complex initialization functions due to retiming [11] whose value the dependency function could replicate with relatively low probability. This illustrates the power of our technique enhance register reduction capability particularly in the presence of complex initial states.

B. Optimal Dependency Identification via Invariants

A key strength of our formulation is its use of unreachability invariants, as illustrated in Figure 2. An unreachability invariant may be synthesized into a gate I over registers in the design such that $I = 0$ only on unreachable states. Through the use of an adequately strong set of unreachability invariants, our formulation may optimally identify all dependent registers. However, due to resource limitations, an incomplete set of unreachability invariants will often be used to identify many but not necessarily all dependent registers. We discuss the invariant generation scheme which we have found effective in this application in Section V.

C. Compatible Dependencies

The `eliminateRegisters` function from Algorithm 1 will attempt to resubstitute each next-state function present in the design. Through this process, a large number of dependency functions may be identified that can replace existing registers as depicted in Figure 3. Unfortunately, the set of dependencies found in this manner are generally not *compatible*, and if multiple dependency simplifications are performed simultaneously then often a combinational cycle will be created in the AIG resulting in an illegal design. A compatible set of dependencies is one in which all dependencies can be applied simultaneously with no resultant combinational cycles. Therefore, once the dependencies have been identified, one must identify a subset of compatible dependencies contained therein, and this chosen subset may impact the size of the resulting design.

Algorithm 2 Selecting a set of compatible dependencies

```
1:  function makeCompatible(design, dependencies)
2:      scored := ∅, compatible := ∅
3:      for all (Dep. D in dependencies) do
4:          red = D.redundant_AIG_node
5:          repl = D.replacement_AIG_node
6:          gain = aigSize(design) - aigSize(design - red + repl)
7:          scored[D] = scoreFunction(gain.regs, gain.ANDs)
8:      end for
9:      sortDescending(scores)
10:     for all (Dep. D in scored) do
11:         red = D.redundant_AIG_node
12:         repl = D.replacement_AIG_node
13:         if (!isCyclic(repl, red, compatible)) then
14:             compatible += D
15:         end if
16:     end for
17:     return compatible
18: end function
```

To illustrate the notion of incompatible resubstitution, consider a design with registers $R_1, \ldots R_n$ which have a one-hot encoding where in every reachable exactly one of these n registers will evaluate to 1. Given adequate invariants to characterize this one-hot condition, the following dependencies may be identified:

$$R_1 = \overline{R_2} \wedge \overline{R_3} \wedge \overline{R_4} \wedge \cdots, \quad R_2 = \overline{R_1} \wedge \overline{R_3} \wedge \overline{R_4} \wedge \cdots$$

It is not possible to express R_1 as a function of R_2 and simultaneously express R_2 as a function of R_1 without creating a combinational cycle.

Finding a compatible subset of dependencies is a computationally difficult task. Finding an optimal subset would entail enumerating and testing every possible subset, and this is feasible for only very small sets of dependencies. In this work we utilize a heuristic to quickly find a near-optimal subset of compatible dependencies.

After the complete set of dependencies is found, we reduce this to a set of compatible dependencies as illustrated in Algorithm 2. Each found dependency consists of two signals: a *redundant* signal that will be eliminated and a *replacement* signal that will be introduced in its place. We first sort the dependencies in the order of their ability to simplify the circuit, computed as a function `scoreFunction`. (Experimentally, we have found that the function $20 \cdot$ *gain*.regs + *gain*.ANDs works well.) The list of sorted dependencies is then iterated over, and a subset of compatible dependencies is greedily found. For each dependency, we test if performing this optimization in the presence of the other *compatible* dependencies will introduce a combinational cycle using `isCyclic`. If so, the candidate merge is discarded. Otherwise the merge is added to the *compatible* set.

While this search is greedy, prioritizing the dependencies by score enables the algorithm to capture most of the optimization potential present in the original set of dependencies. This

TABLE I. Compatible dependencies on a set of IBM benchmarks

Design	Total Deps. Count	Total Deps. Sum Score	Compatible Deps. Count	Compatible Deps. Sum Score
IBM01	376	-1714	325	-91
IBM02	194	-659	154	-725
IBM03	428	-18278	374	-16950
IBM04	678	643	579	909
IBM05	35	312	22	258
IBM06	102	573	58	308
IBM07	142	139	102	-97

is illustrated on a set of industrial designs in Table I. For each design, the found dependencies and compatible subset of these found dependencies are examined. Usually only a small percentage (24%) of the total dependencies must be discarded to form a compatible subset. If the total gain is summed over all possible dependencies, we see that the sum gain from the compatible dependencies is similar. This indicates that most of the AIG optimization potential present in the full set of dependencies was captured by the compatible subset.

D. Reducing Dependency Function Interpolants

The dependency functions are obtained from the interpolant of a proof of unsatisfiability. Using the method given in [7], the resultant logic will have size that is linear in the size of the resolution proof. The proof of unsatisfiability for complex SAT problems may be large, and this may result in an interpolant and resulting dependency function that are very large. Here we explore several ways to control the size of the obtained dependency functions.

The most basic way to control the size of the dependency functions is with combinational synthesis. Logic that comes from interpolants is usually highly redundant and amenable to combinational synthesis techniques [12]. While combinational synthesis is effective in reducing the size of these interpolants, it is too slow to be used on every interpolant prior reducing to a compatible subset in Section IV-C. Here we focus on ways to more directly optimize our dependency functions before combinational synthesis is applied.

One simple way to control the size of the interpolants before synthesis is applied is to use incremental SAT. Our implementation attempts to resubstitute each next-state function in the sequential AIG, and through this process many similar SAT problems are encountered. Using one incremental solver instance to solve all of these problems is advantageous for two reasons:[1]

1) One incremental solver typically learns fewer clauses than many non-incremental solvers. The size of an interpolant is related to the number of learned clauses, and using incremental SAT will result in a reduction in the total size of all interpolants.
2) In incremental SAT the learned clauses from one problem are preserved and may contribute toward the search for a satisfiable solution to a future problem. If the same learned clause participates in two proofs of unsatisfiability then the two interpolants will share common logic. This also reduces the total cost of the logic needed to implement all interpolants.

[1]Incremental SAT is generally preferred, but such solvers can store a large number of learned clauses. If memory is a concern, the solver instance may need to be periodically refreshed.

TABLE II. Dependency function use on a set of IBM benchmarks

Design	Depend. Count	Avg. Score by Repl. Type S	Avg. Score by Repl. Type $\overline{S}$
IBM01	1128	0.70	-0.87
IBM02	582	0.78	0.04
IBM03	1284	-6.56	-20.20
IBM04	2034	1.71	2.11
IBM05	104	3.10	6.06
IBM06	306	2.42	4.44
IBM07	426	1.44	2.21

In addition to using incremental SAT, we propose a more intelligent approach to mitigate logic bloat. Consider the resubstitution framework shown in Figure 2 that is able to eliminate a register S by resubstituting $next(S)$. Copy 1 of the transition relation represents the on-set of $next(S)$, and Copy 2 represents the off-set. Using the partitioning that separates Copy 1 from the rest of the circuit as shown, we get an interpolant I such that $next(S) \implies I \implies \overline{next(S)}$, and the resulting dependency function is able to replace S using the concepts of Section IV-A. However, the problem is symmetric and we could alternatively partition to separate Copy 2 from the rest of the circuit. In this case, $\overline{next(S)} \implies I \implies next(S)$, and the dependency function is be able to replace $\overline{S}$.

Thus, we have the flexibility to compute the interpolant in two different ways to either replace S or $\overline{S}$. These two replacements affect the size of the modified AIG in different ways, and to quantify this each possible replacement is scored in a manner identical to Section IV-C. By selecting the highest-scoring replacement, the dependency function can be used to its best advantage.

This is examined on a suite of industrial benchmarks in Table II. For each design, the number of found dependencies is given. Each dependency is scored as if it were used to replace one of the two signals: S or $\overline{S}$, and the average score for each replacement type is given. A negative score indicates that the number of ANDs introduced to build the dependency function was greater than the cost of the logic being removed. Note that the signal we would prefer to replace is benchmark-dependent. In general, it is also dependency-specific, and our implementation individually scores each dependency in two ways in order to best utilize each dependency function.

V. Invariant Generation

Unreachability invariants are essential to the reduction potential of SAT-based dependent register elimination. Given invariants which adequately characterize the unreachable states of a design, the formulation of Figure 2 is able to optimally identify all dependent registers in a design. However, in practice resource limitations will entail that the set of invariants may be a subset of those which truly hold of the design. The invariant generation approach which we have found useful for dependent register identification is detailed in this section.

The invariant generation algorithm is depicted in function `gatherInvariants` of Algorithm 1. The set of invariants is found over several iterations of a basic invariant discovery

algorithm. By finding invariants over several iterations, the overall framework is made considerably faster as the number of candidate invariants to be proved in each iteration decreases. Also, after each iteration completes a new set of invariants is found, and this provides a place that the computation can be safely terminated if computational resources are exceeded. After each iteration, a set of new invariants have been obtained that can then be added to the global collection of proved invariants.

When discovering new invariants, the first step is gathering the *candidate invariants*, properties that are suspected to hold but have not yet been proved. A set of properties from a particular property family, described in more detail in Section V-A, are first validated against a small set of simulation vectors. Each vector is derived from a random walk on the reachable state space and is therefore guaranteed to visit only reachable states. If any candidate invariants are found invalid by simulation, they are immediately discarded.

Next, the remaining candidates are filtered, as described in Section V-B. The goal is to remove candidates that are easily falsifiable or that are not capable of refining the current reachable state set over-approximation, given by the conjunction of all the already proved invariants. By filtering the candidates in this way, the proof of these candidates is made more scalable while the candidate set's ability to strengthen the current set of proved invariants changes only very little.

The final step involves proving that the candidate invariants hold in all reachable states. There are a variety of unbounded verification algorithms that could be used, but in general induction is the most scalable for large industrial designs. In this work we use k-induction [13] which involves proving the following:

Base Case The candidate invariants hold for all states reachable in k or less transitions from the initial state(s).

Inductive Step For all paths[2] of length k on which the candidate invariants hold, the invariants also hold in all states reachable in the next time step.

A. Property Families

The candidate invariants are local properties over the nodes in the design's AIG. In our implementation there are five property domains we consider: constants, equivalences, k-cuts, implications, and random clauses. Each family exhibits a different proof complexity and ability to refine the set of proved invariants.

Constants are nodes that appear to take the same value in all reachable states.

Equivalences are pairs of nodes that appear equivalent in every reachable state [14].

k-cuts are candidate invariants that are derived from k-feasible cuts of nodes in the network [15]. The set of k-feasible cuts for all nodes in a user-defined part of the AIG are enumerated. For each cut, candidate invariants are derived by forming a clause from the OR of the cut nodes. 2^k such candidates are derived, one for each polarity assignment to the cut nodes. In practice, for small k the number of cuts is approximately linear in the size of the AIG, and therefore the number of candidate invariants is approximately linear as well.

Implications are 2-literal clauses over pairs of gates in a design [16], [17]. Because implications are found exhaustively, there may be a quadratic number of candidate implication invariants.

Random clauses are clauses formed from random sets of nodes. The size of such a set and the circuit location from which the nodes come is parametrized. These invariants are effective in refining the current reachability approximation in ways that the other property families are not capable of.

[2]This can be strengthened to *unique state induction* by only considering simple paths [13]. Here we omit that constraint for computational reasons.

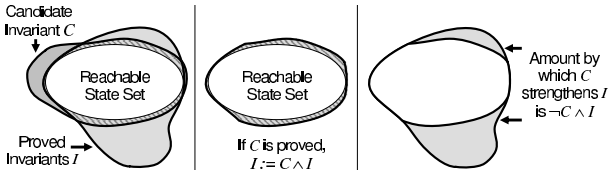


Fig. 4. Filtering candidate invariants

B. Candidate Filtering

The number of candidate invariants generated may be very large, and it may be computationally infeasible to prove that each of these candidates are true in every reachable state. Additionally, not every candidate invariant is effective in characterizing unreachable states which are not already characterized by other already proved invariants. Therefore it is practically necessary to filter the candidates before they are proved.

Let I be the conjunction of all previously proved invariants, as illustrated in Figure 4. The on-set of I contains the set of reachable states, and we would like to find new invariants which are able to reduce this on-set to better approximate the reachable states.

The amount by which a new candidate invariant C may strengthen the current reachability approximation I is equal to the size of the on-set of $\neg C \wedge I$. The size of the on-set is difficult to precisely compute, though it can be estimated with random simulation. For each candidate invariant I, the number of times random simulation asserts $\neg C \wedge I$ is recorded as the candidate's "score. " The top-scoring candidate invariants are selected for the inductive proof attempt. These are the candidates that, if proved, can best refine the current reachability approximation.

C. Invariant Generation Process

In this section we detail the algorithmic parameters which we have found useful for invariant generation. The basic invariant discovery loop (`gatherInvariants` of Algorithm 1) is iterated several times in order to quickly form a high-quality reachability approximation. In our experiments the following cycle was repeated until until a user-specified time limit was reached:

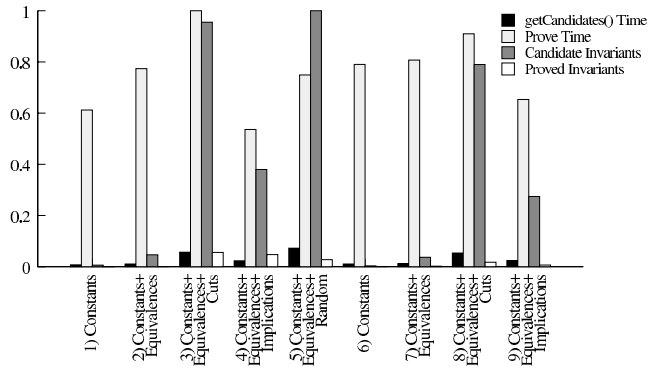


Fig. 5. Invariant generation process

1) Look for constants. Keep the 5000 "best" candidates (Section V-B), and prove them using 1-step induction.
2) Repeat step 1, and enable equivalences.
3) Repeat step 2, and enable 4-cuts. Restrict the search for cuts to the nodes near the registers (lower 8 AIG levels).
4) Repeat step 2, and additionally look for Boolean implications between registers.
5) Repeat step 2, and additionally look for random 3-literal clauses. Restrict the nodes that can participate in these clauses to be near the registers (lower 8 AIG levels).
6) Increment the k in k-step induction, and go to step 1.

These invariant generation iterations are illustrated for one IBM benchmark in Figure 5. Four statistics are shown: 1) `getCandidates()` Time: time needed to derive candidate invariants, 2) Prove Time: time needed to prove those candidates, 3) Candidate Invariants: total number of candidate invariants after filtering, and 4) Proved Invariants: total number of proved invariants. These four statistics were collected over 9 iterations of the basic invariant discovery loop. Iterations 1 - 5 use $k = 1$ induction while 6 - 9 use $k = 2$ induction.

Note that iterations 3 and 8 compute the invariants over the same families, but the number of candidates in iteration 8 is significantly less than in iteration 3. This is because iterations 1-7 have derived a tighter reachability approximation than iterations 1-2 alone, and so in iteration 8 there are fewer candidates that are able to refine this reachability approximation and the filtering described in Section V-B is more effective.

On the industrial design examined in Figure 5, invariants successfully proved in the following three families: k-cuts, random clauses, and implications. In our experience all 5 families provide useful invariants in general, and cycling through different property families allows them to complement each other. The resultant reachability approximation is more effective in strengthening dependent state element elimination than invariants derived from any single family alone.

D. Extracting Synthesis Properties

The constants and equivalences property families give invariants that can be directly used to simplify the circuit. Specifically, each invariant of this type implies that a node in the AIG can be removed and replaced with either another node or a constant.

Our implementation will detect such simple conditions and optimize the AIG. This is not as powerful as sequential SAT sweeping [14] because some useful candidate invariants may be removed by the filtering of Section V-B, but the synthesis optimizations require little overhead and occasionally help to reduce the design size. Additionally, previously proved invariants strengthen our inductive formulation and so occasionally we find more equivalences than sequential SAT sweeping.

VI. Experimental Results

The algorithms discussed in this paper were implemented in the IBM internal verification tool *SixthSense* [11]. The methods are intended to simplify a design in order to reduce the computational expense of subsequent verification algorithms. To demonstrate this ability, two sets of experiments are presented. The first examines the power of our techniques to reduce the size of a design, the second examines the effect of these simplifications on verification algorithms. We additionally provide analysis of the qualitative nature of the reductions achieved in these experiments.

A. Reduction Potential

The proposed synthesis method is able to yield reductions beyond those possible with state-of-the-art synthesis methods. To demonstrate this, our benchmarks were aggressively preprocessed before our dependent register elimination algorithm was applied. The preprocessing steps included: combinational synthesis, min-register retiming, removal of sequentially equivalent gates, and input reparameterization [11], [18]. These steps are able to dramatically reduce the size of the design, and after this preprocessing these existing synthesis methods cannot optimize these designs further.

Our proposed synthesis algorithm, illustrated in Algorithm 1, is evaluated on the preprocessed design in Table III. First we run Algorithm 1 with invariants disabled in order to explore the benefits of dependent state element elimination in the absence of invariants. We limit ourselves to two calls to function `eliminateRegisters` in Algorithm 1. Next, we revert to the preprocessed design then rerun with invariants enabled. The invariant generation algorithm was given 60 seconds to find invariants using the scheme discussed in Section V-C, and the number of proved invariants is given in the *Invars* column.

The effectiveness of our technique is highly dependent on the benchmark set. In *Set1*, the designs have many functionally dependent state elements, even after powerful sequential synthesis techniques were applied. On this benchmark set, dependent register elimination was very effective, even without invariants. However, the invariants did help mitigate the increase in AND gates which may occur through our dependent state element elimination.

In the benchmarks *Set2* and *Set3*, there exist very few registers that can be identified as dependent without the use of invariants. Enabling invariant generation more than doubles the number of dependent state elements that can be eliminated, on average[3]. This indicates that while dependent

[3]Occasionally, enabling invariant generation degrades our results. This is due to the heuristic and greedy nature of the algorithms described in Sections IV-C and IV-D.

TABLE III. Performance on three sets of IBM benchmarks[1]

	Preprocessed Design				Algorithm 1 Without Invariants			Algorithm 1			
Design	Inputs	ANDs	Regs	Time	ANDs	Regs	Time	Invars.	ANDs	Regs	Time
Set1 / IBM01	36	2083	393	158.85	1997	223	15.33	190	1995	224	91.53
Set1 / IBM02	26	906	223	107.77	931	144	9.63	389	911	146	82.05
Set1 / IBM03	44	5977	625	106.95	9602	429	69.90	166	9520	439	197.52
Set1 / IBM04	34	3536	704	16.83	3600	415	52.87	58	3588	414	128.94
Set1 / IBM05	189	19989	743	125.14	13828	733	77.24	514	14611	728	370.94
Set1 / IBM06	40	4373	698	110.98	4330	669	32.44	17	4321	669	104.35
Set1 / IBM07	44	1124	241	27.27	1082	189	10.79	288	1030	190	92.89
Set1 Summary[2]					1.04	0.75			1.03	0.75	
Set2 / IBM08	8	502	101	101.22	499	101	6.22	12	499	101	73.17
Set2 / IBM09	16	3258	683	16.01	3238	678	48.81	10	3232	677	175.71
Set2 / IBM10	89	4463	823	104.53	4471	814	55.29	29	3164	690	141.08
Set2 / IBM11	26	2402	530	102.77	2433	523	37.10	522	2386	521	214.00
Set2 / IBM12	114	770	199	104.69	768	197	7.69	431	754	195	88.15
Set2 / IBM13	189	5111	193	18.17	4878	167	11.84	160	4882	166	100.60
Set2 Summary[2]					0.99	0.97			0.94	0.94	
Set3 / IBM14	38	2773	470	103.15	2773	470	10.76	123	2735	461	100.83
Set3 / IBM15	125	15796	668	120.69	11972	655	113.74	399	13113	655	256.80
Set3 / IBM16	68	2757	680	21.55	2743	675	36.55	16	2740	675	98.53
Set3 / IBM17	127	15867	654	125.49	11882	640	99.87	464	13017	640	521.74
Set3 / IBM18	125	15675	668	128.19	11977	658	127.87	306	13165	658	336.07
Set3 / IBM19	84	5958	776	111.55	6219	762	93.52	146	5560	689	154.88
Set3 Summary[2]					0.88	0.99			0.90	0.97	

[1] All experiments were run on a 1.83 GHz laptop running Linux 2.6. [2] Ratios are relative to the preprocessed AIG size.

state elements abound, their discovery requires the use of reachability invariants.

B. Impact on Verification

Enabling reductions in the number of registers and possibly AND gates in a sequential AIG is advantageous in numerous application domains. Our goal in this paper is to leverage such reductions to simplify an overall verification task. In order to quantify the impact of our reduction on verification complexity, we ran 267 IBM designs, each with a collection of safety properties, through a rugged Transformation-Based Verification (TBV) [11] setup. This setup involves numerous sequential synthesis techniques as mentioned in Section VI-A (Algorithm 1 not included) along with SAT-based bounded model checking (BMC), induction, and interpolation. BMC was limited to 90 seconds, and k-induction was run for 300 seconds, both increasing bounds until the time limit was reached. Interpolation works on each property separately and was limited to 300 seconds per property. It is known that reductions in the number of registers can additionally aid BDD-based verification, but these designs were large and consistently beyond the capacity BDDs.

Of the 267 designs processed in this manner, only 141 had unsolved properties after the rugged TBV script. On these remaining 141 examples, we applied Algorithm 1 and repeated the same rugged TBV script. Algorithm 1 further simplifies the designs, and any properties that are solved in the second TBV pass are a direct result of these simplifications.

The second TBV pass solved properties in 10 of the remaining 141 designs, or 7.1%, as illustrated in Table IV. The cone-of-influence reduced design size after the initial TBV processing is illustrated first, followed by the size after the reduction of Algorithm 1. We next illustrate the extent to which our dependent register elimination may synergistically enhance the reduction potential of subsequent synthesis algorithms, and finally we indicate which verification algorithms were successful in solving properties in the second TBV pass. Note that Algorithm 1 is able to reduce the number of registers in these examples, and also enables further register reductions by more conventional sequential synthesis techniques[4].

In total, 11 properties are solved as a result of the simplifications Algorithm 1. Eight properties are solved with interpolation, and the time needed to solve these properties is well below the 300 second timeout. One property was solved with induction, again far below the 300 second timeout. There were two properties for which a counterexample was "hit." One hit was with BMC, indicating that BMC was able to process more time steps on the reduced design than it was on the original design, and a counterexample was present in these extra time steps. The other hit occurred as simulation run during redundancy removal analysis.

Overall, we have found this technique to be a useful component of our multi-algorithm verification system. Since it is nontrivially expensive in itself, and due to the potential of AND gate increase through interpolant synthesis, in practice we have found it useful to leverage *after* a preprocessing of quicker transformations and proof / falsification algorithms similarly to the algorithm flows used in these experiments. While the percentage of designs for which we were able to solve additional properties through our technique may seem limited (7.1%), we are encouraged by this result since these 141 designs comprise very difficult verification problems which otherwise are extremely difficult or impossible to solve.

C. Characterizing the Simplifications

In order to more fully understand the type of reductions enabled by Algorithm 1, we studied several examples to attempt to characterize the nature of the reductions. Recall that

[4]The occasional bloat in AND count during this final synthesis phase is primarily due to retimed initial value computation and input reparameterization [11], [18].

TABLE IV. Effect on Verification[1]

Specified Design		After TBV			After Algorithm 1		After Resynthesis		
Design	Properties	Properties	ANDs	Regs	ANDs	Regs	ANDs	Regs	After More TBV
IBM20	6	2	758	245	1100	241	697	229	1 proof w/ interpolation (36 sec)
IBM21	17	3	5754	457	5810	455	8173	417	2 proofs w/ interpolation (37 sec, 24 sec)
IBM22	387	5	13620	1147	13550	1137	24917	1126	1 hit w/ BMC (depth 44, 90 sec)
IBM23	6	3	1853	242	2915	223	1815	223	1 proof w/ interpolation (165 sec)
IBM24	17	3	6581	455	6626	449	8067	435	1 proof w/ interpolation (237 sec)
IBM25	1	1	1002	197	1223	195	1010	195	1 proof w/ interpolation (182 sec)
IBM26	6	2	758	245	1100	241	697	229	1 proof w/ interpolation (36 sec)
IBM27	2	2	4350	820	3073	687	3252	680	1 proof w/ induction (k=37, 203 sec)
IBM28	41	24	26932	3259	26953	3249	26872	3247	1 hit with random simulation
IBM29	1	1	1002	197	1223	195	1010	195	1 proof w/ interpolation (179 sec)

[1] All experiments were run on a cluster of IBM workstations running 64-bit AIX 5.3 on POWER 5 CPUs ranging from 2.1 to 2.2 GHz.

our experiments included aggressive synthesis preprocessing, hence most of the simpler reduction potential that could be claimed as dependency was already eliminated.

The designs that we report in our experiments are effectively *testbenches* comprising a hardware logic component, a *driver* which constrains legal input stimulus, and a *checker* which encodes correctness properties. One source of dependencies that we discovered was inherent in this testbench setting: first, the logic used to specify the checker and driver is not highly hand-optimized, instead comprising logic such as sparse decodings of values driven or sampled from the design. Second, in cases dependencies arise between the specification logic and the design; e.g., perhaps the checker tracks a "design is busy" condition which happens to be equivalent to the OR of several state bits inside the design. Finally, in cases the input constraints used in a driver may create dependencies that otherwise would not exist in the design.

We also discovered similar dependencies within the design logic itself. For example, instruction-decode logic and one-hot state machines create sparse state vectors which can often be re-expressed with fewer registers. This redundancy is desirable in high-performance circuitry to minimize combinational delays but presents opportunity for reductions using Algorithm 1.

There is substantial motivation for the development of automated techniques to efficiently identify and eliminate those dependencies. While some of these reduction opportunities could be readily manually identifiable, some of the dependencies identified in our benchmarks were intricate relations over dozens of other registers, which would have been impractical to attempt to identify by hand.

VII. Conclusion

This paper developed a method to eliminate functionally dependent state elements in a sequential design. The method extends prior work in dependency identification [1] with several enhancements:

- Dependent state elements can be identified and directly removed, reducing the number of registers in the design.
- A method to identify a compatible set of dependencies is discussed. This method is fast and effective in reducing the found dependencies to a compatible subset without sacrificing significant optimization potential.
- A method is developed that can effectively mitigate the logic bloat that comes from interpolation.
- The dependent state element elimination is strengthened with an invariant generation framework, enabling the detection of unreachable state invariants which extend this purely SAT-based optimization technique into a sequential synthesis transformation.

Experiments demonstrate that this technique can significantly reduce the number of registers in industrial designs even after powerful sequential synthesis methods such as min-register retiming and redundancy removal have been applied, an area where we have found the technique of [1] to be ineffective. This reduction is also shown to synergistically enable greater synthesis optimizations, and to enhance a variety of verification algorithms.

Acknowledgements

The authors would like to thank Per Bjesse for all his help in preparing this paper for publication.

References

[1] J. Jiang and R.K. Brayton, "Functional Dependency for Verification Reduction", at *CAV* 2004.
[2] M. Wedler, D. Stoffel and W. Kunz, "Exploiting state encoding for invariant generation in induction-based property checking," in *ASP-DAC*, 2004.
[3] C. Lee, J. Jiang, C. Huang and A. Mishchenko, "Scalable exploration of functional dependency by interpolation and incremental SAT solving," in *ICCAD* 2007.
[4] R.K. Brayton, R. Rudell, A. Sangiovanni-Vincentelli and A.R. Wang, "MIS: A Multiple-Level Logic Optimization System," in *TCAD* 1987.
[5] L. Zhang and S. Malik, "Validating SAT solvers using an independent resolution-based checker: practical implementations and other applications," in *DATE* 2003.
[6] W. Craig, "Linear reasoning: A new form of the Herbrand-Gentzen theorem," in *J. Symbolic Logic* 1957.
[7] P. Pudlák, "Lower bounds for resolution and cutting plane proofs and monotone computations", in *J. Symbolic Logic* 1997.
[8] K.L. McMillan, "Interpolation and SAT-Based Model Checking," in *CAV* 2003.
[9] J. Baumgartner and A. Kuehlmann, "Min-Area Retiming on Flexible Circuit Structures," in *ICCAD*, 2001.
[10] A. Mishchenko, S. Chatterjee and R.K. Brayton, "DAG-aware AIG rewriting: A fresh look at combinational logic synthesis," in *DAC*, 2006.
[11] H. Mony, J. Baumgartner, V. Paruthi, R. Kanzelman and A. Kuehlmann, "Scalable Automated Verification via Expert-System Guided Transformations," in *FMCAD* 2004.
[12] G. Cabodi, M. Murciano, S. Nocco, and S. Quer, "Stepping forward with interpolants in unbounded model checking," in *ICCAD* 2006.
[13] N. Eén and N. Sörensson, "Temporal Induction by incremental SAT solving," in *Proc. Workshop on Bounded Model Checking* 2003.
[14] C.A.J. van Eijk, "Sequential equivalence checking based on structural similarities," in *TCAD* 2000.
[15] M.L. Case, A. Mishchenko and R.K. Brayton, "Cut-Based Inductive Invariant Computation," at *IWLS* 2008.
[16] M.L. Case, A. Mishchenko and R.K. Brayton, "Inductively Finding a Reachable State Space Over-Approximation," in *IWLS* 2006.
[17] M.L. Case and R.K. Brayton, "Maintaining A Minimum Equivalent Graph In The Presence of Graph Connectivity Changes," UC Berkeley Technical Report, 2007.
[18] J. Baumgartner and H. Mony, "Maximal Input Reduction of Sequential Netlists via Synergistic Reparameterization and Localization Strategies," in *CHARME* 2005.

Optimal Constraint-Preserving Netlist Simplification

Jason Baumgartner[1] Hari Mony[1,2] Adnan Aziz[2]
[1]IBM Systems & Technology Group, Austin, TX [2]The University of Texas at Austin

***Abstract*— We consider the problem of optimal netlist simplification in the presence of *constraints*. Because constraints restrict the reachable states of a netlist, they may enhance logic minimization techniques such as redundant gate elimination which generally benefit from unreachability invariants. However, optimizing the logic appearing in a constraint definition may weaken its state-restriction capability, hence prior solutions have resorted to suboptimally neglecting certain valid optimization opportunities. We develop the theoretical foundation, and corresponding efficient implementation, to enable the optimal simplification of netlists with constraints. Experiments confirm that our techniques enable a significantly greater degree of redundant gate elimination than prior approaches (often greater than $2\times$), which has been key to the automated solution of various difficult verification problems.**

I. Introduction

Verification testbenches often require the specification of environmental assumptions to prevent uninteresting failures due to illegal input scenarios. For example, a testbench for an instruction buffer may require adherence to assumptions which model the behavior of the instruction fetch unit, such as requiring that instructions are not transferred into the buffer if doing so would cause overflow. Most testbenches require a substantial number of assumptions, many of which involve temporal handshaking with the outputs of the design.

There are two fundamental approaches to modeling environmental assumptions. First, one may utilize an imperative generator-style paradigm, where (possibly sequential) *filter* logic is used to convert nondeterministic data streams into legal input sequences. This filter logic is in turn composed with the design under verification [1]. Second, one may utilize a declarative constraint-based approach, wherein illegal scenarios are enumerated using specific language constructs, and the verification toolset must ensure that these scenarios are not violated in any reported counterexample [2].

Constraint-based testbenches have numerous advantages over generator-based approaches. Most notably, the *checker-assumption duality* paradigm allows an assumption on the inputs of one design component to be directly reused as a checker on the outputs of an adjacent design component. This duality enables the guarantee of compositional correctness by cross-validating assumptions across adjacent components [3]. Constraints may also be used to implement case-splitting strategies to decompose complex verification tasks for computational efficiency [4]. Due to their benefits, constraints have gained wide-spread acceptance, and most verification languages provide constructs to specify constraints – e.g., through the *assume* keyword of SystemVerilog [5].

Despite their prevalence, constraints pose challenges to numerous verification algorithms. For example, consider redundancy removal frameworks, wherein gates which are functionally equivalent in all reachable states may be merged to reduce design size. Because constraints restrict the set of reachable states, they may enhance reduction potential by enabling a pair of gates to be merged which are equivalent within the constrained state space, but may not be equivalent otherwise. Viewed another way, the constraints imply a *don't care* condition which may be exploited when attempting to merge gates, similar to the exploitation of *observability don't cares* (ODCs) for enhanced reduction [6]. However, as discussed in [7], such merging risks weakening the evaluation of the constraints, resulting in spurious property violations.

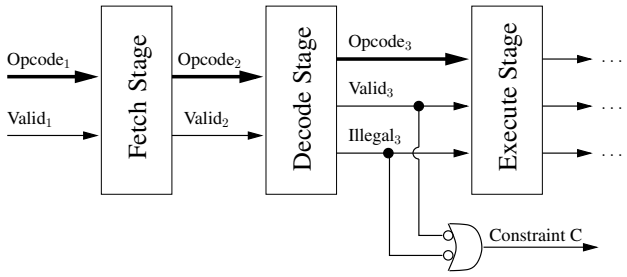


Fig. 1: Constraint weakening through merging

To illustrate how constraints may be weakened through merging, consider the example in Figure 1 which includes the Fetch-Decode-Execute component of a processor. Assume that the testbench for this component must prevent the Execute stage from encountering *valid* instructions with *illegal* opcodes. This may easily be achieved by constraining the output of the Decode stage to be *invalid* if its opcode is *illegal*. Next assume that the logic in the Fetch and Decode stages ensures that if the instruction in the Decode stage is invalid, its opcode is illegal. Coupled with the constraint, which enforces (Valid_3, Illegal_3) $\neq$ (1,1), this *satisfiability don't care* condition that (Valid_3, Illegal_3) $\neq$ (0,0) ensures that instructions in the Decode stage are invalid if and only if they are illegal, within all reachable states. This fact may be used to simplify logic in the Execute stage.

However, this fact will also entail redundancies in the fanin of the constraint, e.g., that the Valid_3 and Illegal_3 gates are antivalent—hence the redundancy removal process may wish to perform the corresponding merges. While the simplification entailed by merging is often advantageous in reducing subsequent verification resources, constraint-enabled merges within the fanin of constraints may lead to an overapproximation in property checking, in that the resulting constraints may lose their ability to prevent invalid counterexamples. In this case, such merging would syntactically simplify the constraint gate to *constant one*, precluding its ability to prevent valid yet illegal instructions from propagating into the Execute stage.

978-1-4244-2735-2/08/$25.00 ©2008 IEEE

We address the simplification of netlists in the presence of constraints through redundancy removal. In particular, we provide the theoretical foundations as well as an efficient implementation of an *assume-then-prove* sequential redundancy removal procedure. Our specific contributions include:

1. an efficient input stimulus generation algorithm that is robust against *dead-end states* (Section III);
2. a sequential redundancy identification framework which allows the identification of all equivalent gates in the presence of constraints (Section IV); and
3. an abstraction-refinement algorithm to optimally leverage the identified redundancy for netlist simplification (Sections V and VI).

Experiments are presented in Section VII to demonstrate that our solution enables a significantly greater degree of redundant gate elimination than prior techniques, which has been key to the automated solution of numerous difficult industrial verification problems. We discuss related work in Section VIII, and conclude this work in Section IX.

II. Preliminaries

Definition 1: A *netlist* is a tuple $\langle\langle V, E\rangle, G, T, C\rangle$ comprising a finite directed graph with vertices V and edges $E \subseteq V \times V$. Function $G : V \mapsto types$ represents a mapping from vertices to $gate\ types$, including primary inputs, registers, and combinational gates with various functions. The register is the only sequential gate type, which has a designated *initial value* (which specifies its value at time 0) as well as *next-state function* (which defines its time $i+1$ behavior). To ensure consistent semantics, it is required that a netlist contain no *combinational cycles*: directed cycles in $\langle V, E\rangle$ comprising no registers. Set $T \subseteq V$ represents the *targets*, corresponding to properties to be checked. Set $C \subseteq V$ represents the *constraints*, the significance of which will be described below.

Definition 2: A *trace* is a temporal sequence of Boolean valuations to vertices in the netlist which is consistent with G, beginning at time 0 which is consistent with initial values. A trace is hereafter understood to be restricted to its valid prefix wherein all constraint gates evaluate to 1. Practically, this prefix is often understood to be significantly shorter, e.g., merely long enough to demonstrate the assertion of a target. The *length* of a trace is 0 if it has no valid time-frames in its prefix, else one plus the last prefix time-frame.

The *verification goal* associated with a netlist is to obtain a trace illustrating an assertion of a target within its valid prefix (such a trace is hereafter referred to as a *counterexample*), or to prove that no such trace exists.

Definition 3: The *fanin* of gate g is the union of g and all gates h for which there exists a directed path from h to g in $\langle V, E\rangle$. The *combinational fanin* of g is merely g if g is of type *register*, else the union of g and all gates h for which there exists a directed path from h to g which includes zero registers, aside possibly from h itself. The *fanout* of g is the set of gates which include g in their fanin. The *combinational fanout* of g is the set of gates which include g in their combinational fanin.

1. Use an arbitrary set of algorithms to compute the *redundancy candidates* of N: sets of equivalence classes of gates, where every gate g in equivalence class $Q(g)$ is suspected to be equivalent to every other gate in the same equivalence class, along every trace.
2. Select a *representative gate* $R\big(Q(g)\big)$ from each equivalence class $Q(g)$.
3. Construct the *speculatively-reduced netlist* N' from N by replacing the source gate g of every edge $(g, h) \in E$ by $R\big(Q(g)\big)$. Additionally, for each gate g, add a *miter* T_g, which is a target representing function $g \not\equiv R\big(Q(g)\big)$.
4. Attempt to prove that each of the miters in N' is semantically equivalent to 0.
5. If any miters cannot be proven equivalent to 0 due to obtaining a trace illustrating their assertion or due to an inconclusive result from the proof algorithm, refine the equivalence classes to separate the corresponding gates and goto Step 2; else proceed to Step 6.
6. All miters have been proven equivalent to 0; return the accurate equivalence classes.

Fig. 2: Sequential redundancy identification framework

Definition 4: A *state* is a valuation to the registers of a netlist. A *reachable state* is one which may occur in a trace, and an *initial state* is a reachable state which may occur at time 0. A *dead-end state* is a state for which no valuation to the primary inputs will satisfy the constraints, hence is unreachable.

Definition 5: Given two netlists N and N', we say that N' *trace-contains* N if every trace of N is valid for N'. If the converse is also true, we say that N and N' are *trace-equivalent*.

Note that if N is trace-equivalent to N', verifying N' in place of N is *sound* and *complete*: a proof or counterexample obtained on N' may be reused as a result for the corresponding target of N. If N is trace-contained by N', verifying N' in place of N is *sound* but *incomplete*: a proof obtained on N' may be reused as a result for the corresponding target of N, though a counterexample on N' may not be valid for N. We refer to such a trace, which illustrates a target assertion on N' but not on N, as a *spurious counterexample*.

A. Redundancy Removal

Definition 6: A *merge* from gate g_1 onto gate g_2 consists of replacing every fanout edge $(g_1, g_3) \in E$ with (g_2, g_3).

To facilitate subsequent reasoning (e.g., trace analysis), after merging, g_1 is made a *buffer* – a single-input gate whose *type* is the identity function. This is done by deleting its fanin edges, and adding edge (g_2, g_1). Without loss of generality we assume that $g_1 \not\equiv g_2$, and that merges will yield valid netlists – i.e., may be initiated only if no combinational cycles will result.

Sequential redundancy removal frameworks attempt to identify functionally redundant gates in a netlist, which may be merged as a trace-equivalence preserving transformation. We refer to redundancy identification as *sound* if the identified gates are truly equivalent in all reachable states, and *complete* if all functionally equivalent gates are identified. Sequential redundancy identification frameworks, e.g., [8], [9], [10], operate as shown in Figure 2. Step 1 of this algorithm typically uses random simulation to compute candidates for equivalence. We address the impact of constraints on this process in Section III.

The speculative merging performed in Step 3 is necessary for scalability, since sequential redundancy identification is PSPACE-complete [11]. To ensure soundness of redundancy

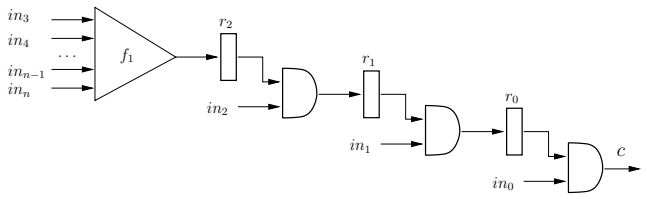


Fig. 3: Sequential constraint example

identification when using speculative reduction, it is required that the redundancy candidates computed in Step 1 be validated as being accurate in the initial states [10]. The selection of representatives must furthermore be performed such that N' contains no combinational cycles. This step requires additional consideration in the presence of constraints, since if the redundancy candidates are not truly equivalent, the speculative merge may alter constraint evaluation resulting in a procedure that is neither sound nor complete. We address this problem in Section IV.

When this algorithm terminates, it will reflect *all* gates which are equivalent across all reachable states. However, in the presence of constraints, additional analysis is necessary before they may be merged to avoid violating netlist semantics. We address this topic in Sections V and VI.

III. Constraint-Preserving Simulation

The use of random simulation is often critical to the effective computation of redundancy candidates. By comparing gate valuations from simulation traces, many inequivalences may be identified with modest runtime [8], [12].

Constraint-preserving input stimulus generation has been widely researched by the simulation community, e.g., in [13], [14]. However, the majority of prior work does not address dead-end states. While some discourage the use of constraints which entail dead-end states [13], such constraints may be readily expressed in various specification languages. We have often seen such constraints in practice due to the manual effort involved in mapping all constraints directly to inputs; e.g., to convert the single-predicate constraint that illegal Decode-stage instructions imply invalidity in Figure 1 into a complex constraint over instructions entering the Fetch stage. When a dead-end state is encountered, the simulation process must halt and begin anew from a shallower state. While the overhead of checkpointing and restoring states is somewhat undesirable, a more significant problem is that dead-end states may outright preclude the ability of simulation to reach deeper states in the design. In a sequential redundancy identification framework, this may entail highly inaccurate initial equivalence classes for gates which may only be differentiated by deep traces.

To illustrate the challenge of stimulus generation in the presence of sequentially-driven constraints, consider the example of Figure 3, where gate c is a constraint and r_0, r_1, r_2 are registers. The stimulus generator must assign values to inputs in_0, in_1, in_2 which prevent c from evaluating to 0 for the desired duration of the simulation run. While the stimulus generator could assign 0 to in_i (for $i \in \{0, 1, 2\}$) at any time-frame j, doing so would result in violating c at time $j+i$. The stimulus generator thus must perform assignments at time j which preclude constraint violations at later time-frames.

Our solution for stimulus generation is based upon k-step satisfiability (SAT) solving. At each time-frame j, we cast a satisfiability check from the current state of the netlist to identify a set of input valuations at time j which may be extended to satisfy each constraint for times j through $j + k$ using a *Sliding Window* approach. For scalability, it is important to keep k as small as possible because satisfiability checking is an NP-complete problem [15]. Our solution, as follows, thus performs design-specific analysis to determine a minimal value of k which avoids dead-end states.

Definition 7: Given a simple directed path in $\langle V, E \rangle$, the *delay* of that path is the number of gates of type *register*.[1] The *minimum input delay* of a gate g is defined as the minimum delay along any simple path from any primary input to g. The *maximum input delay* of a gate g is the minimum delay relative to the deepest primary input: the maximum value among any primary input for the minimum delay along any simple path from that primary input to g.

Empirically, we have observed that setting k to the *maximum input delay* of a corresponding constraint gate is adequate to prevent simulation from encountering dead-end states. The use of maximum input depth for k is intuitive; it represents the minimum delay at which some *furthest* input may affect the evaluation of a given constraint. Depending upon the nature of the design, an input may affect the constraint much later than the input delay, e.g., if the design has a counter. However, we have not encountered any instances for which this depth is inadequate, given a large set of dozens of industrial designs.

A related concern is whether k is larger than necessary, since the performance of the SAT solver is highly sensitive to this value. For example, if the AND gates in Figure 3 are converted to OR gates, the *minimum input delay* of 0 is adequate to avoid dead-end states by assigning in_0 to 1. In practice, the computation of minimum adequate window depth is prohibitively expensive, requiring the solution of a quantified Boolean formula checking whether for every state, there exists an input sequence of a particular depth which satisfies that constraint. Our practical solution to this problem is to break the overall simulation run into multiple phases, using a $\log_2$ search procedure within a minimum and maximum range (initialized as the *minimum* and *maximum input delay*, respectively) to identify a minimal adequate window depth to enable the desired simulation run length. At each phase, we generate stimuli using the minimum range value as the window depth. If a dead-end state is encountered, we update the minimum range to the unsuccessful value plus one and repeat the phase using a median depth between the minimum and maximum range. Otherwise, we update the maximum range to the successful value and proceed to the next phase.

IV. Optimal Redundancy Identification under Constraints

Speculative merging in the presence of constraints raises two concerns:

[1] We assume that oscillating clocks have been factored out of the netlist diameter, e.g. through *phase abstraction* [16]. Otherwise, the *delays* introduced in this section should be multiplied by the periodicity of the clocks.

1. Even if the speculative merges are for truly equivalent gates, the merges may weaken the constraints as per the example of Figure 1. This may entail incomplete redundancy identification, since unreachable states may become reachable under the speculative merges.
2. At the time of speculative reduction, the validity of the redundancy candidates has not been demonstrated. Thus, the speculative merges risk arbitrarily altering constraint evaluation, which may weaken *or strengthen* their evaluation, causing unsound redundancy identification since reachable states may become unreachable.

The following theorem establishes the correctness of a slightly modified form of speculative reduction in netlists with constraints, which ensures that miters for inequivalent redundancy candidates will remain assertable, and accurate miters will remain unassertable.

Theorem 1: Given a netlist N, consider netlist N' formed by adding to N a replication of the combinational fanin of each constraint, and re-labeling the replicated counterpart of each constraint gate as the constraint. Consider netlist N'', derived by speculative reduction of N', though restricting merging to the original gates of N. Using N'' as the basis of sequential redundancy identification is sound and complete in identifying redundant gates in N.

Proof: Consider any trace p over N, and the corresponding trace p'' obtained by simulating the input sequence of p on N''. To demonstrate soundness and completeness of redundancy identification, we prove a stronger condition: that either p illustrates no mismatches within any equivalence class *and* no miters are asserted in p'' (completeness), or that a nonempty subset of earliest-occurring mismatches illustrated in p has a corresponding subset of earliest-occurring miter assertions in p'' (soundness). We prove this condition by induction on this earliest time-frame.

Base Case: Time 0. As discussed in Section II-A, to ensure the soundness of speculative reduction, all equivalence classes must be validated as accurate at time 0. Thus no mismatches may be illustrated in p. Additionally, if p is valid at time 0, the speculative merging cannot alter any valuations in p vs. p'' at time 0 hence there can be no miter assertions at time 0. Furthermore, since there are no speculative merges in the combinational fanin of the constraints, their evaluation and hence their constraining power over registers and inputs cannot be altered, and thus p is valid at time 0 if and only if p'' is valid at time 0

Inductive Case: Time $i+1$. By the induction hypothesis, assume that no mismatches nor miter assertions occurred at times $0, \ldots, i$. We prove that this condition holds at time $i+1$. If no mismatches in p nor miter assertions in p'' manifest at time $i+1$, our proof obligation is trivially satisfied. Otherwise, first consider the case that there is a nonempty set of gates A which differ from their representatives at time $i+1$ in p. Consider the maximal subset $B \subseteq A$ such that no element in $b \in B$ has any element of $A \setminus b$ in the combinational fanin of b or $R(b)$, which is the representative gate from the equivalence class containing b. Clearly B is non-empty since N'' is a netlist hence free of combinational cycles. Note that any speculative merging in the fanin of b or $R(b)$ cannot alter their valuation at time $i+1$, because that merging did not alter the valuation of any gate at times $0, \ldots, i$ by the induction hypothesis, nor at time $i+1$ by the construction of B. Thus, the miters correlating to every gate in B must assert at time $i+1$ in p''. This same argument may be used to demonstrate that a nonempty set B of miter assertions at time $i+1$ in p'' must have a corresponding set of mismatches in p.

We finally must demonstrate that p is valid at time $i+1$ if and only if p'' is valid at time $i+1$. Note that the state of the registers in the combinational fanin of the constraints must be identical across p vs. p'' at time $i+1$, since their next-state functions evaluated the same at times $0, \ldots, i$ by the induction hypothesis. Since there are no speculative merges in the combinational fanin of the constraints, any mismatches at time $i+1$ cannot affect their *valuation* at that time-frame, nor their *evaluation* hence their constraining power over registers and inputs cannot be altered. Thus p is valid at time $i+1$ if and only if p'' is valid at time $i+1$.

We thus conclude that either no mismatches occur in p nor miter assertions in p'' at time $i+1$, or a nonempty set of speculatively merged gates B will illustrate miter assertions in p'' and mismatches in p at time $i+1$. ■

A. Refinement of Equivalence Classes

Because speculative merging of incorrect redundancy candidates may alter the evaluation of gates in their fanout, this may ambiguate the exact set of speculative merges which must be refined given a trace asserting a miter. Such inaccurate merging may cause miters in the fanout of B, the set in the proof of Theorem 1, to assert even though the corresponding redundancy candidates are truly equivalent, which if refined would result in suboptimal redundancy identification. This may also cause such miters to become unassertable even though the corresponding redundancy candidates would be differentiated during resimulation of the mismatch trace on the original netlist, which if not refined would require the computationally redundant step of generating an equivalent miter assertion trace at a future refinement iteration. Thus in practice, miter assertion traces should be resimulated on the original netlist to assess the exact set of gates to refine [10].

V. Redundancy Removal under Constraints

Using the framework we have developed in Section IV, we may compute the exact set of gates which are equivalent in the constrained reachable state space. In theory, this framework is capable of solving every unassertable target, since they correlate to gates which are semantically equivalent to 0. However, either due to computational resource limitations which result in suboptimal redundancy identification, or due to targets which are truly assertable, some targets may remain unsolved after the redundancy identification process.

If any targets remain unsolved after the redundancy identification process, it is generally desirable to leverage the identified redundancy to simplify the netlist so that a subsequent verification strategy may benefit from that simplification [10], [12], [17]. Examples of known algorithmic synergies which benefit from redundancy elimination include: faster and deeper

exhaustive bounded search using SAT; greater reduction potential through transformations and abstractions such as combinational rewriting, retiming, and localization reduction; and enhanced inductiveness [10], [17]. In this section, we discuss how to optimally leverage this identified redundancy.

It was demonstrated in [7] that a merge of gate g_1 onto g_2 is guaranteed to preserve property checking as long as no constraint c_1 in the combinational *or* sequential fanout of g_1 was used to constrain the state space during the proof of $g_1 \equiv g_2$. This implies that certain identified equivalent gates may be safely merged. However, this result is suboptimal since certain merges which do not adhere to this criterion may nonetheless be performed while preserving property checking. For example, industrial verification testbenches often arise which incorporate a large number of constraints and targets, even though some of the constraints may be unnecessary to establish the correctness of some of the targets. If the targets under verification are already unreachable (possibly due to a set of existing constraints $c_1, \ldots, c_i$), adding a constraint c_{i+1} need not preclude merges regardless of fanout connectivity since weakening c_{i+1} cannot cause spurious counterexamples. In practice, it is difficult to assess which of the identified redundancies is conditional upon which subsets of the constraints (e.g., vs. holding due to satisfiability don't cares alone), without performing numerous redundancy identification analyses with different subsets of constraints or per-miter minimal-proof analysis. Such processes are computationally expensive, motivating our technique to optimally leverage the redundancy identified using *all* constraints within a single efficient run of the algorithm of Figure 2.

The following theorem establishes that we may safely perform a greater degree of merging than enabled by [7]

Theorem 2: Consider any set of gate pairs which have been proven equivalent across all reachable states in netlist N, and whose merged *from* gates (vs. merged *onto* gates) do not lie in the combinational fanin of any constraint. Performing the corresponding merges to yield netlist N' will ensure that N' is trace-equivalent to N.

Proof: Similarly to Theorem 1, since no gates within the combinational fanin of constraints are merged onto others, their evaluation and hence their constraining power over registers and inputs cannot be altered. The initial states of N and N' are thus identical; any trace of length 1 in N is valid in N' and vice-versa. Furthermore, since the merges have been verified to reflect equivalence across the constrained reachable states, they cannot alter next-state function valuations. Thus, by simple inductive reasoning, any trace prefix of length $i+1$ in N' is also valid for N and vice-versa. ■

Theorem 2 enables us to perform a significantly greater degree of merging than enabled by the results of [7], particularly for highly cyclic netlists where the fanin of the constraints includes almost all gates. However, we note that this result is also suboptimal, since the combinational fanin of the constraints may be arbitrarily large, and there may still be verification-preserving merging possible therein. We now demonstrate that general constraint-enhanced merging is sound but incomplete.

Given: Netlist N' formed from N by merging equivalent gate pairs $G_f \mapsto G_t$, and traces $p'_1, \ldots, p'_i$ over N'
1. Simulate each p'_i on N to obtain trace p_i
2. If p_i asserts any target t in N, report that result as a valid counter-example and eliminate t from T, the targets of N
3. If T is empty, exit
4. Identify the set of merged nodes $D_1 \subseteq G_f$ of N' which differ in valuation across any p_i and p'_i within the prefix of p'_i, ignoring constraint violations in p_i
5. Construct netlist N'' from N, performing conditional merges for set D_1, and conjuncting each constraint gate with each target in N''
6. Cast a SAT problem conjuncting over each p'_i, checking for the lack of an assertion of targets in N'' under the input sequence of p'_i
7. Define causal merge set D_2 as those whose selector is assigned to 0 from the SAT solution
8. Form refined netlist N''' from N by merging $G_f \setminus D_2$

Fig. 4: Trace refinement algorithm

Theorem 3: Consider any set of gate pairs which have been proven equivalent across all reachable states in netlist N. Performing the corresponding merges to yield netlist N' will ensure that N' trace-contains N.

Proof: The fact that the gate pairs have been proven equivalent across all reachable states ensures that, within all trace prefixes which do not violate constraints in N, these merges do not alter the valuation of any gate whatsoever in N'. Thus, every constraint-satisfying trace prefix in N is also valid in N'. However, the fact that merging may be performed in the combinational fanin of the constraints means that their evaluation relative to registers and inputs may be altered. In particular, for extensions to valid trace prefixes in N – i.e., for time-frames $i+j$ (for non-negative j) where some constraint in N is violated at time i – the merges may alter the evaluation of gates in their fanout. In doing so, these merges may cause constraints in their fanout to evaluate to 1 in N' vs. to 0 in N. Thus, constraint-satisfying trace prefixes in N' may violate constraints in N. Netlist N' thus trace-contains N, but generally may not be trace-equivalent to N. ■

Theorem 3 implies that, if a target t' is proven as unassertable in N', the corresponding target t must be unassertable in N – though counterexamples from N' may be invalid on N. This theorem motivates an abstraction-refinement framework conceptually similar to [18], which retains those merges which do not entail spurious counterexamples for efficiency of a subsequent verification strategy, while discarding the others.

A. Abstraction-Refinement Framework

Definition 8: A *conditional merge* from gate g_1 onto gate g_2 consists of replacing g_1 by a multiplexor whose selector is driven by a newly-created nondeterministic constant[2] gate i_{g_1,g_2}. If the selector evaluates to 1, the value of g_2 is driven at the output of the multiplexor. Otherwise, the original value of g_1 is driven at the output of the multiplexor.

We present an algorithm in Figure 4 for identifying a set of merges (hereafter referred to as *causal merges*) to discard in response to a set of spurious counterexamples. Note that this algorithm is similar to those for automated

[2]A nondeterministic constant may be represented in a netlist by a register whose next-state function is itself (hence it never toggles), and whose initial value is a primary input.

design debugging [19]. The abstracted netlist N' is formed by merging each $g_f \in G_f$ onto the corresponding $g_t \in G_t$ as per the surjective mapping $G_f \mapsto G_t$. Step 1 of this algorithm maps each trace p_i' obtained from N' to trace p_i over N, from which we may assess the behavior of N under the input sequence demonstrated in p_i'. Step 2 checks if any resulting trace constitutes a valid assertion of any target in N. If so, that trace is reported as a counterexample and the corresponding target is eliminated from the set of unsolved targets T. If T becomes empty, then the verification problem has been solved hence the algorithm exits in Step 3. Otherwise, the algorithm proceeds to identify a set of merges which were responsible for the spurious counterexamples. Step 4 discerns an overapproximation D_1 of the set of causal merges, identifying those merged gates whose valuations differ between any p_i and p_i' during the trace-asserting prefix of p_i'. Set D_1 is next minimized in Steps 5-7 by casting a SAT problem which seeks to avoid target assertions under the input sequence of each p_i' within netlist N'' formed from N by performing *conditional merges* for the potentially causal merges. Note that the constraints are conjuncted with each target in N'' vs. being retained natively, so that the resulting SAT instance will be satisfiable and reflect the ability to preclude a target assertion due to constraint violations in N. Step 8 constructs a refined netlist, eliminating those merges which were determined to cause the spurious counterexamples in N'.

Theorem 4: Given trace p_i' which is a counterexample to target t_i' of netlist N', the algorithm of Figure 4 will either yield a valid counterexample p_i to target t_i of N, or produce a refined netlist N''' which will not exhibit a spurious assertion against the input stimuli of p_i'.

Proof: This theorem is trivially true if the algorithm produces a counterexample on N. Otherwise, we note that:

- When constructing D_1, gate inequivalence is checked for all time-frames in the prefix of p_i', which illustrates the spurious assertion of t_i'. This check ignores constraint-based prefixing within p_i to enable detecting *which* merges in N' weakened the constraints in N. Thus D_1 will include all merged gates whose behavior was altered in p_i' vs. p_i.
- The SAT problem is satisfiable, since if the selector of each conditional merge construct is set to 0, the SAT solution will be equivalent to p_i which has been demonstrated not to assert t_i in N.
- Set D_2 by construction enumerates the subset of merges which, if eliminated, will prevent the assertion of t''', the counterpart of t' in N'''. ■

As per Theorem 4, the refinement process of Figure 4 is adequate to ensure that the resulting netlist will not exhibit a spurious assertion against any counterexample used as the basis of the refinement. This result allows us to develop an abstraction-refinement framework which is guaranteed to converge as presented in Figure 5.

Theorem 5: The algorithm of Figure 5 (run in *standard abstraction-refinement* mode) will converge upon a correct verification result for the original netlist N.

Proof: We consider the individual steps of this algorithm.

Given: Netlist N; Initialize $i = 1$
1. Compute equivalence classes in N using a variant of the algorithm of Figure 2 as per Theorem 1, yielding desired merges $G_f \mapsto G_t$
2. Form N' by performing the subset of merges $G_f' \mapsto G_t$ which adhere to Theorem 2
3. Form N_1 from N' by performing the remaining merges $G_f'' \mapsto G_t$
4. Use an arbitrary set of algorithms to attempt to prove or falsify the targets in N_i
5. If any targets were proven unassertable on N_i, report the corresponding targets unassertable for N
6. If a nonempty set of counterexamples P_i were obtained on N_i, use Figure 4 on netlist N^* vs. N and trace set P^* to obtain a valid set of counterexamples and / or a refined netlist N_{i+1}, else exit
7. Report any valid counterexamples that were obtained for N
8. If unsolved targets remain, increment i and goto Step 4, else exit

Fig. 5: Abstraction-refinement framework. $N^* = N_i$ and $P^* = P_i$ in Step 4 implies *standard abstraction-refinement*; $N^* = N_1$ and $P^* = \bigcup_{j \in \{1,\ldots,i\}} P_j$ in Step 4 implies *optimal abstraction-refinement*

.

- Step 1 computes the equivalence classes of gates, which are correct as per Theorem 1. We may select an arbitrary set of desired merges consistent with these equivalence classes, reflected by surjective mapping $G_f \mapsto G_t$.
- Step 2 performs those subset of merges (denoted as $G_f' \mapsto G_t$) which are guaranteed to preserve trace-equivalence as per Theorem 2. Thus, if verification were performed directly upon N', all results would be correct with no need for refinement.
- Step 3 performs the remaining merges, denoted as $G_f'' \mapsto G_t$ where $G_f'' = G_f \setminus G_f'$, to yield abstract netlist N_1.
- Because N_i trace-contains N as per Theorem 3, any target proven unassertable in N_i implies a corresponding unassertable target in N, hence any results reported in Step 5 are correct.
- If counterexamples were obtained for N_i, they may be valid for N or they may be spurious. The algorithm of Figure 4 is used to differentiate these cases in Step 6. The result of this algorithm is a set of valid counterexamples for N and / or a refined netlist N_{i+1}.
- By Theorem 4, any counterexamples reported in Step 7 will be valid.
- By Theorem 4, if any targets remain unsolved, refined netlist N_{i+1} will not exhibit a spurious assertion against traces P_i. Convergence of the refinement loop is guaranteed noting that netlists are finite as per Definition 1, hence $|G_f''|$ is finite and each refinement iteratively eliminates one or more elements from G_f''. ■

VI. Optimality of Reductions

Theorem 5 ensures the correctness and convergence of the overall abstraction-refinement procedure. However, there are several points to consider regarding the *optimality* of the resulting refined netlist, which may have a significant impact on the resources required to compute counterexamples, as well as to prove targets unassertable.

i) While the SAT solution obtained from Step 6 of the algorithm of Figure 4 identifies an adequate set of causal merges for refinement, it does not directly attempt to obtain a solution with a *minimal* set of causal merges, as would be necessary for optimality of the refined netlist.

ii) Compatibility issues with don't-care enabled merging entail that two sets of merges may be independently but not jointly valid or vice-versa [20], [6], [27]. The optimal selection of causal merges may thus entail cumulative suboptimalities across refinement iterations, even if each individual iteration is optimal.

Regarding the first issue: for optimal reductions, one would wish to eliminate as few merges as possible during each refinement. A precise solution to this problem may be attained by solving a max-sat problem [21] in Step 6 of the algorithm of Figure 4, constructed from the original SAT instance augmented with an additional clause for each conditional merge construct representing the selection of the merged value. For enhanced runtime, we have found that a near-optimal initial bound to the max-sat solution may be obtained using a standard SAT solver augmented with a decision procedure which heuristically assigns 1 to primary inputs before assigning 0. Due to enforcing the input sequence of p'_i, note that primary inputs within the resulting SAT instance need only occur within conditional merge instances.

Regarding the second issue: to circumvent the risk that choices at a given refinement iteration will entail cumulative suboptimality across multiple iterations, it is necessary to re-compute refinements relative to the maximally-merged abstraction N_1. This is illustrated by the *optimal abstraction-refinement* mode of the algorithm of Figure 5, where at each refinement iteration i, *all* prior counterexamples $\bigcup_{j\in\{1,\ldots,i\}} P_j$ are used to refine relative to N_1, instead of merely using the final set P_i to refine N_i. The following theorems demonstrate the correctness and optimality of this flavor of the abstraction-refinement process.

Theorem 6: The algorithm of Figure 5 (run in *optimal abstraction-refinement* mode) will converge upon a correct verification result for the original netlist N.

Proof: The correctness of verification results follows from the proof of Theorem 5. To guarantee convergence, we note that we cannot have two identical refined netlists $N_i = N_j$ for $i > j$. This follows by Theorem 4 noting that N_i is formed by refining against *all* prior traces, including P_j which comprises one or more spurious counterexamples on N_j. Additionally, since netlists and hence G''_f are finite, there are a finite number of possible distinct refined netlists. ■

Theorem 7: Assuming that a max-sat procedure is used in the refinement algorithm of Figure 4, the algorithm of Figure 5 (run in *optimal abstraction-refinement* mode) will at every iteration yield an optimal refined netlist which retains as many of the desired merges as possible while not exhibiting a spurious assertion against any counterexample obtained prior to that iteration.

Proof: This proof follows trivially by Theorem 6 and by the construction of the max-sat formulation. ■

The validity of don't-care enabled merges is generally neither symmetric nor transitive [6]. Thus, the selection of desired merges from the computed equivalence classes in Step 1 of the algorithm of Figure 5 may impact the size of the refined netlist. Clearly optimality would be achieved if the algorithm of Figure 5 were run upon every permutation of desired merges consistent with the computed equivalence classes, and selecting the result which retains the most merges. However, doing so would be computationally intractable. This computational expense may be minimized as follows: while *causal* merges must be eliminated upon a refinement, we may attempt to introduce *alternate* merges from within the equivalence classes in their place. Such a framework may be used to generate a refinement retaining an optimal number of merges, provided that the procedure exhaustively attempt alternate merges before outright discarding any set of causal merges. Convergence of such an alternate-merge introduction framework may be guaranteed simply by ensuring that no refinement N_{i+j} repeat the same set of merges reflected in an earlier refinement N_i. While computationally superior to the naïve approach of exhaustive enumeration, since alternative merges need only be explored *on demand*, this approach is also prone to be computationally intractable in practice due to requiring the exploration of all cross-products of alternate merges for the identified causal merges. We thus introduce an efficient technique to generate a nearly-optimal set of desired merges from the equivalence classes in Section VI-A.

A. Incremental Elimination of Constraint-Weakening Merges

SAT-based analysis is often used to search for counterexamples, iteratively checking for failures until computational limits are exceeded. For efficiency, it is desirable to leverage *incremental SAT* in this process, first creating a SAT instance to check for a failure at time 0, then unfolding an additional time-frame onto the existing instance to check for a failure at time 1, etc. This incrementality enables the reuse of learned clauses from earlier time-frames to speed up the SAT solution for later time-frames [22].

In an abstraction-refinement framework, incrementality is more difficult to achieve across refinements. However, if using the *conditional merge* construct instead of outright performing the desired merges in Step 3 of the algorithm of Figure 5, incrementality may be achieved across refinements by merely constraining the causal conditional merge selectors to 0 as also noted in [23]. Such constraints effectively eliminate the causal merges within the SAT instance and allow additional counterexamples to be identified therein. We have found that devoting a small amount of computational resources to such an incremental SAT-based procedure, and using the resulting counterexamples to jump-start the optimal abstraction-refinement process, tends to substantially reduce the resources necessary to arrive at an optimal refined abstraction which is not prone to spurious counterexamples.

Furthermore, one mechanism that we have found to quickly converge upon a nearly-optimal set of compatible merges from within the equivalence classes (Step 1 of the algorithm of Figure 5) is to use the preprocessing mechanism discussed in the prior paragraph with a variant of the conditional merge which enables the selection among *all* gates within an equivalence class. As spurious counterexamples are obtained, we may disable the causal merges and preserve selection among the rest. The desired merges may then be chosen from those which remain at the termination of this preprocessing step, heuristically helping to ensure near-optimal compatibility.

Benchmark	Design Info	SimGen [13]		Sliding Window			SAT-only	
	Gates	Valid Time-Steps	Time (s)	Input Depth: Min; Max; Algo	Valid Time-Steps	Time (s)	Valid Time-Steps	Time (s)
FXU	32903	1	0.13	1; 2; 1	1000	2.6	165	1800
FPU	115037	5	0.39	5; 14; 10	1000	228	902	1800
SCNTL	51504	53	2.94	7; 14; 8	1000	87	72	1800
IBUFF	19230	57	1.05	5; 13; 6	1000	7.4	134	1800
AXU	345518	1	33.24	1; 2; 1	1000	44.15	1000	60.68
IBM_FV_11	4799	2	0.17	4; 8: 6	1000	2.6	337	1800
IBM_FV_24	13391	2	0.59	4; 19; 4	1000	3.2	252	1800

TABLE I: Constraint-preserving simulation results

Benchmark	Design Info		Constraint-Safe Merging [7]			Constraint-Enhanced Merging						
	Gates	Targets	Gates Merged	Unsolved Targets	Resources (s; MB)	Gates Merged	Refined Merges	Refinement # CEXs	CEX Max Depth	Improvement in # Merges	Unsolved Targets	Resources (s; MB)
FXU	32903	8	2218	0	450; 146	2482	62	7	8	9.1%	0	318; 195
FPU	115037	1	2022	1	5465; 690	4928	0	0	0	143.7%	0	1140; 384
SCNTL	51504	551	6638	24	342; 133	6962	4	1	19	4.8%	0	162; 383
IBUFF	19230	303	222	14	77; 91	831	0	0	0	274.3%	0	78; 160
AXU	345518	1	734	1	956; 479	4828	343	6	16	511.0%	1[3]	616; 540
IBM_FV_11	4799	1	228	0	16; 64	747	0	0	0	227.6%	0	34; 69
IBM_FV_24	13391	1	313	0	70; 119	793	13	1	22	149.2%	0	58; 137

TABLE II: Sequential redundancy removal results

VII. Experimental Results

We now provide experiments to illustrate the verification enhancements enabled by our techniques. All experiments were run on a 2.1GHz POWER5 processor, using *Sixth-Sense* [17]. We applied our techniques on a variety of designs, including the subset of the IBM FV Benchmarks [24] which have interestingly large constraint cones, and five diverse and difficult industrial testbenches. FXU is a testbench used to verify the control path of a fixed-point unit. FPU verifies the correctness of bypassing logic in a floating-point unit. SCNTL is a testbench used to verify the control of an instruction-dispatch unit. IBUFF is an instruction buffer. AXU checks the datapath correctness of an arithmetic unit. Each of these has constraints which entail dead-end states.

The experiments of Table I illustrate the power of the Sliding Window algorithm presented in Section III. The first two columns indicate the name of the benchmark and the size of the original netlist. We compare the results of: **(1)** our implementation of SimGen [13] (Columns 3-4) which performs purely combinational constraint solving; **(2)** our Sliding Window approach (Columns 5-7); and **(3)** purely formal analysis using SAT to solve the constraints for the entire duration of the simulation run, using random stimulus generation for unassigned inputs (Columns 8-9). Our goal is to simulate the designs without encountering dead-end states for 1000 time-steps, within a time-limit of 1800 seconds. While fast, SimGen [13] results in constraint violations within 57 time-steps for each design, and often substantially lesser. The SAT-only approach times-out for every design aside from AXU, often completing substantially lesser than 1000 time-steps. In contrast, our Sliding Window approach is able to complete the desired simulation for every design. The window depth used varies across the designs; in Column 5, we report the minimum and maximum input depth, followed by the depth algorithmically converged upon by our $\log_2$ range analysis.

The enhanced redundancy removal enabled by our approach is illustrated in Table II. The first 3 columns indicate the name of the benchmark, the size of the original netlist and the number of targets in the original netlist. We first compare the redundancy removal possible using prior techniques [7] with that of our approach (Columns 4 vs. 7), and then illustrate the impact of this additional reduction on the verification process by using k-induction [25] to attempt to verify the targets on the optimized designs (Columns 5 vs. 12). The resources reported in Columns 6 vs. 13 refer to the combined process of redundancy removal and induction. The induction process was limited to 30 seconds and $k \leq 10$ to avoid significant skew of runtime for cases where targets were left unsolved.

As illustrated in Table II, our approach identifies significantly more redundancy resulting in an increased number of non-refined merges, often more than $2\times$ (indicated by a number greater than 100% in Column 11). The average increase in non-refined merges across all of these designs is 187.8%. This reduction was essential to proving a number of these targets, which otherwise were not inductive and extremely computationally expensive to solve with alternate algorithms.[3] We report the number of merges which were refined during the algorithm of Figure 5 in Column 8 (which is a subset of Column 7), along with the number of counterexamples (CEXs) used during that refinement in Column 9 and the maximum depth at which a refinement occurred in Column 10. Note that only a small percentage of the additional merges enabled by our techniques (4.8%) must actually be refined; nonetheless, in 4 of 7 examples, spurious property failures would have occurred without our refinement process. Our approach in only two cases entails moderate additional run-time due to the larger set of equivalence candidates during the redundancy identification process, though in most cases, particularly the FPU, this results in significantly lesser runtime.

VIII. Related Work

There are similarities between aspects of our theory and prior work on proof decomposition in assume-guarantee reasoning, e.g. [3]. For example, our requirement that the speculatively-merged netlist be acyclic, and our combinational constraint-cone replication of Theorem 1, are closely related to

[3]The AXU target required additional transformations before it became tractable for proof analysis; these details are omitted due to lack of space.

the requirement that combinationally-dependent assumptions vs. properties be used only under a strict dependency relation. However, such prior work in assume-guarantee reasoning has not addressed the problem of automated derivation of equivalent-gate conditions, nor addressed the impact of using such conditions for direct netlist simplification.

The work of [7] discusses redundancy removal in the presence of constraints, allowing gate merges in the fanin of a constraint provided that the proof of the corresponding gate equivalence does not require the state-pruning power of that constraint. Our approach eliminates this suboptimality.

In [26], the authors propose to enhance an inductive SAT solver by performing *all* merges enabled by the constraints representing the induction hypothesis. To compensate for the resulting constraint weakening, they add additional constraints within the SAT solver. Unlike our approach, they do not address constraint-enhanced reduction of sequential netlists; theirs is effectively a run-time optimization to the inductive SAT solver, which our approach may complementarily use if relying upon induction in Step 4 of the algorithm of Figure 2.

There are similarities between our approach and those that optimize relative to other types of don't cares such as ODCs. While most scalable ODC-based techniques rely upon suboptimal local analysis for efficiency (e.g., [6]), the SAT-based technique of [27] uses induction to validate sets of ODC-enhanced merges, enabling unreachability invariants to enhance reduction potential. However, the equivalence boundaries against which the ODC conditions are validated are purely combinational, limiting optimality. The approach of [23] enhances interpolation by optimizing logic resulting from interpolant synthesis relative to don't cares implied by already-reached states. They use incremental SAT to justify that a set of gates may be safely merged to constants, using a construct similar to the conditional merge of Definition 8 so that invalid merges may be disabled within the SAT instance. Neither of these works (nor any others that we are aware of) address efficient yet globally optimal reduction of sequential netlists, leveraging constraints for increased reduction potential while preserving constraint evaluation. Though overall, ODC-based optimization and constraint-enhanced redundancy removal are complementary approaches, and it is a promising area of future research to pursue constraint-enhanced yet constraint-preserving extensions to such complementary techniques.

Constraint-satisfying stimulus generation has been extensively studied, e.g., [13], [14], though little focus has been given to dead-end states. The approach of [28] does address dead-end states, using a BDD-based framework to manipulate a synthesized constraint automaton to avoid dead-end states before they are reached. While demonstrated to be effective, this approach is only applicable if the constraints are specified using temporal logic to reason solely about primary inputs. If arbitrary gates are referenced by the constraints, as in our practical experience they often are, their approach becomes underapproximate, precluding the exploration of arbitrary reachable states. Our techniques do not suffer this limitation.

IX. Conclusion

We have developed a theoretical framework and an efficient implementation for netlist simplification in the presence of constraints. Our solution includes a robust and efficient algorithm for constraint-preserving random stimulus generation, a sound and complete extension to scalable *assume-then-prove* redundancy identification frameworks, and an efficient abstraction-refinement framework to optimally eliminate the identified redundancy for enhanced property checking.

References

[1] Y. Zhu and J. Kukula, "Generator-based verification," in *ICCAD*, Nov. 2003.

[2] C. Pixley, "Integrating model checking into the semiconductor design flow," in *Electronic Systems Technology & Design*, 1999.

[3] T. A. Henzinger, S. Qadeer, and S. K. Rajamani, "Decomposing refinement proofs using assume-guarantee reasoning," in *ICCAD*, Nov. 2000.

[4] C. Jacobi, K. Weber, V. Paruthi, and J. Baumgartner, "Automatic formal verification of fused-multiply-add FPUs," in *DATE*, March 2005.

[5] Accelera, *SystemVerilog Language Reference Manual*. http://www.systemverilog.org/.

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

[7] H. Mony, J. Baumgartner, and A. Aziz, "Exploiting constraints in transformation-based verification," in *CHARME*, Oct. 2005.

[8] C. A. J. van Eijk, "Sequential equivalence checking without state space traversal," in *DATE*, Feb. 1998.

[9] P. Bjesse and K. Claessen, "SAT-based verification without state space traversal," in *FMCAD*, Nov. 2000.

[10] H. Mony, J. Baumgartner, V. Paruthi, and R. Kanzelman, "Exploiting suspected redundancy without proving it," in *DAC*, June 2005.

[11] J.-H. Jiang and R. Brayton, "Retiming and resynthesis: A complexity perspective," in *TCAD*, vol. 25, Dec. 2006.

[12] A. Kuehlmann, "Dynamic transition relation simplification for bounded property checking," in *ICCAD*, Nov. 2004.

[13] J. Yuan, K. Shultz, C. Pixley, H. Miller, and A. Aziz, "Modeling design constraints and biasing in simulation using BDDs," in *ICCAD*, Nov. 1999.

[14] J. Yuan, K. Albin, A. Aziz, and C. Pixley, "Constraint synthesis for environment modeling in functional verification," in *DAC*, June 2003.

[15] S. A. Cook, "The complexity of theorem-proving procedures," in *ACM Symposium on the Theory of Computing*, May 1971.

[16] P. Bjesse and J. Kukula, "Automatic generalized phase abstraction for formal verification," in *ICCAD*, Nov. 2005.

[17] H. Mony, J. Baumgartner, V. Paruthi, R. Kanzelman, and A. Kuehlmann, "Scalable automated verification via expert-system guided transformations," in *FMCAD*, Nov. 2004.

[18] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Counterexample-guided abstraction refinement," in *CAV*, July 2000.

[19] M. F. Ali, A. Veneris, S. Safarpour, R. Dreshler, A. Smith, and M. Abadir, "Debugging sequential circuits using Boolean satisfiability," in *ICCAD*, Nov. 2004.

[20] A. Mishchenko and R. Brayton, "SAT-based complete don't-care computation for network optimization," in *DATE*, March 2005.

[21] D. S. Johnson, "Approximation algorithms for combinatorial problems," *Journal of Computer and System Sciences*, vol. 9, 1974.

[22] O. Shtrichman, "Pruning techniques for the SAT-based bounded model checking problem," in *CHARME*, Sept. 2001.

[23] G. Cabodi, M. Murciano, S. Nocco, and S. Quer, "Stepping forward with interpolants in unbounded model checking," in *ICCAD*, Nov. 2006.

[24] IBM Formal Verification Benchmark Library. http://www.haifa.il.ibm.com/projects/verification/RB_Homepage/fvbenchmarks.html.

[25] N. Eén and N. Sörennson, "Temporal induction by incremental SAT solving," in *Workshop on Bounded Model Checking*, 2003.

[26] F. Lu and K.-T. Cheng, "IChecker: An efficient checker for inductive invariants," in *HLDVT*, Nov. 2006.

[27] M. Case, V. Kravets, A. Mishchenko, and R. Brayton, "Merging nodes under sequential observability," in *DAC*, June 2008.

[28] E. Cerny, A. Dsouza, K. Harer, P.-H. Ho, and H.-K. T. Ma, "Supporting sequential assumptions in hybrid verification," in *ASPDAC*, Jan. 2005.

Recording Synthesis History for Sequential Verification

Alan Mishchenko **Robert Brayton**

Department of EECS, University of California, Berkeley

{alanmi, brayton}@eecs.berkeley.edu

Abstract

Performing synthesis and verification in isolation has two undesirable consequences: (1) *verification runs the risk of becoming intractable, and* (2) *strong sequential optimizations are not applied because they are hard to verify. This paper proposes a format for recording synthesis information and a methodology for sequential equivalence checking using this feedback from synthesis. An implementation is described and experimentally compared against an efficient general-purpose sequential equivalence checker that does not use synthesis information. Experimental results confirm expected substantial savings in runtime and reliability of equivalence checking for large designs.*

1 Introduction

In this paper, we propose a methodology, which records and uses the synthesis history in an efficient and transparent way. It can enable a scalable sequential verification of the result and hence promote the use of sequential synthesis.

Although sequential synthesis can result in considerable reductions in delay (e.g. see [22]) and area; it is mostly avoided for reasons of non-scalability of both synthesis and verification. To circumvent this, we believe that sequential synthesis and verification must go hand-in-hand to make sequential synthesis acceptable, and propose a way to make this happen.

General sequential equivalence checking (GSEC) of two FSMs is PSPACE complete, but certain restrictions on the synthesis transformations allowed are known to make the problem easier. For example, if synthesis is restricted to one series of combinational transformations followed by one retiming or vice versa, the problem is provably simpler. However, iterating retiming and combinational transformations makes the problem again PSPACE complete [14]. Also, problems become simpler if there are structural similarities between the two circuits to be compared. For example, combinational equivalence checking (CEC) [20], although co-NP complete, in practice becomes much simpler in such cases.

This paper has the closest similarities with the following two approaches in the literature. Van Eijk [12] derived an inductive invariant, constructed by a fixed point process, consisting of a set of equivalences between signals in the two circuits under comparison. This invariant characterizes a superset of the reachable states of the product machine. Bjesse [6] and Case [9] extended this to an invariant composed of implications, which can give tighter approximations.

Such methods are dependent on the particular implementation structures of the two machines being compared because equivalences or implications can be stated only between existing signals. To overcome this limitation, Van Eijk proposed creating additional signals, without any fanout, which might be useful in establishing additional equivalences. His proposal involved adding a few nodes which could be obtained by retiming. These signals can help to approximate the reachable state space, thereby simplifying SEC, but do not guarantee that the invariant derived is sufficient to prove sequential equivalence.

Mneimneh et. al. [26] looked at the problem of one retiming and one set of combinational logic transformations (in either order) and proposed a retiming invariant composed of a conjunction of functional relations among latch values derived from atomic retiming moves.

More generally, several methods to keep track of synthesis steps and use them for verification have been proposed. Typically, these record the name and sequence of the transformations used; thus they are very specific to the software synthesis tool used. In contrast, our method is universal and can be adapted to any synthesis package. It produces a byproduct that can be verified independently.

We address the problem when one machine is derived from the other by a sequence of very general synthesis transforms, which may include retiming, combinational synthesis, merging sequentially equivalent nodes, and performing window-based sequential synthesis with don't-cares. We propose to record the synthesis historyin a special way, which will provide the extra signals to aid verification. In contrast to van Eijk, our history aided verification approach (HSEC) has the following characteristics:

- All nodes created during synthesis are recorded, instead of adding a set of ad-hoc signals.
- Each synthesis step records a sequential equivalence relation that *should* hold if the implementation of the synthesis algorithm is correct. A side benefit is that if an equivalence does not hold, the implementation must be incorrect and the source of the error in the software can be located.
- The invariant that exists in our method is the set of all equivalences recorded.
- This invariant is sufficient to prove sequential equivalence of the two machines by induction without counter-examples.
- The invariant can be verified easily by proving each equivalence, one at a time. Typically, the proofs are local and hence fast, and can be done in parallel.

Section 2 surveys the background. Section 3 shows how to efficiently record the history of synthesis by integrating two AIG managers. Section 4 details the use of the recorded history in sequential verification. Section 5 discusses other uses of a recorded history. Section 6 reports experimental results and Section 7 concludes the paper and outlines future work.

2 Background

2.1 Boolean Networks

A *Boolean network* is a directed acyclic graph (DAG) with nodes corresponding to logic gates and directed edges corresponding to wires connecting the gates. The terms Boolean network and circuit are used interchangeably in this paper. If the network is sequential, the memory elements are assumed to be D-flip-flops with initial states. Terms memory elements, flop-flops, and registers are used interchangeably in this paper.

A node *n* has zero or more *fanins,* i.e. nodes that are driving *n*, and zero or more *fanouts*, i.e. nodes driven by *n*. The *primary inputs* (PIs) are nodes without fanins in the current network. The

primary outputs (POs) are a subset of nodes of the network. If the network is sequential, it contains registers whose inputs and output are treated as additional PIs/POs in combinational optimization and mapping. It is assumed that each node has a unique integer called its *node ID*.

A *fanin* (*fanout*) *cone* of node *n* is a subset of all nodes of the network reachable through the fanin (fanout) edges from the given node. A *maximum fanout free cone* (MFFC) of node *n* is a subset of the fanin cone, such that every path from a node in the subset to the POs passes through *n*. Informally, the MFFC of a node contains all the logic used *exclusively* by the node; when a node is removed, the logic in its MFFC can be removed.

Merging node *n* onto node *m* is a structural transformation of a network that transfers the fanouts of *n* to *m* and removes *n* and its MFFC. Merging is often applied to a set of nodes that are proved to be equivalent. In this case, one node is denoted as the *representative* of an equivalence class, and all other nodes of the class are merged onto the representative. The representative can be any node whose transitive fanin cone does not contain any other node of the same class. In this work, the representative is the node of the class that appears first in a topological order.

There are different forms of *sequential equivalence* for FSMs [27]. We use the traditional notion where two FSMs are equivalent if they produce the same output sequences for the same input sequence starting from their two initial states.

2.2 And-Inverter Graphs

A combinational *And-Invertor Graph* (AIG) is a Boolean network composed of two-input ANDs and inverters. *Structural hashing* of AIGs ensures that, for each pair of nodes, all constants are propagated and there is at most one AND node having them as fanins (up to permutation). Structural hashing is performed by one hash-table lookup when AND nodes are created and added to an AIG manager. When an AIG is incrementally rehashed, the changes are propagated to the fanouts, which may lead to rehashing large portions of AIG nodes.

The *size* (area) of an AIG is the number of its nodes; the *depth* (delay) is the number of nodes on the longest path from the PIs to the POs. The goal of AIG optimization by local transformations of an AIG is to reduce both area and delay.

Sequential AIGs add registers to the logic structure of combinational AIGs. The registers are technology-independent D-flops with one input and one output that are assumed to belong to the same clock domain. Previous work on sequential AIGs [2][7] applies on-the-fly forward retiming to the registers along with the combinational structural hashing of the AIG nodes.

However, in this paper, we use *simplified sequential AIGs* where registers are represented traditionally as additional terminal nodes of the AIG. An additional data-structure identifies the I/O pair associated with a register's input and output. The PIs and register outputs are called *combinational inputs* (CIs) and the POs and register inputs are called *combinational outputs* (COs). Although mostly representing the combinational logic, simplified sequential AIGs are still suitable for sequential transformations. For example, for retiming, the operation is decomposed into individual register moves. Each move adds new registers to the register boundary while the old registers are removed.

We assume that the registers have a fixed binary initial state[1]. If a register has an unknown or a don't-care initial state, it can be transformed to have 0-initial state by adding a new PI and a MUX controlled by a special register that produces 0 in the first frame and 1 afterwards.

[1] For a good motivation of this restriction for industrial designs see [3].

2.3 SAT Sweeping and Induction

Combinational SAT sweeping is a technique for detecting and merging nodes that are equivalent up to complementation in a combinational network [15][17] [19][20]. SAT sweeping is based on simulation and Boolean satisfiability. Random simulation is used to divide the nodes into candidate equivalence classes. Next, each pair of nodes in each class is considered in a topological order. A SAT solver is invoked to determine the status of their equivalence. If the equivalence is disproved, a counter-example is used to simulate the circuit, which may result in disproving other candidate equivalences. SAT sweeping can be used as a robust combinational equivalence checking technique and as a building block in *k*-step induction [6].

Bounded model checking (BMC) uses Boolean satisfiability to prove a property true for all states reachable from the initial state in a fixed number of transitions (*BMC depth*). In the context of equivalence checking, BMC checks pair-wise equivalence of the outputs of two circuits to be verified. BMC can be implemented as a combinational SAT sweeping applied to several unrolled timeframes with initial state applied in the first frame.

k-step *induction* over time-frames is a method for proving sequential properties, such as sequential equivalence of two nodes in the network [12]. A property or a set of properties are proved inductively if the following two cases hold:

- **Base Case:** The properties hold true for all inputs in the first *k* frames starting from the initial state.
- **Inductive Case:** If the properties are assumed to be true in the first *k* frames starting from *any* state, then they hold in the $k+1^{st}$ frame.

A SAT-based *inductive prover* [6] is based on simulation and combinational SAT sweeping [20]. Speculative reduction [25] is a key ingredient of an efficient inductive prover because it reduces the runtime by several orders of magnitude and allows sequential SAT sweeping to work for large industrial design. Basically, it uses the simple device of moving all fanouts of a set of candidate equivalent nodes to one representative of the class.

Sequential SAT sweeping is similar to combinational SAT-sweeping, except that it detects and merges sequentially equivalent nodes[2]. In general, combinationally equivalent nodes are also sequentially equivalent, but not vice versa. Thus, it is helpful to apply combinational SAT sweeping before sequential sweeping. The implementation of sequential SAT sweeping uses *k*-step induction and an efficient implementation makes use of a SAT-based inductive prover.

3 Recording Synthesis History

In this paper, we discuss sequential equivalence checking when the synthesis transformations are limited to the following:

1. any combinational synthesis transformation,
2. retiming, both forward and backward
3. any window-based transformation
4. transformations involving observability don't cares
5. sequential SAT sweeping

We first discuss the proposed format for recording synthesis history and then discuss how each of the above transformations can be recorded using this format.

3.1 History Format

AIGs are used increasingly in CAD tools as a unifying data structure for applications dealing with logic synthesis and formal verification. As a circuit representation, AIGs provide uniformity, fast manipulation, low memory requirements, straight-forward

[2] The nodes are *sequentially equivalent* if they compute the same value, up to complementation, in all states reachable from the initial state.

construction for both logic networks and mapped netlists, and the possibility of combining them with efficient simulators and SAT solvers, leading to a *semi*-canonical representation that can replace BDDs in many applications [19].

In the context of AIG-based synthesis, recording synthesis history can be done using two AIG managers: a Working AIG (WAIG) to represent the current state of the synthesis, and a History AIG (HAIG) to save all AIG nodes created by synthesis.

The following rules, which are the standard ones, are used in manipulating a WAIG:

- New logic nodes are added as synthesis proceeds.
- Old logic cones are periodically replaced by new logic cones. When this happens, (a) the old root node is replaced by the new root node, and (b) the fanouts of the old root are transferred to be fanouts of the new root.
- Nodes without fanout (such as the old root) are immediately removed. This helps maintain accurate metrics (node count, logic depth, etc)

The following rules are followed for a HAIG:

- New logic nodes are added as synthesis proceeds.
- Each time a new node is created in the WAIG, a corresponding node is either found or created in the HAIG, **and** a link between the two nodes is established using procedure **setWaigNodeMapping**.
- Old nodes are **not** removed and fanouts are **not** transferred.
- When a node replacement is performed in the WAIG, the two corresponding nodes in the HAIG are **linked** (indicating that they *should* be sequentially equivalent) using procedure **setHaigNodeMapping.**

Thus two node mapping are supported in a WAIG / HAIG pair:

- Each WAIG node points to a corresponding HAIG node, which was created when the WAIG node was created.
- Some of the HAIG nodes point to other HAIG nodes. This node mapping is created between the corresponding HAIG nodes when a WAIG node is replaced by another WAIG node. The resulting pair of HAIG nodes should be sequentially equivalent if synthesis is correct. These equivalences will be proved during HAIG-based verification, as described in Section 4.

Table 1 establishes a correspondence between the AIG procedures of the WAIG and HAIG. These are the only ones needed for implementing any sequential synthesis algorithm.

Table 1. Relation between WAIG and HAIG procedures.

Working AIG	History AIG
aigManagerCreate (the first call)	**aigManagerCreate**
aigManagerCreate (other calls)	do nothing
aigManagerDelete (other calls)	do nothing
aigManagerDelete (the last call)	**aigManagerDelete**
aigNodeCreate	**aigNodeCreate** and **setWaigNodeMapping**
aigNodeReplace	**setHaigNodeMapping**
aigNodeDelete_recursive	do nothing

The first four lines of Table 1 describe what happens when the WAIG is created and deleted. At the first creation of WAIG, the HAIG manager is created also. On subsequent duplications of the WAIG, the HAIG is unchanged, but the CIs/COs of the new WAIG are remapped to point to the CIs/COs of the HAIG. On the last deletion of any associated WAIG, its HAIG is deleted also.

When a WAIG node is created, a corresponding HAIG node is created and put in correspondence with the WAIG node. When one WAIG node replaces another WAIG node, nothing is done in the HAIG, except establishing the mapping between the corresponding HAIG nodes. Finally, when a WAIG node is recursively deleted, the HAIG remains unchanged.

3.2 Recording Combinational Synthesis

Recording the history during combinational synthesis involves three steps shown in Figure 3.1. First, logic cone *A* is re-synthesized, and a new logic cone *B* is constructed. Note that at this point *B* has no fanouts. Both cones are present in both the WAIG and HAIG because creating a new WAIG node always results in creating a matching HAIG node. Second, the fanout of logic cone *A* is transferred to logic cone *B* in the WAIG. The HAIG is unchanged, except the mapping (indicating equivalence) is established between the old root and the new root in the HAIG. Finally in the WAIG, logic node *A* is removed and subsequent new logic may be constructed in the WAIG on top of the new logic cone. No nodes are removed from the HAIG. Subsequent new logic is constructed in the HAIG on top of a new logic cone.

3.3 Recording Retiming

Retiming [16] can be decomposed into forward and backward retiming. Each of these retimings can be decomposed into atomic register moves. An atomic move involves transferring registers forward or backward over one AIG node. In forward retiming, the initial state of the new register is trivial to compute. In backward retiming, the initial state is typically computed by formulating a SAT instance. If the SAT instance is satisfiable, the computed initial state is assigned to the new register.

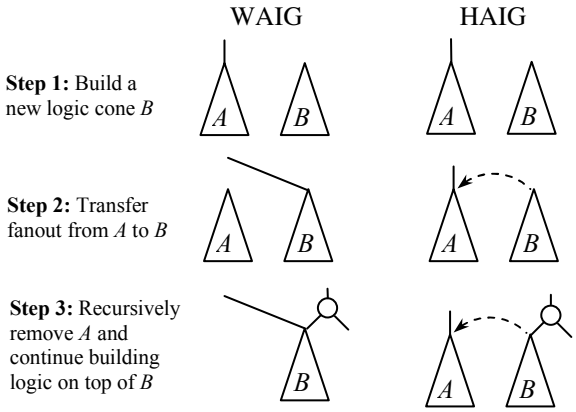


Figure 3.1. Example of history recording in WAIG and HAIG.

Individual register moves are recorded similarly to recording combinational synthesis. In this case, the role of the combination logic cones *A* and *B* is played by the AIG node before and after retiming, as shown in Figure 3.2. Note that, in the case of retiming, the equivalence pointers in the HAIG connecting *A* and *B* are "asserting" sequential equivalence. Also, note that sequential transformations, like retiming can create new registers which create new CIs / COs pairs in the HAIG.

3.4 Recording Window-Based Transformations

To ensure scalability, some synthesis transformations are applied to a node or a group of nodes in the context of a *window* rather than the whole network. A window is computed using a set of user-specified parameters, such as limit on the numbers of levels of logic to be included on the fanin/fanout sides of the node(s), the window size, and the presence and length of

reconvergent paths or sequential loops subsumed in the window. For a overview of windowing, see [23].

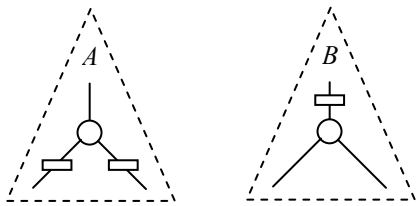


Figure 3.2. Logic cones for one forward retiming move.

For window-based transforms the key is to record the whole logic structure of the window after the transform and only assert, in the HAIG, sequential equivalence of the window's outputs before and after the transformation. Corresponding internal nodes may not be equivalent if don't cares were used.

3.5 Recording Transformations Involving ODCs

Combinational or sequential synthesis may involve the use of observability don't-cares computed for a node or a group of nodes. In this case, nodes after synthesis may have different Boolean functions in terms of the CIs. Such nodes cannot be recorded as equivalent to the original ones in the HAIG. However, for the computation of ODCs to be scalable, there always exists a scope, in which the functionality is preserved. This may include a window, a timeframe, or the whole sequential circuit. In all cases, the primary outputs of the scope should be sequentially equivalent before and after the ODC-based synthesis, and can be recorded as in the case of windowing.

3.6 Recording Sequential SAT Sweeping

When a circuit is transformed by sequential SAT sweeping (SSW), the nodes belonging to an equivalence class are merged onto the class representative. Typically, SSW computes many equivalences (or inverted equivalences) at once. In the implementation, the classes are computed first and then the AIG is duplicated while substituting (in a corresponding polarity) the representative for each node in the equivalence class. The pseudo-code of this procedure is shown in Figure 3.5.

New HAIG nodes are created inside procedure **aigAnd**. The mapping of new HAIG nodes into equivalent old HAIG nodes is set by the procedure **setHaigNodeMapping.** This is the same procedure that is called inside **aigNodeReplace**. The pseudo-code is listed to clarify exactly how this is done.

4 Using the HAIG for Verification

A history AIG (HAIG) is an AIG containing (in AIG form) the original version of the design, the final one, and all the intermediate logic derived during synthesis. Without the set of recorded equivalences, It is a sequential circuit in every sense (e.g. an initial state for every register), but with a lots of redundancy. Sequential verification of the original against the final one can be performed by proving equivalence of all candidate pairs of HAIG nodes recorded during synthesis.

4.1 Theory

Definition: Unlike combinational synthesis, a window in sequential synthesis can cross the register boundary several times. The *sequential depth* of a window-based sequential synthesis transform is the largest number of registers on any path from an input to an output of the window.[3] Proofs of the following theorems are omitted for space considerations.

Theorem 1: If transforms recorded in a HAIG have sequential depth less or equal to *k*, the equivalence classes of the HAIG nodes can be proved by *k*-step induction.

Theorem 2: If the inductive proof of the candidate equivalences in a HAIG passes (no counter-examples), then all synthesis steps have been performed correctly (which implies that the original design and final design are sequentially equivalent).

The proof of Theorem 1 is straightforward. The formal proof of Theorem 2 can be found in [7].

```
node aigNodeCopyWithEquivalences( aig B, node n, classes C ) {
    // if n is already visited, return its copy
    if ( n->copy != NULL )
        return n->copy;
    // if n belongs to an equivalence class, return its representative
    r = getRepr( C, n );
    if ( r != NULL ) {
        if ( n has the same polarity as r )
            return r;
        else
            return aigNot(r);
    }
    // solve the problem for fanins of n
    m0 = aigDuplicationWithEquivalences( B, n->fanin0, C );
    m1 = aigDuplicationWithEquivalences( B, n->fanin1, C );
    // create the copy, save it in the node, and return
    n->copy = aigAnd( m0, m1 );
    setHaigNodeMapping( n, n->copy );
    return n->copy;
}
aig aigCopyWithEquivalences( aig A, classes C ) {
    // start the new AIG manager
    aig B = aigStart();
    // clear the copy pointers for all nodes in the old AIG
    for each object n of aig A
        n->copy = NULL;
    // create combinational inputs and make old nodes point to them
    for each combinational input n of aig A
        n->copy = createCi( B );
    // recursively construct the AIG from the combinational outputs
    for each combinational output n of aig A
        aigNodeCopyWithEquivalences( B, n, C );
    return B;
}
```

Figure 3.5. Copying AIGs while merging equivalent nodes.

4.2 Implementation of Verification

The verification engine should be completely independent of the synthesis engine. In our proposal, the only connection between synthesis and verification is the HAIG, which is a set of suggested equivalences in an AIG. The verification needs only a simple independently-written inductive prover outlined in Figure 4.2. It is used to verify the candidate equivalences recorded in the HAIG, and is much simpler than the general-case prover [12][6][25][18] because it does not rely on the detection and refinement of candidate equivalence classes.

This prover makes use of speculative reduction [25], resulting in substantially reduced runtime. There is no need for iterative refinement of the equivalence classes because, if synthesis was performed correctly, counter-examples are never produced. If a counter-example is detected, the ID of the corresponding synthesis transform can be returned for help in debugging the synthesis code. We note that even the k^{th} copy used in *k*-induction

[3] Currently, loops in a window are not allowed, but this does not seem to be a necessary restriction.

can be speculatively reduced. Further, each equivalence in this copy can be solved in parallel.

```
status inductiveVerification( aig HAIG, int k ) {
    // run BMC for k-1 initialized timeframes
    status = performBMC( HAIG, k-1 );
    // return the status of sequential verification after BMC
    if ( status == "encountered a counter-example" )
        return ID of the synthesis transform that failed BMC;
    // do speculative reduction for k uninitialized timeframes
    aig HAIG_SR = speculativeReduction( HAIG, k-1 );
    // derive SAT solver containing CNF of the reduced timeframes
    solver S = transformAIGintoCNF( HAIG_SR );
    // check candidate equivalences in k-th timeframe
    status = performSatSweepingWithConstraints( S, HAIG );
    // return the status of sequential verification after SAT sweeping
    if ( status == "encountered a counter-example" )
        return ID of the synthesis transform that failed induction;
    return "equivalence check succeeded";
}
```

Figure 4.2. Simple inductive prover to verify a HAIG.

It is significant that the prover that can used in verification of the HAIG can be so simple because as mentioned, it share any code with synthesis, otherwise the same bug occuring in both and may be non-observable.

Memory requirements for a general AIG manager are roughly 32 bytes per AIG node stored. However, a HAIG can get by with only 8 bytes per node. The largest benchmarks in the set had about 20K AIG nodes. Assuming two copies of the circuit stored in a HAIG, yields 2 * 20,000 * 8 = 320Kb. AIGs also lead to significant compaction as shown in the program AIGER [5]. The runtime of HAIG recording is negligible.

5 Other Uses of a HAIG

A HAIG can be used in several other applications, e.g., to improve the quality of technology mapping or to perform incremental changes to netlists after physical design (ECO).

Using synthesis history to overcome structural bias inherent in cut-based structural mapping leads to substantial improvements in delay and area [8]. It was shown that further iterating HAIG-based synthesis and tech-mapping tends to gradually improve the quality of mapping. This happens because the logic structure of the AIG after each iteration of mapping is recorded in the HAIG, and the AIG is gradually synthesized to be compatible with the implementation technology. In [22], it was shown that sequential mapping combining technology mapping and retiming [28] can be extended to use the HAIG similarly.

Another application could be design debugging after physical synthesis, which requires tracing some logic gates back to the lines of the original HDL code, which produced them. For such application, additional APIs would allow the designer to use the HAIG to efficiently iterate through the synthesis steps forward or backward, and trace the dependence of a node in the final AIG to the original source code. Another application may explore the impact of a particular synthesis transform on the final result and possibly incrementally undo that transform to improve the result.

6 Experimental Results

History recording and HAIG-based sequential verification have been partially implemented in ABC [4]. The SAT solver used is a modified version of MiniSat-C_v1.14.1 [10]. The benchmarks used are 20 largest public circuits from the ISCAS'89, ITC'97, and Altera QUIP benchmark suites [1]. The runtimes were measured in seconds on a workstation with two dual-core AMD Opteron 2218 CPUs with 16GB RAM, and runs x86_64 GNU/Linux. Only one core was used in the experiments.

The synthesis included three iterations of balancing, rewriting, and retiming. Balancing is algebraic AND-tree restructuring for minimizing delay. Rewriting stands for one pass of AIG rewriting [21]. Finally, retiming consists of a fixed number of steps of forward retiming. (In the reported experiments, at most 3000 retiming moves were allowed in each iteration. History recording during backward retiming was not reported in these experiments since it had not been implemented yet.)

This script was selected to ensure that synthesis involved several iterations of combinational synthesis and retiming, resulting in networks that are usually difficult to verify, according to [14], by GSEC.

The results of synthesis are shown in Table 1. The three sections of this table show the statistics for the original, final, and HAIG networks respectively. The parameters reported are the number of primary inputs (column "PI"), primary outputs (column "PO"), registers (columns "Reg"), AIG nodes (columns "Node"), and AIG levels (columns "Lev"). The runtime of synthesis is shown in the last column of Table 1.

The results of synthesis were verified with and without using the HAIG. Verification with the HAIG used the approach described in Section 4. Verifying without the HAIG was done by a general-purpose sequential equivalence checking engine [24], which performs a sequence of simplifying transformations, including register sweep, retiming, combinational synthesis, SAT sweeping, register and signal correspondence, etc.

The results of verification are shown in Table 2. The first section shows the statistics of using two time-frames of the HAIG for verification. Since after unrolling, the time-frames are a combinational circuit, listed are only the number of AIG nodes (column "Node") and the number of AIG levels (column "Lev").

The second section shows the number of equivalences enforced in the first timeframe (column "Constr") and the equivalences checked in the second timeframe (column "Property") as well as the total number of equivalences in the HAIG (column "Total"). The first two numbers are less than the total number of node pairs because speculative reduction [25][18], which was used when unrolling the HAIG, makes some equivalences redundant.

The third section of Table 2 shows the parameters of the CNF from the two timeframes of the HAIG using efficient AND-to-CNF conversion [11]. The last section shows the runtimes of SAT-based verification using the HAIG (column "HSEC") and of the general-purpose SEC (column "GSEC") in ABC (command *dsec* [24]). Entry 1000+ indicates a timeout at 1000 seconds.

The last lines in Tables 1-2 list geometric averages of the corresponding parameters. The examples that timed out were given a time of 1000 in computing the runtime ratios.

6.1 Discussion

We discuss the results in the tables with regard to a) size of the HAIG, b) speed of verification, and c) reliability of verification.

HAIG size: To discuss the size of the HAIG, note that it contains both the original and final versions of the design in AIG form. Their total is 1.77 while the HAIG size is 5.13. Thus, on average, the HAIG was about 3x larger (in terms of AIG nodes) than a miter of the circuit that would be created for SEC. While the experiments represent only a medium synthesis effort, the fact that AIGs can be stored in a very compact form suggests that memory blowup during HAIG recording is not going to be a problem (e.g. AIGER [5] uses on average 3 bytes to represent one AIG node). To illustrate this, suppose a huge design, say 10M AIG nodes, on which extensive synthesis was done, and suppose that the HAIG has 200M AIG nodes. The HAIG can be stored

using 16 bytes per node or 3.2G bytes. But using the AIGer format would require only 600M bytes, and compression on the AIGER format, which usually gives a 10X compression, would require only 60M bytes.

Verification: Verification using the HAIG (HSEC) ranged from over 600 times faster, to 4.4 times slower than the general-purpose SEC (GSEC), with an average speed up of 4.59x on the 20 examples (using 1K sec on the 25% that timed out during GSEC) . On five of the examples, GSEC was actually faster than HSEC. We speculate that this is due to GSEC using heavy but scalable pre-processing: min-register retiming, structural sweep, and register correspondence. If this fast pre-processing can reduce or already solve the sequential miter, then general-case SEC does not take much time. HSEC became slower when there were many properties to verify, which was generally due to recording retiming one move at a time. Each gate, over which a register moves, causes an equivalence to be generated and checked later. A possible future investigation would be to see if only recording the equivalences at the final register positions would be sufficient. In addition, we reiterate that HSEC can be formulated so that each property can be checked completely in parallel,

Reliability: 25% of the examples timed out during GSEC, while none timed out during HSEC, although the largest example, *raytracer,* with over 13K registers, took 800 seconds by HSEC. This percentage of time-outs is likely to increase in experiments where heavier synthesis is applied, such as sequential SAT sweeping, min-register retiming, use of reachability don't cares, etc. This is because GSEC is PSPACE-complete. In contrast, HSEC is NP-complete because it is reduced to SAT (Theorem 1).

7 Conclusions and Future Work

We proposed a transparent synthesis process, which efficiently records the history of synthesis transformations. We showed how this history can simplify sequential verification. We proposed a simple format for storing a history as an AIG and described how this can be done easily by orchestrating computations in two related AIG managers. Finally, we demonstrated that the use of a history usually leads to savings in the runtime for sequential verification, compared to the runtime of an efficient general-purpose equivalence checker. More importantly, it leads to a reliable and rugged method for SEC, which is guaranteed to always complete.

Typical questions and concerns about a history-based sequential verification process are:

1) Can't incorrect information be passed inadvertently from the synthesis tool to the verification tool?
2) Might the same bugs in the synthesis tool also exist in the verification tool, thereby cancelling each other out and leading to false positives?
3) Won't the memory required to record the history explode on large examples?
4) If a synthesis tool does not use AIGs can one still use this methodology?

First, we again emphasize that the synthesis history is used simply as a set of hints for verification. Every step recorded in the history must be proved, and should be proved using a *different* prover compared to the one used in synthesis. Fortunately the inductive prover needed in HSEC is much simpler than in GSEC because induction for a HAIG should succeed without counter-examples. A simple HAIG prover in ABC is only about 100 lines of code (not counting the AIG package and the SAT solver), which is much more than about 2000 lines of code needed to implement a general-case inductive prover. The simplicity of the HSEC prover makes it easy to debug and more reliable. Also, at 8 bytes per node, memory requirements for a HAIG are very light, can be compacted significantly, and can be stored on disk without cache interference during history recording. Finally, we envision a history package based on AIGs which is a stand-alone module and can be called by any synthesis tool.

Also, the absence of counter-examples ensures fast and reliable runtimes of the HSEC solver. This is supported by experimental results, although there are cases where GSEC solver can be faster. Mostly, a GSEC prover for large industrial circuits is much slower because of the runtime spent generating and simulating counter-examples, and refining the equivalence classes. For HSEC, a counter-example would be extremely rare, but extremely useful to identify an incorrect synthesis transformation.

The speed of HSEC is helped because speculative reduction effectively reduces the HAIG to a single copy of the original circuit, except for the additional signals that are necessary to state the equivalences. In other words, if these signals were removed, the HAIG will collapse to a single copy of the original circuit. Even in the last, k^{th} timeframe, the circuit can be speculatively reduced. For further speed, all equivalences can be proved in parallel and in the rare case that one does not hold, the first one in topological order identifies a bug in the synthesis code. This is sufficient for debugging the synthesis code.

Although we have not explored other ways of recording synthesis history, the use of AIGs seems to provide an elegant method for doing this. AIGs are becoming increasingly accepted in both synthesis and verification communities, efficient AIG packages are being developed and improved, and AIGs are being used as an intermediate format for circuit logic representation.

Future work in this area will include:

- Completing the HAIG implementation in ABC to include all synthesis transformations; in particular, backward retiming, sequential SAT sweeping, and window-based transforms, such as re-encoding, ODC-based resynthesis.
- Polishing the HAIG interface and releasing it as a stand-alone package for the use in non-AIG-based synthesis tools.
- Conducting extensive experiments on industrial benchmarks while recording long sequences of synthesis transforms.
- Exploring the potential of using a partial HAIG. In particular, (a) developing methods to record a minimal history needed to ensure inductiveness and (b) investigating if only partial history information can be used to speed up the general-case SEC.

Acknowledgements

This work was supported in part by SRC contracts 1361.001 and 1444.001, NSF grant CCF-0702668 entitled "Sequentially Transparent Synthesis", and the California MICRO Program with industrial sponsors Actel, Altera, Atrenta, Calypto, IBM, Intel, Intrinsity, Magma, Synopsys, Synplicity, Tabula, and Xilinx. The authors are indebted to Jin Zhang for her careful reading and useful suggestions in revising the manuscript.

References

[1] Altera Corp., "Quartus II University Interface Program", www.altera.com/education/univ/research/unv-quip.html

[2] J. Baumgartner and A. Kuehlmann, "Min-area retiming on flexible circuit structures", Proc. ICCAD'01, pp. 176-182.

[3] J. Baumgartner, H. Mony, V. Paruthi, R. Kanzelman, and G. Janssen. "Scalable sequential equivalence checking across arbitrary design transformations". *Proc. ICCD'06*.

[4] Berkeley Logic Synthesis and Verification Group. *ABC: A System for Sequential Synthesis and Verification*. Release 70930. http://www-cad.eecs.berkeley.edu/~alanmi/abc

[5] A. Biere. *AIGER Format*. http://fmv.jku.at/aiger/

[6] P. Bjesse and K. Claessen. “SAT-based verification without state space traversal”. *Proc. FMCAD'00.* LNCS, Vol. 1954, pp. 372-389.
[7] R. Brayton and A. Mishchenko, "Scalable sequential verification", ERL Technical Report, EECS Dept., UC Berkeley, 2007. http:// www.eecs.berkeley.edu/~alanmi/publications/2007/tech07_ssv.pdf
[8] S. Chatterjee, A. Mishchenko, R. Brayton, X. Wang, and T. Kam, “Reducing structural bias in technology mapping”, *Proc. ICCAD '05,* pp. 519-526.
[9] M. Case, A. Mishchenko, and R. Brayton, “Inductively finding a reachable state space over-approximation”, *Proc. IWLS ’06*, pp. 172-179.
[10] N. Een and N. Sörensson, “An extensible SAT-solver”. *SAT ‘03*. http://www.cs.chalmers.se/Cs/Research/FormalMethods/MiniSat
[11] N. Een, A. Mishchenko, and N. Sorensson, "Applying logic synthesis to speedup SAT", *Proc. SAT '07*, pp. 272-286. http://www. eecs.berkeley.edu/~alanmi/publications/2007/sat07_map.pdf
[12] C. A. J. van Eijk. Sequential equivalence checking based on structural similarities, *IEEE TCAD*, 19(7), July 2000, pp. 814-819.
[13] J.-H. Jiang and R.Brayton, “Retiming and resynthesis: A complexity perspective”. *IEEE TCAD*, Vol. 25 (12), Dec. 2006, pp. 2674-2686.
[14] J.-H. Jiang and W.-L. Hung, “Inductive equivalence checking under retiming and resynthesis”, *Proc. ICCAD’07*, pp. 326-333.
[15] A. Kuehlmann, “Dynamic transition relation simplification for bounded property checking”. *Proc. ICCAD ’04,* pp. 50-57.
[16] C. E. Leiserson and J. B. Saxe. “Retiming synchronous circuitry“, *Algorithmica*, 1991, vol. 6, pp. 5-35.
[17] F. Lu, L. Wang, K. Cheng, J. Moondanos, and Z. Hanna, “A signal correlation guided ATPG solver and its applications for solving difficult industrial cases," *Proc. DAC `03*, pp. 668-673.
[18] F. Lu and T. Cheng. “IChecker: An efficient checker for inductive invariants”. *Proc. HLDVT ’06*, pp. 176-180.
[19] A. Mishchenko, S. Chatterjee, R. Jiang, and R. Brayton, “FRAIGs: A unifying representation for logic synthesis and verification”, *ERL Technical Report*, EECS Dept., U. C. Berkeley, March 2005.
[20] A. Mishchenko, S. Chatterjee, R. Brayton, and N. Een, "Improvements to combinational equivalence checking", *Proc. ICCAD '06*, pp. 836-843.
[21] A. Mishchenko, S. Chatterjee, and R. Brayton, "DAG-aware AIG rewriting: A fresh look at combinational logic synthesis", In *Proc. DAC '06*, pp. 532-536. http://www.eecs.berkeley.edu/~alanmi/ publications/2006/dac06_rwr.pdf
[22] A. Mishchenko, S. Cho, S. Chatterjee, and R. Brayton, "Combinational and sequential mapping with priority cuts", *Proc. ICCAD '07*, pp. 354-361.
[23] A. Mishchenko, R. Brayton, J.-H. R. Jiang, and S. Jang, "SAT-based logic optimization and resynthesis", *Proc. IWLS '07*, pp. 358-364. http://www.eecs.berkeley.edu/~alanmi/publications/2008/fpga08_im fs.pdf
[24] A. Mishchenko, M. L. Case, R. K. Brayton, and S. Jang, "Scalable and scalably-verifiable sequential synthesis", *Proc. IWLS'08*. http://www.eecs.berkeley.edu/~alanmi/publications/2008/iwls08_seq .pdf
[25] H. Mony, J. Baumgartner, V. Paruthi, and R. Kanzelman. “Exploiting suspected redundancy without proving it”. *Proc. DAC’05*, pp. 463-466.
[26] M. Mneimneh and K. Sakallah, “REVERSE: Efficient sequential verification for retiming”, *Proc. IWLS ’03*, pp. 133-139.
[27] M. N. Mneimneh and K. A. Sakallah. “Principles of sequential-equivalence verification”. *IEEE D&T Comp*. Vol. 22(3), pp. 248-257, 2005.
[28] P. Pan and C.-C. Lin, “A new retiming-based technology mapping algorithm for LUT-based FPGAs,” *Proc. FPGA ’98*, pp. 35-42.

Table 1. Synthesis results.

Bench-mark	Original network					After synthesis			HAIG			Run-time,s
	PI	PO	Reg	Node	Lev	Reg	Node	Lev	Reg	Node	Lev	
s13207	31	121	669	2728	36	1060	2133	25	4763	20598	36	0.36
s35932	35	320	1728	11948	19	2016	9094	11	5046	60771	19	0.71
s38417	28	106	1636	9238	31	1833	8161	27	10636	60156	48	0.83
s38584	12	278	1452	12310	37	2478	9427	25	7731	63638	43	0.98
b14	32	54	245	6070	61	587	4893	61	2630	31296	73	0.32
b15	36	70	449	8448	66	949	7756	94	6377	51139	106	0.67
b17	37	97	1415	27567	93	2271	24386	104	10415	137921	127	1.70
b18	37	23	3320	81710	132	3940	65264	117	12320	354141	132	3.99
aqua	464	3328	1477	25058	276	2032	20710	347	10477	124894	347	1.82
cfft	52	592	1051	13838	80	1165	9184	62	10051	79216	119	0.85
cord1	50	32	719	11846	73	1433	8425	58	6592	60432	119	0.67
cord2	34	40	1015	15773	82	2080	10862	63	8510	79469	142	1.04
desperf	121	64	1976	29905	17	1992	22873	28	10976	145498	31	1.66
ether	192	1171	1272	10820	70	2159	8977	78	7737	60486	111	1.27
fpu	262	280	659	24932	3580	997	16294	1876	9659	126436	3580	3.21
jpeg	1720	3450	3972	56601	89	5788	43712	73	12972	243672	104	6.63
mem	115	152	1825	16727	33	2399	14067	38	8781	85341	45	1.79
radar	3292	17732	6001	78342	173	7557	58759	91	15001	347762	174	8.75
video	1903	3528	3549	46433	95	3422	32852	75	12549	208953	99	4.86
raytracer	4364	10569	13079	187683	338	13624	137974	252	22079	771632	338	13.65
Geomean				1.00			0.77			5.13		

Table 2. Verification results.

Bench-mark	HAIG (2 frames)		CNF statistics			HAIG outputs			Runtime, s	
	Node	Lev	Var	Clause	Literal	Constr	Property	Total	HSEC	GSEC
s13207	9999	44	49631	100078	213073	10821	7526	16557	1.47	1000+
s35932	24230	26	45186	101053	237682	10733	3127	41866	2.08	44.67
s38417	31926	49	99776	210981	463841	24418	7691	47369	7.86	63.74
s38584	27530	44	79499	171554	380874	21279	5443	46931	0.60	18.90
b14	19548	94	55175	125353	276393	12511	6645	22580	9.47	2.18
b15	28958	145	81916	180762	399998	21169	6666	38223	19.85	21.84
b17	72016	147	180428	414631	928526	40450	20253	91526	82.02	48.84
b18	162428	162	388024	919704	2073474	79858	57365	217378	100.45	126.94
aqua	59421	415	159804	356435	795010	39764	14531	90077	119.67	1000+
cfft	44231	150	140962	308619	680245	31522	15495	64259	42.36	1000+
cord1	30202	121	86974	192412	429937	21616	7018	47834	1.52	8.52
cord2	36252	162	108475	236987	526894	27282	9471	61838	2.99	9.62
desperf	66698	35	153539	354900	820614	38235	12181	99651	9.69	4.34
ether	32329	153	94498	206245	457600	21793	9567	45194	4.87	5.83
fpu	54829	2710	177714	390191	856681	44815	19571	94187	5.73	1000+
jpeg	81170	89	278307	606075	1338154	63579	40262	188743	18.07	279.30
mem	44444	60	110632	248315	559943	25050	11004	60230	4.66	43.83
radar	135625	135	362882	823046	1867188	72429	58201	253965	80.29	52.82
video	87497	79	279167	617224	1371529	59229	42531	157531	113.00	69.94
raytracer	288421	487	764810	1757699	3975396	154115	130032	548596	800.55	1000+
Geomean						0.42	0.19	1.00	1.00	4.59

BackSpace: Formal Analysis for Post-Silicon Debug

Flavio M. De Paula[1], Marcel Gort[2], Alan J. Hu[1], Steven J. E. Wilton[2], Jin Yang[3]
[1] Dept. of Computer Science, University of British Columbia, {depaulfm, ajh}@cs.ubc.ca
[2] Dept. of Electrical and Computer Engineering, University of British Columbia, {mgort, stevew}@ece.ubc.ca
[3] Intel Corporation, jin.yang@intel.com

***Abstract*—Post-silicon debug is the problem of determining what's wrong when the fabricated chip of a new design behaves incorrectly. This problem now consumes over half of the overall verification effort on large designs, and the problem is growing worse. We introduce a new paradigm for using formal analysis, augmented with some on-chip hardware support, to automatically compute error traces that lead to an observed buggy state, thereby greatly simplifying the post-silicon debug problem. Our preliminary simulation experiments demonstrate the potential of our approach: we can "backspace" hundreds of cycles from randomly selected states of some sample designs. Our preliminary architectural studies propose some possible implementations and show that the on-chip overhead can be reasonable. We conclude by surveying future research directions.**

I. Introduction

Post-silicon debug (AKA post-silicon validation, silicon debug, silicon validation) is the problem of determining what's wrong when the fabricated chip of a new design behaves incorrectly. The focus of post-silicon debug is *design errors*, whereas VLSI test focuses on random *defects* introduced during the manufacturing process of each chip. Post-silicon debug currently consumes more than half of the total verification schedule on typical large designs, and the problem is growing worse.[1] Even worse, the schedule variability is greatest post-silicon, creating unacceptable uncertainty in time-to-market.

The problem is easy to understand from a typical scenario:

> *After a long development process, first silicon (or second, or third...) comes back from the fab. Yield is mediocre, but several "good" die pass the manufacturing tests and make it to the bring-up lab. Power-on reset and basic functionality tests work. But 30 seconds into booting the OS, the chip crashes. Worse yet, every single "good die" crashes in the same way.*
>
> *Scanning out the state of the crashed chips show an internal data structure (e.g., a routing table or a coherence directory) has been corrupted in an inexplicable manner. The logic analyzer dumps from just before the crash show routine traffic to/from memory and I/O devices. The stimulus to generate the crash (i.e., booting the OS for 30 seconds) is far too deep to replay in simulation or by single-stepping the die, and trying to hit the crash state with full-chip formal verification exceeds the capacity of the formal tools. Increasingly many engineers, from many different teams (pre-silicon verification, design, test, architects, OS, drivers,...) get sucked into the debugging effort. Yet debugging proceeds painfully slowly, because everyone is flying blind, trying to guess what happened.*

[1] Andreas Kuehlmann, personal communication, September 21, 2006. References [1], [6] also report a large fraction of total verification cost occurring post-silicon.

Obviously, the fundamental problem is observability — to debug a chip, we must know what is happening. With increasing technology scaling, higher speeds, and greater integration, there is simply no longer enough I/O to be able to debug effectively: pin counts are limited; I/O pads are too costly (area and power) and slow; and single-stepping and scanning out the state every clock cycle is far too slow.

This paper introduces a novel paradigm for using techniques from formal verification, augmented with some on-chip support logic, to greatly enhance observability of the execution leading up to the observed buggy behavior. Specifically, we allow the chip to run at full speed, yet provide the ability to "backspace" hundreds, perhaps thousands, of cycles from a crash state or a programmable breakpoint, to derive an error trace that led to the crash, which can then be replayed in a simulator or waveform viewer to help understand the bug. In a nutshell, the basic framework consists of adding circuitry to monitor architecturally key operating points and record a small signature of the monitored information. During debug, this signature, as well as the crash state, can be scanned out using existing on-chip test access mechanisms. By itself, the signature provides insufficient information (lossy) and would be meaningless to a human, but we can combine this information with formal analysis of what is possible in the design (by computing pre-images) to determine the unique, or nearly unique, predecessor states that led to the crash state. Such an approach can backspace only a limited number of cycles, so we also add circuitry to the chip to allow programmable breakpoints. By setting a predecessor state as a breakpoint, we can re-run the failing test on-chip (e.g., booting the OS), but crash some number of cycles earlier, then scan out the new crash state and signature, and iterate the entire process to compute arbitrarily long back traces.

This work can revolutionize post-silicon debug, but it is still in its infancy and not yet practical. This paper presents the basic framework and preliminary results demonstrating both potential and limitations. We conclude by surveying future research directions.

978-1-4244-2735-2/08/$25.00 ©2008 IEEE

A. Related Work

There is no closely related work to what is presented in this paper. Under the broad rubric of using formal methods to aid post-silicon debug, however, there are a few related papers. Ahlschlager and Wilkins [2] describe their experience directly using a model checker for post-silicon debug: they write a formal property to describe the observed buggy behavior and ask the model checker to generate a trace. Such an approach is ideal when the model checker can verify the entire chip or when the debug engineer correctly maps the observed chip-level buggy behavior onto a block-level formal specification, neither of which can be counted on. Safarpour et al. [17] address the problem as rectification: they assume that design errors/fixes can be modeled by injecting corrected values at various points in the circuit and use a MAX-SAT solver to find a smallest set of locations to inject values to correct the behavior on a specific set of test cases. This work is simultaneously much more limited than ours (simplistic model of design errors, limited test cases, scalability issues), yet also much more ambitious (attempting to correct the design automatically). Several groups have proposed leveraging the intellectual investment into formal *specifications* during post-silicon debug, by compiling the formal specifications into on-chip monitor circuits (e.g., [13], [8], [4]). Such an approach provides enhanced observability into what went wrong if an on-chip monitor catches an assertion violation, but doesn't leverage formal *verification* technology to aid debugging.

Of course, industry has been doing (and struggling with) post-silicon debugging without the benefit of formal methods since the dawn of VLSI. Very little has been published about current techniques and methodologies, although anecdotally, we know that practices vary enormously between companies, e.g., a small company with a high-margin, low-volume product can't afford specialized high-end test equipment but can tolerate more die area overhead for debugging support, whereas a large company with a high-volume product might spend enormous up-front NRE costs to shave on-chip overheads to the bare minimum. On one extreme, some companies are not even using existing on-chip test access mechanisms to aid debugging, relying instead on manually modifying bring-up programs and observing the results; on the other extreme, some companies have purpose-built bring-up hardware and sophisticated logic analyzers that allow intercepting, recording, and replaying all traffic between the chip and its environment (e.g., [18]). Despite this variation, our work and existing methods share similarities, based on the common underlying constraints:

- It is possible to get a fairly complete snapshot of the internal state of a chip, albeit very slowly. The most basic mechanism is the scan chains [19] present on almost all chips to allow efficient manufacturing test. A chip with scan can be configured into test mode, in which most or all of the latches on the design are connected together into a small number of very long shift registers. At any point in time, the chip can be stopped, and the values of the latches can be shifted in or out. With hold-scan latches, it is even possible to scan out a snapshot of the state of the chip at one point in time, while allowing the chip to continue to execute during the scan-out process, at the cost of substantial on-chip overhead [10]. In our work, we assume the existence of full scan on the chip, but do not require hold-scan latches.
- A very limited history of some number of signals can be recorded at full speed on-chip, and this history can be read out (very slowly), e.g., via the scan chains. These mechanisms ("on-chip logic analyzers", "trace buffers") typically consist of a flexible mechanism to access desired signals on-chip and some way to store the signals for later read out (e.g., RAM or specialized cells [3]). The main trade-off is that considerable die area overhead must be used for each cycle's worth of history for each signal monitored, severely limiting how much history can be stored.

  The signature generation in our method can be viewed as a generalized, optimized trace buffer, using formal verification techniques to enable reconstruction of a fully detailed trace from compressed signatures. We rely on prior work on access mechanisms to observe on-chip signals. In this paper, we consider one efficient, flexible, and reconfigurable architecture that provides this access [15]. Abramovici et al. [1] propose a different reconfigurable architecture, also with the goal of providing efficient signal access for debugging. Also, many companies have their own, in-house access mechanisms to help in debugging (e.g., [7]), but published details are sparse. Nevertheless, the sort of on-chip access we assume are clearly very realistic. For signature storage, our area models assume SRAM (Section IV).
- In many debugging scenarios, standard off-chip logic analyzers are helpful. As mentioned above, in the extreme case, specialized hardware and a great deal of high-end test equipment can be used to record and replay *all* I/O signals between the chip and its environment, allowing deterministic repeatability of the stimulus that triggered a bug on-chip. The main drawbacks are the high cost of the test equipment, the extremely limited ratio of observable I/O pins and pads versus the internal state of the chip, the inability to debug internal IP blocks, and the ability to debug the chip only in the specialized bring-up system — sometimes, a bug will manifest itself only in some OEM system but not in the original bring-up system. Note also that even in the extreme case, the logic analyzer traces alone do not allow us to reproduce the bug in a logic simulator, since we don't know the internal state of the chip to start the simulation. Our method does not require off-chip logic analyzer traces and hence does not suffer the drawbacks. However, if such traces are available, we could use them to reduce our on-chip overhead.
- The only mechanism fast enough to run the actual bring-up of the chip, in an actual system, on actual data, is the chip itself. There is no way to simulate an extremely deep

trace (e.g., even 30 seconds of real execution time), and no way (without formal verification techniques, as in this paper) to go backwards from a state of interest on the actual silicon to determine what happened beforehand.

The goal, of course, is to determine what happened *before* the crash occurs, but we do not know *a priori* when that will be. Accordingly, current methods typically attempt to find some way to scan out a complete state snapshot some number of cycles before a crash state happens, and then use that state along with the recorded I/O behavior to simulate the chip from shortly before the crash up until the crash, e.g.,:

- Periodic Sampling. The chip can be stopped at regular intervals to scan out a snapshot of the internal state and then allowed to continue execution. With hold-scan latches, the chip need not even be stopped. When the crash occurs, the most recent snapshot and the logic analyzer traces of the I/O can be used to recreate the bug in simulation. An obvious problem is that stopping the chip disturbs system-level timing interactions, potentially changing the execution and hiding the bug. Furthermore, because the scan-out process is so slow, the interval between snapshots must be long, meaning that the most recent snapshot might be too far in the past.
- Cycle Counters. If the chip's behavior and the system environment are both deterministic (or the I/O behavior has been recorded and can be replayed), then a simple cycle counter can be used. When a crash occurs, we record when it happened, and then we re-run the system, but scan out a snapshot when the cycle counter is a convenient number of cycles prior to the crash. Non-determinism is the main difficulty for this approach, obviously in the system environment (e.g., when a disk or network interrupt occurs), but also on-chip (e.g., multiple independent clock domains, arbitration, PLL lock times).
- For a system implemented on FPGAs, the problem of system-level non-determinism can be eliminated by duplicating the entire design [11]. One copy of the design runs in the system as usual; the second copy has all of its inputs delayed in a FIFO. When the first copy hits the bug, it triggers trace recording on the second copy. For a non-FPGA design, it's obviously impractical to duplicate the design on-chip, but if two identical dies are available, both of which are fully deterministic in an identical manner, one could imagine building a specialized bring-up board that implements this solution.[2] Aside from the determinism restriction, the obvious problems with this approach are the cost of the purpose-built bring-up system and the likelihood that bug behavior will be different between that system and real OEM systems.

In contrast to the above methods, our method eliminates the challenge of trying to determine when to take a snapshot of the internal state just before the crash is about to happen; instead, the formal analysis allows us to compute the predecessor state directly.

[2]This idea was suggested to us by Igor Markov, June 30, 2008.

Our work is focused on design errors that escape pre-silicon verification and end up on the actual chip. A complementary post-silicon debug problem, for which there is also very little research, is to help identify and repair physical, electrical, and timing errors on-chip. Chang et al. [6] propose a methodological framework for this class of problems. Park and Mitra [14] also focus on electrical bugs and propose a processor-specific technique using summaries of in-flight instructions in the processor. A post-analysis over these summaries helps locate the source of the bug. The main similarity between their work and ours is the collection of information from the design via summaries (signatures), and then using that information in the post-analysis. Their approach demonstrates very low overhead, but is narrowly specific; our approach is not processor-specific, but currently has excessive overhead.

One insight behind our approach is that the actual silicon is so fast that it can be used to re-run some input stimuli *ab initio* repeatedly, to compute the state of the chip at different points in time. This insight echoes a similar computation used for "hardware modeling", in which an actual chip is used in a special modeling system to emulate its own behavior during system-level logic simulation [9].

II. Basic Framework

A. Intuition and Assumptions

The basic problem is that we have observed the chip in some buggy state, and we have no idea how that could have happened. The goal is to explain the inexplicable buggy state, by creating a "backspace" capability — iteratively computing predecessor states in an execution that leads to the bug. The resulting trace can be viewed like a simulation waveform, except it shows what actually happened just before the bug/crash on the real silicon.

We assume that the problem occurs at a depth and complexity not trivially solved by existing methods. For example, if the full chip can be handled in a model checker, we can simply ask the model checker to generate a trace to the observed buggy state. This solution is not realistic for complex designs, because of the capacity limits of model checkers. Alternatively, if the bug occurs extremely shallowly during bring-up, we could run the bring-up tests on the simulator, or via single-stepping the chip (scanning in a state, pulsing the clock, scanning out the state). Such an approach is also not realistic: the roughly billion-to-one speedup of the actual silicon versus full-chip simulation means that one second of runtime on-chip equals decades of run time in simulation, and within seconds of first power-on, the silicon has executed more cycles than months of simulation on vast server farms. Trying to reproduce the bug *ab initio* in simulation is clearly not feasible. Similarly, trying to monitor externally the full execution trace of the chip running full-speed is electrically impossible.

We start with a few simplifying assumptions:

- It must be possible to recover the state of the chip when an error has occurred. For example, this could be done with the chip in test mode, via the scan chain.

- The key assumption is that since we are focusing on *design errors*, we will assume that manufacturing testing has eliminated manufacturing defects. Therefore, we assume that the silicon implements the RTL (or gate-level or layout or any other model of the design that can be analyzed via formal tools).
- The bring-up tests can be run repeatedly and the bug being targeted will be at least somewhat repeatable (one out of every n tries, for a reasonably small value of n).

Later, we discuss how the framework changes when we relax these assumptions, e.g., partial scan, mixtures of design errors and defects, and non-deterministic errors due to marginal circuits, process variability, etc. Even without the relaxations, though, the problem is real, and a solution would be valuable.

Our framework consists of adding some debug support to the chip: a signature that saves some history information but otherwise has no functional effect on the chip's behavior, and a programmable breakpoint mechanism that allows us to "crash" the chip when it reaches a specified state. Given these, the approach repeats the following steps

1) Run the chip until it crashes or exhibits the bug. This could be an actual crash or a programmed breakpoint.
2) Scan out the full crash state, including the signature.
3) Using formal analysis of the corresponding RTL (or other model), compute the predecessor of the crash state. The signature must provide enough information to allow only one (or a few) possible predecessor state.
4) Set the computed predecessor as the new breakpoint.

until we have computed enough of a history trace to debug the design (or Step 3 fails). Each iteration of the loop is like hitting "backspace" on the design – we go back one cycle. The approach exploits the capabilities of different analyses: formal analysis is very slow with limited capacity, but can go forward or backwards equally well; simulation is too slow to run in a real system with actual software, but the visibility of a simulation trace is user-friendly and well-accepted for design understanding and debugging; the actual silicon runs full-speed, rapidly hitting bugs that may have escaped pre-silicon validation, but offers very poor visibility and no way to backspace to see how the chip arrived in some state.

B. Theory

We model the system in the usual manner as a finite state machine M, with S latches and I inputs, initial states Init $\subseteq 2^S$, and transition relation $\delta \subseteq 2^S \times 2^I \times 2^S$. We allow the transition relation to be non-deterministic, so the formalism can handle randomness in the bring-up tests as well as transient errors, race conditions, etc.

Given a state machine M, we can build an augmented state machine M' which has the same behavior as M (when projected onto the original S latches), but has an additional T latches of signature. The T signature latches are not allowed to affect the behavior of M, so the transition relation of M' is a pair of relations: the original $\delta \subseteq 2^S \times 2^I \times 2^S$ as well as a $\delta' \subseteq 2^S \times 2^T \times 2^I \times 2^T$. In other words, the next signature can depend on the signature as well as the state and inputs, but the next state cannot depend on the signature.

Definition 1 (Backspaceable State): A state (s', t') of augmented state machine M' is backspaceable if its pre-image projected onto 2^S is unique, i.e.,:[3]

$$\exists! s \exists t, i[((s, i, s') \in \delta) \wedge ((s, t, i, t') \in \delta')]$$

In general, a signature might contain enough information to allow computing multiple cycles of unique pre-images. In that case, the theory changes in the obvious manner to backspace multiple cycles from each run of the chip. Currently, we envision such multi-cycle signatures to be simply a series of single-cycle signatures, stored via pipelining in the signature collection circuitry or in efficient SRAM structures (Sect. IV). To simplify the exposition in this paper, we describe backspacing only a single cycle at a time.

Definition 2 (Backspaceable Machine): An augmented state machine M' is backspaceable iff all reachable states are backspaceable. A state machine M is backspaceable iff it can be augmented into a state machine M' for which all reachable states are backspaceable.

The algorithm to compute the states prior to the crash state starts from a given crash state and then iteratively computes its predecessors, going backwards in time:

Algorithm 1 (Crash State History Computation): Given a state (s_0, t_0) of a backspaceable augmented state machine M', compute a finite sequence of states $(s_0, t_0), \ldots, (s_k, t_k)$ as follows:

1) *Since M' is backspaceable, let s_{i+1} be the unique pre-image state (in the state bits S) of (s_i, t_i).*
2) *Run M' (possibly repeatedly) until it reaches a state (s_{i+1}, x). Define $t_{i+1} = x$.*

Theorem 1 (Correctness of Trace Computation): If started at a reachable state (s_0, t_0), the sequence of states $s_k, \ldots, s_0$ computed by Algorithm 1 is the suffix of a valid execution of M.

Proof Sketch: For any state (s_i, t_i) in the sequence, we must prove that two properties hold for s_{i+1}: first, that s_{i+1} is a predecessor of s_i in M, and second, that s_{i+1} is a reachable state in M. By the definition of pre-image, there exists x such that (s_{i+1}, x) is a predecessor of (s_i, t_i). By the definition of the augmented state machine, s_i cannot depend on x, so therefore s_{i+1} must be a predecessor of s_i. That establishes the first property. For the second property, because M' is backspaceable, s_{i+1} is the same for all predecessor states of (s_i, t_i). Therefore, any execution σ' of M' that reached (s_i, t_i) must have gone through a state (s_{i+1}, x) for some x. Because the signatures cannot affect the state latches, the projection of σ' onto the state latches is a valid execution of M and goes through the state s_{i+1}. Hence, s_{i+1} is a reachable state of M. ∎

If the state machine as well as the environment/testbench are deterministic, then Algorithm 1 not only gives *a* valid execution, but *the* execution of M that led to the crash state,

[3]The notation $\exists!$ denotes "There exists a unique...."

because the execution σ' will be the same when computing each state in the sequence. In the presence of randomness, the algorithm still works, but with a constant factor expected slow-down: if the bug appears in 1 out of n runs, then we expect to need to repeat n times step 2 per iteration of Algorithm 1. Similarly, if the pre-image is not unique, but there are k states in the pre-image, we can try step 2 for each of the k possible pre-image state, resulting in a constant factor $k/2$ expected slow-down. There is no combinatorial blow-up, as there would be for backward reachability. Definitions 1 and 2 generalize naturally to "k-backspaceable" for a given bound k.

An important caveat is that, under the assumption of true non-determinism, termination of the algorithm is not guaranteed. For example, it is conceivable that setting the programmable breakpoint hardware to target state s_a will result in an execution σ_a that reaches (s_a, t_a) from a state (s_b, x), but if we reprogram the breakpoint hardware to target state s_b, subtle electrical effects might cause the chip to follow a different execution σ_b that reaches (s_b, t_b) from some state (s_a, y). In this case, Algorithm 1 will still compute a valid execution of M, as indicated by the theorem, but this execution won't make any progress toward the initial states. Fortunately, if non-determinism in the model is really randomness, with non-zero probability of choosing all legal transitions, then we can prove termination with probability 1:

Theorem 2 (Probabilistic Termination of Algorithm 1): If we terminate Algorithm 1 when the computed sequence reaches an initial state of M, and if the executions σ' of M' are chosen randomly such that all valid transitions have non-zero probability, then termination occurs with probability 1.

Proof Sketch: At all times, the state being considered in the algorithm is reachable. Hence, there is an execution σ' of length l that reaches the target state, and this execution occurs with non-zero probability. If this execution gets chosen repeatedly l times (an event that also occurs with non-zero probability), then the algorithm will terminate after l iterations. Otherwise, the algorithm continues from another state. Hence, the algorithm is a random walk backwards on the state space, where the initial states are sink states and all states are reachable from the initial states. With probability 1, the random walk must terminate in a sink state.

∎

In practice, we do not expect these issues of non-determinism, randomness and termination to be a problem. The main difficulty with randomness will be the number of trials required to hit a breakpoint state when the chip runs — if the probability is low, many runs will be needed for each backspace step.

Algorithm 1 performs repeated pre-image computation, which can be expensive. We encountered problems in our initial experiments with BDDs and All-SAT. A key insight greatly improved efficiency:

> We need compute only whether a state has a unique pre-image state or not (or whether it has more than k pre-image states for k-backspaceability).

This insight means we can use a state-of-the-art, off-the-shelf SAT solver to search for *any* pre-image state. If one is found, then we add just one blocking clause to eliminate that solution and run the SAT solver a second time, to see if the first solution was unique.

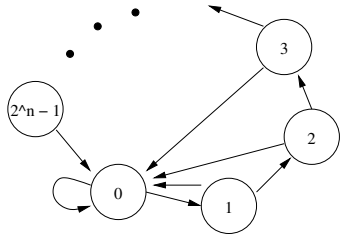


Fig. 1. State Machine Requiring $|S|$ Extra State Bits to Be Backspaceable

To scale to full-size industrial designs, it is likely that not all state bits will be scannable and breakpointable. In that case, the above theory generalizes exactly as abstraction applies to model checking. Algorithm 1 will compute an abstract execution, over the subset of the state bits that are scannable/breakpointable. This execution will be the suffix of a valid execution of the conservative abstraction of M onto the subset of state bits. Hence, there will be the same risk of false abstract execution paths, exactly as in abstract model checking. Standard concretization heuristics from abstract model checking may help. It is also likely that the abstract trace may be suggestive enough of what actually happened on-chip to help the debug engineer understand what caused the crash/error, even if the trace can't be or hasn't been concretized. For example, an abstract trace might indicate that the error occurs when a certain type of transaction encounters a specific exception condition at the exact cycle that another event occurs, but without indicating the specific data values in the trace.

C. Backspace Coverage

Is it always possible to augment any state machine to make it backspaceable? The answer is yes. We can simply make $|T| = |S|$ and set up δ' to copy the values in the latches of S to the latches of T. In other words, we can always backspace to a unique predecessor state because we have stored that state.

Is it possible to do better, to make any state machine backspaceable using fewer than $|S|$ additional latches? Unfortunately, in the worst case, the answer is no. For a simple example, consider the state machine in Figure 1. This example is a simple n-bit counter, with a single input. If the input is low, the counter transitions to the 0 state; otherwise, it counts up. Almost all states have only a single predecessor, making them backspaceable with no additional signature. However, the 0 state has every state as a predecessor. To make the machine backspaceable, we must add the full n additional state bits, just to handle one particularly bad state. We call such states "convergence states" because many incoming transitions converge on them.

Figure 1 shows that in the worst case, we can do no better than by storing a copy of all the state bits. However, it also suggests that we might be able to do much better

for *most* states. Is it good enough if we make most states backspaceable?

Definition 3 (Backspace Coverage): Given state machine M augmented into M', the backspace coverage of M' for M is the fraction of the reachable states of M' that are backspaceable.

Can we get good backspace coverage with much fewer than $|S|$ bits in the signature, or more to the point, can we backspace a long enough trace to be useful before hitting a convergence state? The convergence states are likely to be states that are easy to get to and easy to understand (like reset or idle states); backspacing to a convergence state may be sufficient for debugging purposes. In the next section, we explore whether we can make this theory work on some sample open-source designs.

III. Preliminary Experiments

A. Experimental Setup

We present our experiments on two small processors/microcontrollers. The research is still highly exploratory, so we have chosen to focus on a small number of design examples: we often needed detailed understanding of the designs to generate good research hypotheses. The designs also had to be small enough so that repeated experiments were feasible, and so that the supporting algorithms and tools that are not germane to this research did not need to be highly optimized. On the other hand, the designs must be realistic, to capture characteristics of real designs.

The two processors are a 68HC05 and an 8051. These are both open-source designs from opencores.org that are rebuilds from datasheets of the respective classic 8-bit microcontrollers from Motorola and Intel. The 68HC05 is smaller, with 109 latches. The 8051 implementation has 702 latches. In both cases, we developed a simulation testbench based on the testbenches supplied with the designs: the 68HC05 ran real LED and LCD controller applications, and the 8051 ran some small software routines.

For our experiments, we treated the design running on a commercial logic simulator as if it were the actual chip running on silicon. We simulated the designs for an arbitrary number of cycles and randomly selected 10 states each to serve as "crashed" states for our analysis. In addition, our testbench also recorded the immediate predecessor state before the crash state (which wouldn't be possible in silicon); this predecessor state is the correct answer that our backspace analysis is trying to recover. Thus, we have 10 pairs of states per design to serve as testcases.

B. Signature Functions

As a first step, we needed to find some plausible signature functions. We concentrated on the 68HC05 and tried a variety of approaches. Fig. 2 summarizes the results of our experiments.

Our first idea was to try a quick experimental upper bound on the size of the signature. We created the signature as a randomly selected subset of all state bits. Unfortunately, this approach fared poorly: 90% of the state bits were needed in the signature before the median size of the pre-image was 1, and even with 95% of the state bits in the signature, only 7 out of 10 of our test states had unique pre-image states.

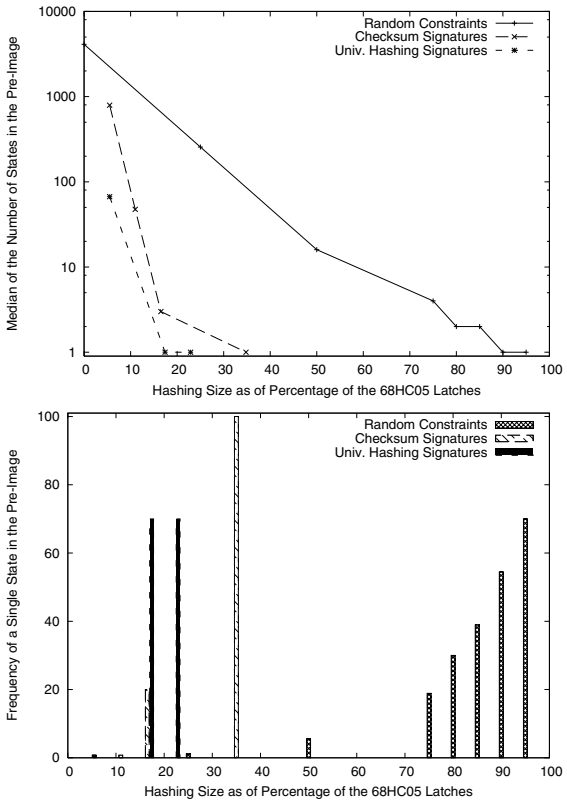


Fig. 2. Results for Compressed Signatures Based on Architectural Insight.

In real life, the designers understand their design, and architectural insight might allow selecting a particularly good subset of the state bits to use as a signature. Based on a careful study of the 68HC05, we identified 38 latches (35% of the design) to use as the signature. This approach was very successful, yielding unique pre-image states for all 10 test cases.

Spurred by that success, we tried some simple checksums on those 38 bits, reducing the number of bits used to 6, then 12, and then 19. These results were not very successful at getting unique pre-image states, but the plot suggested that better compression would be promising.

Accordingly, we tried a perfect hash function — universal hashing [5] (essentially the same as X-Compact [12], which is easier to implement on-chip) to compress the 38 bits to 6, 19, and 25 bits. These results demonstrated the promise of universal hashing.

In all of these experiments, computations were fast, and the SAT solver had no problem computing pre-image states.

C. BackSpacing

With some promising ideas for signature functions, we proceeded to the real test: can we backspace for hundreds of cycles from the random crash states? We created an automatic

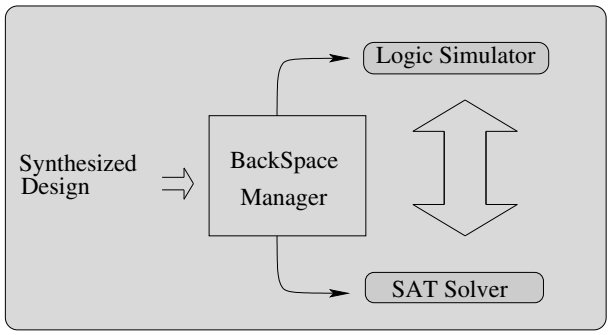


Fig. 3. BackSpace Framework

framework to experiment and explore the BackSpace paradigm (Fig. 3). The components of the framework are the BackSpace Manager, a commercial logic simulator, and a SAT solver. The input to the framework is a synthesized design (gate-level netlist). The logic simulator plays the role of the silicon: we use it to run our testbench, exactly as the real silicon would run bring-up tests. The SAT solver is the engine to compute the required pre-image states. The core of the framework is the BackSpace Manager.

The BackSpace Manager coordinates the logic simulation and the SAT solving tasks by dispatching each task and processing their intermediate results (shown as the double-headed arrow in Fig. 3). For logic simulation, the BackSpace Manager automatically generates a testbench instance based on the synthesized design, dispatches the logic simulation, awaits its termination, and captures the crash state and signature. For SAT solving, given the crash state and the signature, the BackSpace Manager generates a SAT problem instance. When the SAT solver finds a solution, it means there is one (more) state in the pre-image of the crash state. The BackSpace Manager generates a blocking clause based on this solution and asks the SAT solver for another solution. If another solution is found, this process repeats until there are no more solutions. At that point, a single state or a set of states is available as candidate states prior to the crash state. The task now is to find which candidate state is the actual one. The BackSpace Manager dispatches logic simulation, setting a candidate state as a simulation breakpoint. If simulation reaches the breakpoint, it means we have a new crash state and a signature. This process continues until we have "backspaced" some pre-determined number of cycles. If simulation does not reach the breakpoint, it means we need to try another candidate. For our logic simulator, we used Synopsys VCS (version 7.2), and for our SAT solver, we used Minisat (version 2.0). Due to VCS licensing issues and GCC compatibility problems, we had to run these tools on different machines: logic simulation was run on a Sun Fire V880 server (UltraSPARC III at 900Mhz); SAT solving was run on an Intel Xeon at 3.00GHz.

We ran experiments for both the 68HC05 and the 8051. For each, the goal was to see how far we could backspace before the pre-image set got too large or the computation blew-up.

Crash State	# of Cycles BackSpaced	Max States in PreImg	Sim Time	Sat Time	Manager Time
s1	54	4096	63.44	0.93	204.42
s2	1	65536	1.45	86.61	4.38
s3	37	4096	62.65	0.71	139.67
s4	7	4096	37.76	0.52	27.11
s5	53	4096	116.16	0.92	200.34
s6	500	1	1261.48	3.24	1884.31
s7	500	1	2384.29	3.15	1890.91
s8	500	1	4575.41	3.01	1893.89
s9	2	4096	22.93	0.51	22.93
s10	9	65536	2424.55	91.18	34.86

"Sim Time" is the time spent in the logic simulator. This time would be replaced by time running on the actual silicon. "Sat Time" is the time spent in the SAT solver. "Manager Time" is the time spent by the BackSpace Manager to supervise the framework and connect the various tools. Our BackSpace Manager implementation is very preliminary and can be optimized extensively.

TABLE I
68HC05 W/ 38-BIT SUBSET HAND-CHOSEN SIGNATURE

Crash State	# of Cycles BackSpaced	Max States in PreImg	Sim Time	Sat Time	Manager Time
s1	500	2	1097.25	185.57	8524.15
s2	500	2	2011.04	187.21	8397.09
s3	500	2	2737.15	171.57	8335.45
s4	500	2	2988.38	242.89	8477.88
s5	500	2	3358.40	216.81	8398.14
s6	500	1	3176.94	31.89	8175.62
s7	500	1	6247.61	31.42	8280.93
s8	500	1	12207.49	38.58	8297.21
s9	500	2	15280.79	42.31	8173.19
s10	500	1	34084.53	36.63	8125.62

TABLE II
68HC05 W/ 38-BIT UNIVERSAL HASHING SIGNATURE

For the 68HC05, we reused the signature consisting of a hand-selected subset of 38 of the 109 total state bits, chosen based on our insight into the design. We also tried a 38-bit hash generated via universal hashing over the 109 state bits. For the 8051, we hand-selected a 281 bit subset of the 702 total state bits to be the "human architectural insight" signature. We also tried to use a 281 bit universal hash of the 702 state bits.

In these experiments, we used the k-backspaceable computation (i.e., pre-image sets are allowed to have up to k states), with k set to 300 states. To keep our experiments manageable, we also set an upper limit of 500 cycles of backspacing per test crash state.

Tables I and II show the results for the 68HC05. With the hand-chosen subset of bits, we hit our 500 cycle limit on 3 of the 10 test crash states. But on 4 of the 10, we cannot backspace more than a handful of cycles. With a universal hash of the same size, all 10 test crash states can be backspaced to our limit, and all of the pre-images are very small. In Section IV, we will see that a hand-chosen subset of bits is a very low-overhead signature, whereas universal hashing all bits of a large design appears to be prohibitively expensive. We can see the trade-off between quality and cost.

Table III presents the results for the 8051 using the hand-chosen subset of the state bits as the signature. The results are

Crash State	# of Cycles BackSpaced	Max States in PreImg	Sim Time	Sat Time	Manager Time
t1	205	512	2841.07	4.58	6048.46
t2	500	256	21759.74	9.70	14720.71
t3	500	257	8326.66	10.84	14746.10
t4	500	257	10342.40	10.77	14772.03
t5	500	256	11587.21	11.26	14742.81
t6	500	256	11581.93	8.72	14735.07
t7	500	255	25767.40	8.54	14742.60
t8	500	256	13581.20	11.57	14759.73
t9	500	257	22493.04	10.62	14735.48
t10	500	257	24793.42	10.81	14759.77

TABLE III
8051 W/ 281-BIT SUBSET HAND-CHOSEN SIGNATURE

Crash State	# of Cycles BackSpaced	Max States in PreImg	Sim Time	Sat Time	Manager Time
t1	500	8	138616.15	1379.21	55389.29
t2	500	4	497905.92	1350.15	55104.32
t3	500	4	191655.42	1378.15	55462.20
t4	500	4	183283.27	1383.10	55642.82
t5	500	8	431057.79	1377.87	55039.00
t6	500	4	151950.65	1399.62	55601.11
t7	500	4	506787.53	1388.94	55639.58
t8	500	8	506229.79	1368.52	55512.44
t9	500	4	488157.90	1379.14	55049.31
t10	500	4	534870.14	1378.37	55448.52

TABLE IV
8051 W/ 281-BIT UNIVERSAL HASHING SIGNATURE

excellent: we can backspace up to our 500 cycle limit in 9 out of the 10 test crash states. Initially, we were unable to complete results for the 8051 with a 281-bit universal hash. The SAT solver blew up (1 hour timeout and 1GB memory limit) on all 10 test cases. The universal hash function is essentially a matrix-multiplication over GF(2), with a random matrix, so it's not surprising that large instances are challenging for current SAT solvers. However, with some more thought and experimentation, we were successful with this experiment as well. The key is that any full-rank matrix provides correct universal hashing, but a sparse matrix will be easier for the SAT solver, and also reduce area overhead, too. If we generate the random hash matrix with a 0.985 probability of each entry being 0, we can backspace up to our set limit for all 10 test crash states. Furthermore, the number of states in the pre-image is 2 orders of magnitude smaller for all crash states. Table IV gives these results.

To summarize, the overall framework works. We can compute hundreds of cycles of error trace backwards from a crash state. Additional research will need to explore what sorts of signature functions work well, and at what hardware cost.

IV. ARCHITECTURE AND ON-CHIP OVERHEAD

This section describes the circuitry that must be added to the integrated circuit to implement the framework. It also estimates the area overhead of this circuitry.

A. Support Circuitry

Figure 4 shows how a Circuit Under Debug (CUD) can be instrumented with a Breakpoint Circuit, a Signature Creation circuit, and a Signature Collection circuit.

During debugging, as the circuit operates, the Signature Creation circuit monitors N_{mon} of the N_{state} state bits in the CUD. In general, $N_{mon} \leq N_{state}$, but in this analysis, we assume that all state bits are collected and used to form a signature, so $N_{mon} = N_{state}$. Each cycle, the Signature Creation circuit uses these state bits to construct a signature of size S_{width}; the construction of the signature will be described below. The signature is then stored in a memory within the Signature Collection circuit. The memory is arranged as a FIFO buffer composed of an SRAM block and read/write circuitry. The depth of this FIFO buffer dictates how many consecutive states can be stored. Meanwhile, the Breakpoint Circuit also monitors the state bits. When the state bits match a predetermined state (the target state), a signal is sent to stop the collection of signatures. The signature(s) stored in the buffer can then be read out and processed as described in Section II.

The heart of the architecture is the Signature Collection circuit. The simplest way to construct a signature is to simply use the state bits directly. If $N_{mon} = S_{width}$, then the history of all flip flops is stored, and the circuit becomes trivially backspaceable. If $N_{mon} > S_{width}$, then missing bits must be reconstructed using off-chip analysis as described in Section II.

If the set of N_{mon} signals cannot be determined at fabrication time, the selection of these signals can be made programmable at debug-time using a concentrator network [15]. Such a network would programmably connect a subset of the N_{mon} monitored signals for use in the signature. On-chip SRAM bits (similar to configuration bits in an FPGA) can be used to store the configuration of the concentrator. As debugging proceeds, the configuration can be changed, so that a different set of N_{mon} bits can be used in the signature. An example of the use of a concentrator in a debugging application can be found in [16]. Unlike the concentrators described in previous work in which each bit can be switched independently, we assume that the concentrator switches 8-bit wide words; this reduces the area of the concentrator by approximately 50%, while suffering only a small decrease in flexibility.

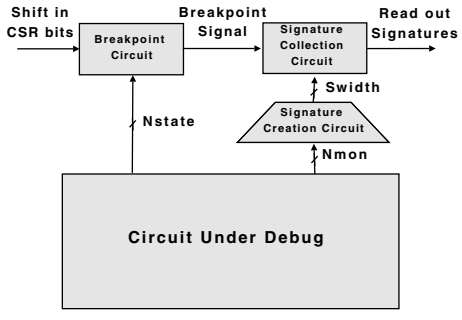
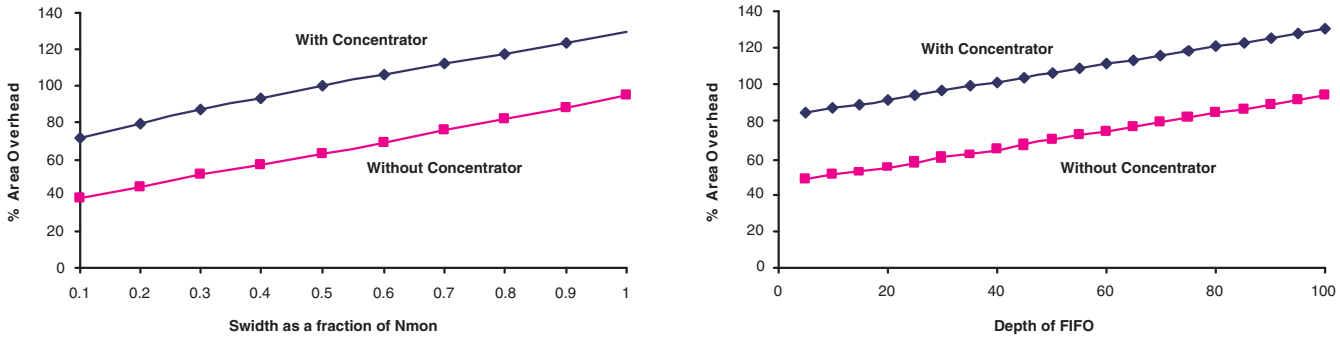


Fig. 4. Debugging Architecture



a) As a function of stored bits

b) As a function of FIFO depth

Fig. 5. Area overhead for instrumented LEON3 processor

As described in Section II, a Universal Hash Function can be used to compress the signature. This can be implemented as an array of XOR gates within the Signature Creation Circuit, and can be used with or without a concentrator network.

B. Area Overhead

In this subsection, we estimate the area overhead of our circuitry. To make our results concrete, we present an estimate of the area required to instrument a specific processor. In Section III, we used implementations of 68HC05 and 8051 processors to illustrate the technique, but these processors are too small to give meaningful area overhead estimates. Instead, in this section, we focus on a typical instantiation of the LEON3 open source processor,[4] because it represents a typical small-but-modern RISC processor, which is the natural next step beyond the small microcontrollers of our initial experiments. The LEON3 is a synthesizable, pipelined 32-processor that is certified SPARC V8 conformant. It is highly configurable, including support for multiprocessing, making it an attractive testbed as we scale this research to increasingly challenging designs. In this subsection, our LEON3 configuration has an area equivalent of 40,000 2-input nand-gates and 2,500 flip flops (so, $N_{state} = N_{mon} = 2500$). This area estimate does not include any RAM used by the LEON3 (but we will include the area of the SRAM used to store the signatures when computing the overhead of our method).

We first present results assuming that the Universal Hash function is not used, and then discuss the overhead of the hash function.

Figure 5(a) shows the area overhead as a function of the ratio between S_{width} and N_{mon}, for an architecture without the Universal Hash function. Intuitively, if this ratio is 1, all state bits are stored as a signature, and so the area is maximum. As the ratio drops, the size of the memory decreases, reducing the area overhead. The figure shows results for an architecture with and without a concentrator; as described above, if a concentrator is not present, the decision of which N_{mon} bits must be fixed before fabrication, while if a concentrator is present, this decision can be made during debugging. The difference between the two lines in Figure 5(a) indicates the area cost of this post-fabrication flexibility.

Figure 5(b) shows the area overhead results as a function of the depth of the FIFO, assuming $S_{width} = 0.3N_{mon}$. The larger the depth, the higher the area overhead, but the more cycles that can be backspaced per run of the CUD.

Much of the area overhead in our architecture is due to the Breakpoint Circuit. This circuit requires storing each bit of the target state. As described in Section II, we could use abstraction and match only a subset of the flip flops. Doing so could greatly reduce the size of the circuit, at the cost of losing precision in the debug traces.

Adding the universal hash circuitry to the Signature Creation circuit increases the overhead dramatically. For small designs (such as the 68HC05 described earlier), such circuitry may be feasible. However, the size of a straightforward implementation of the hash circuit grows quadratically with the number of inputs. For the LEON3, if we use a hash circuit similar to what we used for the 8051, the estimated area overhead would be unacceptable (almost 150% *overhead*). By intelligently combining signals (such that the number of XOR gates grows linearly with respect to the number of inputs), we might be able to reduce the area overhead of this structure considerably.

More generally, we believe that signatures can be made much smaller, likely by exploiting architectural insight and design-specific characteristics. For example, Park and Mitra [14] produce information-rich signatures with only 2% area overhead, but narrowly tuned for an Alpha-like processor. We are hopeful we will be able to achieve similarly low overheads in future research.

V. Extensions and Research Directions

This paper introduces a novel paradigm for using formal analysis in post-silicon debug and demonstrates its potential. However, it is only a start. As with any new paradigm, considerable further research remains to be done.

The primary direction for further research is scalability and reducing overhead. As mentioned, we have started work on implementing the BackSpace paradigm on physical hardware, with a multi-thousand latch design. As the idea scales to

[4]http://www.gaisler.com

larger designs, new research challenges will reveal themselves. Some ideas already mentioned include abstraction (with research questions of finding effective abstraction and concretization techniques), better signature functions (that are effective at constraining the pre-images, scalably solvable via SAT, and efficiently realizable in hardware), and reconfigurable BackSpace architectures (how much smaller can the signatures be if they are tailored to a specific target state).

A specific idea, which we have yet to explore but which we believe can greatly reduce signature size, is to use external constraints to prune the pre-images. For example, on toy benchmark circuits, constraining the pre-image by the reachable states of the circuit results in vastly smaller pre-images (equivalently, allows a smaller signature size). Obviously, if we could compute the exact reachable state set, then we would use pure formal verification to hit the target state, obviating the need for the BackSpace approach. But a crude over-approximation of the reachable states might be efficiently computable even for a large design, yet provide useful pruning of the pre-images. Similarly, we are not currently using constraints on the primary inputs to reduce pre-image size. For example, logic analyzer traces or knowledge about the bring-up tests could be used to constrain the primary inputs when computing pre-images.

The other main direction for further research is better handling of realistic designs: partial state monitoring, randomness/non-determinism, multiple clock domains, and circuit marginality and faults. For each of these, there is a straightforward, brute-force attack (partial states and non-determinism were described already, multiple clock domains can be conservatively approximated by a single state machine, and circuit marginality/faults can be handled analogously to fault simulation: for each postulated circuit fault, we repeat the entire BackSpace framework), but much more elegant and efficient approaches are likely necessary and possible.

ACKNOWLEDGMENTS

This work was funded by the Semiconductor Research Corporation (SRC) Task ID: 1586.001. We would also like to thank Brad Quinton for helpful advice on the circuit modeling.

REFERENCES

[1] M. Abramovici, P. Bradley, K. Dwarakanath, P. Levin, G. Memmi, and D. Miller, "A Reconfigurable Design-for-Debug Infrastructure for SoCs," in *43rd Design Automation Conference*. ACM/IEEE, 2006, pp. 7–12.

[2] C. Ahlschlager and D. Wilkins, "Using Magellan to Diagnose Post-Silicon Bugs," *Synopsys Verification Avenue Technical Bulletin*, vol. 4, no. 3, pp. 1–5, September 2004.

[3] ARM, *Embedded Trace Macrocell Architecture Specification*, July 2007, vol. 20, ref: IHI0014O.

[4] M. Boulé, J.-S. Chenard, and Z. Zilic, "Adding Debug Enhancements to Assertion Checkers for Hardware Emulation and Silicon Debug," in *International Conference on Computer Design*. IEEE Computer Society Press, 2006, pp. 294–299.

[5] J. L. Carter and M. N. Wegman, "Universal Classes of Hash Functions," in *9th ACM Symposium on Theory of Computing*, 1977, pp. 106–112.

[6] K.-H. Chang, I. L. Markov, and V. Bertacco, "Automating Post-Silicon Debugging and Repair," in *International Conference on Computer-Aided Design*. IEEE/ACM, 2007, pp. 91–98.

[7] T. Glökler, J. Baumgartner, D. Shanmugam, R. Seigler, G. Van Huben, B. Ramanandray, H. Mony, and P. Roessler, "Enabling Large-Scale Pervasive Logic Verification through Multi-Algorithmic Formal Reasoning," in *Formal Methods in Computer-Aided Design (FMCAD)*. IEEE, 2006, pp. 3–10.

[8] A. J. Hu, J. Casas, and J. Yang, "Efficient Generation of Monitor Circuits for GSTE Assertion Graphs," in *International Conference on Computer-Aided Design*. IEEE/ACM, 2003, pp. 154–159.

[9] N. F. Kelly and H. E. Stump, "Software Architecture of Universal Hardware Modeler," in *European Design Automation Conference (EDAC)*. IEEE, 1990, pp. 573–577.

[10] R. Kuppuswamy, P. DesRosier, D. Feltham, R. Sheikh, and P. Thadikaran, "Full Hold-Scan Systems in Microprocessors: Cost/Benefit Analysis," *Intel Technology Journal*, vol. 8, no. 1, pp. 63–72, February 2004.

[11] M. Larouche, *Infusing Speed and Visibility into ASIC Verification*, January 2007. [Online]. Available: www.synplicity.com/literature/whitepapers/pdf/totalrecall_wp_1206.pdf

[12] S. Mitra and K. S. Kim, "X-Compact: An Efficient Response Compaction Technique for Test Cost Reduction," in *International Test Conference (ITC)*. IEEE, 2002, pp. 311–320.

[13] J. A. M. Nacif, F. M. de Paula, C. N. Coelho, Jr., F. C. Sica, H. Foster, A. O. Fernandes, and D. C. da Silva, "The Chip is Ready, Am I done? On-chip Verification using Assertion Processors," in *International Conference on Very Large Scale Integration of System-on-Chip (VLSI-SoC)*. IFIP WG 10.5, 2003, pp. 111–116.

[14] S.-B. Park and S. Mitra, "IFRA: Instruction Footprint Recording and Analysis for Post-Silicon Bug Localization in Processors," in *45th Design Automation Conference*. ACM/IEEE, 2008, pp. 373–378.

[15] B. Quinton and S. Wilton, "Concentrator Access Networks for Programmable Logic Cores on SoCs," in *IEEE International Symposium on Circuits and Systems*, 2005, pp. 45–48.

[16] ——, "Programmable Logic Core Based Post-Silicon Debug For SoCs," in *4th IEEE Silicon Debug and Diagnosis Workshop*, Germany, May 2007.

[17] S. Safarpour, H. Mangassarian, A. Veneris, M. H. Liffiton, and K. A. Sakallah, "Improved Design Debugging Using Maximum Satisfiability," in *Formal Methods in Computer-Aided Design (FMCAD)*. IEEE, 2007, pp. 13–19.

[18] R. Singhal, K. S. Venkatraman, E. R. Cohn, J. G. Holm, D. A. Koufaty, M.-J. Lin, M. J. Madhav, M. Mattwandel, N. Nidhi, J. D. Pearce, and M. Seshadri, "Performance Analysis and Validation of the Intel Pentium 4 Processor on 90nm Technology," *Intel Technology Journal*, vol. 8, no. 1, pp. 34–42, February 2004.

[19] 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, no. 1, pp. 46–60, January 1973.

Automatic Formal Verification of Block Cipher Implementations

Eric Whitman Smith and David L. Dill
Gates Bldg. 3A, Stanford University, Stanford, CA 94305
{ewsmith,dill}@cs.stanford.edu

***Abstract*—This paper describes an automatic method for proving equivalence of implementations of block ciphers (and similar cryptographic algorithms). The method can compare two object code implementations or compare object code to a formal, mathematical specification. In either case it proves that the computations being compared are bit-for-bit equivalent.**

The method has two steps. First the computations are represented as large mathematical terms. Then the two terms are proved equivalent using a phased approach that includes domain-specific optimizations for block ciphers and relies on a careful choice of both word-level and bit-level simplifications. The verification also relies on STP [5], a SAT-based decision procedure for bit-vectors and arrays. The method has been applied to verify real, widely-used Java code from Sun Microsystems and the open source Bouncy Castle project. It has been applied to implementations of the block ciphers AES, DES, Triple DES (3DES), Blowfish, RC2, RC6, and Skipjack as well as applications of the cryptographic hash functions SHA-1 and MD5 on fixed-length messages.

I. Introduction

A block cipher is a cryptographic algorithm that encrypts a small block of data using a shared, secret key. Block ciphers are critical infrastructure underlying important national security and e-commerce applications. Therefore, their correctness is crucial, and formal verification of block cipher implementations is worthwhile, especially if it can be accomplished with reasonable effort.

The verification method described in this paper can prove mathematically that two computations are equivalent. For block ciphers this means that they produce identical outputs for every possible message and key. To verify an implementation, one can prove it equivalent to either a reference implementation that is strongly believed to be correct, or to a carefully written formal mathematical specification. Such a specification is typically written independently of any implementation and usually very closely resembles the official informal specification of the cipher. Even if there is no formal specification or golden model, proving the equivalence of two implementations produced independently can increase one's confidence in both of them. Two implementations proven equivalent are either both correct, or they both have the same bugs.

A key property of block cipher code is that the loops can be completely unrolled. That is, they have static upper limits on the number of times that they may execute. Usually this is because the cipher consists of a fixed number of "rounds" (e.g., 10 rounds for AES-128 encryption). The verification method unrolls all loops and recursion in the implementations and specifications of the ciphers, generating large loop-free terms. Unrolling avoids the need to specify loop invariants or verify the loops using inductive methods. It remains to prove equivalence of the resulting large terms, and the automation of that process is the focus of this paper.

This paper makes several new research contributions. Most importantly, it demonstrates the feasibility of automatically formally verifying existing block cipher implementations written in a real, widely-used language. It also identifies the key issues in performing such verifications and explains how they can be addressed.

II. Results

Our method has been applied to verify many real-world block cipher implementations written in Java, including all of the block ciphers in Sun's implementation of the Java Cryptography Extension (AES, DES, Triple DES, Blowfish, and RC2) and several ciphers from the open source Bouncy Castle project (three AES implementations, Blowfish, DES, Triple DES, RC2, RC6, and Skipjack). All implementations have been shown to be bit-for-bit equivalent, for all messages and all keys of the given length, to the corresponding formal mathematical specifications. Also, several proofs were done equating Sun and Bouncy Castle implementations of the same ciphers directly, to show that implementations can be compared without the use of formal specifications.

For the AES cipher, four implementations have been verified (three from Bouncy Castle and one from Sun). For each implementation both the encryption and decryption operations were verified for all three official key lengths. The implementations are heavily optimized. Even the simplest one (AESLightEngine) packs the sixteen bytes of the cipher state into four 32-bit machine words and partially unrolls the loops to perform two rounds of AES per loop iteration. AESEngine, AESFastEngine and Sun's AESCrypt are further optimized with lookups into large pre-computed tables. For the ciphers that support many key lengths (RC2 and Blowfish), we have verified only one key length (but other key lengths should be similar).

We also verified implementations of the cryptographic hash functions MD5 and SHA-1 from Bouncy Castle. Unlike block ciphers, these functions take input messages of unknown length. Thus, the method does not directly apply, because the number of input bits is not fixed and the loops cannot be completely unrolled. However, one can fix the length of the

978-1-4244-2735-2/08/$25.00 ©2008 IEEE

input messages and then apply the method. We constrained the MD5 and SHA-1 implementations to each run on 32-bit and 512-bit messages, and then proved them equivalent to the formal specifications.

The proof of a typical cipher can be rechecked by a computer in a matter of minutes or a few hours. For AES-128 encryption, unrolling the specification takes about 7 seconds on a fast machine. Symbolically executing the Java code for the Bouncy Castle's "light'," "regular," and "fast" implementations takes 14, 10, and 16 seconds, respectively. The equivalence proofs connecting the unrolled implementations to the unrolled formal specification take 51, 178, and 167 seconds respectively.

We found that we can often verify a new cipher implementation in a day or two. Our most recent cipher verification, for Skipjack, took less than three hours, including time spent writing and debugging the formal specification.

III. Background and Related Work

The standard approach to validating block cipher implementations is testing. For example, putative AES implementations must pass a suite of tests from the National Institute of Standards and Technology [9]. However, the number of possible inputs to a block cipher is too large to test exhaustively; there are 2^{256} possible inputs for 128-bit AES, which takes a 128-bit message and a 128-bit key. Indeed, any block cipher that did allow for exhaustive testing would likely be insecure, since one could recover a plaintext by trying all possible decryption keys.

There have been other efforts to apply formal methods to block ciphers. For example, Duan, Hurd, Li, Owens, Slind, and Zhang used an interactive theorem prover to prove that decryption inverts encryption for several block ciphers expressed in higher order logic [4]. Their approach seems to require significant manual effort to guide the theorem prover. Also, the inversion property, while important, is fairly weak; insecure implementations that do not conform to the AES standard may still satisfy inversion (consider the trivial implementation that does nothing at all for both encryption and decryption). By contrast, our verification method proves bit-for-bit equivalence between each implementation and a formal specification or a reference implementation. Also, our approach applies to real-world implementations in Java bytecode (rather than the native language of a theorem prover).

Toma and Borrione used the ACL2 interactive theorem prover to verify a hardware implementation of SHA-1 [12]. Their work also seems to involve significant manual effort to guide the prover.

Galois Connections is developing Cryptol, a domain specific language for cryptography which can be compiled down to an implementation using a verifying compiler [3] [10]. (The same seems possible for the specifications of block ciphers analyzed by Duan et. al.) In contrast, our work verifies pre-existing block cipher implementations not written using a correct-by-construction framework.

Yin, Knight, Nguyen, and Weimer, have applied their Echo tool to verify an implementation of AES [14] by repeatedly refactoring the code to remove optimizations. Their approach seems less automatic than ours because the user must often identify the transformations to be performed (e.g., finding instances of word packing or specifying the patterns encoded in the values of lookup tables). Our work makes no such requirements.

Sean Weaver, in a presentation posted online, has described a verification method similar to the equivalence checking phase of our method [13]. Both methods decompose large equivalence proofs by identifying pairs of probably equal nodes using random test cases, and both make calls to boolean satisfiability solvers. Weaver has applied his method to verify an AES implementation, but his presentation does not address other ciphers. We found AES to be among the easiest ciphers to verify, and many of the complications we address later in this paper arose in the verification of other ciphers.

Our approach borrows several techniques from combinational equivalence checking, which is commonly used in hardware verification. In particular, the use of random test cases to find internal equivalences goes back at least to Berman and Trevillyan's work in 1989 [2]. Their approach, as does Kuehlmann's [6], involves proving the discovered equivalences in a bottom-up fashion – much like our method. Early approaches used binary decision diagrams, but state-of-the-art work on combinational equivalence checking, including Kuehlmann [15]'s later work and the checker ABC [7], involves "SAT-sweeping" or "fraiging," in which functionally equivalent nodes in a miter are identified through simulation and proved equivalent by a satisfiability solver. Our equivalence checking phase is very similar.

A key distinguishing feature of our approach is that it uses word-level simplifications to normalize terms as much as possible. (By "word-level" we mean simplifications performed before "bit-blasting," when the terms involved are still entire bit-vectors rather than single bits.) Our simplifications are not guaranteed to give identical representations for equivalent terms, but our approach proves to work well in practice; simplifying transformations alone suffice to verify many block cipher implementations, with no equivalence checking phase required.

IV. The Verification Method

A. Inputs to the method

The verification method takes as input a collection of compiled Java class files that together implement a cryptographic algorithm (e.g., the cipher engine class, its helper classes, and its ancestor classes and interfaces). Verifying class files instead of source code has several advantages. First, class files contain the actual bytecodes that will run on the machine, and object code verification can thus catch errors introduced during compilation. Second, source code may not always be available for analysis. Finally, it is convenient to model Java bytecode using operational semantics, using an executable formal model of the Java Virtual Machine. (The model is

derived from the M5 model of Moore et al. [8].) Unlike many formal verification frameworks, no programmer annotation of the source code is required.

Our method can compare two Java implementations of the same algorithm, but often a Java implementation is instead compared to a formal, mathematical specification. Such a specification must be written by hand by translating the document that defines the cipher (e.g., FIPS-197 for AES [1]) into a formal language. Writing and debugging a specification for a new cipher typically takes a few hours. Once the specification is written it can be reused to verify many implementations of the algorithm. Formal specifications have been written for AES, DES, Triple DES, Blowfish, RC2, RC6, Skipjack, MD5, and SHA-1. Each is written in the native language of the ACL2 theorem prover, a side-effect-free dialect of Common Lisp with well-defined semantics. The specifications are written without optimizations, so as to be as close to the official documents as possible. They are also executable and can be validated using test cases. For example, we have tested our AES specification by running it on all the test cases given in the standard.

The user of our verification method must provide a few simple inputs. First, the user must write a driver that calls the cryptographic algorithm to be verified. The driver is a small Java method that takes an input message and a key and calls the cipher in the typical way. Usually this involves allocating a new object of the cipher class, perhaps wrapping its key in a "key parameter" object (for the Bouncy Castle ciphers), perhaps calling an initialization routine, then calling the main encryption routine, and finally perhaps calling a finalization routine (for the cryptographic hash functions). Such a driver would be easy to write for anyone capable of writing code that uses the cipher. Also, drivers for different ciphers implementing the same Java interface or abstract class (e.g., org.bouncycastle.crypto.BlockCipher) are often very similar.

The user must also specify the Java class files containing the code to be executed. (We could improve our tool to generate this list automatically.)

Finally, the user must list the input variables and their sizes and indicate how the inputs of the two computations match up (as must always be done when making a miter). Handy macros are provided to help. Given these inputs, the method runs automatically.

B. Terms and simplifications

The verification method has two steps. First, the two computations to be compared are represented as large mathematical terms including bit-vector and array operations. Then the terms are compared to establish that they compute identical outputs for all possible inputs. The comparison is the main focus of this paper, but first we briefly describe how terms are represented and simplified, and how suitable terms are generated from Java bytecode and formal specifications.

Computations are represented as large mathematical terms. Identical subterms are shared, so the terms are really directed acyclic graphs. The "nodes" of a term include its input variables (which together determine its value), constant nodes, and nodes which represent function applications. A function application includes an operator and a list of arguments (constants and other nodes called the node's "children"). Nodes are numbered, and we require that a node have a higher number than any of children. Thus our terms are acyclic. Our terms differ from operator trees in an important way: shared subterms are represented only once (that is, we perform "structural hashing"). This is crucial because block cipher computations perform extensive mixing operations in which results from previous rounds appear many times in the expressions for future rounds. Sharing subterms results in exponential space savings when representing these computations. Our representation of terms is also generic. The operator of a node can be any function in the ACL2 logic, such as a recursive Lisp function from a formal specification or one of the functions used to model the complicated state of the Java Virtual Machine. Through unrolling and simplification, terms containing only bit-vector and array operators can be obtained.

The loop-free terms representing block ciphers are large. For example the term for Sun's Blowfish encryption operation using a 64-bit key has 220,811 nodes after simplification (and before word operations are bit-blasted into bit operations).

It is often important to simplify or transform terms, and a novel tool is employed to do so by applying simplification rules. The rules can range from simple identities to sophisticated transformations that handle common coding idioms. Each simplification replaces a subterm with an equivalent but simpler term, and successful simplifications often enable further simplifications later on. The careful choice of which simplifications to perform, and the order in which to perform them, is important to successful verification. Simplifications serve several purposes. First, they reduce the size of terms involved in the verification effort. More importantly, they often allow for the normalization of terms; that is, logically equivalent terms are transformed to have the same syntactic form (and so are clearly equal by inspection). We have found normalization to be crucial to the verification effort. In many cases, normalizing terms using our rules is sufficient in itself to prove the equivalence of two cipher implementations. This is the case for the RC2, RC6, Blowfish, Skipjack, and "light" AES implementations from Bouncy Castle and for the RC2 and Blowfish implementations from Sun. For these examples, two completely independent representations of the same computation (one in Java bytecode and the other in ACL2's language of recursive functions) can be proved equivalent just by applying simplifications. The tool used to simplify terms is similar to the simplifier of the ACL2 theorem prover and borrows several key ideas from it. However, a crucial difference is that our tool can efficiently handle terms with many shared subterms. ACL2's inability to do so makes it unable to handle the unrolling of block ciphers. The simplification rules applied by the term simplifier are stated as ACL2 theorems and can be proved. (We have proved most of our rules and are adding proofs for the rest.)

C. From bytecode and specifications to mathematical terms

In order to verify a Java bytecode implementation of a block cipher, one must first extract a term representing the core computation of the cipher. Our tool for doing so uses symbolic execution and the term simplifier described above. While not the main focus of this paper, we briefly describe the process here.

Extracting the computation performed by a bytecode program is a non-trivial task. The Java bytecode language includes many complicated concepts, including field and method resolution, allocation of new heap addresses, static initializers of classes, the use of values from the runtime constant pool, string interning, the throwing of exceptions, etc. Surprisingly, many of these complicated language features are used incidentally in common block cipher implementations. To extract the core computation performed by a program, the tool symbolically simulates the operation of the Java Virtual Machine (JVM). The simulation is done by using the term simplifier to repeatedly simply an application of the "run until return" function when the machine is executing the driver method. The details are beyond the scope of this paper, but the simulation essentially steps through the computation one instruction at a time until the driver method finally returns. This amounts to a full unrolling of all the loops and a full inlining of all subroutine calls. Our approach is very similar to that used by J Moore et. al. with their ACL2 models of the JVM. Key differences are that we use an improved version of Moore's M5 machine model which is more amenable to formal reasoning and that our terms share common subterms. Our tool also handles conditional branches in a sophisticated way to prevent some simulations from splitting into exponentially many cases. (Briefly, it simulates the branches independently until flow reconverges and then combines the resulting states, using an if-then-else operator for any state components that differ between the branches.)

An important feature of our symbolic simulation is that it simplifies terms as it goes. This keeps the term size manageable and can prevent the simulations from encountering exceptions or error states. For example, the JVM performs a bounds check for each array access and throws an exception if the check fails. The on-the-fly application of simplifications allows such bounds checks to be discharged during the simulations. The symbolic simulation process can extract bit-accurate representations of the computations performed by long symbolic executions (tens of thousands of Java bytecodes, with tens of thousands of nodes in the resulting mathematical terms).

Formal specifications must be unrolled into terms involving only bit-vector and array operations. The term simplifier is used to expand and unroll all other functions, including recursive functions. At each stage of the unrolling, it is usually possible to determine whether a recursive function terminates in a base case or whether another recursive call occurs. Non-recursive functions are simply expanded in place.

D. Proving equivalence of terms

The equivalence proof for two terms is done in phases. A (miter) term is built to represent the equality of the two terms, and the goal of equivalence checking is to reduce the equality to true.

In some cases, simplifications at the word-level suffice to prove the equality of the two computations. If not, the next phase, bit-blasting, is applied. Bit-blasting splits word-level operations into bit-level operations; it helps regularize the structure of the two computations and prepares them for the next phase. Simplifying both before and after bit-blasting is important, because transformations obvious in one phase often are not obvious in the other. For example, it is convenient to apply commutativity of 32-bit addition at the word-level, but after the addition operations have been bit-blasted (turned into ripple-carry adders), the commutativity property is no longer easy to apply. If simplifying and bit-blasting are not enough to prove the computations equivalent, the final phase, equivalence checking, is applied to the simplified term.

A vast amount of work has been done on expression optimization and logical transformations, and many of the transformations and simplifications applied before equivalence checking are well known. The challenge is in choosing which transformations to apply (and which ones not to apply) and in what order to apply them. Seemingly minor changes in the simplifications applied or in their order can make the difference between the equivalence checking process taking a long time (or never finishing at all) and finishing quickly. Also, transformations to handle certain coding idioms (discussed later) sometimes prove to be crucial. The transformations used in our verification method have been motivated by real problems encountered in verifying block cipher code. When we found that STP was unable to discharge a proof obligation, we analyzed the failure and added transformations to handle the relevant issues. We now have a set of transformations which we believe to be robust and effective. Most of the transformations are currently performed outside of STP, but we may soon modify STP to perform them.

Block cipher implementations have several characteristics which make them difficult to verify, and we discuss these in the following sections.

E. Handling bit manipulation

Block ciphers perform extensive bit manipulation (shifting, masking, concatenating, extracting, rotating, etc.). For example, the AESLightEngine class achieves efficiency by packing data bytes into machine words; each group of four bytes is concatenated into a 32-bit word. The bytes are repeatedly extracted from the packed representation, processed, and then repacked. A complication in verifying implementations that manipulate bits is that different programmers use different idioms to perform equivalent bit manipulations. For example, bit-vector concatenation is usually expressed by shifting one value and then combining it with the other value using an operation such as OR.

However, instead of the OR operation, the XOR operation is sometimes used. Since there is always at least one zero in every pair of bits to be combined (due to the shift), OR and XOR behave the same. Addition is also sometimes used, since the presence of zeros guarantees that there is never any carry from one bit position to the next. The use of different operations to implement concatenation means that equivalent operations may not have the same syntactic form. So our method recognizes these patterns and changes them into calls of the bit-vector concatenation operator, which provides a unique representation for this operation and best reflects the computation actually being performed. The rules to turn occurrences of OR, XOR, and addition into equivalent concatenation operations are a bit complicated; they require the presence of zeros in such a way that the combination operations never combine two ones. Yet they reflect common coding idioms and are not specific to any particular block cipher. Bit manipulation issues such as these could be handled by simply bit-blasting all word-level operations immediately. However, it is more efficient to do these transformations at the word level first, which may enable further word-level simplifications.

F. Handling bit rotations

Another common operation in block cipher code is bit rotation, in which bits shifted out are not dropped but rather used to fill in the positions opened up by the shift. (Since no bits are lost, the operation, like the cipher as a whole, is reversible.) Java and C programs implement rotation using other operations. A common idiom is to perform two shifts followed by a combination operation (OR, XOR, or addition, as discussed in the previous section). Again, the existence of several different but equivalent idioms requires special handling when normalizing terms.

When the rotation amount is a known constant, a rotation operation can be turned into a concatenation of two bit slices and then handled using the techniques for concatenation. However, some block ciphers (e.g., RC6) contain "variable" or "data-dependent" rotations," in which the rotation amounts are not constants but rather depend on the inputs to the cipher [11]. Such operations cannot be easily turned into concatenations of slices because the indices in the slices and the bit-widths of the resulting values would not be constants, which most bit-vector theories require. Also, variable rotations cannot be bit-blasted, since one doesn't know which bit will end up in which position.

So variable rotations are handled specially. The rotation idiom (two shifts and a combination using OR, XOR, or addition) is recognized and transformed into a call to a special LEFTROTATE operator. (Any right rotation can also be expressed as a left rotation.) If subsequent simplifications do not simplify the whole overarching term to true, the rotations must be expanded before equivalence checking. This can be done with if-then-else operators that test the rotation amount. However, wrapping occurrences of the rotation idiom into LEFTROTATE operations sometimes enables the entire overarching term to be simplified down to true (as is the case for RC6), with no equivalence checking phase required.

G. Handling lookup tables

A common optimization in block cipher code is the use of pre-computed lookup tables to replace sequences of logical operations (e.g., XORs, shifts, masks, etc.). Such optimizations can interfere with the comparison of an optimized implementation with an unoptimized reference implementation or specification. Lookup tables are represented in our terms as array subterms with constant values for all of their elements, and our verification method can in some cases exploit patterns in the array values to rewrite table lookups into the equivalent logical operations.

Consider for example a three-bit bit-vector $x = x_2x_1x_0$ for which one wants to compute the three bit result $(x_2 \oplus x_1)@(x_2 \oplus x_0)@(x_1 \oplus x_0)$. Here "@" denotes concatenation and $\oplus$ denotes the exclusive-or operation. To compute the XOR of two of the bits of x would require several operations (a shift, an XOR, a mask, and possibly another shift to put the result into position). Instead of doing that for each bit of the result, one could simply use x as an index into the lookup table T:

```
T[000] = 00000000  T[100] = 00000110
T[001] = 00000011  T[101] = 00000101
T[010] = 00000101  T[110] = 00000011
T[011] = 00000110  T[111] = 00000000
```

where the table elements are bytes expressed in binary. This example is representative of what appears in practice in block cipher code: a lookup table in which each bit of the values encodes the XOR of some subset of bits from the table index. Our method translates such tables to logical terms when it is simple to do so. First, tables (such as T) whose elements are more than one bit wide are blasted into tables with one-bit elements. Table T would be blasted into 8 tables, one for each column of bits; the table T_0 for the least significant bits would contain the rightmost bits of T's values. A lookup, $T[x]$ would be suitably transformed into the equivalent concatenation: $T_7[x]@T_6[x]@T_5[x]@T_4[x]@T_3[x]@T_2[x]@T_1[x]@T_0[x]$. Note that after the blasting of T, five of the resulting tables (T_7 down through T_3) contain all zeros. Clearly a lookup into a table with all zeros returns zero, and the method makes that simplification. More generally, if all the values in a table are the same, a lookup into the table is simplified to that value.

It remains to turn the nonzero single-bit tables, such as T_0 above, into logical formulas. The method recognizes when table entries correspond to XORs of certain index bits, and so can transform $T_0[x]$ into $x_1 \oplus x_0$. One could include other simplifications as well (e.g., identifying index bits that are ANDed in or ORed in), but the ones described above are sufficient to handle the examples that we have seen in practice (since XOR is very common in block ciphers).

H. Normalizing XOR terms

The XOR operation is very common in cryptography because it is easily invertible, and the verification method

normalizes nested XOR terms. It is important to handle XOR "nests" before translating to conjunctive normal form; SAT-based tools often handle XORs poorly, and the XOR nests sometime involve many operands. Since XOR is associative and commutative there are many ways to express the result of XORing several values together (i.e., there are many XOR trees of various shapes for a given set of leaves), and the verification method ensures that equivalent nests are transformed to be syntactically identical. The simplifications ensure that, after normalization, an XOR nest has the following properties: All constituent XOR operations are binary and are associated to the right. Values being XORed are sorted (by their node number in the overarching term), with constants at the front. Multiple constants are combined, and a constant of 0 is dropped (since XORing with 0 has no effect). Pairs of the same value are removed (since XORing a value with itself gives 0). Negations of values being XORed are turned into XORs with ones (since negating a bit is equivalent to XORing it with the value 1).

For example, if an XOR nest contains both the bit x and its negation, the latter will become an XOR of x with the constant 1, the 1 will get moved to the front (and perhaps combined with other constants), and the two x's will be removed as a pair of duplicates. After normalization, two XOR nests which could be shown to be equivalent using only the properties of XOR and negation are transformed to be syntactically identical. This normalization can enable further simplifications and help the rest of the verification succeed.

In many cases, applying word-level simplifications, bit-blasting, and then simplifying again suffices to complete the verification. If not, the simplified term is passed to the final verification phase, equivalence checking. In such a case, the normalizing transformations done during the simplification phase may aid the equivalence checking phase because the two implementations have been made more similar.

V. Equivalence Checking

Equivalence checking is the final phase of the verification method. Often the terms involved are large (tens of thousands of distinct nodes, even after the simplifications of the previous phases). Simply calling an off-the-shelf satisfiability solver, combinational equivalence checker, or decision procedure for bit-vectors and arrays rarely produces good results.

To prove equivalence of the large terms that represent block cipher computations one must use correspondences between internal nodes of the two computations to break down the large equivalence proof into a sequence of smaller ones. This is a well known technique in equivalence checking, and it works quite well for block ciphers and similar computations. One reason is that such computations are usually structured as a sequence of rounds. A cipher typically has a set of bits that represent its internal state, and each round modifies these bits. Different implementations may compute the rounds differently (e.g., using different optimizations or by executing operations in a different order), but the state bits almost always match up between rounds. Since computations are bit-blasted before equivalence checking is done, differences in data packing will not prevent internal correspondences from being detected, since the nodes considered will be single bits.

In our approach, random test cases are used to identify nodes which match up between the two implementations. Clearly if any test case falsifies the top equality node, a counterexample has been found and the verification attempt has failed. Otherwise, nodes which agree in value for all test cases are considered to be "probably equal". In practice a few dozen to a few hundred test cases usually suffice to eliminate most spurious "probably equal" pairs. (We also detect "probably constant" nodes, but such nodes rarely survive simplification.)

Pairs of "probably equal" nodes are used to split the large equivalence proof into a sequence of smaller equivalence proofs, one for each pair. When two "probably equal" nodes are proved to be actually equal, they are merged. That is, one node is chosen as the representative, and all parents of the other node are changed to depend on the representative. The proving and merging proceeds up the term, from the inputs toward the root, until the nodes for the two computations are proven equivalent and merged. When that happens the root node becomes an equality of some node with itself, which is trivially true. All this is very similar to the SAT-sweeping (also called "fraiging") done by combinational equivalence checkers such as ABC [7].

The equivalence proofs for pairs of nodes are done by calling out to STP, a decision procedure for bit-vectors and arrays. However, the terms involved are often very large, so abstraction is used to keep the formulas sent to STP small when possible. Proofs are cut by replacing heuristically-chosen nodes with primary inputs, allowing large shared subterms to be ignored and not given to STP. Adding primary inputs for a proof results in an STP goal that is more general than the original, so if STP proves it, the original equivalence is also established. Cutting equivalence proofs in this way is a known technique, and it suffers from false negatives. When STP fails to prove the cut formula, less aggressive cuts are attempted until either STP is given a formula it can prove or it reports a counterexample for the full formula. If a counterexample is found, then the "probable equality" must in fact be false. In such a case, the nodes incorrectly thought to be equal are simply not merged, and the equivalence check continues with the next pair of "probably equal" nodes (after printing a message to the user about the failure). Such false negatives are particularly rare with block ciphers. Usually a false negative (represented as an assignment of values to the nodes along a cut) occurs because those nodes cannot actually assume that particular combination of values when the surrounding context is considered. But the nodes representing the state of a block cipher can usually take on all combinations of values.

Since verifying block cipher implementations amounts to proving combinational miters, we compared our method to ABC, a state-of-the-art hardware equivalence checker. Of course, ABC supports neither Java bytecode nor recursive functions, so we again used our term simplifier to do the symbolic simulations and unroll the specifications. To test

the effect of our word-level transformations, we refrained from performing them before calling ABC. We did have to perform transformations to get rid of all array operations (since ABC does not support them) and to bit-blast all word-level operations into simple logic gates. We didn't "charge" any of the translation time to ABC. Still, ABC took significantly longer than our tool. For "light" AES, our tool took 72 seconds total (including the symbolic simulation and specification unrolling). ABC took 385 seconds just for the equivalence check (or longer, depending on which set of simplifications we applied before calling it). For "regular" AES, our tool took 196 seconds total, but ABC took 2149 seconds for the equivalence check (or more, again, depending on which set of simplifications we performed first). For RC6 (a more challenging example), our method took 48 seconds total, and ABC took at least 1.5 days (at which point we got a "bus error").

VI. Conclusions and Future Work

The verification method of this paper demonstrates the feasibility of highly automated correctness proofs of block cipher implementations. The correctness results are strong (bit-for-bit equivalence with the official specification or between two implementations), and the effort and computer time required are relatively low. Thus, our method may be worth applying on a routine basis.

The techniques described in this paper should apply to implementations of block ciphers in other programming languages, machine languages, or hardware circuits, as long as one can provide appropriate translators to unroll the computations into mathematical terms. Future work might involve dealing with loops in cryptographic code in a better way, so that we can verify cryptographic hash functions on messages of arbitrary lengths.

VII. Acknowledgment

This work was supported by the National Science Foundation, under the ACCURATE program (sponsor reference number CNS-0524155-003). Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the view of the National Science Foundation.

References

[1] Federal Information Processing Standard Publication 197. http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf.

[2] C. L. Berman and L. H. Trevillyan. Functional comparison of logic designs for vlsi circuits. In *Digest of Technical Papers of the IEEE International Conference on Computer-Aided Design*, Nov. 1989.

[3] Galois Connections. Cryptol. http://www.cryptol.net/.

[4] J. Duan, J. Hurd, G. Li, S. Owens, K. Slind, and J. Zhang. Functional correctness proofs of encryption algorithms. In G. Sutcliffe and A. Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning (LPAR 2005)*, volume 3835 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, December 2005.

[5] Vijay Ganesh and David L. Dill. A decision procedure for bit-vectors and arrays. In *Computer Aided Verification (CAV '07)*, Berlin, Germany, July 2007. Springer-Verlag.

[6] Andreas Kuehlmann and Florian Krohm. Equivalence checking using cuts and heaps. In *Design Automation Conference*, pages 263–268, 1997.

[7] Alan Mishchenko, Satrajit Chatterjee, Robert Brayton, and Niklas Een. Improvements to combinational equivalence checking. In *ICCAD '06: Proceedings of the 2006 IEEE/ACM international conference on Computer-aided design*, pages 836–843, New York, NY, USA, 2006. ACM.

[8] J Moore. Proving Theorems about Java and the JVM with ACL2. http://www.cs.utexas.edu/users/moore/publications/marktoberdorf-02/index.html.

[9] N.I.S.T. and Lawrence E. Bassham III. The Advanced Encryption Standard Algorithm Validation Suite (AESAVS). http://csrc.nist.gov/groups/STM/cavp/documents/aes/AESAVS.pdf.

[10] Lee Pike, Mark Shields, and John Matthews. A verifying core for a cryptographic language compiler. In *ACL2 '06: Proceedings of the sixth international workshop on the ACL2 theorem prover and its applications*, pages 1–10, New York, NY, USA, 2006. ACM.

[11] Ronald L. Rivest, M.J.B. Robshaw, R. Sidney, and Y.L. Yin. The RC6 Block Cipher. http://theory.lcs.mit.edu/~rivest/rc6.pdf.

[12] Diana Toma and Dominique Borrione. SHA formalization. In *Fourth International Workshop on the ACL2 Theorem Prover and Its Applications (ACL2 '03)*, July 2003.

[13] Sean Weaver. Equivalence checking. http://gauss.ececs.uc.edu/Courses/C702/Weaver/ec.01.08.07.ppt.

[14] Xiang Yin, John C. Knight, Elisabeth A. Nguyen, and Westley Weimer. Formal verification by reverse synthesis, to appear in SAFECOMP 2008.

[15] Qi Zhu, Nathan Kitchen, Andreas Kuehlmann, and Alberto Sangiovanni-Vincentelli. Sat sweeping with local observability don't-cares. In *DAC '06: Proceedings of the 43rd annual conference on Design automation*, pages 229–234, New York, NY, USA, 2006. ACM.

Verifying an Arbiter Circuit

Chao Yan and Mark R. Greenstreet

Department of Computer Science
University of British Columbia
{chaoyan, mrg}@cs.ubc.ca

***Abstract*—This paper presents the verification of an asynchronous arbiter modeled at the circuit level with non-linear ordinary differential equations. We use Brockett's annulus to represent the allowed families of continuous waveforms for input and output signals and show that the metastability filter of the arbiter can be understood as a "Brockett annulus transformer." Improvements to the Coho verification tool are described that reduce the over approximation errors when working with non-convex reachable regions. The verification shows that the arbiter observes a four-phase handshake protocol with its clients and maintains mutual exclusion. We also show several liveness properties including bounded time response to uncontested requests and that grants are issued fairly.**

Keywords: Arbiters, formal verification, metastability dynamical systems.

I. Introduction

Deep submicron designs present designers with increasing challenges. On the one hand, correct circuit operation depends on properly accounting for a myriad of effects including leakage currents, crosstalk, power-supply noise and random parameter variations. On the other hand, the large device count enables the implementation of complex systems on chip, requiring designers to rely on higher levels of abstraction. Formal methods can help address these challenges by verifying proper behaviors at low-levels of abstraction and ensuring that the abstractions of these analog behaviors to discrete ones are sound.

This paper presents a case study of such verification: we verify the correct operation of an asynchronous arbiter. We start with a specification of a discrete arbiter including that it maintains mutual exclusion, handshakes properly with its clients, fairness, and bounded time response to uncontested requests. Verifying the arbiter raises interesting issues because the clients may make requests concurrently, and the arbiter may take arbitrarily long to respond to such requests due to metastability. We describe an approach for creating abstraction mappings from continuous waveforms to discrete sequences. We show how this framework allows us to describe the allowed continuous inputs to the arbiter, including signals with varying rise and fall times and arbitrary relationships between the assertions of requests by the two clients. Section II presents the specification of the discrete arbiter, abstraction mappings between continuous and discrete state spaces, and the arbiter circuit that we use as an example. We then summarize previous work on arbiters in circuit level verification in Section III.

We verified the arbiter using the Coho verification tool. Coho computes over approximations of the reachable space to guarantee the soundness of its results. We found that some of these approximations became so large that they prevented successful verification of the arbiter. This is mainly because of the non-convexity of the reachable space for the arbiter. While Coho was capable in principle of representing non-convex objects, in practice, Coho's approximations often filled-in these non-convexities. Section IV shows how we modified Coho to reduce these over approximations, and thus enable Coho to compute non-convex regions that bound the reachable space for the arbiter and thereby verify the arbiter's correctness.

Section V presents the actual verification of the arbiter, and Section VI summarizes these results and identifies areas for further research.

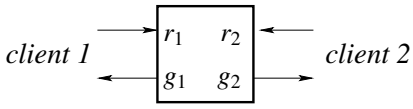


Fig. 1. An Arbiter

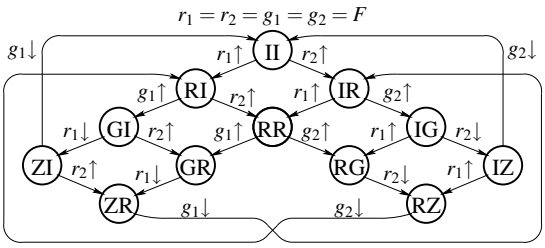


Fig. 2. Transition Diagram for a Discrete Arbiter

II. Arbiters

An arbiter provides mutual-exclusive access to a resource for some set of clients. In this paper, we consider an asynchronous arbiter with two clients as shown in Figure 1. The two clients interact with the arbiter using a four-phase handshake protocol: client i raises r_i to request the privilege; the arbiter raises g_i to grant client i the privilege; when the client is done with the privilege, it lowers r_i; and finally the arbiter lowers g_i to complete the handshake. The arbiter must guarantee mutual exclusion; in other words, signals g_1 and g_2 may not both be high at the same time.

A. Discrete Arbiters

We first present a specification for arbiters where behaviors are modeled as discrete sequences of discrete values. In particular, we specify safety properties with a transition diagram,

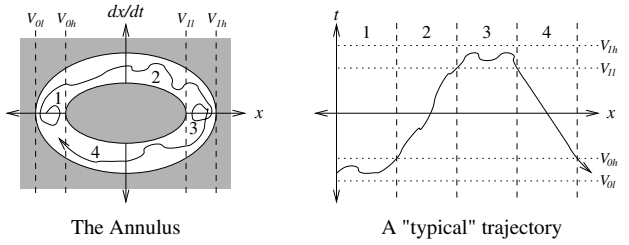


Fig. 3. Brockett's Annulus

and then state additional liveness and fairness requirements. Figure 2 shows the transition diagram. Each state has a two-letter label: the first letter indicates the state of the handshake between client 1 and the arbiter; and the second letter denotes the state of the handshake with client 2. In particular,

> "I" indicates idle: both the request and grant are false;
> "R" indicates requesting: the request signal is true, but the arbiter has not yet asserted the corresponding grant;
> "G" indicates granted: both the request and grant are true;
> "Z" indicates resetting: the request signal is false, but the arbiter has not yet lowered the corresponding grant.

Signals r_1 and r_2 are inputs. If either of these signals make a transition that is not in the diagram, that is a *failure of the environment*; in other words, the client for that input has violated the handshake protocol, and after the specification places no restriction on the behavior of the arbiter. In other words, diagram has implicit edges for all such events to a sink state for environment failures. Conversely, if either g_1 or g_2 make a transition that is not in the diagram, that is a *failure of the circuit*, and the diagram has implicit edges for all such events to a sink state for circuit failures. Thus, the arbiter verification problem includes showing that the circuit failure state is unreachable. By inspection of the transition diagram, this implies that the arbiter follows the handshake protocol and guarantees mutual exclusion.

We also want to establish liveness properties for the arbiter. In particular, we require that any uncontested request is eventually granted, and that once a client drops a request, the grant for that client will eventually be withdrawn. We do this by requiring that no trace can end in states RI, IR, ZI, ZR, IZ or RZ. Obviously, it would be desirable if the arbiter were guaranteed to eventually issue a grant when contested requests occur; i.e., that state RR cannot be terminal. It is well known that a real arbiter cannot satisfy this requirement along with the safety requirements described above (see, for example, [1]). Thus, here we do not give a liveness requirement for traces with contested requests. Finally, we require that the arbiter is fair: if client i has made a request, the other client can only receive a finite number of grants before the client i is granted. In fact, we will show that the arbiter considered in this paper issues at most one grant to one client while the other is waiting.

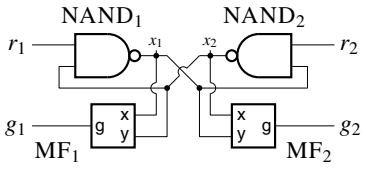
B. Continuous Arbiters

To specify the continuous behavior of the arbiter, we need an abstraction mapping from the continuous model to the discrete one. In the continuous model, signals will be modeled as continuous functions of time, i.e., as functions from $\mathbb{R}^+$ (time) to $\mathbb{R}$ (voltage or some other physical quantity). We assume that the circuit is modeled as a system of ordinary differential equations (ODEs) and that the time derivatives of all signals are defined and bounded everywhere along any trajectory.

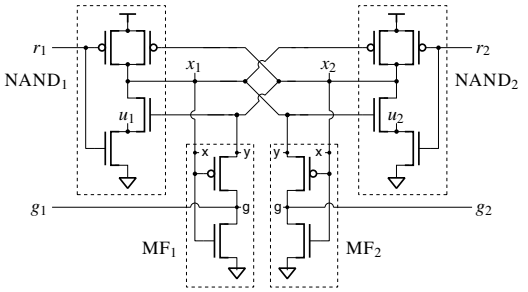
Our abstraction is based on the Brockett annulus construction [2] depicted in Figure 3. When a variable is in region 1, its value is constrained but its derivative may be either positive or negative. When the variable leaves region 1, it must be increasing; therefore, it enters region 2. Because the derivative of the variable is positive in region 2, it makes a monotonic transition leading to region 3. Regions 3 and 4 are analogous to regions 1 and 2 corresponding to logically high and monotonically falling signals respectively. This provides a topological basis for discrete behaviors – the hole in the middle of the annulus forces transitions to be unambiguous. A signal cannot meander to some value between the low and high values and then return to where it came from without making a complete transition. Furthermore, the horizontal radii of the annulus define the maximum and minimum high and low levels of the signal (i.e. V_{0l}, V_{0h}, V_{1l}, and V_{1h} in Figure 3). The maximum and minimum rise time for the signal correspond to trajectories along the upper-inner and upper-outer boundaries of the annulus respectively. Likewise, the lower-inner and lower-outer boundaries of the annulus specify the maximum and minimum fall times. We also add constraints specifying the minimum low and high times for signals, i.e., the minimum duration of sojourns in regions 1 and 3.

Brockett annuli provide a basis for mapping continuous trajectories to discrete sequences. We will say that a signal makes a rising transition when it enters region 2 of its Brockett annulus and a falling transition when it enters region 4. Because Brockett annuli imposes minimum rise and fall times for signals, the number of rising and falling transitions is countable. To obtain a discrete sequence from a continuous trajectory, we "sample" the trajectory at the times when any signal transitions from region 1 to region 2, or from region 3 to region 4. At each such sampling time, t_s, we map a signal to "true" if its right limit at time t_s is in region 2 or 3 and to false if its right limit is in region 1 or 4. These limits exist by our assumption that the derivative function is defined everywhere along any trajectory.

Note that a Brockett annulus describes an entire family of trajectories. Given any trajectory, $x(t)$, with a trajectory that is contained in the interior of the annulus, any trajectory $x'(t)$ that is obtained from a small, differentiable perturbation of $x(t)$ is also in the annulus. This is in contrast with traditional circuit simulators that test a circuit for specific stimuli such as piecewise linear or sinusoidal waveforms. Thus, a Brockett annulus can be given that contains all trajectories that will occur during actual operation, something that traditional simulation cannot achieve. Of course, such an annulus also includes trajectories that will never occur during actual operation. Thus,



(a) Gate-level schematic for an arbiter.



(b) Transistor-level schematic for the arbiter above.

Fig. 4. An Arbiter Circuit

this abstraction is sound in that false positives are excluded, but false negatives could, in principle, occur. In our experience, the Brockett annulus abstraction has not been a cause for false negatives; these arise instead due to the over approximations in the reachability computation as described in Section IV.

Verification that an ODE model for a circuit satisfies a discrete specification proceeds in four steps:

1) Give a Brockett annulus specification for each continuous signal.
2) Compute an invariant set that contains all reachable trajectories of the continuous system. Here, we assume that all inputs to the circuit satisfy their Brocket annuli, and that their discrete abstractions do not produce environment failures.
3) Verify that each signal produced the circuit satisfies its Brockett annuli.
4) Verify that the discrete abstractions for all trajectories in the invariant step computed in step 2 satisfy the discrete specification.

We apply this process to an asynchronous arbiter in the remainder of this paper. First, we describe the circuit that we will verify.

C. An Arbiter Circuit

Figure 4 shows the schematic for a commonly used arbiter circuit. The arbiter consists of two parts: an R/S latch implemented by a pair of cross-coupled NAND gates, and a "metastability filter" for each grant output. When both request inputs are low, both NAND gate outputs are driven high, which in turn causes both grant outputs to be driven low. If client i makes an uncontested request, then x_i goes low in response to r_i going high which in turn causes g_i to go high. At this point the RS latch is set, and a transition on the other request input will have no effect until r_i goes low.

If both requests go high at roughly the same time, then metastability may occur. In this case, both x_1 and x_2 drop to an intermediate voltage that is the unstable equilibrium for the feedback cycle through the two NAND gates. The arbiter may remain in such a state arbitrarily long although the probability of unresolved metastability drops exponentially with time for a properly designed arbiter. Propagating such intermediate values back to the clients could cause them to malfunction. Thus, the arbiter includes *metastability filter* circuits to ensure that the grant outputs make clean transitions even if the NAND-gate outputs do not.

The metastability filter is a modified inverter. Consider circuit MF_1. The gates of the transistors in the inverter are connected to x_1. Unlike a traditional inverter, the source of the PMOS pull-up is connected to x_2. With this configuration, the pull-up transistor remains in cut-off until x_1 is at least the PMOS threshold voltage below x_2. This is to prevent g_1 from moving any significant amount above ground until client 1 has clearly won the arbitration.

In our framework for mapping continuous trajectories to discrete sequences, the metastability filter can be understood as a "Brockett annulus transformer." If the arbiter is correctly designed, then x_1 and x_2 will satisfy a Brockett-annulus specification, but the lower limit for region 3 (logically high signals) must be below the metastable voltage. This is incompatible with the input requirements of typical logic elements. The metastability filter takes as inputs signals with these shifted Brockett-annuli, and outputs signals that satisfy the requirements of typical logic circuits.

III. Related Work

Metastable behavior in digital circuits has been studied since Chaney and Molnar's original paper on synchronizer failures [3]. Hurtado [4] analyzed metastability from a dynamical systems perspective. Seitz [5] gives a nice introduction to metastability issues, and Marino [6] provides a fairly comprehensive treatment.

Arbiters have also been studied from a formal verification perspective. Kurshan and McMillan [7] use the arbiter from [5] as their main example in proposing a way to verify digital circuits modeled by differential equations. Their arbiter is the nMOS counterpart of the CMOS design presented in Section II-C. They formulated the verification problem in terms of language containment. To model the continuous behavior of the circuit, they divided the possible values for each continuous state variable into 10 to 20 intervals, and computed the set of reachable hyper-rectangles using such a grid. This is similar to the d/dt tool [8], [9]. Although the total number of possible hyper-rectangles is large, Kurshan and McMillan used COSPAN to construct the reachable space, and the next hyper-rectangle relation is only computed for reachable hyper-rectangles. Unlike our Brockett annulus approach for specifying signal transitions, Kurshan and McMillan model the inputs as making instantaneous transitions. These transitions were allowed at arbitrary times that satisfied the handshake protocol.

Mendler and Stroup [1] gave a formal specification for a continuous system to implement an arbiter. They used a dynamical systems theory argument to show that the specification is unsatisfiable by any continuous system. Mitchell

and Greenstreet [10] used measure-theory based arguments to show that Mendler and Stroup's specification is satisfiable if the liveness requirement is relaxed to an "almost-surely" version when concurrent requests are made.

Martin [11] gave a delay insensitive construction for implementing a fair arbiter from unfair ones. The "delay insensitivity" means that it functions correctly for any bounded delays in its components or wires. He uses the circuit from Section II-C as his "unfair" arbiter. We show in Section V-C that this arbiter circuit is, in fact, fair. Thus, the extra hardware and delay of Martin's construction is unnecessary if this circuit is used as an arbiter.

Formal methods for verifying analog and mixed-signal designs have received intense attention in the past five years. We summarize some of this work here noting that a comprehensive survey is in [12]. Frehse implemented *PHAVer* [13] which provides more efficient and robust implementations of the HyTech algorithms [14]. *Checkmate* [15] computes convex polyhedral approximations of the reachable regions for systems with non-linear dynamics by using numerical optimization methods to find extremal trajectories. The *d/dt* tool [8] performs reachability analysis of continuous or hybrid systems modeled by linear differential inclusions of the form of $dx/dt = Ax + Bu$, where u is an external input taking values in a bounded convex polyhedron. *d/dt* represents the reachable sets as non-convex orthogonal polyhedra [16], i.e. finite unions of full-dimensional, fixed size hyper-rectangles, and approximates the reachable state using numerical integration and polyhedral approximation. Finally, Al-Sammane *et al.* have applied symbolic rewriting and bounded-model checking techniques to several analog verification problems [17]. Each of these tools have been used to verify several analog circuits with low-dimensional state spaces including a tunnel diode oscillators, an VCOs and Sigma-Delta modulators [9], [18]–[20].

IV. Coho

We used the Coho tool to verify the arbiter circuit presented in Section II-C. This section first briefly describes how Coho computes over approximations of the reachable space for circuits modeled by ODEs. It then describes the improvements that we made to Coho in order to verify the arbiter. These changes were required to reduce approximation errors for the highly non-convex reachable regions that arise in the operation of the arbiter.

A. Coho Overview

Coho uses *projectagons* to represent reachable regions in continuous spaces. A projectagon represents an object by its projection onto two-dimensional subspaces. Conversely, given a set of projection polygons, the projectagon is the object obtained by intersecting the prisms obtained by inverse-projecting each projection polygon back into the full-dimensional space. Most operations on projectagons can be performed by manipulating the individual polygons, avoiding any need to explicitly construct high-dimensional objects. Furthermore, projectagons can represent non-convex objects; this property is essential for the verification of the arbiter.

Coho's reachability algorithm has been described previously [21]–[24]; we summarize it here. Coho computes reachability through a sequence of time steps. In each time step, Coho computes a projectagon that contains all reachable points at the end of the time step. The key to Coho's approach is that extremal trajectories originate from the boundary of the faces of the projectagon, and these faces correspond to edges of the projection polygons. This allows Coho to compute reachability by working on one edge at a time.

Coho first computes the LP for the intersection of the convex hulls of all of the projection polygons and then expands it outward by an amount Δ. Let LP_Δ denote this LP. Then, for each edge, Coho constructs a linear program (LP) whose feasible region contains the face corresponding to the edge. This LP is simply the intersection of LP_Δ with the linear constraints that describe the edge. Coho then computes an LP for the "bloated face" whose feasible region contains a small neighborhood around the face. Let $\mathrm{LP}_{neighborhood}$ denote this LP. The circuit model then produces a *linear differential inclusion* that is valid in the feasible region of $\mathrm{LP}_{neighborhood}$; these inclusions have the form:

$$Ax + b - u \quad \leq \quad \dot{x} \quad \leq \quad Ax + b + u \quad . \tag{1}$$

Where A is a matrix; b is a vector; $Ax + b$ is a linear approximation of the circuit model for the neighborhood of the face; and u is vector that upper bounds the approximation error. Coho finds the maximum magnitude of the derivative, $\dot{x}$, in the feasible region of $\mathrm{LP}_{neighborhood}$. Let D denote this magnitude. Then Δ/D is an upper bound on the valid size for a time step using this bloat. Coho uses the differential inclusion computed above and the LP for the face to construct an LP that contains all points reachable from the face at the end of the time step. This reachable space for this LP is then projected back to the subspace for the original projection polygon. The union of these projections for each edge of the original polygon gives the boundary of the polygon at the end of the time step.

Coho's method of turning an edge into an LP, advancing the LP by the differential inclusion, and projecting it back down tends to produce a dramatic increase in the number of projectagon edges at each time step. Thus, Coho performs polygon simplification at the end of each time step. This simplification step produces a polygon with a smaller number of vertices that contains the polygon obtained from the projection operations.

B. Improvements to Polygon Simplification

Coho has two methods for simplifying polygons: it can delete a vertex at a concave corner, and it can replace a pair of vertices at consecutive convex corners with a single vertex. Both transformations produce a polygon that strictly contains the original. In earlier versions of Coho, the cost of these operations were measured in terms of the increase in the area of the polygon and the increase in the area of its convex hull. This metric took into account that over approximating the convex hull can introduce over approximations in the reachability calculations for all edges, and not just for part

of a particular polygon. Thus, Coho preferentially eliminated concavities in the projection polygons.

When verifying the arbiter, we observed that reachability computations often started with convex regions that should evolve into concave ones due to the dynamics of the ODE model. However, these concavities tend to be initially very small, and Coho encounters a very small area cost to fill them in. Unfortunately, this prevented the projectagons from evolving highly non-convex shapes and prevented verification of the arbiter. We also noted that reducing the number of polygon vertices in the convex hull is more important for performance than eliminating concave vertices. The reason for this is that Coho spends a large percentage of its time solving LPs. Simplifying the convex hulls reduces the number of constraints in these LPs and allows them to be solved more rapidly.

Our solution was to simplify the polygon and its convex hull separately. The earlier version of Coho simplified each polygon and then computed their convex hulls. Now, Coho computes the hulls before simplification. This allows the polygons and the hulls to be simplified using objective criteria that are appropriate for each case. With this change, we obtained a small speed-up in Coho and, more importantly, projectagons can now evolve into concave shapes.

C. Interval Closure

As described above, earlier version of Coho derived an LP for each projectagon face by intersecting the constraints for the face's edge with the bloated convex hulls for each of the other projection polygons. If these polygons are non-convex, then this can produce a large over approximation of the face. Our solution is to perform an interval closure calculation. We can view each projection polygon edge as defining interval bounds for the two variables of the projection. The closure algorithm then applies these intervals to other polygons that include one of these variables in their basis to obtain bounds for other variables. This process continues until no further tightening of the interval bounds is possible. The algorithm is simple, fast and greatly reduces approximation error when the projection polygons are non-convex. However, some care must be taken to preserve the soundness of the reachability algorithm.

The problem is that although all extremal trajectories originate from faces of the projectagon, an extremal trajectory for one projection may emanate from a face whose edge only appears in another projection. Figure 5 shows an example. When edge AB of the (x,y) projection polygon is moved forward, the edge itself gives intervals of $[x_1,x_2]$ and $[y_1,y_2]$ for x and y respectively. The (x,z) projection then yields the interval $[z_1,z_2]$ for z. Now, consider what happens if point A (equivalently P) moves to the left by distance Δ in the current time step and point Q moves to the right by Δ. If interval closure were applied naïvely, then at the end of the time step, trajectories from point Q could reach points with larger x values that any point admitted by the end-of-timestep (x,y) projection polygon.

Figure 6 shows pseudocode for our interval closure calculation. The key idea is to bloat the bounding rectangle

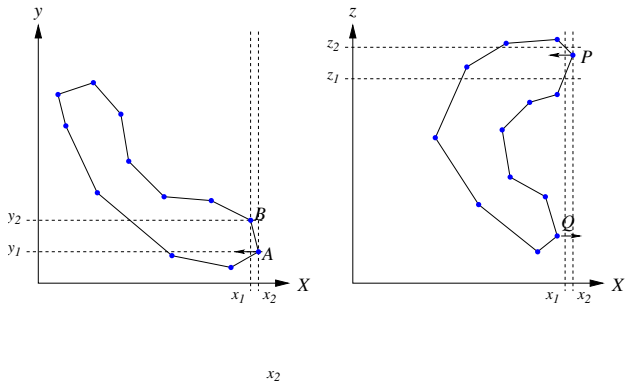


Fig. 5. Interval closure example.

```
HyperRectangle IntervalClosure(Edge e, ProjectionPolygon p, Real Δ) {
    /* e is an edge of polygon p.
        Δ is the bloat amount for the current time step.
        Return the hyper-rectangle of interval closure bounds
            for e for the current time step.
    */
    Let r be the oriented rectangle that contains all points within
    distance 2Δ of e by the ℓ∞ metric.
    Let q be the intersection or r and p.
    Let b be the bounding box of q.
    Let h0 be the hyper-rectangle obtained by interval closure
        starting with b and using all of the other projection polygons.
    Let h = bloat(h0, 2Δ). return(h).
}
```

Fig. 6. Computing interval closure for edges.

for each edge by Δ, intersect the resulting rectangle with the edge's projection polygon, and use the bounding box for this intersection as the starting point for the interval closure calculation. Coho uses the intersection of $\text{LP}_{neighborhood}$ and h to describe region where the differential inclusion for the current edge must hold. Likewise, Coho uses the intersection of h and the current edge to obtain an LP that contains all points on the current face.

To establish the soundness of the interval closure method from Figure 6, we consider a projectagon, and assume that one of the projections has the basis (x,y). We show that all points reachable from the projectagon by the end of the timestep are contained in the (x,y) projection polygon at the end of the timestep. Let p be an arbitrary point of the projection polygon at the beginning of the timestep. If p is further than 2Δ from the boundary of the (x,y) projection polygon at the beginning of the timestep, then any points reachable from p at the end of the timestep will be inside the time advanced polygon, because trajectories from p can move at most Δ units outward and points on the boundary of projection polygon can move at most p units inward during the timestep.

Otherwise, let e be an edge of the (x,y) projection polygon that is within distance 2Δ of p, and let h be the hyper-rectangle computed by the interval closure algorithm for e. By construction, h contains p. Accordingly, the LPs that Coho constructs for the faces of the projection polygon have feasible regions that extend by 2Δ in all of the other dimensions beyond the nearby (i.e. within 2Δ) points of projectagon.

These extensions create a "parapet" to ensure that trajectories from faces for other projection polygons can't "escape" this polygon. In particular, the face may shrink by at most Δ along any dimension, and any point can only reach other points that are within distance Δ (by the ℓ_∞) metric of itself. Thus, to reach a point outside of the (x,y) polygon, a trajectory from p would have to touch the feasible region for one of the faces arising from the (x,y) polygon. This means that points reachable from p are also reachable from the face that it touched, and therefore project to points on the (x,y) plane that are inside the time advanced projection polygon.

V. Verifying the Arbiter

This section describes how we modeled the clients of the arbiter and then describes the verification of the arbiter circuit that was described in Section II.

A. Modeling Concurrent Input Transitions

Rising transitions of the request signals for the two clients can occur concurrently. These requests can start at different times and have different rise-times. For example, r_1 can start a slow rising at some time, and r_2 can start a fast rising signal at slightly later and overtake r_1. In these scenarios, the arbiter may issue a grant to either client, and the grant may be delayed due to metastability in the arbiter. Verifying correct operation of the arbiter requires accounting for all allowed transitions of the inputs, including overlapping ones.

We sub-divided the Brocket annulus for each request signal into sixteen disjoint regions: one region for logical low values (region 1 from Section II-B, henceforth referred to as B_1); one for a logical high values (region 3, henceforth B_3); seven for rising transitions (dividing region 2 into $B_{2,1}$ through $B_{2,7}$); and seven for falling signals (henceforth $B_{4,1}\ldots B_{4,7}$). We modeled the concurrent behaviors of the two request signals using cross-product of these partitions. Applied directly, this would require Coho to solve 256 reachability problems. We reduced this to 136 problems by exploiting the symmetry of the arbiter and its clients and further down to 108 regions by noting that there must be a failure of the arbiter or its clients if both requests are falling at the same time – this would imply that either the arbiter had violated the mutual-exclusion requirement or that at least one client had violated the handshake protocol.

We divided the verification into three phases: the first phase starts with both requests in region B_1 and ends with at least one in B_3. The second phase starts with at least one request in region B_3 and ends with that request back in B_1. The final phase starts with one request having just entered region B_1 and the other in region B_3, and ends when the second request returns to region B_1. We use an assume-guarantee approach for verification [25]: for each phase, we posit bounding boxes for entering trajectories and verify bounding boxes for exiting trajectories. We show that the boxes bounding boxes for exiting trajectories for each phase are contained in the corresponding bounding boxes boxes for entering trajectories for the other phases. This establishes an invariant set that we use to verify the arbiter.

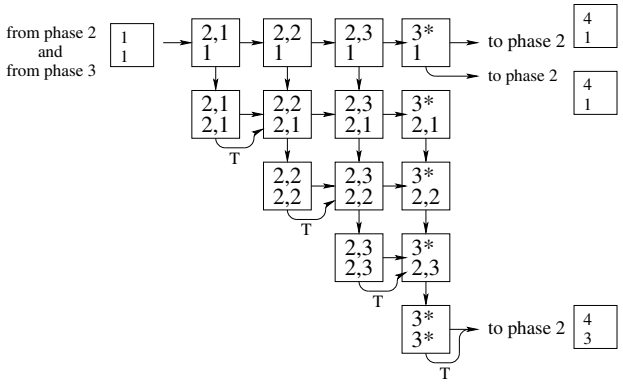


Fig. 7. First Verification Phase.

Figure 7 shows the first phase of this verification process; the other two phases are similar. Boxes are labeled with the regions for the r_1 (the upper label) and r_2, the lower label. A directed edge between boxes indicates that trajectories can leave that source box and enter the destination. We exploit the symmetry of the arbiter and only consider states where r_1 is further along in its handshake cycle than r_2. We take the reachable regions computed that would flow out the bottom of squares along the main diagonal, swap their (r_1,x_1,g_1) and (r_2,x_2,g_2) components, and include the result in the input to the cell to the right; these are the arrows labeled with T to indicate this "transposition" of the state. Finally, states labeled with $1*$ (resp. $3*$), indicate that the signal is low (resp. high) and has not completed the minimum high (resp. low) time requirement. For simplicity, we assume that these minimum dwell times are greater than the maximum rise and fall times. Thus, the boxes where r_1 enters B_3 while r_2 is in $B_{2,1}$ has an outgoing edge for r_2 entering $B_{2,2}$ but no outgoing edge for r_1 entering $B_4 - r_2$ is guaranteed to complete its rising while r_1 remains high.

We used Coho to compute reachable sets for trajectories exiting each box based on reachable sets for trajectories entering the box. For each phase, Coho computes these sets in topological order, using the "assume" bounding boxes for trajectories entering the phase, and verifying that trajectories that exit the phase satisfy the "guarantee" bounding boxes. This establishes an invariant set of trajectories that we use to verify safety and liveness properties of the arbiter in Section V-C.

B. Modeling the Circuit

For the most part, modeling the circuit is straightforward. Given the LP for a face, the modeling code computes a linear inclusion for the drain-source current of each transistors. For each node, summing the inclusions for the currents flowing into the node via transistors gives the total current to charge the node capacitance. Dividing this current by the node capacitance yields a linear differential inclusion for the node voltage.

The problem that we encountered is associated with nodes u_1 and u_2. These nodes have much smaller capacitances than the other nodes in the circuit; in fact, many designers would

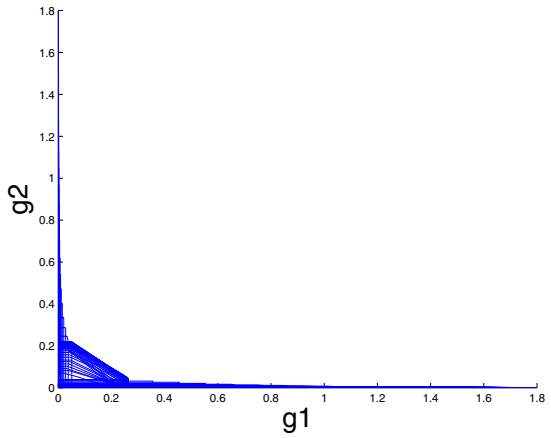


Fig. 8. Mutual Exclusion

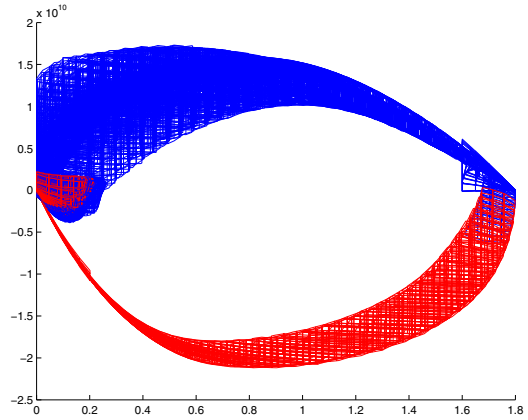


Fig. 9. Brockett Annulus for g

instinctively ignore the contributions of these tiny capacitors. Unfortunately for our verification work, these outlier node capacitances produce a stiff system of differential equations which exacerbate Coho's over approximation of the reachable space.

Coho has two principle causes of over approximation. First, there is an over approximation when producing a linear differential inclusion for a non-linear ODE. Second, over approximations are introduced when projecting the reachable region for a face back down to the basis for the projection polygon. For the arbiter, large time steps result in large linearization over approximation for nodes u_1 and u_2. Small time steps, on the other hand, lead to large over approximations for the other nodes due to the large number of projection operations. Either way, we were unable to verify the arbiter. Our solution was to follow the example of typical designers and treat nodes u_1 and u_2 as if they had no capacitance. We did this by creating a model for a nMOS tetrode with source connected to ground, gates connected to r_1 and x_2, and drain connected to u_1, and another such tetrode for the two pull-down transistors for u_2.

C. Results

Using the methods described above, Coho computed an invariant region for the arbiter. This set allows us to establish the correct operation of the arbiter as described below. Our circuit parameters correspond to the TSMC 1.8V, 180nm bulk CMOS process. Our Brockett annuli for r_1 and r_2 have inner and outer boundaries that are ellipses. Logically low signals are constrained to lie in $[0.0V, 0.2V]$ and logically high signals must be in $[1.6V, 1.8V]$. At the midpoint of rising and falling transitions ($0.9V$), the time derivative must be in $[1.5,2]*10^{10}$ volts/sec for rising transitions and $-[1.5,2]*10^{10}$ volts/sec for falling ones.

Safety Properties: Mutual Exclusion:
Figure 8 shows the reachable space projected onto the signals g_1 and g_2. Everywhere in the reachable space, either g_1 or g_2 is in $[0,0.2]$ region 1 of their Brockett annuli. Thus, at least one is false in the discrete abstraction, and the arbiter satisfies the mutual exclusion requirement.

Handshake Protocol:
In a similar fashion, projecting the reachable space is projected onto the signals g_1 and r_1 shows that g_1 enters B_2 only when r_1 is in B_3. Likewise g_1 enters B_4 when $r_1 \in [0.0, 0.22]$. When $r_1 \in [0.2, 1.6]$, $\dot{r_1} < 0$, thus $r_1 \in B_4$. This shows that g_1 starts to fall only when the discrete abstraction of r_1 is a logically low signal. Thus, the grants both rise and fall in accordance with the handshake protocol.

Brockett Annuli:
Figure 9 shows the Brockett annulus for g_1; it is slightly larger than the one that we used for the requests. We plan to re-run the verification using this slightly larger ring for the requests to show that the outputs of the arbiter satisfy the constraints that we imposed on the inputs, and thus show that this circuit could be part of a library of circuits sharing a common specification for signal transitions. The Brockett annulus for the grants shows an artifact of the over approximations used by Coho. Note that the left edge of the annulus is vertical. This is because over approximations that arise when linearizing the ODE model produce a differential inclusion that allows the grant signals to go to negative voltages. However, the model has an invariant that all signals must have values in $[0, V_{dd}]$ (we model all capacitances as being to ground). Coho clips the projectagon at the end of each time step to satisfy this invariant. This clipping produces the vertical left edge seen in the figure.

As shown in Figure 10, signal x_1 only satisfies a much less restrictive Brockett annulus than the ones we used for the request or established for the grants. The bulge in the lower right part of the annulus is a consequence of metastability in the arbiter: x_1 may drop to a value near the metastable voltage; remain their for an arbitrarily long time; and then either return to a high value (if client 1 loses the arbitration), or resolve to a low value (if client 1 wins). Comparing the Brockett annuli for x_1 and g_1, we clearly see the role of the metastability filter as a Brockett annulus transformer.

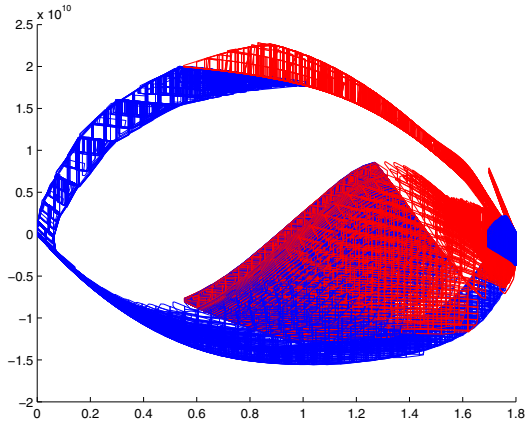


Fig. 10. Brockett Annulus for x

Liveness Properties: Initialization:
We used Coho to compute the reachable space when r_1 and r_2 were both low (i.e. in region B_1) starting from a state where x_1, x_2, g_1 and g_2 could be anywhere in $[0, 1.8]$. Coho establishes that within 200ps, x_1 and x_2 enter $[1.6V, 1.8V]$ (i.e. B_3), and g_1 and g_2 enter $[0.0V, 0.2V]$ (i.e. B_2). Thus, the arbiter can be initialized simply by not asserting any requests for a short time – no additional reset signal is needed.
Uncontested Requests:
We consider the reachable space with the additional restriction that r_2 remains within region B_1 (i.e. a logically low value). Coho shows that g_1 is asserted within 343 ps of r_1 rising (i.e. entering B_2). This shows that the arbiter is guaranteed to respond to contested requests within a bounded amount of time.
Contested Requests:.
If r_1 and r_2 are asserted at nearly the same time, the arbiter may exhibit metastable behavior and the arbiter may remain in a metastable state for an arbitrarily long period of time. Coho can show that metastability can only occur in the hyper-rectangle where:

$$\begin{array}{lll} r_1 \in B_3 & x_1 \in [0.55, 1.3] & g_1 \in B_1 \\ r_2 \in B_3 & x_2 \in [0.55, 1.3] & g_2 \in B_1 \end{array} \quad (2)$$

Outside of this region, the dot product of the derivative vector with the final stable state of granting client 1 or granting client 2 is unambiguous.
Reset:
Coho shows that if client i has a grant and lowers its request signal then the arbiter lowers the grant for client i within 270 ps. This shows that the arbiter satisfies the liveness requirement for withdrawing grants.
Fairness:
If client i wins a grant while the other client is making a request and subsequently client i drops its request, then Coho shows that the other client receives a grant within 420 ps. This shows that the other client receives at most one grant while the current client has a pending request; therefore, this simple arbiter is fair.

Other Observations:
As noted in Section V-A, we divided the rising and falling transitions for the arbiter inputs into seven regions each. After verifying the arbiter, we tried again using 1, 2, 3, 5, 9, 11, and 14 regions. For one region (i.e. no partitioning of B_2 or B_4, the approximation errors are significantly larger). For two or more regions, the approximation error decreases slightly with an increasing number of partitions.

VI. Conclusions

We have presented the verification of an asynchronous arbiter circuit. Our specification is a discrete transition diagram with additional liveness requirements. The circuit is modeled as a system of non-linear differential equations using foundry provided SPICE models for the transistors. The abstraction mapping from the continuous trajectories of the circuit model to the discrete sequences of the specification use Brockett's annulus construction. Our model allows the inputs to transition at arbitrary times. Unlike [7], we also allow the input transitions to have arbitrary waveforms that satisfy the rise and fall time and monotonicity requirements of the Brockett annulus.

We presented improvements to the Coho tool that made this verification possible. In particular, we presented an interval closure algorithm for obtaining tighter bounds for projectagon faces, and we sketched a proof that using these bounds preserves the soundness of Coho. We described how we modified the polygon simplification step to allow concavities to evolve in the projectagons. Verification of the arbiter requires representing highly non-convex regions (e.g., see Figure 8). The changes that we made to Coho enabled the successful verification of the arbiter, and we believe that similar non-convexities will be important when verifying other circuits.

The safety properties that we verified establish that the arbiter observes the handshake protocol with its clients and maintains mutual exclusion. For liveness, we showed that the arbiter can be initialized simply by leaving both requests low for at least 186 ps; an uncontested request is granted within 343 ps; a grant is lowered within 270 ps of lowering the request; and that a waiting request is granted with 420 ps of the other client dropping its request. The last of these establishes that the arbiter is strongly fair: no client will receive two or more grants while the other client is waiting.

This verification points to further work for verifying that circuits with continuous models satisfy discrete specifications. Here we mention possible areas for future research, both for the discrete and continuous aspects of the verification problem. Our discrete abstraction uses Brockett's annulus construction, and we verified safety properties by manually checking that the reachable trajectories were contained in the specified annulus. This was a relatively easy and straightforward task that could be readily automated. More generally, we would like to develop a tool to automatically translate "data sheet" style specifications to the corresponding topological properties to check by reachability analysis. For circuits more complicated than the arbiter, the discrete behavior could be specified using standard methods such as state-machine descriptions, assume-guarantee techniques or guarded command programs. For the

arbiter example in this paper, we used the Brockett annulus directly for the visual intuition that it provides, and because we believe that the space to introduce an additional notation for such a simple problem would only serve to distract from the key issues in circuit-level verification.

The arbiter provides an important generalization over our prior verification of a toggle flip-flop [23] in that the arbiter has two-inputs, and the relationship between these inputs and the arbiter outputs must be specified to capture the hand-shake protocol. We extended the Brockett annulus abstraction to capture the allowed behaviors of these inputs. We plan to apply similar techniques to handle synchronous circuits, adding the extra timing constraints that input signal transitions must satisfy timing requirements such as set-up and hold times. We believe that our specification framework can be readily adapted to synchronous design, and that the verification tasks may actually be simpler. Extending our techniques to a high-performance, synchronous design family such as domino circuits [26] would make these techniques of use to a large range of designers.

We would like to be able to show an "almost-surely" result for liveness when the two clients make concurrent requests. The proof in [10] was based on arguments about the eigenvectors of the Jacobian matrix for the time derivative function. The A matrix in our differential inclusions (see Equation 1) is roughly the same as this Jacobian matrix. However, the u term in the differential inclusion prevents making a direct correspondence. Verifying the almost-sure liveness is a topic for future work.

The other key area that we see for further research is developing methods to cope with stiff ODE models. As described in Section V-B, the small capacitance on the internal nodes of the NAND gates have associated time constants that are much smaller than the other nodes in the circuit. This is a classic example of a stiff system of ODEs. For Coho, this stiff system makes it impossible to choose a time step that produces an acceptably small over approximation of the reachable space. We are currently exploring methods for handling stiffness and expect that they will be useful for other circuits and for verification problems for hybrid systems.

REFERENCES

[1] M. Mendler and T. Stroup, "Newtonian arbiters cannot be proven correct," in *Proceedings of the 1992 Workshop on Designing Correct Circuits*, Jan. 1992.

[2] R. Brockett, "Smooth dynamical systems which realize arithmetical and logical operations," in *Three Decades of Mathematical Systems Theory: A Collection of Surveys at the Occasion of the 50th Birthday of J. C. Willems*, ser. Lecture Notes in Control and Information Sciences, H. Nijmeijer and J. M. Schumacher, Eds. Springer, 1989, vol. 135, pp. 19–30.

[3] T. Chaney and C. Molnar, "Anomalous behavior of synchronizer and arbiter circuits," *IEEE Transactions on Computers*, vol. C-22, no. 4, pp. 421–422, Apr. 1973.

[4] M. Hurtado, "Structure and performance of asymptotically bistable dynamical systems," Ph.D. dissertation, Sever Institute, Washington University, Saint Louis, MO, 1975.

[5] C. L. Seitz, "System timing," in *Introduction to VLSI Systems* (Carver Mead and Lynn Conway). Addison Wesley, 1979, ch. 7, pp. 218–262.

[6] L. Marino, "General theory of metastable operation," *IEEE Transactions on Computers*, vol. C-30, no. 2, pp. 107–115, Feb. 1981.

[7] R. Kurshan and K. McMillan, "Analysis of digital circuits through symbolic reduction," *IEEE Transactions on Computer-Aided Design*, vol. 10, no. 11, pp. 1356–1371, Nov. 1991.

[8] E. Asarin, T. Dang, and O. Maler, "The d/dt tool for verification of hybrid systems," in *Proceedings of the* 14^{th} *Conference on Computer Aided Verification*. Copenhagen: Springer, July 2002, pp. 365–370.

[9] T. Dang, A. Donzé, and O. Maler, "Verification of analog and mixed-signal circuits using hybrid system techniques," in *Proceedings of the* 5^{th} *International Conference on Formal Methods in Computer Aided Design*, Nov. 2004, pp. 21–36.

[10] I. Mitchell and M. Greenstreet, "Proving Newtonian arbiters correct, almost surely," in *Proceedings of the Third Workshop on Designing Correct Circuits*, Båstad, Sweden, Sept. 1996.

[11] A. J. Martin, "A delay insensitive fair arbiter," Computer Science Department, California Institute of Technology, Tech. Rep. 5193-TR-85, 1985.

[12] M. H. Zaki, S. Tahar, , and G. Bois, "Formal verification of analog and mixed signal designs: A survey," *Microelectronics Journal*, 2008.

[13] G. Frehse, "PHAVer: Algorithmic verification of hybrid systems past HyTech," in *Proceedings of the Fifth International Workshop on Hybrid Systems: Computation and Control*. Springer-Verlag, 2005, pp. 258–273, LNCS 3414.

[14] R. Alur, T. Henzinger, and P.-H. Ho, "Automatic symbolic verification of embedded systems," *IEEE Transactions on Software Engineering*, vol. 22, no. 3, pp. 181–201, 1996.

[15] B. Silva, K. Richeson, *et al.*, "Modeling and verifying hybrid dynamical systems using *checkmate*," in *Proceedings of the* 4^{th} *International Conference on Automation of Mixed Processes (ADPM 2000)*, 2000, pp. 323–328.

[16] O. Bournez, O. Maler, and A. Pneuli, "Orthogonal polyhedra: Representation and computation," in *Proceedings of the Second International Workshop on Hybrid Systems: Computation and Control*. Springer, 1999, pp. 46–60, LNCS 1569.

[17] G. Al-Sammane, M. H. Zaki, and S. Tahar, "A symbolic methodology for the verification of analog and mixed-signal designs," in *Proceedings of the 10th Design Automation and Test Europe Conference (DATE'2007)*, Apr. 2007, pp. 249–254.

[18] S. Gupta, B. H. Krogh, and R. A. Rutenbar, "Towards formal verification of analog designs," in *Proceedings of 2004 IEEE/ACM International Conference on Computer Aided Design*, Nov. 2004, pp. 210–217.

[19] G. Frehse, B. H. Krogh, and R. A. Rutenbar, "Verifying analog oscillator circuits using forward/backward abstraction refinement," in *Proceedings of Design Automation and Test Europe*, Mar. 2006, pp. 257–262.

[20] M. H. Zaki, G. Al-Sammane, S. Tahar, and G. Bois, "Combining symbolic simulation and interval arithmetic for the verication of AMS designs," in *Proceedings of the* 7^{th} *Conference on Formal Methods in Computer Aided Design (FMCAD'2007)*, Nov. 2007, pp. 207–215.

[21] M. R. Greenstreet and I. Mitchell, "Integrating projections," in *Proceedings of the First International Workshop on Hybrid Systems: Computation and Control*, T. A. Henzinger and S. Sastry, Eds., Berkeley, California, Apr. 1998, pp. 159–174.

[22] ——, "Reachability analysis using polygonal projections," in *Proceedings of the Second International Workshop on Hybrid Systems: Computation and Control*. Berg en Dal, The Netherlands: Springer, Mar. 1999, pp. 103–116, LNCS 1569.

[23] C. Yan and M. R. Greenstreet, "Circuit level verification of a high-speed toggle," in *Proceedings of the* 7^{th} *Conference on Formal Methods in Computer Aided Design (FMCAD'2007)*, Nov. 2007.

[24] ——, "Faster projection based methods for circuit-level verification," in *Proceedings of the 2008 Asia and South Pacific design automation conference (ASPDAC'08)*, Jan. 2008, pp. 410–415.

[25] T. A. Henzinger, S. Qadeer, and S. K. Rajamani, "You assume, we guarantee: Methodology and case studies," in *Proceedings of the* 10^{th} *International Conference on Computer Aided Verification*, 1998, pp. 440–451, LNCS 1427.

[26] R. Krambeck, C. Lee, and H. Law, "High-speed compact circuits with CMOS," *IEEE Journal of Solid-State Circuits*, vol. SC-17, pp. 614–619, June 1982.

Formal Verification of Hardware Support For Advanced Encryption Standard

Anna Slobodová
Intel Corporation

***Abstract*—The Advanced Encryption Standard (AES), approved by National Institute of Standards and Technology, specifies a cryptographic algorithm that can be used to protect electronic data. The next generation of Intel micro-processor introduces a set of instructions known as AES-NI, that promises multi-folded acceleration of the AES encryption and decryption process. In this paper, we report about the formal verification of hardware support for these new instructions. The verification is based on use of Symbolic Trajectory Evaluation that lies at the base of formal verification methodology used by Intel Corporation. To our knowledge, this is the first formal verification of AES hardware support.**

I. Introduction

In the competitive world of computer business, each generation of micro-processors is expected to come up with something new – more performance, lower power, higher degrees of parallelism and, last but not least, higher degrees of security. The new features of the next generation of Intel's product include hardware support for encryption and decryption algorithms. The Federal Information Processing Standards Publication 197 [7] specifies the Advanced Encryption Standard (AES) that is based on the Rijndale algorithm [5], [6] – a block cipher encryption and decryption algorithm that uses Galois Field arithmetic. The latest Intel architecture introduces six Intel$^{(R)}$ SSE instructions known as AES-NI that promise threefold-ed speed-up of the AES algorithms. Their description can be found on Intel's official web site [2], [3]. Four of the instructions – AESENC, AESENCLAST, AESDEC, and AESDECLAST correspond to one round of encryption/decryption algorithm. The other two facilitate the key expansion procedure: AESKEYGENASSIST is used for generating the round keys for encryption; AESIMC is used for converting the encryption round keys to a form usable for decryption.

Each of these instructions is implemented using AES micro-operations – primitives understandable by hardware. Our focus is on the verification of these micro-operations whose complexity matches the complexity of the AES-NI instructions.

For the verification, we use an internal formal verification system Forte [1]. Our specification of micro-operations is written in Reflect [8] – a ML-like functional language that is a successor of FL. Reflect includes Binary Decision Diagrams as first-class objects and Symbolic Trajectory Evaluation (STE) [16] as built-in functions. For details on Intel's verification methodology using Forte, we refer the reader to the published papers [10], [11], [12], [17].

For each micro-operation, a proof constitutes a run of Symbolic Trajectory Evaluation on a formal model derived from the Register Transfer Level model of Execution Cluster (EXEC) – a part of design responsible for the execution of micro-operations. All external assumptions – the assumptions made about hardware outside EXEC – are checked by traditional simulation on the full-chip model.

To our knowledge, this is the first verification of HW support for AES. There is a previous work describing verification of Galois Field Algorithms. Morioka *et.al.* proposed a logic system for the verification of algorithms over Galois fields $GF(2^m)$ and carried out a proof for One Shot Reed-Solomon Decoding Algorithm [14]. Mukhopadhyay *et. al.* suggested [15] an improvement to the theorem-based approach in [14] by using an observation that the verification of a property in $GF(2^m)$, where $m = np$, can be reduced to the verification of the property in a composite field $GF(2^n)^p$. This observation allowed the authors to take advantage of a hierarchical approach to the verification problem. For the transformation of elements between the fields, they used Reduced Ordered Functional Decision Diagrams (ROFDDs) [9]. They demonstrated their approach on the same Reed-Solomon decryption algorithm. Recently, a group in Galois Connections developed a domain-specific language Cryptol and a tool set for formally specifying, implementing and verifying cryptographic algorithms. It has been applied to the verification of some aspects of the AES algorithm. Their verification methodology uses And-inverter graphs and combinational equivalence checking and requires both specification and implementation be written in Cryptol.

Our work differs from the papers mentioned above in several aspects. Our focus is not on the AES algorithms but on the verification of hardware that implements a basic step of the algorithm, which is equivalent to one round of the AES encryption/decryption. An important aspect of our work is that the hardware has been designed with high performance rather than verification in mind. The main message of this paper is that the verification of the AES implementation can be performed using general purpose BDD-based symbolic simulation engine instead of a special-purpose diagrams ROFDDs tailored for XOR-AND-based functions.

In the next section, we explain all terms needed to understand the specification of AES instructions. Section III describes verification methods.

[1] A publicly available version of the tool that can be used for non-commercial purposes can be downloaded from *http://www.intel.com/software/products/opensource/*

978-1-4244-2735-2/08/$25.00 ©2008 IEEE

II. AES And Its HW Support in Intel(R) New Generation Microprocessors

Our specifications of micro-operations were written based on the Micro-Architecture Specification for the Intel(R) new generation microprocessor – Westmere; while the specification of each particular transformation is defined as in FIPS 197 document [7] to avoid replication of possible errors made by designers. It is comprised of about five hundred lines of Reflect code. Although the MAS is confidential, the description of AES instructions can be found on Intel public web-pages [2], [3].

The FIPS 197 document [7] specifies a symmetric block cipher that can process data blocks of 128 bits, using cipher keys with lengths of 128, 192, and 256 bits. The AES instructions operate on one or on two 128 bits sources: State and Round Key. From the architectural point of view, the state is input in an xmm register (xmm1) and the Round key is input either in an xmm register (xmm2) or in memory (m128).

Because of lack of space we will describe just one representative AES instruction – AESENC that performs one round of AES encryption. This will give the reader a flavor of the complexity of the design and the specification. We will start with the definition of the instruction, followed by basic terms and operations that the instruction is composed of. Note, that because of the symmetric character of the AES encryption and decryption algorithm, for each transformation, there is an inverse transformation that is used for decryption. Here we introduce only transformations that are needed for the definition of the chosen instruction.

Instruction AESENC xmm1, xmm2/m128, performs one round of AES encryption applied to the content of a xmm register xmm1 using a 128-bit key from the register xmm2 or memory m128 (we will show the version for xmm2).

```
AESENC xmm1, xmm2
Tmp:= xmm1;
RoundKey:= xmm2;
Tmp:= ShiftRows (Tmp);
Tmp:= SubState (Tmp);
Tmp:= MixColumns (Tmp);
xmm1:= Tmp xor_bitwise RoundKey
```

Any sequence (of bits, bytes or words) is represented as a list. **Byte** is a sequence of 8 bits with lsb indexed by 0. **Word** is a sequence of 4 bytes $[w_0, w_1, w_2, w_3]$. Note that this differs from the usual notion of a computer word as a sequence of two bytes. **State** is a sequence of words in Big Endian notation. State might be viewed as a matrix, so it makes sense to talk about **columns** (that match words) and **rows**. If not stated otherwise, in $s = [s_0, s_1, s_2, s_3]$, si denote columns.

Example: A state in hexadecimal notation and viewed as a matrix:
$[[h00, h01, h02, h03], [h04, h05, h06, h07],$
$[h08, h09, h0a, h0b], [h0c, h0d, h0e, h0f]]$

word3	word2	word1	word0
h00	h04	h08	h0c
h01	h05	h09	h0d
h02	h06	h0a	h0e
h03	h07	h0b	h0f

Elements in $GF(2^8)$ can be represented as 7-degree polynomials with binary coefficients. **Addition** and **subtraction** over $GF(2^8)$ are equivalent operations and are defined as bitwise XOR, denoted by $\oplus$.

Multiplication in $GF(2^8)$ is multiplication of polynomials modulo an irreducible polynomial $m(x) = x^8+x^4+x^3+x+1$ and it results in an element in $GF(2^8)$. $m(x)$ is represented by a binary vector $\text{AES}_M = [1,0,0,0,1,1,0,1,1]$. In our specification, multiplication over $GF(2^8)$ is implemented as suggested in the FIPS 197 document using an efficient iterative algorithm, where each iteration step consists of a multiplication by polynomial x. **Inverse element** of Galois field is computed using **Extended Euclidean Algorithm** with irreducible polynomial $m(x)$.

Another structure we use is a four-term **polynomial with coefficients from** $GF(2^8)$. $a(x) = a_3x^3 + a_2x^2 + a_1x + a_0$ can be represented by 4 bytes: $[a_0, a_1, a_2, a_3]$ i.e., one word. These polynomials behave differently from the polynomials we mentioned above:

Addition of such polynomials is defined as bit-wise XOR applied to the corresponding bytes:

$$(a_3x^3 + a_2x^2 + a_1x + a_0) + (b_3x^3 + b_2x^2 + b_1x + b_0) = (a_3 \oplus b_3)x^3 + (a_2 \oplus b_2)x^2 + (a_1 \oplus b_1)x + (a_0 \oplus b_0)$$

Or

$$[a_0, a_1, a_2, a_3]+[b_0, b_1, b_2, b_3] = [a_0 \oplus b_0, a_1 \oplus b_1, a_2 \oplus b_2, a_3 \oplus b_3]$$

Multiplication of two polynomials (of degree less than 4) is computed modulo $x^4 + 1$. To distinguish this multiplication from the $*$-multiplication defined above, we denote it by $\otimes$ and call a **modular multiplication**.
$a \otimes b$ can be expressed as a left multiplication of b by matrix

$$M(a) = \begin{pmatrix} a_0 & a_3 & a_2 & a_1 \\ a_1 & a_0 & a_3 & a_2 \\ a_2 & a_1 & a_0 & a_3 \\ a_3 & a_2 & a_1 & a_0 \end{pmatrix}$$

$a \otimes b = M(a) * b^t$, where * is usual matrix multiplication with * applied to the elements of the field, and b^t is a vector transposition of b.

ShiftRows is a transformation that rotates i-th row by i bytes to the left (rows are labeled in top-down manner 0 to 3).

$$\begin{pmatrix} S_0 & S_4 & S_8 & S_c \\ S_1 & S_5 & S_9 & S_d \\ S_2 & S_6 & S_a & S_e \\ S_3 & S_7 & S_b & S_f \end{pmatrix} \longmapsto \begin{pmatrix} S_0 & S_4 & S_8 & S_c \\ S_5 & S_9 & S_d & S_1 \\ S_a & S_e & S_2 & S_6 \\ S_f & S_3 & S_7 & S_b \end{pmatrix}$$

SubByte transformation applied to a byte b consists of finding its multiplicative inverse b^{-1} and applying affine transformation defined by matrix S and vector c. S is generated by left-rotation of vector $[1,1,1,1,0,0,0,1]$:

$$\begin{pmatrix} 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \end{pmatrix}$$

and $c = [0,1,1,0,0,0,1,1]$ is a binary representation of h63. SubByte $b = S * b^{-1} + c$, where b^{-1} is inverse element of b. Note that the bits in the rows of the S matrix published in the FIPS 197 document are in reversed order. This is because our bytes have bits ordered $[b_7, b_6, b_5, b_4, b_3, b_2, b_1, b_0]$.
SubWord is a byte-wise application of SubByte to the word.
SubState is a word-wise application of SubWord to the state.
MixColumn_word $w = q \otimes w = M(q) * w^t$,
where $q = [h02, h01, h01, h03]$.
MixColumns is defined as a word-wise application of MixColumn_word to a state.

Another polynomial used in the algorithm is x^3 represented as $x3 = [h01, h00, h00, h00]$. The corresponding matrix $M(x3)$ is used to implement the cyclic shift of the bytes of the word to the left:
RotWord $w = x3 \otimes w = M(x3) * w^t$
Example:
RotWord $[h03, h02, h01, h00] = [h02, h01, h00, h03]$

III. Formal Verification of AES Support in Intel(R) New Generation Processor Design

Verification of AES was a part of an extensive verification effort undertaken by our team on Execution Cluster of the next generation micro-processor. After several years of experience in verification of arithmetic units described in numerous publications, Intel has in place a solid methodology that is based on Forte system [10], [12], [11], [17]. Even though the methodology advanced substantially in the past couple of years, its base remains the same. Our proofs consist of data-path proofs (one for each particular micro-operation) and control proofs that assure that there is no write-back conflicts between the unit that implements AES and the other units. All of them were completed using Symbolic Trajectory Evaluation [16].

Since we haven't had any previous experience with arithmetic over Galois fields, we were not sure about the complexity of the proof. Because the representation of elements in Galois Field use module-2 sum-of-products, it seems to be natural to use ROFDDs for their representation. However, our STE engine is based on Reduced Ordered Binary Decision Diagrams (BDDs) [4]. The involvement of multiplication in some operations made us uncertain whether the capacity of the tool will be sufficient to carry out the verification. As it turned out, with the right variable ordering, we had no BDD-size blow-up issues and no decomposition of the proof/design was needed, which eliminated the need for using theorem proving.

The data-path proof for each micro-operation was performed in one STE run that starts with sources (after bypass) and control signals, and ends at the write-back signals at the boundary of EXEC. Since all AES instructions have fixed latency, STE is a suitable verification engine. We assumed that sources carry legal values, and control signals have values that correspond to a valid AES micro-operation in the pipe, i.e. micro-operation code and additional control signals (power-up, reset, valid signal, etc.) have specific values. Assumptions that concern the design outside execution cluster are called *external*. Examples of external assumptions are:
– Reset signal is not asserted.
– When a micro-operation is executed, the resepective unit is powered up.
– Scheduling of Micro-operations are non-conflicting.
External assumptions used in our proofs were identical to assumptions used in the verification of other micro-operations inthe EXEC. They were checked dynamically by means of simulation based full-chip validation. The bypass logic has been formally verified, but the verification is out of the scope of this paper.

There were some of the usual challenges in writing the specification from incomplete and imprecise documents. Some confusion also came from the use of different byte and bit ordering formats in the design and in the specification. Since we did not want to duplicate potential errors made by designers, we wrote our definition of GF operations based on the FIPS 197 document [7]. However, the FIPS 197 document is Big Endian oriented, while Intel hardware is Little Endian oriented. That means that the behavior of the hardware requires inputs and outputs to be byte-reflected, in order to match the FIPS specification.
Therefore, in the specification, we translate source values from Little Endian to Big Endian, transformations are applied to these translated values; then results are translated back to Little Endian to be compared to values taken from write-back signals.

Variable Ordering

To nobody's surprise, part of the success of any BDD-based verification lies on a right choice of variable ordering. Let us look closer at the transformation included in our instruction:

An important observation is that all of the transformations just shuffle bits and bytes around without "re-using" them in multiple places. The intutition behind this vague statement is that we will never come into a conflict that one variable should be in two differnt positions in the order. The other observation is that the most operations are defined on bytes or words.

Let us consider the ShiftRows transformation first. It is important to note that it does not change relative position of bytes within words. The first byte of each new word was the first byte of some other word; the second byte of each new word was the second byte of some other word; and so on. If we

consider a variable ordering where words are interleaved, the variable order of bits in original and transformed work remains consistent with this order. When we look at our example, word [S0, S1, S2, S3] is transformed to [S0, S5, Sa, Sf]. Bits of S0 are followed by bits of S1 and S5 interleaved, those are followed by bits of S2 and Sa interleaved, and those are followed by S3 and Sf interleaved. Obviously, such order is a good variable ordering for checking the ShiftRows transformation.

Let us consider SubState transformation. This state transformation consists of byte-wise substitution. The FIPS 197 document published a pre-computed SubByte transformation in a form of a table labeled by possible values of half-bytes. In our specification, we actually choose to generate the table using the operations over GF as described in the specification, which has been compared to the published table. That provided an additional assurance that our specification of the operations is correct. For any variable order with variables bound to inputs on the top, a BDD for S-table consists of a complete tree of depth 8, with 8-node BDDs at each tip. A BDD for one bit of the result is even smaller.

MixColumns is defined as a word-wise application of Word-Column_word. The words neither overlap, nor are combined. What we deal here with are four 8x8 bit multiplications (mod $m(x)$), followed by bit-wise XOR. Since for each multiplication, multiplicand is a constant, there is no BDD growth to expect. The XOR-combination of bytes would call for interleaving bytes in the variable ordering, if we would want to write our check as AND of XOR-ed bits, but we don't need to do it. Each bit of the result can be checked separately. Because of that, XOR of bytes works fine even if the variables are in order of sequential concatonation of the bytes. As a result, the variable order where words are interleaved is a good variable ordering for this transformation as well.

To summarize, we choose variable ordering with variables for control signals (such as reset signal, power-up signal, micro-operation code, etc.) on the top, followed by 128-bit data (xmm1) chopped by words and interleaved, and 128-bit key (xmm2) closes the list. In fact, the order of variables bound to the key does not matter. However, for the instructions that involve transformations applied to the key, we use the order where the key is chopped into words and interleaved.

IV. Conclusion and final remarks

Our formal proofs cover correct execution (as defined in [7] and Westmere Micro-Architecture Specification) of all valid micro-operations issued to the AES unit for all legal inputs. Following assume/guarantee principle, we also proved conjecture of other EXEC units which assume that AES does not interfere with their write-backs. Our conjecture that other units do not interfere with AES write-back has been proven within the respective units as well. With our choice of variable ordering we did not encounter BDD blow-ups in the course of the verification. Time and memory consumption of the proofs were very moderate. The data-path and control proofs for all six micro-operations took about 12 minutes and the peak memory usage was less than 2GB on a 64-bit Intel(R) Xeon (TM) machine). As a collateral of doing FV, the Micro-Architecture Specification has been improved and extended with examples. The proof scripts were added to our rich library of proofs that we routinely apply to IA32/64 micro-processor designs and are expected to be reused on the successor projects. To our knowledge, there have been no published reports on the verification of AES hardware support.

Acknowledgment

I would like to thank Naren Narasimhan for challenging me with the problem and Brent Boswell, designer of the AES unit, for answering all of my annoying questions about the design and its intended functionality.

References

[1] AES page available via http://www.nist.gov/CryptoToolkit
[2] Intel(R) Advanced Vector Extensions Programming Reference. March 2008. 319433-002. http://softwarecommunity.intel.com/isn/downloads/intelavx/Intel-AVX-Programming-Reference-31943302.pdf
[3] S. Gueron: Advanced Encryption Standard (AES) Instruction Set, http://softwarecommunity.intel.com/articles/eng/3788.htm
[4] Bryant, R.E.: Graph-based Algorithms for Boolean Function Manipulation. *IEEE Trans. on Comps.*, C-35, pp. 677-691, 1986.
[5] J. Daemen and V. Rijmen, AES Proposal: Rijndael, AES Algorithm Submission, September 3, 1999, available at [1].
[6] J. Daemen and V. Rijmen, The block cipher Rijndael, Smart Card research and Applications, LNCS 1820 (Springer) pp. 288-296.
[7] Federal Information Processing Standards Publication 197, http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf
[8] Jim Grundy, Thomas F. Melham, John W. O'Leary: A reflective functional language for hardware design and theorem proving. J. Funct. Program. 16(2): 157-196 (2006)
[9] T. Kropf: Introduction to Formal Hardware Verification: Methods and Tools for Designing Correct Circuits and Systems, Springer, 1999.
[10] R. Kaivola, M, Aagard: Divider Circuit Verification with Model Checking and Theorem Proving. *TPHOL 2000, (Springer) LNCS 1869, pp.338-355.*
[11] R. Kaivola, K. Kohatsu: Proof Engineering in the Large: Formal Verification of Pentium(R) 4 Floating-Point Divider. *Software Tools for Technology Transfer, Springer, 2003, Vol.4, Issue 3, pp 323-335.*
[12] R. Kaivola, N. Narasimhan: Formal Verification of the Pentium(R) 4 Floating-Point Multiplier *Proc. of the conference on Design, automation and test in Europe (DATE), 2002.*
[13] J. Lewis: Cryptol: specification, implementation and verification of high-grade cryptographic applications. In *FMSE'07: Proceedings of the 2007 ACM workshop on Formal methods in security engineering, pp. 41.*
[14] S. Morioka, Y. Katayama, T. Yamane: Towards Efficient Verification of Arithmetic Algorithms over Galois Fields $GF(2^m)$, in Proc. CAV 2001, LNCS 2102, pp. 465-477.
[15] D. Mukhopadhyay, G. Sengar, D.r. Chowdhury: Hierarchical Verification of Galois Field Circuits. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 26, No. 10, October 2007.
[16] C.-J.H. Seger, R.E. Bryant: Formal Verification by Symbolic Evaluation of Partially-Ordered Trajectories. *Formal Methods in System Design, 6(2):147-189,1995.*
[17] A. Slobodová: Challenges for Formal Verification in Industrial Setting. In Proc. of FMICS/PDMC 2006, pp.1-22.
or *Formal Methods: Applications and Technology, 11th International Workshop*, FMICS 2006 and 5th International Workshop PDMC 2006, Bonn, Germany, August 26-27, and August 31, 2006, Revised Selected Papers. Springer (2007), LNCS 4346.

BACH : *B*ounded Reach*A*bility *CH*ecker for Linear Hybrid Automata

Lei Bu, You Li, Linzhang Wang, Xuandong Li
State Key Laboratory for Novel Software Technology, Nanjing, Jiangsu, P.R.China, 210093
Department of Computer Science and Technology, Nanjing University, Nanjing, Jiangsu, P.R.China, 210093
Email: bl@seg.nju.edu.cn, lxd@nju.edu.cn

***Abstract*—Hybrid automata are well studied formal models for hybrid systems with both discrete and continuous state changes. However, the analysis of hybrid automata is quite difficult. Even for the simple class of linear hybrid automata, the reachability problem is undecidable. In the author's previous work, for linear hybrid automata we proposed a linear programming based approach to check one path at a time while the length of the path and the size of the automaton being checked can be large enough to handle problems of practical interest. Based on this approach, in this paper we present a prototype tool BACH to perform *b*ounded re*a*chability *ch*ecking of linear hybrid automata. The experiment data shows that BACH has good performance and scalability, and supports our belief that BACH could become a powerful assistant to design engineers for the reachability analysis of linear hybrid automata.**

I. Introduction

Hybrid automata [1] are well studied formal models for hybrid systems with both discrete and continuous state changes. However, the analysis of hybrid automata is very difficult. Even for the simple class of *linear hybrid automata (LHA)*, the reachability problem is undecidable [1] [2] [3]. Several model checking tools have been developed for LHA, but they do not scale well to the size of practical problems. The state-of-the-art tool HYTECH [9] and its improvement PHAVer [10] need to perform expensive polyhedra computation, and their algorithm complexity is exponential in number of variables in the automata.

In recent years, bounded model checking (denoted as BMC) [4] has been presented as an alternative technique for BDD-based symbolic model checking, whose basic idea is to encode the next-state relation of a system as a propositional formula, and unroll this formula to some integer k, using SAT idea to search for a counterexample in the model executions whose length is bounded by k. As extensions to BMC, there are several related works [5] [6] using linear programming technique with SAT to check linear hybrid systems. In these works, the bounded model checking problems are reduced to the satisfiability problem of a boolean combination of propositional variables and linear mathematical constraints. Based on these works, several tools were proposed accordingly, such as HySAT [5], MathSAT [6]. All of these solvers are built based on a SAT-solver that calls a linear program solver for conjunctions of the linear continuous-part constraints. As this technique requires to encode the whole problem space firstly, when the system size or the given step threshold is large, the object problem will be very huge, which greatly restricts the size of the problem that can be solved.

In our previous study [7], for LHA we proposed an linear programming based approach to develop an efficient path-oriented tool to check one path at a time while the length of the path and the size of the automaton being checked can be large enough to handle problems of practical interest. Based on this approach, in this paper we present a prototype tool BACH to do bounded reachability checking of linear hybrid automata. The main function of BACH includes

- *Path-Oriented Reachability Analysis*: to check a specific path in a LHA using linear programming;
- *"Full" Bounded Reachability Analysis*: to check all the paths in a LHA in a given step threshold of the system by traversing in depth first search (DFS).

We have conducted some case studies using BACH, and the experiment data shows that:

- for the path-oriented reachability analysis, the length of the path being checked can be made especially large and the size of the automaton can be made large enough to handle problems of practical interest; and
- for the "full" bounded reachability analysis, BACH has much better performance and scalability than the SAT-style solver HySAT.

The rest of the paper is organized as follows: Section II gives a simple description of the underlying techniques of BACH. Section III depicts the main functionality of BACH. Section IV describes several case studies to show the processing ability of BACH. Finally the conclusion is stated in Section V.

II. The Underlying Techniques

Here we use one simple case scenario to illustrate the main underlying techniques of BACH. The automaton in Fig. 1 describes a model of water level monitor cited from [3]. The water level is controlled through a monitor, which continuously senses the water level and turns a pump on and off. When the pump is off the water level falls by 2 per second, and when the pump is on the water level rises by 1 per second. And there is a delay of 2 seconds between the time monitor signals to change the status of the pump and the time that the change becomes effective. When the system is in location v_1, if the monitor finds the water level is 0, the system will jump to location v_5 and stop.

978-1-4244-2735-2/08/$25.00 ©2008 IEEE

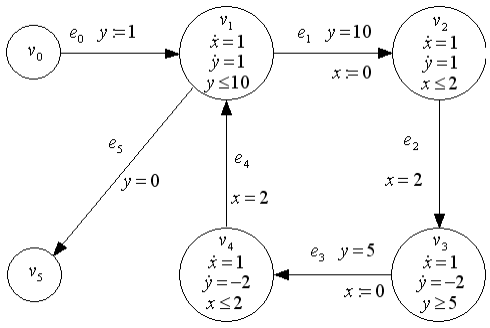


Fig. 1. The automaton modeling water-level monitor system (WLM)

A. Path-Oriented Reachability Analysis

The main decision procedure under BACH is the path-oriented rechability analysis approach given in our previous study [7]. In order to illustrate our method succinctly without loss of generality, the path we choose is a traverse of each transition for exactly once, that is $v_0\hat{}v_1\hat{}v_2\hat{}v_3\hat{}v_4\hat{}v_1\hat{}v_5$. The problem we concern is to check the reachability of location v_5 along this path, which can be reduced to a linear program whose linear inequality set is defined as follows:

1) We represent the timed behavior of the path by the form $(v_0,t_0)\hat{}(v_1,t_1)\hat{}(v_2,t_2)\hat{}(v_3,t_3)\hat{}(v_4,t_4)\hat{}(v_1,t_5)\hat{}(v_5,t_6)$ where t_j stands for the time spent in the jth location in the path. Thus, we will generate 6 variables t_i $(1 \le i \le 6)$, and 6 corresponding constraints in the linear program $t_i \ge 0$. As initial location fires its transition immediately, the time spent in it is set to 0, which means $t_0 = 0$, so we don't need to generate variable t_0.
2) For each location invariant and transition guard in the path, we will generate the corresponding linear constraints, e.g., for $y \le 10$ in location v_1 and $y = 10$ in transition e_1, as the change rate of y in v_1 is 1, we can generate four constraints: $y_{v_{1in}} \le 10$, $y_{v_{1out}} \le 10$, $y_{v_{1in}} + t_1 = y_{v_{1out}}$, and $y_{v_{1out}} = 10$, where $y_{v_{1in}}$ stands for the value of y when the automaton comes into location v_1, and $y_{v_{1out}}$ stands for the value of y when the automaton is going to jump to v_2 after having stayed at location v_1 for time t_1.
3) For each reset action in the path, we will also generate the corresponding constraints, e.g., for $y := 1$ in the transition e_0, we get $y_{v_{1in}} = 1$.

According to the above steps, the problem of checking if v_5 is reachable along the path will be reduced into an linear programming problem with 43 constraints and 12 variables.

Besides of the above original technique, we also develop two optimization techniques in [7], *path decomposition* and *path shortening*, to reduce the size of the resulting linear program so that the tool can be used to solve problems of size as large as possible. The *path decomposition* technique is to decompose the linear program corresponding to the checked path into several separate smaller linear programs so that the tool can check longer paths. Since the size of the linear program corresponding to the checked path is proportional to the length of the path, the *path shortening* technique is to find a shorter path in lieu of the checked one such that both of them are equivalent with respect to the given reachability specification - if one of them satisfies the reachability specification, so does the other.

B. "Full" Bounded Reachability Analysis

The bounded reachability analysis is to look for a system trajectory in a given length threshold which can reach a given target. As we can already check the reachability of one single path, we can traverse the system structure directly in DFS and check all the potential paths one by one until a feasible path to the reachability target is found or the length threshold is reached. Instead of encoding the whole problem space to a group of formulas like SAT-style solver, which will suffers a lot when dealing with big problems, this plain DFS style approach only needs to keep the discrete structure and current visiting path in memory, and check each potential path one by one, which makes it possible to solve big problems as long as enough time is given.

We have also devised some optimizations for decreasing the number of paths that need to be checked using linear programming. In general, BACH checks each path p in a given length threshold for the reachability by solving the corresponding linear program. However, sometimes there is no target location in p at all so that we can simply falsify p for the reachability. For example, when we check whether location v_5 is reachable in Fig.1, the path v_1, $v_1\hat{}v_2$, $v_1\hat{}v_2\hat{}v_3$, $v_1\hat{}v_2\hat{}v_3\hat{}v_4$, and $v_1\hat{}v_2\hat{}v_3\hat{}v_4\hat{}v_1$ can simply be falsified for the reachability. With this simple optimization technique, BACH only needs to check the reachability for the paths reaching the target location. Furthermore, as BACH does the bounded reachability analysis by a DFS based traversing of paths, once we find that the current checked path is infeasible (no timed behavior corresponding to the path) we should backtrack. We can take advantage of the irreducible infeasible subset (IIS) technique [11] to find the constraints which result in no solution of the linear program, which can help us to return to the most reasonable branch location for starting new search.

Thanks to the advancement in computing during the past decade, linear programs with several hundreds variables and constraints are considered "small" nowadays, and the ones with tens or hundreds of thousand of variables can be solved very efficiently, which makes it possible for BACH to check considerable number of long paths within tolerable time.

III. Functionality of BACH

The ***B***ounded reach***A***bility ***CH***ecker of LHA (BACH) is implemented in Java, and the latest version of BACH can be downloaded from (http://seg.nju.edu.cn/BACH/BACH.html). Several parts of GUI are shown in Fig.2 with the system structure shown in Fig.3 as well. The linear programming software package integrated in BACH is from OR-Objects of DRA Systems [8] which is a free collection of Java classes for developing operations research, scientific and engineering applications.

BACH is composed of two main parts as follows:

- Graphical LHA Editor: It allows users to construct, edit, and perform syntax analysis of LHA interactively. This Editor can also save the graphical representation of LHA to a human readable text file which is used as the input file for reachability checker.
- Bounded Reachabilty Cheker: It accepts a LHA file generated by the Editor, and perform the bounded reachability checking. BACH supports two kinds of reachability checking:
  - Path-Oriented Reachability Checking: The checker requires users to select a specific path in a LHA, and use linear programming to check the path for reachability.
  - "Full" Bounded Reachability Checking: The checker requires users to input the bounded threshold for a LHA, traverses all the paths in the bounded threshold by DFS, and check the related paths for reachability using linear programming. Once the reachability specification is satisfied, a witness will be given also by the checker.

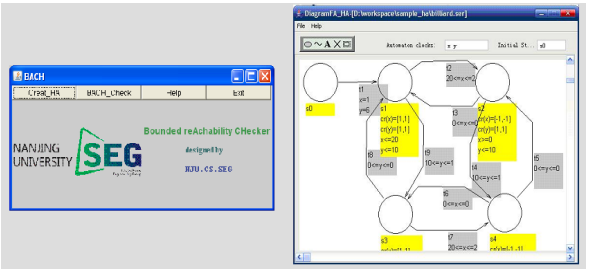


Fig. 2. BACH GUI

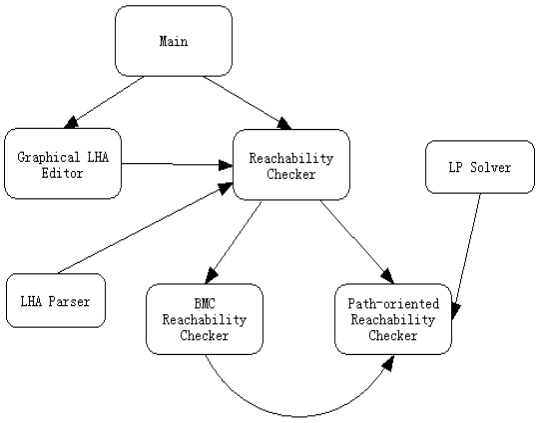


Fig. 3. BACH System Structure

IV. Case Studies

BACH has been used for the bounded reachability analysis of two well-known examples: the water level monitor and the temperature control system [3]. The experiments are conducted on a Fedora Server (Intel Xeon CPU 1.8GHz/3.78GB).

TABLE I
PATH-ORIENTED CHECKING RESULTS ON THE WATER-LEVEL MONITOR

Path: $v_0\hat{}v_1\hat{}(v_2\hat{}v_3\hat{}v_4\hat{}v_1)^k\hat{}v_5$					
k	Original technique			Decomposing	Shortening
	constraints	variables	time	time	time
200	7207	2002	503.140s	1.562s	0.031s
400	14407	4002	4202.421s	3.187s	0.031s
500	Java.lang.out of memory error			4.109s	0.031s
10000	Java.lang.out of memory error			38.047s	0.031s

TABLE II
PATH-ORIENTED CHECKING RESULTS ON THE TEMPERATURE CONTROL SYSTEM

Path: $v_0\hat{}v_1\hat{}(v_2\hat{}v_1\hat{}v_3\hat{}v_1)^k\hat{}v_4$					
k	Original technique			Decomposing	Shortening
	constraints	variables	time	time	time
200	8815	2004	686.938s	686.938s	686.938s
400	17591	3998	5574.312s	5574.312s	5574.312s
450	Java.lang.out of memory error			Java.lang.out of memory error	

For the water level monitor whose model is depicted in Fig.1, we want to check whether the tank will be empty in the future (the location v_5 is reachable). For the temperature control system whose model is depicted in Fig.4, which describes a cooling system with two control rods, we plan to check whether a complete shutdown is required after the system has run for a while (the location v_4 is reachable). We set up the parameters such that both of the locations are not reachable in the models.

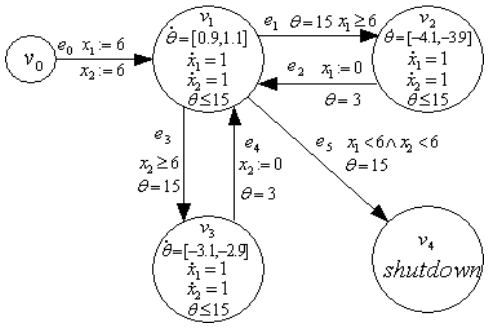


Fig. 4. The automaton modeling temperature control system (TCS)

The experimental data in Table.I and Table.II illustrate the processing ability of the Path-Oriented Checking in BACH. It is shown that the length of the single path analyzed by BACH without optimization can be as long as 1600 $(400 * 4)$ steps. Furthermore, if the optimization techniques are applicable which split a big linear program into several independent small ones, the problem size solved by BACH can be extremely large. For example with the path decomposition technique, BACH can analyse paths of length up to 40000 $(10000 * 4)$ in only 0.031 seconds, as shown in Table.I. Actually the inherent reason for such good performance and scalability is that the size of linear program solved in the Path-Oriented Checking is linear in the path length and the variable number in the model.

We have also evaluated the the processing ability of the "Full" Bounded Reachability Checking (BRC) in BACH using the same examples, and conduct a comparison with the SAT-style solver HySAT [5]. The experiment data are shown in

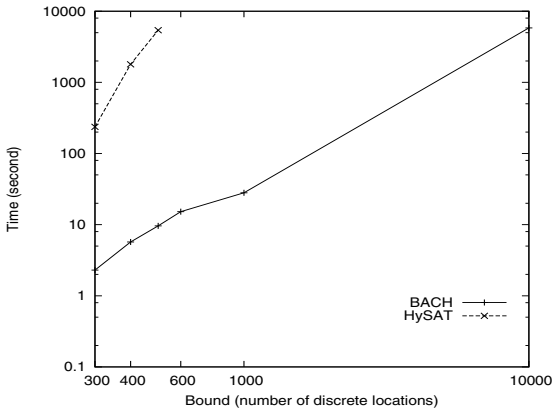


Fig. 5. BRC Results of water level monitor system of BACH and HySAT

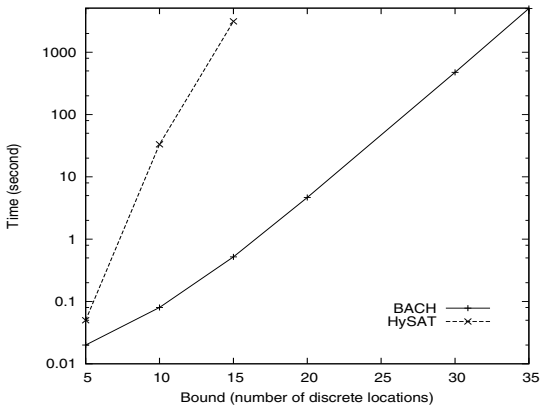


Fig. 6. BRC Results of temperature control system of BACH and HySAT

Fig.5 and Fig.6, from which we can see that

- *BACH has much better scalability than HySAT*. HySAT encodes the whole problem space to a group of formulas before checking. When the system size or the given step threshold is large, the object problem will be very huge, which greatly restricts the solvable problem size. As the underlying technique of BACH is to recursively traverse and check all the paths in given length one by one, no matter how big the problem size is, in memory BACH only needs to save the discrete LHA structure and the current visiting path; and
- *BACH has much better performance than HySAT*. BACH goes through each path directly from discrete LHA structure and check it efficiently by linear programming, while HySAT uses a SAT-solver to choose the discrete path and calls another linear program solver for conjunctions of the continuous-part constraints. We think this kind of interaction between SAT solver and linear program solver might be time consuming.

The above opinions contain some intuition, and the more deep reason that BACH works better than HySAT merits further investigation.

We also attempt to compare BACH with another SAT-style solver MathSAT [6]. But because from [6] we cannot get the details of encoding LHA with MathSAT language, we have to leave the comparison in the future work when we gain a mastery of encoding LHA in MathSAT.

V. Conclusion

In this paper, we present a bounded reachability checker BACH for linear hybrid automata. BACH provides a convenient GUI to construct LHA model, a powerful path-oriented reachability checker to analyze one single LHA path, and a "full" bounded reachability checker to do reachability checking in a given threshold. The experiment data shows that BACH has good performance and scalability, and supports our belief that BACH could become a powerful assistant to design engineers for the reachability analysis of linear hybrid automata.

Acknowledgment

We would like to thank Prof. Tom Melham and the anonymous reviewers for their valuable comments and suggestions. This work is supported by the National Natural Science Foundation of China (No.60425204 and No.60721002), the National 863 High-Tech Programme of China (No.2007AA010302), and by the Jiangsu Province Research Foundation (No.BK2007714).

References

[1] Thomas A. Henzinger. The theory of hybrid automata. In *Proceedings of the 11th Annual IEEE Symposium on Logic in Computer Science (LICS 1996)*, IEEE Computer Society Press, 1996, pp. 278-292.

[2] Thomas A. Henzinger, Peter W. Kopke, Anuj Puri, and Pravin Varaiya. What's Decidable About Hybrid Automata? In *Journal of Computer and System Sciences*, 57:94-124, 1998.

[3] R. Alur, C. Courcoubetis, N. Halbwachs, T.A. Henzinger, P.-H.Ho, X. Nicollin, A. Olivero, J. Sifakis, S. Yovine. The algorithmic analysis of hybrid systems. In *Theoretical Computer Science*, 138(1995), Elsevier Science, pp.3-34.

[4] Armin Biere, Alessandro Cimatti, Edmund Clarke, Ofer Strichman, Yunshan Zhu. Bounded Model Checking. In *Advance in Computers*, Vol.58: 118-149, Academic Press, 2003.

[5] Martin Franzle, Christian Herde. Efficient Proof Engines for Bounded Model Checking of Hybrid Systems. In *Proceedings of the Ninth International Workshop on Formal Methods for Industrial Critical Systems (FMICS 2004)*, Electronic Notes in Theoretical Computer Science, Vol. 133, Elsevier Science, 2005, pp. 119-137.

[6] Gilles Audemard, Marco Bozzano, Alessandro Cimatti, Roberto Sebastiani. Verifying Industrial Hybrid Systems with MathSAT. In *Proceedings of the 2nd International Workshop on Bounded Model Checking (BMC2004)*, Electronic Notes in Theoretical Computer Science, Vol.119, Issue 2, Elsevier Science, 2005, pp. 17-32.

[7] Xuandong Li, Sumit Kumar Jha, and Lei Bu. Towards an Efficient Path-Oriented Tool for Bounded Reachability Analysis of Linear Hybrid Systems using Linear Programming. In *Proceedings of the Fourth International Workshop on Bounded Model Checking (BMC06)*, Electronic Notes in Theoretical Computer Science, Vol.174, Issue 3, Elsevier Science, 2007, pp.57-70.

[8] OR-Objects. http://OpsResearch.com/ OR-Objects/index.html.

[9] T.A. Henzinger, P.-H. Ho, and H. Wong-Toi. HYTECH: a model checker for hybrid systems. In *Software Tools for Technology Transfer*, 1:110-122, Springer, 1997.

[10] Goran Frehse. PHAVer: Algorithmic Verification of Hybrid Systems past HyTech. In Manfred Morari and Lothar Thiele, editors, *Hybrid Systems: Computation and Control (HSCC'05)*, Lecture Notes in Computer Science 2289, Springer-Verlag, 2005, pp.258-273.

[11] Chinneck, J., Dravnieks, E.: Locating minimal infeasible constraint sets in linear programs. In ORSA Journal on Computing, 3 (1991), 157-168.

Going with the Flow: Parameterized Verification using Message Flows

Murali Talupur
Intel
murali.talupur@intel.com

Mark R. Tuttle
Intel
tuttle@acm.org

***Abstract*—A message flow is a sequence of messages sent among processors during the execution of a protocol, usually illustrated with something like a message sequence chart. Protocol designers use message flows to describe and reason about their protocols. We show how to derive high-quality invariants from message flows and use these invariants to accelerate a state-of-the-art method for parameterized protocol verification called the CMP method. The CMP method works by iteratively strengthening and abstracting a protocol. The labor-intensive portion of the method is finding the protocol invariants needed for each iteration. We provide a new analysis of the CMP method proving it works with any sound abstraction procedure. This facilitates the use of a new abstraction procedure tailored to our protocol invariants in the CMP method. Our experience is that message-flow derived invariants get to the heart of protocol correctness in the sense that only couple of additional invariants are needed for the CMP method to converge.**

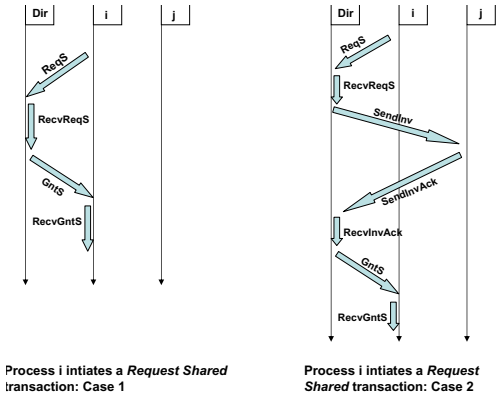


Fig. 1. Two typical message flows

I. Introduction

Invariants play a crucial role in most approaches to verification, but finding good invariants is a challenging problem. The methods proposed for finding invariants fall into roughly two classes. The manual approach is to begin with a simple candidate for an invariant, and to elaborate the invariant based on counterexamples produced by a mechanical checker or prover. There are also automatic approaches such as invisible invariants [23], [4], indexed predicates [16], interpolant-based invariant generation [21], and split invariants [12] that try to deduce invariants more or less automatically. In both approaches, the resulting invariant is often unwieldy and rarely insightful. The invariants are usually low-level expressions formulated in terms of the protocol variables and the high-level understanding of why the protocol works is either unused or obscured.

In this paper, we show that *message flows* are a succinct and readily available source of high-level invariants that usually go unused in protocol verification. A message flow is a sequence of messages sent among processors following a protocol that logically constitutes a single transaction of the protocol. Flows are often illustrated by protocol designers in the form of message sequence charts. Figure 1, for example, illustrates two flows from the German cache coherence protocol.

Message flows are linear, local, and typically involve only a small number of processes. They arise naturally in the context of protocols and embody local ordering relations among the actions of different agents. Message flows are found everywhere from protocol design documents to conference papers to informal explanations on white boards. Yet, in spite being a compelling way of understanding protocols, flows almost never appear in that ultimate explanation of why a protocol works, namely, the correctness proof itself. We show that message flows easily yield high-quality invariants, and use them to accelerate a state-of-the-art approach to parameterized protocol verification called the CMP method.

The *CMP method* [20], [9], [15] is one of the most successful methods for parameterized protocol verification. Most protocols are naturally described in terms of a few parameters such as the number of processors n. Parameterized verification is the problem of proving protocol correctness for all values of n, and not just the small values that can be checked with modern model checkers. The CMP approach to parameterized verification is a combination of *data type reduction* [19] and *compositional reasoning* [18] first proposed by McMillan [18], later elaborated by Chou, Mannava, and Park [9], and formalized by Krstic [15]. In this approach, a model checker is used as proof assistant and the user guides the proof by supplying invariants or "noninterference lemmas." Speaking informally, CMP works by taking a protocol for n processors, retaining two or three processors and replacing the remaining processors with a single, highly-nondeterministic processor called $Other$, and then formulating a set of protocol invariants that serve to constrain the behavior of $Other$ sufficiently to allow model checking to validate the desired property.

The CMP method scales very well. It has been used to verify Flash protocol [20], [9], a protocol sufficiently complex that

978-1-4244-2735-2/08/$25.00 ©2008 IEEE

only two or three other methods have been able to fully validate the protocol parametrically. At Intel, we have used CMP to verify an industrial-strength cache protocol several orders of magnitude larger than even the Flash protocol [24]. Although manual guidance is necessary in the form of noninterference lemmas, the lemmas supplied by the user do not have to add up to an inductive invariant. Consequently, CMP is much easier on the user than other theorem-proving style methods such as [22].

The hardest part of using CMP is finding a set of protocol invariants that enable CMP to converge. We show that invariants derived from message flows constrain the *Other* process so well that the burden of manual addition of further invariants is significantly reduced. We could verify the control and data safety properties of German's protocol and Flash protocol by adding manually at most two lemmas. Augmenting the basic CMP method with flow based invariants thus results in an *even more scalable and a much easier to use method.*

In this paper we make the following contributions:

1) *We show how to derive invariants from message flows.* We give a simple language for describing message flows, and show how adding a few auxiliary variables to the protocol description allows us to describe these message flows as ordinary state predicates.
2) *We show that CMP can be generalized.* The CMP method works by iteratively *strengthening* and *abstracting* a protocol. We provide a new analysis of CMP that cleanly separates the strengthening and abstraction procedures used by CMP. We prove that CMP works with any sound abstraction procedure, in contrast to the original analysis that depended heavily on protocol symmetry and the use of data type reduction as the abstraction procedure.
3) *We show how to use message flow invariants with CMP.* We provide a new abstraction procedure for use with our message flow invariants, and show how to use them with CMP.
4) *We demonstrate that message flow invariants simplify and accelerate CMP.* We apply our ideas to the German and Flash protocols. We show that starting with the invariants derived from message flows, couple of additional invariants are enough for CMP to validate both the data and control properties, in contrast to the several complex invariants needed in the original application of CMP [9], and in contrast to recent automated methods [7], [17] that can verify only the control safety properties.

A. Related Work

Parameterized verification, mainly of cache coherence and mutual exclusion protocols, has received considerable attention. Recent methods have been based on invisible invariants [4], counter abstraction [23], indexed predicates [16], ordinary model checking [3], WS1S [5], learning [14], strengthening split invariants [12], and environment abstraction [10], [11]. These techniques have had varying degrees of success, but none of them has been applied to a large industrial-strength protocol like Flash (although environment abstraction [11] has been applied to a simplified version of Flash).

McMillan introduced the CMP method in a series of papers [18], [20], [9] (the proof in [20] does not completely follow the CMP framework: to get BDDs to scale, the Flash model had to be pruned by hand.) Chou, Mannava, and Park [9] elaborated on the method and showed how it can be performed in conjunction with any model checker. Krstic [15] gave a formalization of the method.

As far as we are aware, the CMP method is one of the few methods to handle the full complexity of the Flash protocol. A transaction-based method [22] and a predicate abstraction-based method [13] have both verified the safety properties of Flash. Both the methods required extensive manual guidance and took several days to complete the verification. Recently, some nearly automatic methods based on BDDs have been proposed [17], [7]. As with the other BDD-based methods [20], they have trouble scaling up: they have been able to verify only a control property, but not data. For all of these methods applied to Flash, model checking has taken roughly a couple hours. In contrast, our work is able to check both the data and control properties of Flash in just 120 seconds.

We note that our method of deriving invariants from message flows is not specific to CMP, and can potentially be used in conjunction with any other method of protocol verification.

Finally, the compositional reasoning principle at the heart of our new analysis of CMP is similar to principles proposed by Abadi and Lamport [1], [2], McMillan [18], Krstic [15], Bhattacharya et al. [6] and Chen et al. [8].

We introduce preliminary concepts in Section II, derive protocols invariants from message flows in Section III, present a new analysis of CMP in Section IV, show how to use flow-derived invariants with CMP in Section V, give experimental results in Section VI, and conclude with Section VII.

II. Parameterized Protocols

A parameterized protocol $P(N)$ is a protocol for N processes with unique ids in $\mathbb{N}_N = [1..N]$. We follow Krstic [15] and define the simplest model sufficient to express cache coherence protocols, but our work depends only on the protocol being symmetric and described with guarded commands.

Index variables: Fix a set $\mathcal{I}$ of index variables for quantifying over process ids. When we write $T(i, j)$, we mean that i and j are the only index variables appearing free in T.

States: The state of a protocol consists of global and local variables that can hold either Boolean values or process ids. A global variable (like a directory entry) is a scalar variable, and a local variable (like a cache entry) is an array variable indexed by process id. Formally, the state is determined by four sets of variables, W, X, Y, Z where variables in W are of type $\mathbb{B}$, in X are of type $\mathbb{N}_N \to \mathbb{B}$, in Y are of type $\mathbb{N}_N$ and those in Z are of type $\mathbb{N}_N \to \mathbb{N}_N$. Note that the types of the variables are determined by the parameter N.

Expressions: An expression is made up of combination of the four *basic expressions*

$$w \qquad y = i \qquad x[i] \qquad z[i] = j$$

where $i, j \in \mathcal{I}$ are index variables and $w \in W, x \in X, y \in Y, z \in Z$ in the usual fashion by taking their Boolean combinations and possibly quantifying out some free index variables.

Assignments: An assignment is one of the four *basic assignments*

$$w := b \qquad y := i \qquad x[i] := b \qquad z[i] := j$$

where $i, j \in \mathcal{I}$, where b is either *true* or *false*, and $w \in W, x \in X, y \in Y, z \in Z$, together with the *quantified assignment*

$$\forall I.\phi \Rightarrow v := e$$

where $v := e$ is one of the basic assignments. The latter is just a set of conditional assignments: consider all possible assignments of values to indices in I, and for each such assignment, perform $v := e$ if ϕ is true.

Rules: A rule is a *guarded command* written as

$$rl : \forall i, j.\rho \rightarrow a \qquad \text{or} \qquad rl(i,j) : \rho(i,j) \rightarrow a(i,j)$$

where rl is the *rule name*, ρ is an expression called the *guard*, and a is a list of assignments called the the *action*. We assume that i and j are the only free index variables in ρ and a. Note that we use the ordered pair (i, j) to emphasize that the ordering of these indices is important. We assume that an action never assigns two values to the same variable, ruling out asymmetric assignments like $\forall k.true \Rightarrow y := k$.

Protocols: A *protocol* is a state transition system (S, Θ, T) where S and Θ are the sets of states and initial states and $T \subseteq S \times S$ is the transition relation given by a set of rules. There is a transition from s to s' if there is a rule $rl(i,j) : \rho(i,j) \rightarrow a(i,j)$ and an instantiation p, q such that s satisfies $\rho(p,q)$ and s' is the result of starting with s and performing the assignments in $a(p,q)$.

From these restrictions on expressions and assignments it is clear that the protocol P must be symmetric. The following lemma, which simply restates symmetry in a different form, will be used later. Let R_i denote the set of all reachable states of P within i steps.

Lemma 1: For all $i \geq 0$ and $M \leq N$:

- $\forall s \in R_i.s \models \phi(1,..,M)$ implies $\forall s \in R_i.s \models \forall i_1,..,i_M.\phi(i_1,..,i_M)$.
- $P \models \phi(1,..,M)$ implies $P \models \forall i_1,..,i_M.\phi(i_1,..,i_M)$.

III. Message Flows

Protocol designers use message flows, like the two flows from German's protocol illustrated in Figure 1, to describe the basic organization of a protocol and to reason about the protocol itself. Message flows implicitly impose constraints on the order in which the actions appearing within them can happen. In the flow on the left of Figure 1, the cache sends the directory a *ReqS* message (requesting shared access). The directory executes a *RecvReqS* action to receive the *ReqS* message, and then sends the cache a *GntS* message (granting shared access). Finally, the requesting cache does a *RecvGntS* action to complete the transaction. Each action can happen only after the preceding action has been executed. For instance, the directory cannot send a *GntS* message before it has executed a *RecvReqS* action. In the flow on the right of Figure 1, the directory can send a *GntS* message only after all the *SendInv* subflows have completed. Thus, flows impose ordering constraints on the rules and subflows appearing in them, and the protocol invariants we define in this section are intended to capture these constraints.

ReqShare(i):
SendReqS(i), RecvReqS(i), **SendInval(k),**
SendGntS(i), RecvGntS(i)
ReqExcl(i):
SendReqE(i), RecvReqE(i), **SendInval(k),**
SendGntE(i), RecvGntE(i)
SendInval(i):
SendInv(i), SendInvAck(i), RecvInvAck(i)

Fig. 2. Three flows from German's protocol. Flow names are in bold.

A. Language for Describing Flows

Each flow is of the form:

$$fl(i,j) : a_1, \ldots, a_n, \ldots, a_m$$

where fl is the name of a flow with two parameters i and j from $\mathcal{I}$ and each a_k is either

- a *rule* of the form $rl(i,j)$ where rl is name of a ruleset with two parameters instantiated with i and j, or
- a *subflow* of the form $sfl(k,l)$ where sfl is the name of another flow in the system with two parameters instantiated with k and l, possibly different from i and j.

We assume for the sake of exposition that every flow and rule has two parameters. In practice, the definition of a rule $rl(i,j)$ might omit reference to one of the parameters and reduce to $rl(i)$ or $rl(j)$, but the modifications to our presentation required to deal with this possibility are elementary. A rule $rl(i,j)$ appearing in a flow represents just a single firing of the rule rl with the quantifiers instantiated to i and j. A subflow $sfl(k,l)$ appearing in a flow can be instantiated zero or more times with varying instantiations of the parameters. For example, Figure 2 illustrates three flows from German's protocol, including the *ReqShare* flows illustrated in Figure 1 in which the subflow *SendInval* occurs zero times on the left and one time on the right.

The reason for allowing flows to occur within other flows is as follows. While flows in a protocol usually involve at most two agents, some times it can happen that additional processes get involved as well. For instance, this happens in the *ReqShare* transaction shown on the right side of Figure 1. The directory has to send an invalidate message to any process holding the data item in an exclusive state. This series of rule firings involving agents different from the primary agents of the flow is modeled in our language by having a *subflow* inside a flow. Note from Figure 2 that the flow *SendInval* occurs in two different flows. In case of the *ReqExcl* transaction, several

$update(p, fl(i,j), rl(i,j))$ =
  assert($p \in \{i,j\}$);
  let $last\_rule$ =
    if $rl'(i,j)$ preceeds $rl(i,j)$ in $fl(i,j)$
    then $\{(fl, rl')\}$
    else $\{\}$;
  let $next\_rule$ =
    if $rl(i,j)$ ends $fl(i,j)$
    then $\{\}$
    else $\{(fl, rl)\}$;
  $Aux(p) := Aux(p) \setminus last\_rule \cup next\_rule$

Fig. 3. Tracking flows with auxiliary variables. If $rl(i,j)$ is a rule appearing in a flow $fl(i,j)$, then $update(p, fl(i,j), rl(i,j))$ updates the value of $Aux(p)$ when the rule fires. Since $Aux(p)$ is a multiset, the set difference operator removes just one copy of $last\_rule$.

instantiations of *SendInval* flow might be forked from the main flow. In case of the *ReqShare* transaction at most only one instantiation of *SendInval* is spawned from the main flow.

Given a cache coherence protocol we can associate a set of *flows* $\{fl_1, fl_2, .., fl_k\}$ with it. For this set of flows to be *legal*, there should be a topological ordering $\prec$ of the flows such that a flow fl_m can occur inside another flow fl_n only if $fl_m \prec fl_n$. This prevents circular specification of flows.

Remark 1: We can enrich our language by specifying the precise number of times a subflow can be instantiated within a flow. Or by having more than two entities be involved with in the main part of a flow. While many such generalizations are possible, the information contained in these simple flows is enough for the application in this paper.

B. Tracking Flows using Auxiliary Variables

We add auxiliary variables to a protocol to track the state of the various flows. Let P be a protocol, and let P_a be the augmented protocol obtained by adding auxiliary variables to P as follows. For each process p, let $Aux(p)$ be a multiset consisting of pairs (fl, rl) where $fl(i,j)$ is a flow and $rl(i,j)$ is a rule appearing in $fl(i,j)$. Initially, $Aux(p)$ is empty.

Whenever a rule instantiation $rl(p_1, p_2)$ fires the sets $Aux(p_1)$ and $Aux(p_2)$ are updated using the procedure given in Figure 3 by calling the $update(p, fl(p_1,p_2), rl(p_1,p_2)$ for both $p = p_1$ and $p = p_2$. In the update procedure we say a rule $rl'(i,j)$ or subflow $sfl(k,l)$ *immediately precedes* or *precedes* $rl(i,j)$ in flow $fl(i,j)$ if there is no other rule between them (although there can be subflows between them).

C. Noninterference Lemmas from Flows

A flow implicitly orders the rules appearing in the flow. A flow $fl(i,j)$ induces two preconditions on the firing of any rule $rl(i,j)$ appearing within it:

1) If rule $rl'(i,j)$ precedes $rl(i,j)$, then rl can fire only after rl'. That is,

$$(fl, rl') \in Aux(i) \wedge (fl, rl') \in Aux(j).$$

If no rule precedes $rl(i,j)$, then the precondition is *true*.

2) If subflow $sfl(k,l)$ precedes $rl(i,j)$, then all instances of sfl started by fl must have ended before rl can fire. That is, assuming sfl occurs only in fl,

$$\forall k \in \mathbb{N}_N.(sfl, \_) \notin Aux(k).$$

If no subflow precedes $rl(i,j)$, then the precondition is *true*.

Thus, each rule $rl(i,j)$ in a flow $fl(i,j)$ leads to a condition $c(fl, rl)$ that is a conjunction of the two conditions above. If the rule $rl(i,j)$ appears in more than one flow, then we find $c(fl, rl)$ for each flow $fl(i,j)$ that $rl(i,j)$ appears in and the disjunction of all such conditions is defined to be the precondition $p_{rl}(i,j)$ for the firing of rule $rl(i,j)$.

We transform the preconditions $p_{rl}(i,j)$'s into noninterference lemmas as follows. Consider the rule rl and the associated precondition $p_{rl}(i,j)$. Suppose rl is of the form $rl : \forall(i,j).\rho \rightarrow a$. The noninterference lemma is simply

$$\forall i, j.\rho(i,j) \Rightarrow p_{rl}(i,j)$$

This lemma says if the rule rl is enabled for processes i and j — if $\rho(i,j)$ is true — then the precondition associated with the rule must be true as well. Note that the lemmas derived from flows involve both the variables of the protocol itself and the auxiliary variables. *Thus, they are bridging two levels: the informal, metalevel consisting of the rule names and the formal level consisting of the model under verification itself.*

As we will see in Section V, these lemmas are used exactly like any other noninterference lemma in the CMP method. If any of the lemmas is wrong then our framework, which both uses and validates the lemmas, will catch the violating lemma.

Remark 2: In practice, only protocol actions that involve the directory yield useful constraints and to optimize the abstract models we use only those in our experiments.

IV. Generalized CMP

The CMP method consists of two basic steps — *abstraction* and *strengthening* — that are applied iteratively to a protocol. The abstraction procedure used in CMP retains detailed information on a small number of processes, and abstracts away the remaining processes. Since our protocols are symmetric there is no loss in generality in focussing on processes 1 and 2. Given a symmetric protocol P with N processors $[1..N]$ and a property $I = \forall i, j \in [1..N].I(i,j)$ that we want to prove is an invariant of P, the method is as shown below. We assume that the abstraction procedure

CMP(P,I) =
  $P^{\#} = P$; $I^{\#} = I$
  while $abstract(P^{\#}) \not\models abstract(I^{\#}(1,2))$ do
    examine counterexample *cex*
    exit if *cex* is a real counterexample
    find $L = \forall i, j.L(i,j)$ ruling out *cex*
    $P^{\#} = strengthen(P^{\#}, L)$
    $I^{\#}(i,j) = I^{\#}(i,j) \wedge L(i,j)$
  end

If the loop terminates normally, the method and protocol symmetry allow us to conclude that $I^{\#}$ and consequently I are invariants of P. If the loop terminates via the exit, then either I or one of the proposed lemmas L is not an invariant of the protocol, and the user must back up and try again.

In this section, we present a new analysis of CMP that improves over the original in several ways. The main advantage of our analysis is that it allows any sound abstraction to be used as the abstraction step, and not just the data type reduction used in the original paper. In particular, in Section V, we will use message flow invariants as one of the proposed properties L in the strengthening step, but we will define a new abstraction (a variant on data type reduction) that treats in a special way the auxiliary variables used to track message flows.

A. Abstraction

An abstraction is a procedure that transforms one protocol P to another protocol $\hat{P} = abstract(P)$, and transforms one property ϕ over states of P to another property $\hat{\phi} = abstract(\phi)$ over states of $\hat{P}$. Apart from being sound, we make the additional requirement that the abstraction focuses on two processes 1 and 2 and abstracts the rest away [1]. One such well-known abstraction is *data type reduction* [19] that reduces data types with large or unbounded ranges to small, finite ranges. Given a variable v with a large range, say $[1..L]$, it can be abstracted to a variable $\hat{v}$ with a small range $\{1, 2, o\}$ which retains two values, 1 and 2, and the rest of the values are lumped into an abstract value o.

Denoting the syntactic abstraction operation in the original CMP by A_{red}, the abstract protocol $A_{red}(P)$ is constructed via a data type reduction which retains a small number of processors, say processors 1 and 2, and replaces the remaining processors $3, .., N$ with a single, highly nondeterministic process called the $Other$ process. The abstraction process is a simple, syntactic procedure: any condition in the protocol code involving processors $3, .., N$ is replaced with *true*, *false* or a nondeterministic Boolean variable as appropriate to conservatively *over-approximate* it. Any assignment to the state variables of $[3..N]$ is deleted. The abstraction $A_{red}(\phi)$ of a property ϕ is defined similarly except that any condition involving processors $3..N$ in property ϕ are replaced with *true*, *false* or a nondeterministic Boolean variable as appropriate to conservatively *under-approximate* it.

This abstraction is best explained with an example. Consider the following ruleset from the Murphi description of German's protocol. This ruleset is just a collection of guarded commands indexed by a process id $i \in$ `NODE`, where `NODE` is the parameterized range $[1..N]$.

[1]Generalization to more than 2 processes is simple.

```
ruleset i : NODE; do
  rule "RecvGntS"
    Chan2[i].Cmd = GntS
    ==>
      begin BODY endrule;
endruleset;
```

We replace this ruleset with N independent rules and apply data type reduction to each rule independently. For processors 1 and 2, the rules remain unchanged (ignoring the effect of data type reduction on the body of the rule), but for processors $i > 2$ the abstracted rule becomes

```
rule "RecvGntS"
  true
  ==>
    begin BODY' endrule;
```

This is because the condition $Chan2[i].Cmd = GntS$ refers to the state variable of an abstracted process and it is conservatively over-approximated to *true*. Thus, after applying data type reduction to the ruleset we will end up with two rulesets: the abstract rule shown above and a ruleset identical to the original ruleset except that the quantifier i ranges over $[1..2]$. The example gives a flavor of the syntactic nature of data type reduction and it is clear that the abstract rules obtained data type reduction are sound abstractions.

Theorem 1: The abstraction A_{red} is sound for $\phi(1, 2)$ for every expression $\phi(i, j)$:

$$A_{red}(P) \models A_{red}(\phi(1, 2)) \;\; \Rightarrow \;\; P \models \phi(1, 2).$$

We refer the reader to Krstic [15] for a proof.

B. Strengthening

A strengthening is a procedure that transforms one protocol P to another protocol $P^{\#} = strengthen(P, \psi)$ by replacing each guarded command $\rho \rightarrow a$ of P with the guarded command $\rho \wedge \psi \rightarrow a$ whose guard has been strengthened by ψ. Given a property ϕ, a strengthening is said to be *sound for* ϕ if it satisfies the property

$$P^{\#} \models \phi \Rightarrow P \models \phi.$$

Returning to the abstraction of the *RecvGntS* ruleset, the abstract rule is clearly too abstract. The guard *true* does not constrain the $Other$ process in any way. This leads to spurious counterexamples, and to eliminate such counterexamples, CMP depends on user-provided noninterference lemmas. Suppose we have the following lemma that we think might be useful.

```
forall p : NODE; i: NODE do
  Chan2[i].Cmd = GntS ->
    i != p -> Cache[p].State != E
end
```

Strengthening the protocol with this lemma — applying it to the *RecvGntS* rule set — and abstracting, we get the following rule set for the concrete processors

```
ruleset i : [1..2]; do
  rule "RecvGntS"
    Chan2[i].Cmd = GntS &
    forall p: NODE do
       i != p -> Cache[p].State != E
    ==>
      begin BODY endrule;
endruleset;
```

and the following rule set for the abstracted process

```
rule "RecvGntS"
  forall p: NODE do
     Cache[p].State != E
  ==>
    begin BODY' endrule;
```

Note that the abstract rule now has a meaningful guard and thus the abstract rule is more refined than previously. This process is continued iteratively by adding more and more lemmas until either a real counterexample is found or all the lemmas and the property of interest are proved.

C. Correctness

The new insight in our analysis of CMP is a generalized understanding of when a strengthening can be declared sound. Looking back at the definition of CMP, in proving $\phi(1,2)$ we are assuming $\forall i,j.\phi(i,j)$ unlike earlier compositional reasoning principles, which would assume only $\phi(1,2)$. This is explained by a property we call *entailment*.

Lemma 2: Let R_i be the states of P reachable within i steps, and let $P^{\#} = strengthen(P,\psi)$. If

$$\forall i.(\forall s \in R_i.s \models \phi) \;\; \Rightarrow \;\; (\forall s \in R_i.s \models \psi)$$

then $P^{\#} \models \phi$ implies $P \models \phi$.

Proof: The proof is similar to the proof of the guard strengthening principle in [15]. Denote by $R_i^{\#}$ the set of all states reachable in $P^{\#}$ within i steps. We will prove by induction on i that $\forall s \in R_i.s \in R_i^{\#}$ and consequently $\forall s \in R_i.s \models \phi$.

For $i = 0$, if s is an initial state of P, then it is an initial state of $P^{\#}$ as well. So the base case for induction is true.

Assume we have proved the inductive hypothesis for $i = k$. That is, $\forall s \in R_k.s \models \phi$ and $\forall s \in R_k.s \in R_k^{\#}$. We will prove that $\forall s \in R_{k+1}.s \models \phi$ and consequently $\forall s \in R_k.s \in R_k^{\#}$.

Consider any state $s' \in R_{k+1}$ reachable from $s \in R_k$ via a rule $\rho \rightarrow a$. For the rule to fire we must have $s \models \rho$. By the inductive hypothesis, $s \in R_k^{\#}$ as well. Moreover, from $\forall s \in R_k.s \models \phi$ and the condition $\forall i.(\forall s \in R_i.s \models \phi) \Rightarrow (\forall s \in R_i.s \models \psi)$ we have $\forall s \in R_k.s \models \psi$. Consequently, we have $s \models \psi$. Putting all the facts together, we have s is reachable in $P^{\#}$ within k steps and the rule $\rho \wedge \psi \rightarrow a$ is enabled at s. Therefore, s' is reachable in $P^{\#}$ within $k+1$ steps. Since $P^{\#} \models \phi$ we immediately have $s' \models \phi$. ■

We use the phrase *entailment* to refer to the condition

$$\forall i.(\forall s \in R_i.s \models \phi) \Rightarrow (\forall s \in R_i.s \models \psi)$$

It is because of this notion of entailment that our lemma for compositional reasoning given above differs subtly from the compositional reasoning principles considered by McMillan [18], Abadi and Lamport [1], [2], Krstic [15], Bhattacharya et al. [6] and Chen et al. [8]. In our case, in proving a property ϕ not only can we assume ϕ, but also the metaconsequence ψ of ϕ [2]. Moreover, ψ does not have to be discharged explicitly. Including metaconsequences of assumptions allow us to exploit domain specific knowledge such as symmetry.

Theorem 2: CMP is sound for any symmetric protocol and any sound abstraction procedure.

Proof: Since the protocol is symmetric, Lemma 1 proves the entailment precondition

$$\forall i.(\forall s \in R_i.s \models I^{\#}(1,2)) \Rightarrow (\forall s \in R_i.s \models \forall j,k.I^{\#}(j,k))$$

of Lemma 2. Thus,

$$A_{red}(P^{\#}) \models A_{red}(I^{\#}(1,2)) \Rightarrow P^{\#} \models I^{\#}(1,2)$$

by the soundness of abstraction and

$$P^{\#} \models I^{\#}(1,2) \Rightarrow P \models I^{\#}(1,2)$$

by the soundness of strengthening (Lemma 2) and

$$P \models I^{\#}(1,2) \Rightarrow P \models \forall j,k.I^{\#}(j,k)$$

by symmetry. ■

The significance of this analysis is that it shows that CMP is sound for any sound abstraction procedure not just data type reduction. The earlier analysis of CMP in [9], [15] proved

$$A_{red}(P^{\#}) \models A_{red}(I^{\#}(1,2)) \Rightarrow P \models I^{\#}(1,2)$$

with a single, complex proof that depended heavily on symmetry and the use of data type reduction as the abstraction procedure. What has happened here is to realize the soundness of strengthening depends only on entailment (the hypothesis of Lemma 2) which happens to be satisfied by symmetric protocols. Realizing this, we can prove the soundness of strengthening independent of the specific abstraction procedure being used.

Remark 3: Note that, unlike the usual counterexample guided refinement approaches, CMP requires the abstract model be sound only for $I^{\#}(1,2)$ and not for the full property $\forall i,j.I^{\#}(i,j)$. This means the abstraction can record much less information and thus scale to larger examples.

V. Parameterized Verification using Flows

We now show how to use flow-derived lemmas with CMP. We begin with the augmented protocol P_a obtained by augmenting P with auxiliary variables as described in Section IV. We then run CMP, strengthening with the flow-derived lemmas, and abstracting with a new abstraction procedure A^a_{red} tailored to auxiliary variables.

We augment our language of expressions to include the flow-derived lemmas as follows. We define an *augmented*

[2]For ϕ to be a logical consequence $\phi \Rightarrow \psi$ should hold. It is clear the set of metaconsequences is richer than the set of logical consequences.

expression to be a Boolean combination with quantification over free indices of the original basic expressions together with

$$(fl, rl) \in Aux(i) \qquad \forall k.(sfl, \_) \notin Aux(k)$$

To ensure sound abstraction we will require that in any augmented expression $\phi(i, j)$, variables $Aux(l), l \notin \{i, j\}$ appear only as part of a condition of the form $\forall k.(sfl, \_) \notin Aux(k)$. We augment our language of assignments in the obvious way to include the assignments to auxiliary variables used in the update procedure in Figure 3.

We note that the proof of Theorem 2 depends only on the soundness of abstraction and on the protocol being described with guarded commands. The syntactic restrictions on expressions and assignments are needed only to prove that A_{red} is a sound abstraction. Thus, we can augment expressions and assignments, and prove than an augmented abstraction procedure A^a_{red} is sound, and conclude by Theorem 2 that running CMP with this augmented abstraction A^a_{red} is sound for augmented expressions.

The abstraction A^a_{red} leaves the sets $Aux(i)$ unchanged for $i = 1, 2$ and replaces $Aux(i), i \geq 3$ with a single multiset S_{aux}, but otherwise is identical to A_{red}.

For conditions appearing in the protocol code, the operation A^a_{red} abstracts expressions involving auxiliary variables as follows:

$$(fl, rl) \in Aux(i)$$

is unchanged for $i = 1, 2$ and is abstracted to

$$(fl, rl) \in S_{aux}$$

for $i \geq 3$, and

$$\forall k.(sfl, \_) \notin Aux(k)$$

is abstracted to

$$(sfl, \_) \notin Aux(1) \wedge (sfl, \_) \notin Aux(2) \wedge (sfl, \_) \notin S_{aux}.$$

For properties, the syntactic operation A^a_{red} treats the auxiliary variables in the same as well. Finally, A^a_{red} abstracts assignments involving auxiliary variables by leaving assignments to $Aux(i)$ unchanged for $i = 1, 2$ and modifying assignments to $Aux(i)$ for $i \geq 3$ to update S_{aux} instead.[3]

Theorem 3: The abstraction A^a_{red} is sound for $\phi(1, 2)$ for every augmented expression $\phi(i, j)$:

$$A^a_{red}(P_a) \models A^a_{red}(\phi(1, 2)) \quad \Rightarrow \quad P_a \models \phi(1, 2).$$

The proof is given in the full version of this paper [25].

This theorem says we can use flow-derived lemmas with CMP as follows. First, we augment the protocol P with auxiliary variables to obtain P_a. Then we run CMP strengthening with augmented expressions and abstracting with the augmented abstraction A^a_{red}. When choosing augmented expressions for the strengthening, however, one source of augmented expressions are the flow-derived lemmas. We note that even using all the flow-derived lemmas may not be enough to cause CMP to converge. We claim, however, that the flow information is powerful enough than running CMP with the flow-derived lemmas is likely to require fewer and simpler lemmas that running CMP without them, and our experiments in the next section support this intuition.

[3]In practice, we bound the size of S_{aux} by keeping track of whether an item appears $0, 1,$ or $2+$ times. The remove operator choses between the transitions from $2+$ to $2+$ or 1 non-deterministically.

VI. Experimental Results

We applied the ideas presented in this paper to verify the standard control and data properties of the German's protocol and Flash cache coherence protocol. The control property we are interested in verifying is

$$\forall i. \forall j. i \neq j \Rightarrow (state[i] = E \Rightarrow \neg(state[j] \in \{E, S\}))$$

In words, if process i is in an exclusive state then no other process can be in exclusive or shared state.

The data properties were

$$\begin{array}{l} \forall i : (state[i] = E \Rightarrow data[i] = currdata) \\ \wedge \\ (state[i] = S \Rightarrow (collecting \Rightarrow data[i] = prevdata) \wedge \\ (!collecting \Rightarrow data[i] = currdata)) \end{array}$$

and $!dirty \Rightarrow memdata = currdata$.

The variables $currdata$ and $collecting$ are auxiliary variables introduced to specify the data properties [9]. The first property essentially says that the data value at a cache matches the last written value. The second property says the data value at the directory matches the last written value. (These are data properties for Flash; German data property was similar.) Verification of data properties is harder than control property as it involves a model with bigger state space.

We built a tool in OCaml that takes the Murphi description of a cache protocol along with the associated flows and builds an abstract model with the auxiliary variables and the invariants from flows added. (This was built on top of a pre-existing program that parsed and applied data type reduction to Murphi code.)

To verify German's protocol we used the three flows described in Table 2. The abstract model created using these flows gives us a spurious counterexamples for both the control and data property. To eliminate these we added two lemmas similar to the ones in [9]. With these lemmas added the proofs go through in about 6 seconds. In Chou et al. [9], the work closest to ours, two lemmas were needed as well though their lemmas were a little bit more complicated than ours.

The real strength of our approach is seen in the verification of Flash protocol, which is much larger than German's protocol. For this protocol, we extracted the 6 flows in a couple of hours. To get the proofs for safety properties through we had to add two lemmas on top of the flow constraints. One lemma was the same as a lemma used in [9]. The quantifier free part of our second lemma, written in CNF, has 3 clauses. The rest

of the lemmas in [9] have 12 clauses all together.[4] The running time for our abstract model is just 120 seconds compared to the couple of hours taken by the abstract model in [9]. We ran our models and the Murphi model for Flash protocol used in [9] on a 3 GHz machine with 512 MB memory. The Murphi models used in our experiments are avaiable online [25].

We believe these experimental results are significant and point to the ability of flows to give us the precise constraints needed to reason about correctness of protocols. The hardest invariants usually characterize what messages could be in transit between two processors based on the information recorder in the processors' states. Flows give us such invariants automatically which prevent the *Other* process from sending messages out of turn. This accounted for most of the lemmas used in [9]. Lemmas which related the state of local cache process to the state of the directory could not be replaced using flow constraints.

VII. Conclusion

We have demonstrated that message flows are a readily available source of powerful invariants and how these invariants can replace complicated invariants used in a typical application of the CMP method. The pragmatic implications of the applications in this paper are very encouraging. Message flows — similar to message sequence charts — are a natural, local, linear way for us to think about system behavior in place of global, monolithic, system-wide invariants. It seems likely that many systems — not just cache coherence protocols — can be understood in terms in terms of message flows. In fact, this work opens up many paths to investigate in the future.

Message flows are interesting because they are so common in industrial design documents, but message flows are not the only depictions of dependencies that appear in these documents. Timing diagrams, pipeline diagrams, block diagrams, and simple controllers with small transition systems are other common ways of describing aspects of system behavior, and dependencies described there should be just as useful as message flows when describing system constraints.

Communication events — sending and receiving messages — have a natural partial order, but so do many other system events. Reading and writing shared memory locations, acquiring and releasing a lock, starting, committing, and aborting a transaction, these are all important system events that can contribute to a flow. More abstractly, any system in which there is a clearly defined interaction among system agents should be open to a flow-based analysis and flow-inspired invariants. We have focused on message-passing systems, but it would be very interesting to extend the ideas here to concurrent systems with shared memory, transactional memory, and other systems where the notion of event is more abstract but still well-defined.

Finally, since our formulation of the CMP method separates the strengthening and abstraction phases of CMP so cleanly, we plan to investigate combining CMP with other forms of strengthening and abstraction in the future. For example, we could consider replacing data type abstraction with a form of abstraction that is less coarse and reduce the need for user supplied lemmas even further.

[4]The proof in [9] is infact incomplete; so more lemmas will be needed in their case.

References

[1] M. Abadi and L. Lamport. Composing specifications. In *ACM Transactions on Programming Languages and Systems*, 1993.

[2] M. Abadi and L. Lamport. Conjoining specifications. In *ACM Transactions on Programming Languages and Systems*, 1993.

[3] P. Abdullah, A. Buojjani, B. Jonsson, and M. Nilsson. Handling Global Conditions in Parameterized Verification. In *Proc. CAV*, 1999.

[4] T. Arons, A. Pnueli, S. Ruah, and L. Zuck. Parameterized Verification with Automatically Computed Inductive Assertions. In *Proc. CAV*, 2001.

[5] K. Baukus, Y. Lakhnech, and K. Stahl. Parameterized Verification of Cache Coherence Protocols: Safety and Liveness. In *Proc. VMCAI*, 2002.

[6] R. Bhattacharya, S. M. German, and G. Gopalakrishnan. Exploiting symmetry and transactions for partial order reduction of rule based specifications. In *SPIN*, 2006.

[7] J. Bingham. Automatic invariant generation for parameterized verification. In *Submitted to FMCAD*, 2008.

[8] X. Chen, Y. Yang, M. DeLisi, G. Gopalakrishnan, and C.-T. Chou. Hierarchical Cache Coherence Protocol Verification One Level at a Time Through Assume Guarantee. In *HLDVT*, 2007.

[9] C.-T. Chou, P. K. Mannava, and S. Park. A simple method for parameterized verification of cache coherence protocols. In *Proc. FMCAD)*, 2004.

[10] E. Clarke, M. Talupur, and H. Veith. Environment Abstraction for Parmeterized Verification. In *Proc. VMCAI*, 2006.

[11] E. Clarke, M. Talupur, and H. Veith. Proving Ptolemy Right: Environment Abstraction Principle for Parameterized Verification. In *Proc. TACAS*, 2008.

[12] A. Cohen and K. Namjoshi. Local Proofs for Global Safety Properties. In *Proc. CAV*, 2007.

[13] S. Das, D. L. Dill, and S. Park. Experience with Predicate Abstraction. In *CAV*, 1999.

[14] O. Grinchtein, M. Leucker, and N. Piterman. Inferring Network Invariants Automatically. In *Proc. IJCAR*, 2006.

[15] S. Krstic. Parameterized system verification with guard strengthening and parameter abstraction. In *Automated Verification of Infinite State Systems*, 2005.

[16] S. K. Lahiri and R. Bryant. Constructing Quantified Invariants. In *Proc. TACAS*, 2004.

[17] Y. Lv, H. Liu, and H. Pan. Computing invariants for parameter abstraction. In *MEMOCODE*, 2007.

[18] K. L. McMillan. Verification of an implementation of tomasulo's algorithm by compositional model checking. In *Proc. Computer Aided Verification*, 1998.

[19] K. L. McMillan. Verification of infinite state systems by compositional model checking. In *CHARME*, 1999.

[20] K. L. McMillan. Parameterized verification of the FLASH cache coherence protocol by compositional model checking. In *Proc. Conf. on Correct Hardware Design and Verification Methods (CHARME '01)*, volume 2144 of *LNCS*, pages 179–195. Springer, 2001.

[21] K. L. McMillan. Quantified invariants using interpolants. In *TACAS*, 2008.

[22] S. Park and D. L. Dill. Verification of flash cache coherence protocol by aggregation of distributed transactions. In *SPAA '96: Proceedings of the eighth annual ACM symposium on Parallel algorithms and architectures*, pages 288–296. ACM Press, 1996.

[23] A. Pnueli, J. Xu, and L. Zuck. Liveness with $(0, 1, \infty)$ Counter Abstraction. In *Proc. CAV*, 2002.

[24] M. Talupur, S. Krstic, J. O'Leary, and M. R. Tuttle. Parameteric Verification of Industrial Strength Cache Coherence Protocols. In *Proc. Workshop on Design of Correct Circuits (DCC)*, 2008.

[25] M. Talupur and M. R. Tuttle. Going with the Flow: Parameterized Verification using Message Flows. `www.markrtuttle.com/fmcad08`.

Automatic Non-interference Lemmas for Parameterized Model Checking

Jesse Bingham

Abstract—*Parameterized model checking* refers to any method that extends traditional, finite-state model checking to handle systems with an arbitrary number of processes. One popular approach to this problem uses abstraction and so-called *guard strengthening*. Here a small number of processes remain intact, while the rest are abstracted away. This initially causes counter-examples, but the user can write *non-interference lemmas*, which eliminate spurious behavior by the abstracted processes. The technique is sound in that if the user writes a lemma that is not invariant, the proof will never succeed. Though the non-interference lemmas are typically much simpler than writing a full inductive invariant, there is still a non-trivial amount of human insight needed to analysis counter-examples and concoct the lemmas. In our work, we show how the process of inferring appropriate non-interference lemmas can be automated. Our approach is based on a very general theory that simply assumes a *Galois connection* between the concrete and abstract systems. Effectively, we start with the non-interference conjecture *False*, and iteratively weaken it until it is provable using the Galois connection. This produces the *strongest* non-interference lemma provable in the Galois connection. Hence, if the approach fails to prove the property, then no human lemma would help, since it is the strongest possible lemma. We instantiate this theory to a class of symmetric parameterized systems, and show how BDDs can be used to perform all involved computations. We also show how BDD-blow up that can arise when concretizing can be mitigated by using a sound over-approximation. We successfully applied the resulting tool to three parameterized verification benchmarks: the GERMAN protocol with data path, the GERMAN2004 protocol, and the FLASH protocol. To our knowledge, this is the first time automatic parameterized model checking has been done on GERMAN2004.

1 Introduction

Consider the problem of verifying a safety property of a system S with an arbitrary (but fixed) number n processes. One compelling method to solve this problem involves model checking a finite state system A_0 involving a small number m (typically 2 or 3) of unmodified *concrete* processes, along with an abstracted process [21], [6], [15], [18], [19], [27]. The abstracted process over-approximates the behaviors of an arbitrary number of processes. Because of this approximation, the abstracted process usually interferes with the concrete processes in some undesirable way, not possible in the unabstracted system. Hence the invariant property typically fails in A_0. However, the method allows the human to write a so-called *non-interference lemma* ψ_1 which is a conjectured invariant of S manually culled from counterexample (cex) analysis. One uses ψ_1 to restrict the abstracted process in A_0 via a technique called *guard strengthening*, which yields a new system A_1, which again may or may not satisfy the safety property. If not, the user iterates the process, writing non-interference lemmas $\psi_1, \psi_2, \ldots$ and producing a succession of abstract systems $A_0, A_1, A_2, \ldots$ that are model checked. Assuming a system A_k is found to satisfy the property, one must still confirm that $\bigwedge_i \psi_i$ is an invariant of S. Surprisingly, the theory allows one to decide this by checking that $\bigwedge_i \psi_i$ is an invariant of A_k; this is counter-intuitive because A_k is in a sense constructed under the assumption that $\bigwedge_i \psi_i$ is an invariant. Assuming that this check also passes, we have proved that the system S is safe for arbitrary n.

In this paper we automate this process. We compute the *strongest possible* non-interference lemma that is provable using the underlying abstraction. We then simply check if this invariant implies the safety property. If this approach fails to prove the property, i.e. the computed invariant does not imply the property, then one must necessarily use a better abstraction, for example by adding more concrete processes or axillary variables. In particular since there is no stronger invariant that can be proved invariant using the approach, no manually selected lemmas would have helped out.

Roughly, our approach involves starting with the strong non-interference conjecture *False*. We strengthen the concrete system with this conjecture, abstract, and compute the reachable states in the abstract domain. We then concretize the reachable states, which will yield some weaker conjecture. We iterate, using the successively weaker conjectures found by concretizing the reachable abstract states as lemmas for strengthening the concrete system. Hence our approach involves an inner loop that computes forward reachability in abstract systems, and an outer loop that concretizes the reachable abstract states, strengthens the concrete system with the result, and re-abstracts. We terminate when the outer loop reaches a fix-point in the sense that no new reachable abstract states are found.

Two primary contributions of this paper are a general theory that supports this technique, and an instantiation of the theory for a class of symmetric parameterized protocols, similar to that of Krstić's [15]. This class admits

• *Jesse Bingham is with Intel Corporation, Hillsboro, OR. jesse.d.bingham@intel.com.*

978-1-4244-2735-2/08/$25.00 ©2008 IEEE

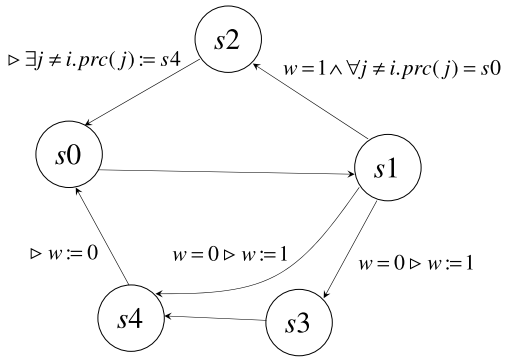


Fig. 1. A process in our example parameterized system

a *small model theorem* not unlike that of Pnueli et al. [23], which allows us to perform all necessary computations using BDDs and symbolic model checking [4]. For complex examples, BDD blow up can become an issue. To combat this well-known problem, we propose a variant on our approach wherein only the abstracted transitions (rather than *all* transitions) are strengthened with the non-interference conjectures and a (user-selected) subset of the protocol variables are omitted from the lemmas.

We've implemented an instance of approach using the *forte* [26] formal verification system. The tool takes a protocol described in forte's functional language reFLect, and automatically performs parameterized verification. We have successfully verified properties of three caching protocols: the original GERMAN protocol, a significantly different version GERMAN2004 [19], [13], and the Stanford FLASH protocol [16].

The paper is organized as follows. Sect. 2 provides a motivating example, which contrasts the classical guard strengthening approach with our approach. Related work is discussed in Sect. 3. The general theory behind our approach is presented in Sect. 4, which again contrasts the classical approach with our own. This theory is instantiated for a class of symmetric parameterized protocols in Sect. 5. Case studies and further discussion are presented in Sects. 6 and 7, respectively. We note that proofs and some technical details have been relegated to an extended version [1].

2 Motivating Example

An example parameterized system is given in Fig. 1. The system is the interleaved composition of an arbitrary number n of identical processes.The system has two state variables: $prc : \mathbb{N}_n \rightarrow \{s0, s1, s2, s3, s4\}$, where $\mathbb{N}_n = \{0, \ldots, n-1\}$ are the process IDs (PID), and w, which is a Boolean. So $prc(i)$ is the "local" state of process i, and w is a "shared" global variable. Initially, all processes are in state $s0$ and w is *false*. The transitions of an arbitrary process i are shown in Fig. 1; each transition is labeled with a guarded command $G \triangleright C$, which means that the transition can occur whenever G is true and C is the resulting update. If G is omitted, then the transition is always enabled; if $\triangleright\ C$ is omitted, then there is no update other than process i's local state change. Note that some guards and updates mention other processes $j \neq i$. We wish to establish that the following is an invariant:

$$\forall i \neq j. \neg(prc(i) = s4 \wedge prc(j) = s4) \quad (1)$$

The states of the abstract systems are same as the parameterized system, except that there are only two processes 0 and 1. Hence the domain of prc is restricted to the two PIDs $\mathbb{N}_2$. We call these abstract states *views* and write them as a triple $(w, prc(0), prc(1))$. We now outline how both the classic approach and our own approach tackles parameterized verification of this example.

The Classic Approach starts by model checking an initial abstract system, created by conservatively abstracting away all processes except 0 and 1. This yields the following cex (i.e. a view sequence)

$$(0,s0,s0), (0,s0,s1), (0,s4,s1), (1,s4,s4)$$

The human realizes that the cex stems from the transition from the 2nd to 3rd view. This transition occurs because an abstracted process transitions from $s2$ to $s0$, with $j = 0$ in the update (see Fig. 1). By inspection, the human concludes that $s2$ is unreachable, and writes the lemma

$$\psi_1 \ = \ \forall i. prc(i) \neq s2$$

Strengthening the the concrete system with ψ_1 effectively removes the transition from $s2$ to $s0$, which is justified since if $s2$ is unreachable, we don't need any transitions out of it. Abstracting and then model checking, we find that ψ_1 holds, but we get another cex of (1):

$$(0,s0,s0), (0,s1,s0), (0,s1,s1),$$
$$(1,s4,s1), (0,s4,s1), (1,s4,s4)$$

Here an abstracted process transitions from $s4$ to $s0$, which "interferes" by clearing w, allowing process 1 to get to $s4$. The user writes another lemma

$$\psi_2 \ = \ \forall i \neq j. w \Rightarrow \neg(prc(i) = s4 \wedge prc(j) = s4)$$

Strengthening with $\psi_1 \wedge \psi_2$ allows us to label the $s4$-$s0$ transition with $\psi_2 \triangleright w := 0$. This disallows an abstracted process from transitioning from $s4$ to $s0$ whenever a concrete process is in $s4$. Model checking for the third time, we find all three of (1), ψ_1, and ψ_2 are invariants. The theory allows us to conclude that (1), as well as ψ_1 and ψ_2, is invariant in the original parameterized system.

Our Approach employs a concretization function which takes a set of views V to a universally quantified formula $\gamma(V)$ that is used to strengthen the concrete system. Intuitively, $\gamma(V)$ characterizes the concrete states such that for all distinct PIDs i and j, the view $(w, prc(i), prc(j))$ is in V. We first model check the abstract system obtained by strengthening the concrete system with the "non-interference conjecture" $\gamma(\emptyset)$, which equals *False*. Strengthening with $\gamma(\emptyset)$ disables all transitions, and hence the set of reachable states is just the

singleton of the initial view $R_1 = \{(0,s0,s0)\}$. We now take $\gamma(R_1)$ as our new, weakened conjecture. Strengthening with $\gamma(R_1)$, abstracting, and computing the reachable states yields $R_2 = R_1 \cup \{(0,s1,s0), (0,s0,s1), (0,s1,s1)\}$. We iterate, now using $\gamma(R_2)$ to strengthen. The reachable views are now

$$R_3 = R_2 \cup \left\{ \begin{array}{l} (1,s3,s0), (1,s4,s0), (1,s3,s1), (1,s4,s1), \\ (1,s0,s3), (1,s0,s4), (1,s1,s3), (1,s1,s4) \end{array} \right\}$$

Strengthening with $\gamma(R_3)$ yields reachable views R_4 where $R_4 = R_3$. Since we've reached a fix-point, our theory tells us that $\gamma(R_3)$ is an invariant of the concrete system. Next we simply check that $\gamma(R_3)$ implies our property, which it does. We note that $\gamma(R_3)$ *properly* implies $\psi_1 \wedge \psi_2$; this is no coincidence as our algorithm computes the strongest lemma provable in the given abstraction/concretization.

3 Related Work

There have been many approaches to parameterized model checking and the more general paradigm of infinite state model checking, for examples well-structured transition systems [11], [12], regular model checking [3], the work of Emerson and Kahlon [9], and that of Pong and Dubois [24]. To our knowledge, none of these have been applied to a system with the complexity of the FLASH protocol. The rest of this section focuses on approaches that are closer to our own.

The idea of using non-interference lemmas for parameterized model checking is attributed to McMillan [20], [21], who added support for this style of reasoning into Cadence SMV. The idea was later used along with Murphi by Chou et al. [6] and formalized further by Krstić [15] and Li [18]. Both papers [21], [6] verify the FLASH coherency protocol [16] using several user-supplied lemmas and the addition of history variables. Similar types of reasoning have been applied by Chen et al. to verify hierarchical protocols [5]. Finally, a simultaneous FMCAD paper [27] simplifies the process of concocting non-interference lemmas by incorporating protocol *flows*. We view this work as complementary to ours, since if our aggressive automatic approach blows-up, then it is desirable to fall back onto an intuitive manual lemma framework.

Closely related work was done recently by Lv et al. [19]. Like us, they employ BDD-based methods to generate non-interference lemmas. Essentially they use the heuristic for generating candidate "invisible invariants" [23] to generate possible non-interference lemmas. Our work also draws from the invisible invariants work [23] . In particular, our BDD-based concretization computation is inspired by their techniques. Furthermore, we employ a similar small model theorem, though we use slightly tighter syntactic restrictions in order to make the small smaller. We compare against these papers [19], [23] further in Sect. 7.

Other related work includes that of Pandav et al. [22], who have proposed a set of heuristics to aid in constructing invariants for caching protocols. Also, *environment abstraction* [7] uses what amounts to existentially quantified predicate abstraction on the environment of a (concrete) process. Finally, a similar approach has been explored for software verification [14]; however our system class is incomparable with theirs (e.g. we allow universal quantification, they allow spawning of processes, etc.).

4 Theoretical Foundation

The method of guard strengthening for parameterized verification has been formalized by others [6], [15], [18]. Our re-formalization here is motivated by three factors. First, our formalization is tailored to show how automation is achieved. Second, ours is more general than parameterized systems; it in fact applies to a finite abstraction of any finite or infinite state transition system that is a Galois connection. Finally, we believe this formalization in terms of Galois connections is more succinct than previous explanations.

Given a set S called a *state space*, a *transition system* over S is a triple $\mathcal{T} = (S, I, T)$ where $I \subseteq S$ are called the *initial states* and $T \subseteq S \times S$ is called the *transition relation*. A state $s \in S$ is said to be *reachable* if there exists states $s_0, s_1, \ldots, s_\ell$ such that $s_0 \in I$, $s_\ell = s$, and $(s_i, s_{i+1}) \in T$ for all $0 \leq i < \ell$. The set of all reachable states is denoted $\mathsf{Reach}(\mathcal{T})$. A set $\psi \subseteq S$ is call an *invariant* if $\mathsf{Reach}(\mathcal{T}) \subseteq \psi$. A transition relation $T \subseteq S \times S$ induces a *post image operator* $\mathsf{post}[T] : 2^S \to 2^S$ defined by $\mathsf{post}[T](\phi) = \{s' \mid \exists s \in \phi.(s, s') \in T\}$. We will identify subsets ϕ of S with logic formulas that characterize them, and call such formulas *state formulas*.

Our goal is to verify that a state formula p is an invariant of $\mathcal{T} = (\mathcal{C}, I, T)$, which we call the *concrete system*. The concrete state space $\mathcal{C}$ consists of the type-consistent assignments to some set of state variables. In the case of parameterized systems, these types depend on a natural parameter, but this is not relevant for the theoretical development of this section.

One can conjoin a state formula ψ with the transition relation T to strengthen it. Formally, let us define

$$\mathcal{T} \diamond \psi = (\mathcal{C}, I, T \cap (\psi \times \mathcal{C}))$$

We call $\mathcal{T} \diamond \psi$ the *strengthening* of $\mathcal{T}$ by ψ. Intuitively, $\mathcal{T} \diamond \psi$ is the same as $\mathcal{T}$, except that any state $c \in \mathcal{C}$ in which ψ does not hold has no transitions. The following theorem is well-known (see e.g. Krstić [15]) and underlies the approach of using strengthening:

Theorem 1: ψ is invariant in $\mathcal{T}$ if and only if ψ is invariant in $\mathcal{T} \diamond \psi$.

The concrete system is verified by constructing a succession of abstract systems that over-approximate the concrete system. The abstract systems are over a finite set $\mathcal{A}$ called the *abstract domain*, which, along with a partial order $\sqsubseteq$, is a lattice. Each element of $\mathcal{A}$ is an object that represents a certain set of concrete states, and $\sqsubseteq$

corresponds to inclusion of the represented sets. Often $\mathcal{A}$ will be a powerset and $\sqsubseteq$ is $\subseteq$. We assume a *Galois connection* (α, γ) between $2^{\mathcal{C}}$ and $\mathcal{A}$. This means that $\alpha : 2^{\mathcal{C}} \to \mathcal{A}$ and $\gamma : \mathcal{A} \to 2^{\mathcal{C}}$ are monotonic functions such that for all $\phi \subseteq \mathcal{C}$ and $A \in \mathcal{A}$, we have that $\alpha(\phi) \sqsubseteq A$ if and only if $\phi \subseteq \gamma(A)$. We add the additional requirement that $\alpha \circ \gamma$ is the identity.The functions α and γ are respectively referred to as *abstraction* and *concretization*. Intuitively, α defines the mapping that associates elements of the abstract domain with sets of concrete states. We say that a concrete state formula ψ is *representable* if $\psi = \gamma(\alpha(\psi))$. The representable formulas are those that can be abstracted without loosing information.

Since the objects in $\mathcal{A}$ represent sets of concrete states, it is natural to approximate the concrete post image operator by a function over $\mathcal{A}$ called an *abstract post-image*. For $U \subseteq \mathcal{C} \times \mathcal{C}$ there is the notion of a *best* abstract post-image $\mathsf{bap}[U] : \mathcal{A} \to \mathcal{A}$ defined by

$$\mathsf{bap}[U] \;=\; \alpha \circ \mathsf{post}[U] \circ \gamma \tag{2}$$

Intuitively, $\mathsf{bap}[U]$ first concretizes, then takes the post-image of U in the concrete domain, and then abstracts the result. We say that a function $\mathsf{post} : \mathcal{A} \to \mathcal{A}$ *abstracts* the concrete transition system $(\mathcal{C}, I, U)$ if for all $A \in \mathcal{A}$ we have that $\mathsf{bap}[U](A) \sqsubseteq \mathsf{post}(A)$. We extend Reach to apply to abstract post-images: if $A_0 \in \mathcal{A}$ and $\mathsf{post} : \mathcal{A} \to \mathcal{A}$ is a monotonic function, then $\mathsf{Reach}(A_0, \mathsf{post}) = \mathsf{post}^k(A_0)$, where k is minimal such that $\mathsf{post}^{k+1}(A_0) \sqsubseteq \mathsf{post}^k(A_0)$ (note that k must exists since $\mathcal{A}$ is finite). We say that $A \in \mathcal{A}$ is an invariant of (A_0, post) if $\mathsf{Reach}(A_0, \mathsf{post}) \sqsubseteq A$.

Both the classic approach and our own approach employ a mechanism for performing strengthening and abstraction of the concrete transitions. We will denote this mechanism by Ω, which is not made explicit in the classic papers. We do so here since it is a key ingredient in our method, and serves to compare the two approaches. Given $\psi \subseteq \mathcal{C}$, Ω first strengthens the concrete system $\mathcal{T}$ with ψ and then builds an abstract post-image for the strengthened system $\mathcal{T} \diamond \psi$. Formally, Ω takes a representable state formula ψ and returns a function $\Omega(\psi) : \mathcal{A} \to \mathcal{A}$ such that $\Omega(\psi)$ abstracts $\mathcal{T} \diamond \psi$. We now describe how the classic approach and our approach manifest in this formal setting.

The Classic Approach. Some papers that use the classic approach compute their Ω automatically [21], [19] while others involve the human manually selecting guarded commands to strengthen [6], [27]. Also, the user may strengthen with a formula weaker than ψ [6], [15], [27]; this is permissible with our Ω.

Let us say that a representable concrete state formula p is a *provable invariant* if there exists a representable concrete state formula ψ (i.e. the non-interference lemma) such that both $\mathsf{Reach}(\alpha(I), \Omega(\psi)) \sqsubseteq \alpha(\psi)$ and $\mathsf{Reach}(\alpha(I), \Omega(\psi)) \sqsubseteq \alpha(p)$. It is straightforward to show that a provable invariant deserves this title:

Theorem 2: If p is a provable invariant, then p is an invariant.

Of course the converse does not hold; the Galois connection or Ω is necessarily too weak to prove many (semantically true) invariants. Theorem 2 succinctly justifies the classical approach, although it hides the fact that the non-interference lemma ψ used to prove p is typically constructed iteratively via counter-example analysis as a conjunction $\bigwedge_i \psi_i$.

Our Approach makes a minor additional monotonicity requirement on Ω. We call Ω *monotonic* if for all representable concrete state formula ψ_1 and ψ_2 such that $\psi_1 \Rightarrow \psi_2$, and for all $A \in \mathcal{A}$, we have that $\Omega(\psi_1)(A) \sqsubseteq \Omega(\psi_2)(A)$. This effectively disallows Ω from giving us a better abstract post-image from a weaker ψ.

Our algorithm can now be described. Let us assume a monotonic Ω, and define the function $\mathsf{Reach} \circ \Omega : \mathcal{A} \to \mathcal{A}$ to be the composition of $\mathsf{Reach}(\alpha(I), \cdot)$ with Ω. I.e. for any $A \in \mathcal{A}$ we define $\mathsf{Reach} \circ \Omega(A) \;=\; \mathsf{Reach}(\alpha(I), \Omega(\gamma(A)))$ So $\mathsf{Reach} \circ \Omega$ maps an element of the abstract domain to the best abstract approximation (afforded by Ω, α, and γ) of the reachable states of the concrete system strengthened with $\gamma(A)$. We note that $\mathsf{Reach} \circ \Omega$ is monotonic and acts on a finite domain, hence it has a well defined *least fixed point* (LFP). This LFP is computed as $(\mathsf{Reach} \circ \Omega)^k(\bot)$, where $k \geq 0$ is minimal such that $(\mathsf{Reach} \circ \Omega)^{k+1}(\bot) \sqsubseteq (\mathsf{Reach} \circ \Omega)^k(\bot)$ and $\bot$ is the bottom element of $\mathcal{A}$, which is $\emptyset$ if $\mathcal{A}$ is a powerset.

Theorem 3: Let p be a representable concrete state formula, let Lfp be the LFP of $\mathsf{Reach} \circ \Omega$. and assume that Ω is monotonic, Then p is a provable invariant if and only if $\mathsf{Lfp} \sqsubseteq \alpha(p)$.

Theorem 3 is the crux of our approach. We simply compute Lfp, and test if $\mathsf{Lfp} \sqsubseteq \alpha(p)$. If so, we can conclude that p is an invariant of the concrete system $\mathcal{T}$. If $\mathsf{Lfp} \sqsubseteq \alpha(p)$ does not hold, then either p is not an invariant, or Ω, α, and/or γ are too weak to prove that p is an invariant.

Automation aside, our approach is in a sense dual to the classical approach. The latter initially doesn't strengthen *at all*, i.e. the strengthened is done with *True*. Then the user repeatedly conjuncts constraints, i.e. strengthens the formula, until a lemma is found that proves the property p. On the other hand, we start with the conjecture *False*, and iteratively weaken it until we have a provable lemma. Since we are left with the strongest lemma (afforded by Ω, α, and γ), we know that no human intervention (respecting Ω) could help.

We can eliminate the possibility that Ω is too weak by using a best Ω relative to (α, γ). Say that Ω is *best* if it always yields the best abstract post image. Formally, this means that for all $A \in \mathcal{A}$, $\mathsf{post}[\Omega(A)] = \mathsf{bap}[T \wedge \gamma(A)]$. If Ω is best, it follows that Lfp is the strongest non-interference lemma that is provable under Galois connection (α, γ). In our instantiation of this theory in the next section, we will first define Ω to be best. However, we have found that our BDD-based methods can potentially blow-up when using a best Ω. In this case we propose weaker variants that are not best, but are still good enough to automatically prove the desired invariant.

A simple optimization over the algorithm described here is as follows. Rather than computing the LFP of $\mathsf{Reach} \circ \Omega$, it is more efficient to compute the LFP of the sequence $A_0, A_1, \ldots$ defined by $A_0 = \alpha(I)$ and $A_{i+1} = \mathsf{Reach}(A_i, \Omega(\gamma(A_i)))$. Hence rather than starting each abstract Reach computation with the $\alpha(I)$, we start with the abstract object A_i computed in the previous iteration. This converges on the same fix point Lfp.

variable set	type in protocol states $\Sigma(n)$	type in view
W	$\mathbb{B}$	$\mathbb{B}$
X	$\mathbb{N}_n \to \mathbb{B}$	$\mathbb{N}_m \to \mathbb{B}$
Y	$\mathbb{N}_n$	$\mathbb{N}_m^{oth}$
Z	$\mathbb{N}_n \to \mathbb{N}_n$	$\mathbb{N}_m \to \mathbb{N}_m^{oth}$

TABLE 1
The four types allows for protocol variables, and how they are typed in views.

5 Parameterized Protocols

In this section, we instantiate the general theory of Sect. 4 for a class of symmetric parameterized protocols. We restrict the state variables to be of four basic types, which are the same used by others [6], [15], [27], [23]. The types are Booleans, functions from PIDs to Booleans, PIDs, and functions from PIDs to PIDs; clearly multiple Booleans can be used to to encode any finite enumerated type. The PIDs are $\mathbb{N}_n$ for some arbitrary natural n and we denote a protocol by $\mathcal{P}(n)$.

Our protocol class is defined by a syntax (similar to that of Krstić [15]) that enforces symmetry and allows for certain small model theorems. The syntax is expressive enough to admit well-known cache protocol case studies, such as the three protocols we consider in Sect. 6. The class accommodates so-called *conjunctive guard* and *broadcast* communication primitives, which renders even invariant verification undecidable [10]. Thus using an incomplete verification approach such as abstraction is necessary.

The abstract domain is (a certain subset of) the powerset of a set of objects called *views* [6]. A view is essentially a state of $\mathcal{P}(m)$, where m is a small constant that the user selects,[1] except that PID-valued variables can take on a special value called *oth*, which represents an unknown PID in $\{m, m+1, \ldots, n\}$. The set of views, and hence its powerset, are both finite. The Galois connection we employ maps a set of protocol states to its set of constituent views, as well as all permutations of such. Similarly, concretization of a given set of views returns (a formula characterizing) the set of states having its views contained in the given set. A small model theorem asserts that there exists a small constant N, such that the views of the strengthened transition relation of $\mathcal{P}(N) \diamond \gamma(V)$ (where V is a set of views) capture the views of the transition relation of $\mathcal{P}(n) \diamond \gamma(V)$ for any $n \geq N$. The key corollary of the small model theorem is that we can use $\mathcal{P}(N) \diamond \gamma(V)$ to compute a Ω for the Galois connection. This is done using BDD-operations on the transition relation of $\mathcal{P}(N)$. We note that theorem assumes that $n \geq N$, hence our parameterized model checking only establishes safety of $\mathcal{P}(n)$ for all $n \geq N$, rather than all $n \geq 0$. We regard this as a negligible limitation.

We now elaborate on these ideas; first describing the concrete syntax in Sect. 5.1, then the abstract domain of sets of views in Sect. 5.2, then the Galois connection in Sect. 5.3, and finally how we compute our Ω in Sect. 5.4. To reduce the flood of notation, we simply denote the Galois connection of this section by (α, γ).

1. typically at least the number of universally quantified PID variables in the property of interest.

5.1 Concrete System

A parameterized protocol $\mathcal{P}(n)$ has variables from four sets W, X, Y, and Z. We call the variables in $W \cup X \cup Y \cup Z$ the *protocol variables*. The variables are typed according to the middle column of Table 1, where $\mathbb{B}$ is the set of the two Boolean constants. The states $\Sigma(n)$ of $\mathcal{P}(n)$ are the set of type-consistent assignments to $W \cup X \cup Y \cup Z$. For $s \in \Sigma(n)$ and protocol variable ν, we write $s[\nu]$ for the value assigned to ν by s. For $\nu \in X \cup Z$, we write $s[\nu(i)]$ for the ith entry in the array $s[\nu]$.

The transition relation τ of $\mathcal{P}(n)$, and strengthenings thereof, are expressed as formulas we call *protocol transition formula*. A rigorous definition of protocol formula is given in [1]; here we give an intuitive description. The important point is that the syntax provides a balance between the two conflicting requirements of expressiveness and having a small enough *small model theorem*, which will be stated in Sect. 5.4. A protocol transition formula is a restricted first order formula over atomic propositions called atoms. These *atoms* perform various queries on the protocol variables, such as indexing into an array, comparison between PID variables, and evaluating a Boolean. Atoms also allow for priming of protocol variables to refer to the next state. We employ three sets of PID variables (disjoint from Y) to quantify over: $E = \{e_0, \ldots, e_{\ell-1}\}$, $U = \{u_0, \ldots, u_{k-1}\}$, and $Q = \{q_0, \ldots, q_{m-1}\}$. We will write $\vec{e}$ for the tuple $e_0, \ldots, e_{\ell-1}$ and similarly for $\vec{u}$ and $\vec{q}$. Protocol transition formulas are of the form

$$(\exists \vec{e}.\forall \vec{u}.\phi_0) \wedge (\forall \vec{q}.\phi_1) \tag{3}$$

Where ϕ_0 and ϕ_1 are quantifier-free Boolean combinations of atoms (see [1] for details) . Here ϕ_0 characterizes the unstrengthened transition relation; the existential quantification allows for "Murphi"-like rulesets [8] constructs over PIDs. Hence $\ell = |E|$ is the maximum level of nesting of PID rulesets. The second conjunct $\forall \vec{q}.\phi_1$ is a strengthening formula. We will see in Sect. 5.3 that our concretization function produces formula of this form; we note that m here is the number of "concrete processes" used in views. The initial states of $\mathcal{P}(n)$ will also be required to be protocol transition formula, except with no primed variables.

5.2 Abstract Domain: Symmetric Sets of Views

A *view* is an assignment to the protocol variables, but with typing according to the rightmost column in Table 1. Here $\mathbb{N}_m^{oth} = \{0, \ldots, m-1\} \cup \{oth\}$, where oth is a fresh symbol that will represent an arbitrary element of $\mathbb{N}_n \setminus \mathbb{N}_m$. Our abstract domain is simply the set of *symmetric* sets of views, along with the inclusion ordering $\subseteq$. This notion of symmetric is best defined using the Galois connection, which is the focus of the next subsection. Hence we postpone the definition momentarily.

5.3 Galois Connection

Let τ be a protocol transition formula (3) and let $s_0, s_1 \in \Sigma(n)$. We write $(s_0, s_1) \vDash \tau$ to indicate that τ is satisfied when unprimed and primed protocol variables are valuated by s_0 and s_1, respectively. Call an injection $\pi : \mathbb{N}_m \to \mathbb{N}_n$ a *view map*. For any $i \in \mathbb{N}_n$ and view map π, define $\hat{\pi}(i)$ to be $\pi^{-1}(i)$ if i is in the image of π, otherwise $\hat{\pi}(i) = oth$. Now, given view map π, the π-*view* of $s \in \Sigma(n)$ is the view $v = \mathsf{View}_\pi(s)$ defined by

- for all $w \in W$ define $v[w] = s[w]$
- for all $x \in X$ and $i \in \mathbb{N}_m$ define $v[x(i)] = s[x(\pi(i))]$
- for all $y \in Y$ define $v[y] = \hat{\pi}(s[y])$
- for all $z \in Z$ and $i \in \mathbb{N}_m$ define $v[z(i)] = \hat{\pi}(s[z(\pi(i))])$

So, essentially, v permutes PIDs so that PIDs in the image of π become the first m PIDs $\mathbb{N}_m$, and PIDs in $\mathbb{N}_n \setminus \mathbb{N}_m$ are replaced with oth.

We now introduce the abstraction function α that takes subsets of $\Sigma(n)$ to sets of views and a corresponding concretization function γ. For $\varphi \subseteq \Sigma(n)$, define

$$\alpha(\varphi) \;=\; \{\mathsf{View}_\pi(s) \mid s \in \varphi \text{ and } \pi \text{ is a view map}\}$$

Thus $\alpha(\varphi)$ is the set of all views that are the π-views of some state in φ for some π. Computing α using BDDs is straightforward, especially when φ is symmetrical. In this case, $\alpha(\varphi)$ can be computed by $\mathsf{View}_{\mathsf{id}}(\varphi)$, where id is the identity function on $\mathbb{N}_m$. In our algorithm, α is only directly applied to such sets, i.e. the initial states and the property being checked.

Concretization is expressed by

$$\gamma(V) \;=\; \forall \vec{q}.\mathsf{distinct}(\vec{q}) \Rightarrow \bigvee_{v \in V} \Pi(v) \qquad (4)$$

Here $\mathsf{distinct}(\vec{q})$ means that $q_i \neq q_j$ for all $0 \leq i < j < m$. For a view v, the formula $\Pi(v)$ is a quantifier-free Boolean combination of atoms that precisely characterizes the concrete states having π-view being v, where π takes $i \in \mathbb{N}_m$ to (the valuation of) q_i. Thus $\gamma(V)$ says that for any m distinct PIDs $\{q_0, \ldots, q_{m-1}\}$ one selects, the corresponding view must be an element of V. Defining $\Pi(v)$ is straightforward but tedious, hence we relegate it to the extended paper [1] . We note that (4) is given for exposition purposes only and is never used directly during computation.

In Sect. 5.2 we mentioned that our abstract domain is a subset of the powerset of views, i.e. those that are symmetric. We are now equipped to define this notion. A set of views V is said to be *symmetric* if $\alpha(\gamma(V)) = V$.

Theorem 4: (α, γ) is a Galois connection between $2^{\Sigma(n)}$ and symmetric sets of views.

We conclude this section by noting that our (α, γ) is quite similar to the Galois connection used for universally quantified predicate abstraction [17].

5.4 Computing Ω with BDDs

This section explains how we compute the best abstract post-image $\mathsf{bap}[\tau]$ for a protocol transition formula τ. In turn, this allows us to compute the best Ω. In summary, our protocol transition formula admit a *small model theorem*. Recall that ℓ is the number of existentially quantified variables in τ (3), and m is the number of "concrete" processes in the views. Let us write τ_N for the set of all $(s, s') \in \Sigma(N) \times \Sigma(N)$ such that $(s, s') \vDash \tau$.

Theorem 5 (Small Model Theorem): Let τ be a protocol transition formula and let $N = m + \ell + 1$. Then $\mathsf{bap}[\tau] = \mathsf{post}[T]$, where $T = \{(\mathsf{View}_{\mathsf{id}}(s), \mathsf{View}_{\mathsf{id}}(s')) \mid (s, s') \in \tau_N\}$

Theorem 5 allows us to compute a BDD representation of $\mathsf{bap}[\tau]$ as follows. First, we build a BDD for the transition relation τ_N. Next we massage this BDD into a BDD for T by applying $\mathsf{View}_{\mathsf{id}}$ to both the present and next state variables; this involves two steps. First, all X-type and Z-type variables (both primed and unprimed) with indices in $\{m, \ldots, N-1\}$ are existentially abstracted out of τ_N. Second, oth is represented by *any* value $\geq m$, thus we canonicallize so that whenever a PID variable is $\geq m$, we allow it to be any such value; this is accomplished through straightforward BDD operations. The result represents the T in the statement of Theorem 5. Finally, $\mathsf{post}[T]$ is computed with standard BDD techniques [4].

Applying our approach of Sect. 4 requires us to map an arbitrary symmetric set of views V to a best abstract post-image $\Omega(\gamma(V))$. This post-image must abstract the formula

$$\tau \;=\; (\exists \vec{e}.\forall \vec{u}.\phi_0) \wedge \gamma(V)$$

Note that τ is a valid protocol transition formula (3) since $\gamma(V)$ is of the form $\forall \vec{q}.\phi_1$. Let $\gamma_N(V)$ be the BDD for the elements of $\Sigma(N)$ that satisfy $\gamma(V)$. Once we have $\gamma_N(V)$, computing τ_N is straightforward, and thus so too is $\mathsf{bap}[\tau]$ via Theorem 5.

The BDD $\gamma_N(V)$ is computed as follows. Recall that $\gamma(V)$ is defined by (4). Since $\gamma_N(V)$ restricts us to PIDs in $\mathbb{N}_N$, we can replace the universal quantifier and the antecedent $distinct(\vec{q})$ with a conjunction like so:

$$\gamma_N(V) \;=\; \bigwedge_{\vec{p}} \left(\bigvee_{v \in V} \Pi(v)[\vec{p}/\vec{q}] \right) \qquad (5)$$

where $\vec{p}$ ranges over all m-tuples of *distinct* elements of $\mathbb{N}_N$. For each such m-tuple $\vec{p}$, we compute the BDD for $\bigvee_{v \in V} \Pi(v)[\vec{p}/\vec{q}]$ as a post-image $\mathsf{post}[\Theta(\vec{p})](V)$, using a

technique due to Pnueli et al. [23]. Here $\Theta(\vec{p})$ is defined

$$\begin{aligned}\Theta(\vec{p}) \;=\; & \bigwedge_{w\in W} w' \Leftrightarrow w \\ \wedge\; & \bigwedge_{x\in X}\bigwedge_{0\le i<m} x'(p_i) \Leftrightarrow x(i) \\ \wedge\; & \bigwedge_{y\in Y}\bigwedge_{0\le i<m} y' = p_i \Leftrightarrow y = i \\ \wedge\; & \bigwedge_{z\in Z}\bigwedge_{0\le i,j<m} z'(p_i) = p_j \Leftrightarrow z(i) = j\end{aligned} \tag{6}$$

In effect, $\Theta(\vec{p})$ constrains the π-view of the primed state to some view in the unprimed state, where $\pi(i) = p_i$ for each $i \in \mathbb{N}_m$. By taking the post-image of $\Theta(\vec{p})$ with respect to the view set V, we get the set of all $s \in \Sigma(N)$ such that $\mathsf{View}_\pi(s) \in V$. Finally, by conjuncting over all $\vec{p}$ as in (5), we obtain $\gamma_N(V)$.

We conclude this section by noting that when BDD blow-up becomes an issue, a remedy is to soundly weaken Ω, which can be done in two orthogonal ways. First, one can remove various conjuncts from (6). This amounts to identifying that certain variables are not needed in the non-interference lemma. Second, when the protocol transition formula is a disjunction of guarded commands (GC), one can apply different subsets of the conjuncts in (5) to each command. This allows one to leave unabstracted GCs unstrengthened, while applying strengthening to abstracted GCs. This can be done purely algorithmically. Previous approaches [19], [15], [6], [27] allow the human to use such heuristics on a lemma-by-lemma basis; in contrast we write a simple weakening scheme once and our algorithm then proceeds to compute the strongest lemma provable under the weaker Ω. Weakening Ω tends to decrease the number of iterations of the outer loop while increasing the number of iterations of the inner loop of our algorithm.

6 Case Studies

We applied our technique to three coherency protocols: GERMAN, GERMAN2004, and FLASH. The GERMAN protocol was originally poised as a challenge problem for parameterized model checking by German [13], it is rather simple and has been verified many times in the literature [19], [6], [2], [23], [22]. GERMAN2004 is a significant modification of GERMAN and is considerably more complex; it was first formally verified by Lv et al. [19], though they required some manually added history variables. Finally, FLASH [16] is quite well-known. The control property for FLASH was first automatically verified by Lv et al. [19]. As pointed out by Chou et al. [6]: *"FLASH is a good test for any proposed method of parameterized verification; if the method works on FLASH, then there is a good chance that it will also work on many real-world cache coherence protocols."*

We implemented our approach using the *forte* formal verification system [26]. We hand translated the protocols from their Murphi descriptions into *reFLect*, forte's functional language. They were written as protocol transition formula, which required some minor refactoring in some cases[2]. A tarball including our implementation and the case study models is available [1].

example	time	mem	outter	inner
GERMAN	0.3	1.1	19	2,2,. . .,2
GERMAN2004	15.0	1.6	8	55, 74, 57, 30, 5, 6, 2, 1
FLASH	57.0	3.2	7	75, 51, 43, 39, 36, 9, 1

TABLE 2
Case studies results

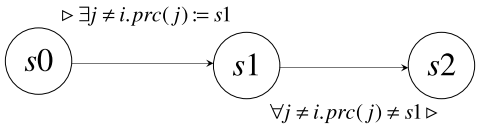


Fig. 2. Example for which our algorithm fails.

We successfully verified the control and data properties for GERMAN, and the control properties for GERMAN2004 and FLASH. For GERMAN, we used a best Ω and BDD-blow-up was a non-issue For the other two protocols, the ability to weaken Ω was very useful to curb BDD blow-up issues, as was dynamic variable reordering [25]. Weakening Ω involves identifying some variables to omit from (6); finding a good set was simply a matter of trial and error and does not involve any insights about the protocol.[3] We note that for these experiments we used a slightly different small model theorem than Theorem 5 that effectively reduces N by 1 to $m + \ell$, at the expense that $\mathsf{post}[T]$ is not quite best. This alternate small model theorem is stated and proved in the extended version [1].

Our results are given in Table 2. All runs were done on a 64-bit linux machine. The columns "time" and "mem" give the runtime in minutes and the memory usage in GB, respectively. The "outer" column gives the number of iterations of the outer loop of our algorithm, i.e. the number of iterations to find the fix-point of $\mathsf{Reach} \circ \Omega$. The "inner" column gives the number of abstract post-image iterations to compute Reach, for each iteration of the outer loop. Since the best Ω was used for GERMAN, the inner loop always iterates just twice.

7 Discussion

7.1 Example of Insufficiency

Fig. 2 gives a very simple parameterized protocol for which our approach of Sect. 5 fails, in the sense that the property holds but our algorithm is unable to prove it. Fig. 2 uses the same notation as Sect. 2. Initially, all processes are in $s0$, and we wish to prove the (clearly

2. For example, protocol formula don't allow for a Y variable to be used to index into an array; however by using an existential variable to index into the array, and adding the constraint that the existential equals the Y variable, the same effect can be achieved.

3. For example, for GERMAN2004 all Z variables where omitted.

true) invariant $\forall i.prc(i) \neq s2$. Our algorithm fails at this verification; this stems from the fact the existentially quantified lemma $\forall i.\exists j \neq i.prc(i) = s1 \Rightarrow prc(j) = s1$ is needed. Our automatically generated lemmas are restricted to be of the form (4), which only allows for universal quantification. We note that adding history variables can help in this situation; if done correctly our algorithm can easily verify this example.

7.2 Comparison with Invisible Invariants

The most salient difference between our approach and that of invisible invariants [23] and Lv et al. [19] is that the lemma we generate is guaranteed to be an invariant, while the other methods heuristically produce a *possible* invariant. This possible invariant is constructed by generalizing the reachable states of a small instance of the concrete system. This leads to the question: are there systems for which invisible invariants fail, while our approach succeeds?

The answer is positive. Consider a parameterized system where each process has a single bit, initially all *False*. There is a global 15-bit counter initially 0. Any process that has its bit clear can set it, and increment the counter (unless it is saturated). We want to prove that if any two processes have their bits set, then the counter is at least 2. The invisible invariants method would compute the reachable states of the concrete system with 2 processes. Generalizing this space does not yield an invariant, However, our algorithm produces an invariant that is strong enough to prove the property. Of course, if a large enough concrete system is used to generate the candidate invisible invariant, their approach will succeed. However, this would require model checking the instance of size 2^{15}, which has over 2^{15} state variables.

It is interesting to note that one can extend the invisible invariant method by feeding back non-inductive formula (i.e. failed invariant candidates) as initial conditions, and then iterating the method until an inductive invariant is found.[4] The resulting "fix-point" invisible invariant method would be very similar to our approach, with the exception that the invisible invariant small model theorem is not as small as our own.

REFERENCES

[1] J. Bingham. Appendices and code for this paper. `http://www.cs.ubc.ca/~jbingham/fmcad08.html`.

[2] J. Bingham and A. J. Hu. Empirically efficient verification for a class of infinite-state systems. In *Tools and Algorithms for the Construction and Analysis of Systems*, 2005.

[3] A. Bouajjani, B. Jonsson, M. Nilsson, and T. Touili. Regular model checking. In *Computer Aided Verification*, 2000.

[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*, 98(2), 1992.

[5] X. Chen, Y. Yang, G. Gopalakrishnan, and C.-T. Chou. Reducing verification complexity of a multicore coherence protocol using assume/guarantee. In *FMCAD '06: Proceedings of the Formal Methods in Computer Aided Design*, pages 81–88, 2006.

4. The author would like to thank an anonymous reviewer for suggesting this extension and its connection to this work.

[6] C. Chou, P. K. Mannava, and S. Park. A simple method for parameterized verification of cache coherence protocols. In *5th International Conference on Formal Methods in Computer-Aided Design (FMCAD)*, 2004.

[7] E. Clarke, M. Talupur, and H. Veith. Environment abstraction for parmeterized verification. In *7th International Conference on Verification, Model Checking and Abstract Interpretation*, 2006.

[8] 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*, pages 522–525, 1992.

[9] E. A. Emerson and V. Kahlon. Exact and efficient verification of parameterized cache coherence protocols. In *Correct Hardware Design and Verification Methods (CHARME)*, 2003.

[10] E. A. Emerson and V. Kahlon. Model checking guarded protocols. In *18th IEEE Symposium on Logic in Computer Science (LICS)*, pages 361–370, June 2003.

[11] J. Esparza, A. Finkel, and R. Mayr. On the verification of broadcast protocols. In *14th IEEE Symposium on Logic in Computer Science (LICS)*, pages 352–359, 1999.

[12] A. Finkel and P. Schnoebelen. Well structured transition systems everywhere! *Theoretical Computer Science*, 256(1-2):63–92, 2001.

[13] S. German. Personal correspondence. 2008.

[14] T. A. Henzinger, R. Jhala, and R. Majumdar. Race checking by context inference. In *Programming Language Design and Implementation (PLDI)*, pages 1–13, 2004.

[15] S. Krstić. Parameterized system verification with guard strengthening and parameter abstraction. In *Automated Verification of Infinite-State Systems*, 2005.

[16] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy. The stanford FLASH multiprocessor. In *21st Annual International Symposium on Computer Architecture (ISCA)*, pages 302–313, 1994.

[17] S. K. Lahiri and R. E. Bryant. Constructing quantified invariants via predicate abstraction. In *5th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI)*, pages 267–281, 2004.

[18] Y. Li. Mechanized proofs for the parameter abstraction and guard strengthening principle in parameterized verification of cache coherence protocols. In *SAC '07: Proceedings of the 2007 ACM symposium on Applied computing*, pages 1534–1535, 2007.

[19] 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*, pages 29–38, 2007.

[20] K. L. McMillan. Verification of infinite state systems by compositional model checking. In *Correct Hardware Design and Verification Methods (CHARME)*, pages 219–234, 1999.

[21] K. L. McMillan. Parameterized verification of the FLASH cache coherence protocol by compositional model checking. In *Correct Hardware Design and Verification Methods (CHARME)*, pages 179–195, 2001.

[22] S. Pandav, K. Slind, and G. Gopalakrishnan. Counterexample guided invariant discovery for parameterized cache coherence verification. In *CHARME*, pages 317–331, 2005.

[23] A. Pnueli, S. Ruah, and L. Zuck. Automatic deductive verification with invisible invariants. In *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems*, pages 82–97, 2001.

[24] F. Pong and M. Dubois. Formal automatic verification of cache coherence in multiprocessors with relaxed memory models. *IEEE Transactions on Parallel and Distributed Systems*, 11(9):989–1006, September 2000.

[25] R. Rudell. Dynamic variable ordering for ordered binary decision diagrams. In *Proceedings of the International conference on Computer-aided design*, pages 42–47, 1993.

[26] C.-J. H. Seger, R. B. Jones, J. W. O'Leary, T. Melham, M. D. Aagaard, C. Barrett, and D. Syme. An industrially effective environment for formal hardware verification. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 24(9):1381–1405, September 2005.

[27] M. Talupur and M. R. Tuttle. Going with the flow: Parameterized verification using message flows. In *Formal Methods in Computer-Aided Design (FMCAD)*, 2008.

Model Checking Nash Equilibria in MAD Distributed Systems

Federico Mari, Igor Melatti, Ivano Salvo, Enrico Tronci
Dep. of Computer Science
University of Rome "La Sapienza"
Via Salaria 113, 00198 Roma, Italy
Email: {mari,melatti,salvo,tronci}@di.uniroma1.it

Lorenzo Alvisi, Allen Clement, Harry Li
Dep. of Computer Science
University of Texas at Austin
1 University Station C0500, Austin, Texas, USA
Email: {lorenzo,aclement,harry}@cs.utexas.edu

***Abstract*—We present a symbolic model checking algorithm for verification of Nash equilibria in finite state mechanisms modeling *Multiple Administrative Domains* (MAD) distributed systems.**

Given a finite state mechanism, a *proposed protocol* for each agent and an *indifference threshold* for rewards, our model checker returns *PASS* if the proposed protocol is a Nash equilibrium (up to the given indifference threshold) for the given mechanism, *FAIL* otherwise.

We implemented our model checking algorithm inside the NuSMV model checker and present experimental results showing its effectiveness for moderate size mechanisms.

I. Introduction

Cooperative services are increasingly popular distributed systems in which nodes (agents) belong to *Multiple Administrative Domains* (MAD). Thus in a MAD distributed system each node owns its resources and there is no central authority owning all system nodes. Examples of MAD distributed systems include Internet routing [25], [49], wireless mesh routing [40], file distribution [16], archival storage [41], cooperative backup [6], [17], [37].

In traditional distributed systems, nodes may deviate from their specifications (*Byzantine nodes*) because of bugs, hardware failures, faulty configurations, or even malicious attacks. In MAD systems, nodes may also deviate because their administrators are *rational*, i.e. selfishly intent on maximizing their own benefits from participating in the system (*selfish nodes*). For example, selfish nodes may change arbitrarily their protocol if that is at their advantage.

Cooperative file distribution (e.g. see [16]) is a typical example of the above scenario. Every peer will be happy to download file chunks from other peers. However, in order to save bandwidth, a *selfish* peer may modify its protocol parameters to disallow upload of its file chunks.

In this paper we present an automatic verification algorithm for MAD distributed systems. That is, given a protocol P for a MAD system and a property φ for P we want to automatically verify if φ holds for P.

Note that in a MAD system *any* node may behave selfishly. This rules out the classical approach (e.g. see [48]) of modeling nodes deviating from the given protocol as *Byzantine* [34]. In fact, doing so would leave us with a system in which all nodes are Byzantine. Very few interesting protocols (if any) work correctly under such conditions. Thus, in order to verify MAD systems, we first need a model for them in which protocol correctness can be formally stated and hopefully proved. This issue has been studied in [1], [15] where the *BAR* model has been introduced.

In BAR, a node is either *Byzantine*, *Altruistic*, or *Rational*. Byzantine nodes, as usual, can deviate from their specification in any way for any reason. Altruistic nodes follow their specification faithfully, without considering their self-interest. Rational nodes deviate selfishly from a given protocol if doing so improves their own utility. In the BAR framework correctness of a protocol with respect to a given property is stated as BAR tolerance. Namely, a protocol is *BAR tolerant* if it guarantees the desired property despite the presence of Byzantine and rational players.

Several BAR tolerant protocols have since been proposed [1], [36] to implement cooperative services for p2p backup and live data streaming. Taking into account how hard it is to formally prove correctness for classical distributed protocols it is not surprising that formally proving that a given protocol is BAR tolerant is indeed quite a challenge (e.g. see [15]).

This motivates investigating if the model checking techniques devised for classical distributed protocols can also be used in our framework. To this end we note that in order to show that a protocol is BAR tolerant, it is sufficient to show that it satisfies the given property when all rational nodes follow the protocol *exactly* and then to show that all rational nodes do, in fact, follow the protocol *exactly*.

If all rational nodes follow the given protocol *exactly* we are left with a system with only Byzantine and altruistic nodes. Well known model checking techniques (e.g. see [14] for a survey) are available to verify that such systems satisfy a given property despite the presence of a limited number of Byzantine nodes. It suffices, as usual, to model Byzantine nodes with nondeterministic automata.

Unfortunately, to the best of our knowledge, no model checking algorithm or tool is available to address the second BAR tolerance requirement, namely proving that all rational nodes do follow the given protocol *exactly*.

To fill this gap in this paper we present a symbolic model checking algorithm to automatically verify that it is in the best interest of each rational agent to follow *exactly* the given protocol. This is usually accomplished by proving that no

978-1-4244-2735-2/08/$25.00 ©2008 IEEE

rational agent has an incentive in deviating from the proposed protocol. This, in turn, is done by proving that the proposed protocol is a *Nash equilibrium* (e.g. see [25], [11]).

A. Our contribution

First of all we need a formal definition of *mechanism* suitable for model checking purposes and yet general enough to allow modeling of interesting systems. Accordingly in Sect. III we present a definition of *Finite State Mechanism* suitable for modeling of finite state BAR systems as well as for developing effective verification algorithms for them. We model each agent with a *Finite State Machine* defining its admissible behavior, that is, using the BAR terminology, its *Byzantine* behavior. Each agent action yields a real valued *reward* which depends both on the system state and on agents' actions. The *proposed protocol* constrains which actions should be taken in each state. This protocol defines the behavior of *altruistic* agents.

The second obstruction to overcome is the fact that we need to handle infinite games since nodes are running, as usual, nonterminating protocols. As a result, we need to *rank* infinite sequences of agents' actions (*strategies*). This is done in Sects. V, VI by using a *discount factor* (as usual in game theory [26]) to decrease relevance of rewards *too far* in the future. In Prop. 1 we give a dynamic programming algorithm to effectively compute the value of a finite strategy in our setting.

To complete our framework we need a notion of equilibrium that can be effectively computed by only looking at finite strategies and that accommodates Byzantine agents. Accordingly, in Sect. VII we give a definition of mechanism Nash equilibrium accounting for the presence of up to f Byzantine players (along the lines of [21]) and for agent *tolerance* to small ($\varepsilon > 0$) differences in rewards (along the classical lines of, e.g. [23], [26]). This leads us to the definition of ε-f-Nash equilibrium in Def. 4.

Sect. VIII gives our main theorem on which correctness of our verification algorithm rests. Theor. 2 shows that ε-f-Nash equilibria for finite state BAR systems can be automatically verified within any desired precision $\delta > 0$ by just looking at *long enough* finite sequences of actions.

Sect. IX presents our symbolic model checking algorithm for Nash equilibria. Our algorithm inputs are: a finite state mechanism $\mathcal{M}$, a proposed protocol for $\mathcal{M}$, the tolerance $\varepsilon > 0$ for agents to differences in rewards, the maximum number f of allowed Byzantine agents, our desired precision $\delta > 0$. Our algorithm returns *PASS* if the proposed protocol is indeed a $(\varepsilon + \delta)$-f-Nash equilibrium for $\mathcal{M}$, *FAIL* otherwise.

We implemented (Sect. X) our algorithm on top of NuSMV [46] using ADDs (*Arithmetic Decision Diagrams*) [18] to manipulate real valued rewards.

Finally in Sect. XI we present experimental results showing effectiveness of our approach on moderate size mechanisms. For example, within 22 hours using 5 GB of RAM we can verify mechanisms whose global present state representation requires 32 bits. The corresponding normal form games for such mechanisms would have more than 10^{22} entries.

B. Related works

Design of mechanisms for rational agents has been widely studied (e.g. [49], [45], [13]). Design methods for BAR protocols have been investigated in [1], [36], [15], [21]. We differ from such works since our focus here is on automatic verification of Nash equilibria for finite state BAR systems rather than on design principles for them.

Algorithms to search for pure, mixed (exact or approximate) Nash equilibria in games have been widely studied (e.g. see [24], [19]). We differ from such works in two ways. First, all such line of research addresses explicitly presented games (*normal form games*) whereas we are studying implicitly presented games (namely, mechanisms defined using a programming language). Thus, (because of *state explosion*) the explicit representation of a mechanism has size exponential in the size of our input. This is much the same as the relationship between reachability algorithms for directed graphs and reachability algorithms for finite state concurrent programs. As a result the algorithms and tools (e.g. [27]) for explicit games cannot be used in our context. Second, we are addressing a verification problem, thus the candidate equilibrium is an input for us whereas it is an output for the above mentioned works.

The relationship between model checking and game theory has been widely studied in many settings.

Game theoretic approaches to model checking for the verification of concurrent systems have been investigated, for example, in [32], [35], [30], [39], [51]. An example of game based model checker capable of CTL, modal μ-calculus and specification patterns is [29].

Model checking techniques have also been applied to the verification of knowledge and belief logics in game theoretic settings. Examples are in [7], [8], [12]. An example of a model checker for the logic of knowledge is MCK [28].

Applications of model checking techniques to game theory have also been investigated. For example, model checking techniques have been widely applied to the verification of games stemming from the modeling of multi-agent systems. See for example [31], [33], [38], [20]. An example of model checker for multi-agent programs is CASP [9]. An example of model checking based analysis of probabilistic games is in [5].

Note that the above papers focus on verification of temporal-like (e.g. temporal, belief, knowledge) properties of concurrent systems or of games whereas here we focus on checking Nash equilibria (of BAR protocols).

Synthesis of winning strategies for the verification game leads to automatic synthesis of correct-by-construction systems (typically controllers). This has been widely investigated in many settings. Examples are in [22], [47], [4], [2], [50], [52], [53], [3]. Note that the above papers focus on automatic synthesis (of systems or of strategies) whereas our focus here is on checking Nash equilibria (of BAR protocols).

Summing up, to the best of our knowledge, no model checking algorithm for the automatic verification of Nash equilibria of finite state mechanisms modeling BAR systems has been previously proposed.

II. Basic Notations

We denote an n-tuple of objects (of any kind) in boldface, e.g. $\boldsymbol{x}$. Unless otherwise stated we denote with x_i the i-th element of the n-tuple $\boldsymbol{x}$, $\boldsymbol{x_{-i}}$ the $(n-1)$-tuple $\langle x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n\rangle$, and with $\langle \boldsymbol{x_{-i}}, x\rangle$ the n-tuple $\langle x_1, \ldots, x_{i-1}, x, x_{i+1}, \ldots, x_n\rangle$.

We denote with $\mathbb{B}$ the set $\{0,1\}$ of boolean values (0 for *false* and 1 for *true*). We denote with $[n]$ the set $\{1,\ldots,n\}$. The set of subsets of X with cardinality at most k will be denoted by $\mathcal{P}_k(X)$.

III. Finite State Mechanisms

In this section we give the definition of *Finite State Mechanism* by suitably extending the usual definition of the synchronous parallel of finite state transition systems. This guarantees that all mechanisms consisting of finite state protocols can be modeled in our framework.

Definition 1 (Mechanism Skeleton): An n player (agent) *mechanism skeleton* $\mathcal{U}$ is a tuple $\langle \boldsymbol{S}, \boldsymbol{I}, \boldsymbol{A}, \boldsymbol{B}, \boldsymbol{h}, \boldsymbol{\beta}\rangle$ whose elements are defined as follows.

$\boldsymbol{S} = \langle S_1, \ldots, S_n\rangle$ is an n-tuple of nonempty sets (of *local states*). The *state space* of $\mathcal{U}$ is the set (of *global states*) $S = \prod_{i=1}^{n} S_i$.

$\boldsymbol{I} = \langle I_1, \ldots, I_n\rangle$ is an n-tuple of nonempty sets (of *local initial states*). The set of *global initial states* is $I = \prod_{i=1}^{n} I_i$.

$\boldsymbol{A} = \langle A_1, \ldots, A_n\rangle$ is an n-tuple of nonempty sets (of *local actions*). The set of *global actions* (i.e. n-tuples of local actions) is $A = \prod_{i=1}^{n} A_i$. The set of *i-opponents actions* is $A_{-i} = \prod_{j=1, j\neq i}^{n} A_i$.

$\boldsymbol{B} = \langle B_1, \ldots, B_n\rangle$ is an n-tuple of functions s.t., for each $i \in [n]$, $B_i : S \times A_i \times S_i \to \mathbb{B}$. Function B_i models the *transition relation* of agent i, i.e. $B_i(\boldsymbol{s}, a, s')$ is true iff agent i can move from (global) state $\boldsymbol{s}$ to (local) state s' via action a. We require B_i to be *serial* (i.e. $\forall \boldsymbol{s} \in S\ \exists a \in A_i\ \exists s' \in S_i$ s.t. $B_i(\boldsymbol{s}, a, s')$ holds) and *deterministic* (i.e. $B_i(\boldsymbol{s}, a, s') \wedge B_i(\boldsymbol{s}, a, s'')$ implies $s' = s''$). We write $B_i(\boldsymbol{s}, a)$ for $\exists s'\ B_i(\boldsymbol{s}, a, s')$. That is, $B_i(\boldsymbol{s}, a)$ holds iff action a is allowed in state $\boldsymbol{s}$ for agent i. For each agent $i \in [n]$, function B_i models the *underlying behavior* of agent i. That is, the set of all possible choices of *rational* player i. As a result, B_i defines the transition relation for the *Byzantine* behavior for agent i.

$\boldsymbol{h} = \langle h_1, \ldots, h_n\rangle$ is an n-tuple of functions s.t., for each player $i \in [n]$, $h_i : S \times A \to \mathbb{R}$. Function h_i models the *payoff (reward) function* of player i. Note that $\boldsymbol{h}$ may be seen as a function $\boldsymbol{h} : S \times A \to \mathbb{R}^n$ s.t. $\boldsymbol{h}(\boldsymbol{s}, \boldsymbol{a}) = (h_1(\boldsymbol{s}, \boldsymbol{a}), \ldots, h_n(\boldsymbol{s}, \boldsymbol{a}))$ for all global states $\boldsymbol{s} \in S$ and global actions $\boldsymbol{a} \in A$.

$\boldsymbol{\beta} = \langle \beta_1, \ldots, \beta_n\rangle$ is an n-tuple of *discounts*, that is of real values such that for each $i \in [n]$, $\beta_i \in (0,1)$.

Definition 2 (Mechanism): An n player *mechanism* $\mathcal{M}$ is a pair $(\mathcal{U}, \boldsymbol{T})$ where: $\mathcal{U} = \langle \boldsymbol{S}, \boldsymbol{I}, \boldsymbol{A}, \boldsymbol{B}, \boldsymbol{h}, \boldsymbol{\beta}\rangle$ is a mechanism skeleton and $\boldsymbol{T} = \langle T_1, \ldots, T_n\rangle$ is an n-tuple of functions s.t., for each $i \in [n]$, $T_i : S \times A_i \to \mathbb{B}$. We require T_i to satisfy the following properties: 1) $T_i(\boldsymbol{s}, a)$ implies $B_i(\boldsymbol{s}, a)$; 2) (*nonblocking*) for each state $\boldsymbol{s} \in \boldsymbol{S}$ there exists an action $a \in A_i$ s.t. $T_i(\boldsymbol{s}, a)$ holds. Function T_i models the *proposed protocol* for agent i, that is its *obedient* (or *altruistic*, following [1], [36]) behavior. More specifically, if agent i is *altruistic* then its transition relation is $B_i(\boldsymbol{s}, a, s') \wedge T_i(\boldsymbol{s}, a)$.

Often we denote an n player mechanisms $\mathcal{M}$ with the tuple $\langle \boldsymbol{S}, \boldsymbol{I}, \boldsymbol{A}, \boldsymbol{T}, \boldsymbol{B}, \boldsymbol{h}, \boldsymbol{\beta}\rangle$. Furthermore we may also call mechanism a mechanism skeleton. The context will always make clear the intended meaning.

Remark 1 (Finite State Agents): In order to develop our model checking algorithm we model each agent as a *Finite State Machine* (FSM). This limits agent knowledge about the past. In fact, the system *state* represents the system past history. Since our systems are nonterminating ones, histories (and thus agent knowledge) are in general unbounded. As for verification of security protocols (e.g. see [43], [44]) it is the modeler responsibility to develop a suitable finite state approximation of knowledge.

Definition 3: Let $\mathcal{M} = \langle \boldsymbol{S}, \boldsymbol{I}, \boldsymbol{A}, \boldsymbol{T}, \boldsymbol{B}, \boldsymbol{h}, \boldsymbol{\beta}\rangle$, be an n player mechanism and $Z \subseteq [n]$. Let $BT : \mathcal{P}([n]) \times \boldsymbol{S} \times \boldsymbol{A} \times \boldsymbol{S} \to \mathbb{B}$ be such that $BT(Z, \boldsymbol{s}, \boldsymbol{a}, \boldsymbol{s'}) = \bigwedge_{i=1}^{n} BT_i(Z, \boldsymbol{s}, a_i, s'_i)$, where

$$BT_i(Z, \boldsymbol{s}, a_i, s'_i) = \begin{cases} B_i(\boldsymbol{s}, a_i, s'_i) & \text{if } i \in Z \\ B_i(\boldsymbol{s}, a_i, s'_i) \wedge T_i(\boldsymbol{s}, a_i) & \text{otherwise.} \end{cases}$$

BT models the transition relation of mechanism $\mathcal{M}$, when the set of Byzantine players is Z and all agents not in Z are altruistic.

IV. An Example of Mechanism

In order to clarify our definitions we give an example of a simple mechanism. Consider the situation in which a set of agents cooperate to accomplish a certain job. The job, in turn, consists of n tasks. Each task is assigned to at least one agent which may carry out the assigned task or may deviate by not doing any work. Carrying out the assigned task entails a cost (negative reward) for the agent. On the other hand, if all tasks forming the job are completed (and thus the job itself is completed) all agents that have worked to a task get a reward greater than the cost incurred to carry out the assigned task. If the job is not completed no agent gets anything. The mechanism skeleton (Def. 1) is defined as follows.

All agents have the same discount, say $\beta = 0.5$, and the same underlying (Byzantine) behavior, defined by the automaton B_i in Fig. 1.

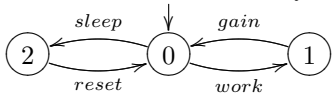


Fig. 1. Underlying behavior B_i for agent i.

From Fig. 1 we have: $S_i = \{0,1,2\}$, $I_i = \{0\}$, $A_i = \{\text{reset}, \text{sleep}, \text{work}, \text{gain}\}$. Let $\boldsymbol{S} = \langle S_1, \ldots, S_n\rangle$, $\boldsymbol{s} = \langle s_1, \ldots, s_n\rangle \in S$ and $\boldsymbol{a} = \langle a_1, \ldots, a_n\rangle \in A$.

The mechanism skeleton is $\mathcal{U} = \langle \boldsymbol{S}, \boldsymbol{I}, \boldsymbol{A}, \boldsymbol{B}, \boldsymbol{h}, \boldsymbol{\beta}\rangle$ where the payoff function h_i for agent i is defined as follows:

$$h_i(\boldsymbol{s}, \langle \boldsymbol{a}_{-i}, \text{work}\rangle) = -1$$
$$h_i(\boldsymbol{s}, \langle \boldsymbol{a}_{-i}, \text{sleep}\rangle) = h_i(\boldsymbol{s}, \langle \boldsymbol{a}_{-i}, \text{reset}\rangle) = 0$$
$$h_i(\boldsymbol{s}, \langle \boldsymbol{a}_{-i}, \text{gain}\rangle) = \begin{cases} 4 & \text{if } (s_i = 1) \text{ for all } i \in [n] \\ 0 & \text{otherwise} \end{cases}$$

The mechanism (Def. 2) is $\mathcal{M} = \langle \boldsymbol{S}, \boldsymbol{I}, \boldsymbol{A}, \boldsymbol{T}, \boldsymbol{B}, \boldsymbol{h}, \boldsymbol{\beta} \rangle$. where the *proposed protocol* T_i for agent i requires agent i to cooperate, that is to carry out the assigned task. Formally: $T_i(\boldsymbol{s}, a_i) = ((s_i = 0) \wedge (a_i = \text{work})) \vee ((s_i = 1) \wedge (a_i = \text{gain})) \vee ((s_i = 2) \wedge (a_i = \text{reset}))$.

V. Paths in Mechanisms

Let $\mathcal{M} = \langle \boldsymbol{S}, \boldsymbol{I}, \boldsymbol{A}, \boldsymbol{T}, \boldsymbol{B}, \boldsymbol{h}, \boldsymbol{\beta} \rangle$ be an n player mechanism and let $Z \subseteq [n]$ be a set of (Byzantine) agents.

A *path* in ($\mathcal{M}$, Z) (or simply a *path* when ($\mathcal{M}$, Z) is understood from the context) is a (finite or infinite) sequence $\pi = \boldsymbol{s}(0)\boldsymbol{a}(0)\boldsymbol{s}(1)\ldots\boldsymbol{s}(t)\boldsymbol{a}(t)\boldsymbol{s}(t+1)\ldots$ where, for each t, $\boldsymbol{s}(t)$ is a global state, $\boldsymbol{a}(t)$ is a global action and $BT(Z, \boldsymbol{s}(t), \boldsymbol{a}(t), \boldsymbol{s}(t+1))$ holds.

The *length* of a path is the number of global actions in a path. We denote with $|\pi|$ the *length* of path π. If π is infinite we write $|\pi| = \infty$. Note that if $\pi = \boldsymbol{s}(0)$ then $|\pi| = 0$. Thus a path of length 0 is not empty.

In order to extract the t-th global state and the t-th global action from a given path π, we define $\pi^{(s)}(t) = \boldsymbol{s}(t)$ and $\pi^{(a)}(t) = \boldsymbol{a}(t)$. To extract local actions, we denote with $\pi_i^{(a)}(t)$ the action $a_i(t)$ at stage t of agent i and with $\pi_{-i}^{(a)}(t)$ the actions $\boldsymbol{a_{-i}}(t)$ at stage t of all agents but i.

For each agent $i \in [n]$, the *value* of a path π is $v_i(\pi) = \sum_{t=0}^{|\pi|-1} \beta_i^t h_i\left(\pi^{(s)}(t), \pi^{(a)}(t)\right)$. Note that for any path π and agent $i \in [n]$ the *path value* $v_i(\pi)$ is well defined also when $|\pi| = \infty$ since the series $\sum_{t=0}^{\infty} \beta_i^t h_i\left(\pi^{(s)}(t), \pi^{(a)}(t)\right)$ converges for all $\beta_i \in (0, 1)$. The *path value vector* is defined as: $\boldsymbol{v}(\pi) = \langle v_1(\pi), \ldots, v_n(\pi) \rangle$.

A *path* π in ($\mathcal{M}$, Z) is said to be *i-altruistic* if for all $t < |\pi|$, $T_i(\pi^{(s)}(t), \pi_i^{(a)}(t))$ holds.

Given a path π and a nonnegative integer $k \leq |\pi|$ we denote with $\pi|_k$ the *prefix* of π of length k, i.e. the finite path $\pi|_k = \boldsymbol{s}(0)\boldsymbol{a}(0)\boldsymbol{s}(1)\ldots\boldsymbol{s}(k)$ and with $\pi|^k$ the *tail* of π, i.e. the path $\pi|^k = \boldsymbol{s}(k)\boldsymbol{a}(k)\boldsymbol{s}(k+1)\ldots\boldsymbol{s}(t)\boldsymbol{a}(t)\boldsymbol{s}(t+1)\ldots$.

We denote with $\mathrm{Path}_k(\boldsymbol{s}, Z)$ the set of all paths of length k starting at $\boldsymbol{s}$. Formally, $\mathrm{Path}_k(\boldsymbol{s}, Z) = \{\pi \mid \pi \text{ is a path in } (\mathcal{M}, Z) \text{ and } |\pi| = k \text{ and } \pi^{(s)}(0) = \boldsymbol{s}\}$.

We denote with $\mathrm{Path}_k(\boldsymbol{s}, f)$ the set of all paths of length k feasible with respect to all sets of Byzantine agents of cardinality at most f. Formally, $\mathrm{Path}_k(\boldsymbol{s}, f) = \bigcup_{|Z| \leq f} \mathrm{Path}_k(\boldsymbol{s}, Z)$. We write $\mathrm{Path}_k(\boldsymbol{s})$ for $\mathrm{Path}_k(\boldsymbol{s}, n)$.

Unless otherwise stated in the following, we omit the subscript or superscript horizon when it is ∞. For example, we write $\mathrm{Path}(\boldsymbol{s}, Z)$ for $\mathrm{Path}_\infty(\boldsymbol{s}, Z)$.

Note that if $i \notin Z$, all paths in $\mathrm{Path}_k(\boldsymbol{s}, Z)$ are i-altruistic, that is, agent i behaves accordingly to the proposed protocol.

VI. Strategies

Let $\mathcal{M} = \langle \boldsymbol{S}, \boldsymbol{I}, \boldsymbol{A}, \boldsymbol{T}, \boldsymbol{B}, \boldsymbol{h}, \boldsymbol{\beta} \rangle$ be an n player mechanism and let $Z \subseteq [n]$ be a set of (Byzantine) agents.

As usual in a game theoretic setting, we need to distinguish player actions (i.e. local actions) from those of its opponents. This leads to the notion of strategy.

A *strategy* σ is a (finite or infinite) sequence of local actions for a given player. The *length* $|\sigma|$ of σ is the number of actions in σ (thus if $|\sigma| = 0$, the strategy is empty).

A strategy σ for player i *agrees* with a path π (notation $\pi \simeq^i \sigma$) if $|\sigma| = |\pi|$ and for all $t < |\sigma|$, $\sigma(t) = \pi_i^{(a)}(t)$.

Given a path π, the strategy (of length $|\pi|$) for player i associated to π is $\sigma(\pi, i) = \pi_i^{(a)}(0)\pi_i^{(a)}(1)\ldots\pi_i^{(a)}(t)\ldots$.

The set of *Z-feasible strategies* of length k for player i in state $\boldsymbol{s}$ is: $\mathrm{Strat}_k(\boldsymbol{s}, Z, i) = \{\sigma(\pi, i) \mid \pi \in \mathrm{Path}_k(\boldsymbol{s}, Z)\}$.

The set of *f-feasible strategies* of length k for player i in state $\boldsymbol{s}$ is: $\mathrm{Strat}_k(\boldsymbol{s}, f, i) = \{\sigma(\pi, i) \mid \pi \in \mathrm{Path}_k(\boldsymbol{s}, f)\}$.

As for paths, a strategy $\sigma \in \mathrm{Strat}_k(\boldsymbol{s}, Z, i)$ is said to be *i-altruistic* if $i \notin Z$. We use the notations $\sigma|_k$ and $\sigma|^k$ to denote, respectively, the k-prefix and the tail after k steps of a strategy.

The set of paths that agree with a set of strategies Σ for an agent i is defined as follows. $\mathrm{Path}(\boldsymbol{s}, Z, i, \Sigma) = \{\pi \in \mathrm{Path}_k(\boldsymbol{s}, Z) \mid \exists \sigma \in \Sigma.\ k = |\sigma| \wedge \pi \simeq^i \sigma\}$. When Σ is the singleton $\{\sigma\}$, we simply write $\mathrm{Path}(\boldsymbol{s}, Z, i, \sigma)$.

The *guaranteed outcome* (or the *value*) of a strategy σ in state $\boldsymbol{s}$ for player i is the minimum value of paths that agree with σ. Formally: $v_i(Z, \boldsymbol{s}, \sigma) = \min\{v_i(\pi) \mid \pi \in \mathrm{Path}(\boldsymbol{s}, Z, i, \sigma)\}$

The *value* of a state $\boldsymbol{s}$ at horizon k for player i is the guaranteed outcome of the best strategy of length k starting at state $\boldsymbol{s}$. Formally: $v_i^k(Z, \boldsymbol{s}) = \max\{v_i(Z, \boldsymbol{s}, \sigma) \mid \sigma \in \mathrm{Strat}_k(\boldsymbol{s}, Z, i)\}$.

The *worst case value* of a state $\boldsymbol{s}$ at horizon k for player i is the outcome of the worst strategy of length k starting at state $\boldsymbol{s}$. Formally, $u_i^k(Z, \boldsymbol{s}) = \min\{v_i(Z, \boldsymbol{s}, \sigma) | \sigma \in \mathrm{Strat}_k(\boldsymbol{s}, Z, i)\}$.

As usual, we will write $v_i(Z, \boldsymbol{s})$ for $v_i^\infty(Z, \boldsymbol{s})$, and $u_i(Z, \boldsymbol{s})$ for $u_i^\infty(Z, \boldsymbol{s})$.

The finite horizon value of a state can be effectively computed by using a dynamic programming approach (Prop. 1). This is one of the main ingredients of our verification algorithm (Sect. IX). We omit proofs because of lack of space.

Proposition 1: Let $\mathcal{M} = \langle \boldsymbol{S}, \boldsymbol{I}, \boldsymbol{A}, \boldsymbol{T}, \boldsymbol{B}, \boldsymbol{h}, \boldsymbol{\beta} \rangle$ be an n player mechanism, $i \in [n]$, $Z \subseteq [n]$ and $\boldsymbol{s} \in S$. The state values at horizon k for player i can be computed as follows:

- $v_i^0(Z, \boldsymbol{s}) = u_i^0(Z, \boldsymbol{s}) = 0$;
- $v_i^{k+1}(Z, \boldsymbol{s}) = \max_{a_i \in A_i} \min_{\boldsymbol{a}_{-i} \in A_{-i}} \{h_i(\boldsymbol{s}, \langle \boldsymbol{a}_{-i}, a_i \rangle) + \beta_i\, v_i^k(Z, \boldsymbol{s'}) \mid BT(Z, \boldsymbol{s}, \langle \boldsymbol{a}_{-i}, a_i \rangle, \boldsymbol{s'})\}$
- $u_i^{k+1}(Z, \boldsymbol{s}) = \min_{a_i \in A_i} \min_{\boldsymbol{a}_{-i} \in A_{-i}} \{h_i(\boldsymbol{s}, \langle \boldsymbol{a}_{-i}, a_i \rangle) + \beta_i\, u_i^k(Z, \boldsymbol{s'}) \mid BT(Z, \boldsymbol{s}, \langle \boldsymbol{a}_{-i}, a_i \rangle, \boldsymbol{s'})\}$

VII. Nash Equilibria in Mechanisms

Our notion of *Nash equilibrium* for a mechanism combines those in [23], [21]. Intuitively, a mechanism $\mathcal{M}$ is ε-f-Nash, if as long as the number of Byzantine agents is no more than f (e.g. see [21]), no rational agent has an interest greater than ε (e.g. see [23], [26]) in deviating from the proposed protocol in $\mathcal{M}$.

Definition 4 (ε-f-Nash): Let $\mathcal{M} = \langle \boldsymbol{S}, \boldsymbol{I}, \boldsymbol{A}, \boldsymbol{T}, \boldsymbol{B}, \boldsymbol{h}, \boldsymbol{\beta} \rangle$ be an n player mechanism, $f \in \{0, \ldots, n\}$ and $\epsilon > 0$. $\mathcal{M}$ is ε-f-Nash for player $i \in [n]$ if $\forall Z \in \mathcal{P}_f([n] \setminus \{i\})$, $\forall \boldsymbol{s} \in I$, $u_i(Z, \boldsymbol{s}) + \varepsilon \geq v_i(Z \cup \{i\}, \boldsymbol{s})$. $\mathcal{M}$ is ε-f-Nash if it is ε-f-Nash for each player $i \in [n]$.

Note that ε-0-Nash is the ε-Nash equilibrium defined in [23], [26]. Furthermore, stretching Def. 4 by setting $\varepsilon = 0$, we see that 0-f-Nash is the f-Nash equilibrium defined in

[21] whereas 0-0-Nash is the classical Nash equilibrium (e.g., see [26]).

Observe that, for each agent i, we compare agent i best reward when it considers deviating from the protocol ($v_i(Z \cup \{i\}, \boldsymbol{s})$), with agent i worst reward when it obeys the protocol ($u_i(Z, \boldsymbol{s})$). The reason for tolerating a small (ε) tolerance on rewards when deviating from the proposed protocol in Def. 4 is that our aim is to verify Nash equilibria by looking only at finite strategies. It is well known (e.g. see Sect. 4.8 of [26]) that ε-0-Nash equilibria have been introduced to get within a finite horizon equilibria that are only available with an infinite horizon. This means that a finite horizon may not suffice to check that a mechanism is 0-0-Nash. The following example aims at clarifying the above well known game-theoretical issue framing it in our context.

Example 1: Let $\mathcal{M}$ be a one agent (named 1) mechanism defined as follows. The underlying behavior B_1 for agent 1 is shown in Fig. 2 where on the automaton edges we show action names as well as payoff values since in this simple case the payoff function depends only on local states and local actions of the agent. The discount factor for agent 1 is $\beta_1 = \frac{1}{2}$. Let the *proposed protocol* $\boldsymbol{T}$ of $\mathcal{M}$ be defined as follows: $T_1(\boldsymbol{s}, \boldsymbol{a}) = ((\boldsymbol{s} = 0) \wedge (\boldsymbol{a} = \text{a})) \vee ((\boldsymbol{s} = 1) \wedge (\boldsymbol{a} = \text{d})) \vee ((\boldsymbol{s} = 2) \wedge (\boldsymbol{a} = \text{e})) \vee (\boldsymbol{s} \geq 3)$. We focus on the case $f = 0$, that is there are no Byzantine agents and hence $Z = \varnothing$. For all $k > 0$ we have: $u_1^k(\varnothing, 0) = -1 + \frac{3}{2} \sum_{i=0}^{k-2} (-\frac{1}{2})^i = (-1)^k \frac{1}{2^{k-1}}$ (the protocol $\boldsymbol{T}$ prescribes to follow the strategy $a(de)^\omega$) and $v_1^k(\{1\}, 0) = \frac{1}{2^{k-1}}$ (v_1 will use strategy $\sigma_1 = a(de)^\omega$ when k is even and $\sigma_2 = c(gh)^\omega$ when k is odd). Therefore, if k is odd $u_1^k(\varnothing, 0) < v_1^k(\{1\}, 0)$, and if k is even $u_1^k(\varnothing, 0) = v_1^k(\{1\}, 0)$, Thus there is no $\bar{k} > 0$ s.t. for all $k \geq \bar{k}$, $u_1^k(\varnothing, 0) \geq v_1^k(\{1\}, 0)$.

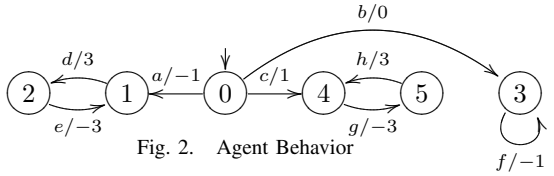


Fig. 2. Agent Behavior

Finding ε-0-Nash equilibria even for finite horizon games is not trivial (e.g. see [19]). As for infinite games, we note that game-theory results focus on showing that an ε *small enough* can be found so that an infinite horizon 0-0-Nash equilibrium becomes a finite horizon ε-0-Nash one (e.g. see [26]). Our concern here is different. We are given ε (fixed) and want to verify if the given mechanism is ε-0-Nash (actually, ε-f-Nash).

From Example 1 one may conjecture that ε-f-Nash ($\varepsilon > 0$) equilibria can be verified by just looking at *long enough* finite horizons. Unfortunately this is not always the case as shown by the following example.

Example 2: Consider again the mechanism $\mathcal{M}$ in Example 1. Let the *proposed protocol* $\boldsymbol{T}$ of $\mathcal{M}$ be defined as follows: $T_1(\boldsymbol{s}, \boldsymbol{a}) = ((\boldsymbol{s} = 0) \wedge (\boldsymbol{a} = \text{b})) \vee (\boldsymbol{s} \geq 1)$. Also in this case we focus on the case in which there are no Byzantine agents, that is $f = 0$ and $Z = \varnothing$. For all $k > 0$ we have: $u_1^k(\varnothing, 0) = \sum_{i=1}^{k} -\frac{1}{2^i} = -1 + \frac{1}{2^k}$ and $v_1^k(\{1\}, 0) = \frac{1}{2^{k-1}}$. For all k we have: $\Delta(k) = v_1^k(\{1\}, 0) - u_1^k(\varnothing, 0) = (1 + \frac{1}{2^k})$. Now, $\mathcal{M}$ is clearly 1-0-Nash however there is no $\bar{k} > 0$ s.t. for all $k \geq \bar{k}$, $\Delta(k) \leq 1$. That is, there is no finite horizon that allows us to conclude that $\mathcal{M}$ is 1-0-Nash. Note, however, that for all $\delta > 0$ there exists a $\bar{k} > 0$ s.t. for all $k \geq \bar{k}$, $\Delta(k) \leq 1 + \delta$. Thus, for all $\delta > 0$, by just considering a suitable finite horizon $\bar{k} > 0$, we can verify that $\mathcal{M}$ is $(1 + \delta)$-0-Nash.

VIII. Verifying ε-f-Nash Equilibria

In this Section we give our main theorem (Theor. 2) on which correctness of our verification algorithm (Sect. IX) rests.

Example 2 shows that, in general, using finite horizon approximations, we cannot verify ε-f-Nash equilibria. However the very same example suggests that we may get arbitrarily close to this result. This is indeed our main theorem.

Theorem 2 (Main Theorem): Let $\mathcal{M} = \langle \boldsymbol{S}, \boldsymbol{I}, \boldsymbol{A}, \boldsymbol{T}, \boldsymbol{B}, \boldsymbol{h}, \boldsymbol{\beta} \rangle$ be an n player mechanism, $f \in \{0, \ldots, n\}$, $\varepsilon > 0$ and $\delta > 0$. Furthermore, for each agent $i \in [n]$ let:

1) $M_i = \max\{|h_i(\boldsymbol{s}, \boldsymbol{a})| \mid \boldsymbol{s} \in S \text{ and } \boldsymbol{a} \in A\}$.
2) $E_i(k) = 5\, \beta_i^k \frac{M_i}{1-\beta_i}$.
3) $\Delta_i(k) = \max\{v_i^k(Z \cup \{i\}, \boldsymbol{s}) - u_i^k(Z, \boldsymbol{s}) \mid \boldsymbol{s} \in I, Z \in \mathcal{P}_f([n] \setminus \{i\})\}$
4) $\varepsilon_1(i, k) = \Delta_i(k) - 2E_i(k)$
5) $\varepsilon_2(i, k) = \Delta_i(k) + 2E_i(k)$

For each agent i, let k_i be s.t. $4\, E_i(k_i) < \delta$. Then we have:

1) If for each $i \in [n]$, $\varepsilon \geq \varepsilon_2(i, k_i) > 0$ then $\mathcal{M}$ is ε-f-Nash.
2) If there exists $i \in [n]$ s.t. $0 < \varepsilon \leq \varepsilon_1(i, k_i)$ then $\mathcal{M}$ is not ε-f-Nash. Of course in such a case a fortiori $\mathcal{M}$ is not 0-f-Nash.
3) If for each $i \in [n]$, $\varepsilon_1(i, k_i) < \varepsilon$ and there exists $j \in [n]$ s.t. $\varepsilon < \varepsilon_2(j, k_j)$ then $\mathcal{M}$ is $(\varepsilon + \delta)$-f-Nash.

Proof: Because of lack of space we omit the proof. See [42] for the complete proof. ∎

IX. ε-f-Nash Verification Algorithm

Resting on Prop. 1 and on Theor. 2, Algorithm 1 verifies that a given n agent mechanism $\mathcal{M}$ is ε-f-Nash.

In Algorithm 1, $\boldsymbol{s}$ and $\boldsymbol{a}$ are vectors (of boolean variables) ranging respectively on (the boolean encoding of) states ($\boldsymbol{S}$) and actions ($\boldsymbol{A}$).

The set Z of Byzantine agents in Algorithm 1 can also be represented with a vector of n boolean variables $\mathbf{b} = \langle b_1, \ldots b_n \rangle$ such that for each agent $i \in [n]$, agent i is Byzantine iff $b_i = 1$. Accordingly, the constraint $Z \in \mathcal{P}_f([n] \setminus \{i\})$ in Def. 4 becomes a constraint on $\mathbf{b}$, namely: $(\sum_{i=1}^{i=n} b_i \leq f \wedge b_i = 0)$. Along the same lines, the set $Z \cup \{i\}$ (used in Algorithm 1) can be represented with the boolean vector $\mathbf{b}[b_i := 1]$ obtained from $\mathbf{b}$ by replacing variable b_i in $\mathbf{b}$ with the boolean constant 1. For readability in Algorithm 1 we use Z rather than its boolean representation $\mathbf{b}$.

First of all, in line 3 of Algorithm 1, we compute the horizon k needed for agent i to achieve the required accuracy δ.

Lines 4–11 use Prop. 1 to compute state values at horizon k of state $\boldsymbol{s}$ with the set of Byzantine players Z when player i obeys the protocol ($u_i^k(Z, \boldsymbol{s})$) as well as when player i behaves arbitrarily within the underlying behavior ($v_i^k(Z \cup \{i\}, \boldsymbol{s})$).

Line 12 computes the max difference between initial state values for a rational player and those for a player following the proposed protocol, by also maximizing over all $Z \in \mathcal{P}_f([n] \setminus \{i\})$. (hypothesis 3 of Theor. 2, see also Def. 4). Line 13 computes the values in hypotheses 4 and 5 of Theor. 2.

Line 14 returns *FAIL* as soon as the hypothesis of thesis 2 of Theor. 2 is satisfied. In such a case from Theor. 2 we know that the given mechanism is not ε-f-Nash and thus it is not f-Nash.

Line 15 (16) returns, *PASS with* ε (*PASS with* $\varepsilon + \delta$) when the hypothesis of thesis 1 (3) of Theor. 2 is satisfied. In such a case from Theor. 2 we know that the given mechanism is ε-f-Nash ($(\varepsilon + \delta)$-f-Nash).

Algorithm 1 Checking if a mechanism is ε-f-Nash

1: *CheckNash*(mechanism $\mathcal{M}$, int f, double ε, δ)
2: **for all** $i \in [n]$ **do**
3: Let k such that $4\, E_i(k) < \delta$
4: Let $\boldsymbol{s} \in \boldsymbol{S}$ and $Z \in \mathcal{P}_f([n] \setminus \{i\})$
5: $v_i^0(Z, \boldsymbol{s}) \leftarrow 0$; $u_i^0(Z, \boldsymbol{s}) \leftarrow 0$;
6: **for** $t = 1$ **to** k **do**
7: $v_i^t(Z \cup \{i\}, \boldsymbol{s}) \leftarrow \max_{a_i \in A_i} \min_{\boldsymbol{a}_{-i} \in A_{-i}} [h_i(\boldsymbol{s}, \langle a_i, \boldsymbol{a}_{-i}\rangle) +$
8: $+\beta_i v_i^{t-1}(\boldsymbol{s'}, Z \cup \{i\})]$,
9: $BT(Z \cup \{i\}, \boldsymbol{s}, \langle a_i, \boldsymbol{a}_{-i}\rangle, \boldsymbol{s'})$
10: $u_i^t(Z, \boldsymbol{s}) \leftarrow \min_{a_i \in A_i} \min_{\boldsymbol{a}_{-i} \in A_{-i}} [h_i(\boldsymbol{s}, \langle a_i, \boldsymbol{a}_{-i}\rangle) +$
11: $+\beta_i u_i^{t-1}(Z, \boldsymbol{s'})], BT(Z, \boldsymbol{s}, \langle a_i, \boldsymbol{a}_{-i}\rangle, \boldsymbol{s'})$
12: $\Delta_i \leftarrow \max\{v_i^k(Z \cup \{i\}, \boldsymbol{s}) - u_i^k(Z, \boldsymbol{s}) \mid \boldsymbol{s} \in I, Z \in \mathcal{P}_f([n] \setminus \{i\})\}$
13: $\varepsilon_1(i) \leftarrow \Delta_i - 2E_i(k)$; $\varepsilon_2(i) \leftarrow \Delta_i + 2E_i(k)$
14: **if** $(\varepsilon < \varepsilon_1(i))$ **return** (FAIL)
15: **if** $(\forall i \in [n](\varepsilon_2(i) < \varepsilon))$ **return** (PASS with ε)
16: **else return** (PASS with $(\varepsilon + \delta)$)

X. Implementation

We implemented Algorithm 1 within the NuSMV [46] model checker. Here we briefly describe the main ideas bridging the gap between Algorithm 1 and its NuSMV implementation.

First of all, we extended the SMV language so as to be able to define mechanisms. We should keep in mind that, once we have verified that the proposed protocol is a Nash equilibrium for the given mechanism, then we will undertake a standard CTL verification in order to check that the proposed protocol satisfies the desired safety and liveness properties. Accordingly, we confined most of our extensions to the SMV language inside SMV comments so that mechanism models can also be used for standard CTL verification again with NuSMV.

As a second step we implemented Algorithm 1 using *Ordered Binary Decision Diagrams* (OBDDs) [10] resting on the CUDD [18] OBDD package (which is also the one used in NuSMV). Note that all functions in Algorithm 1 depend only on boolean variables, namely those representing states, actions and sets of Byzantine agents.

From Algorithm 1 we see that we only have two kinds of functions: *b2b* functions, taking booleans and returning a boolean value, and *b2r* functions, taking booleans and returning a real value. As usual *b2b* functions can be effectively represented using OBDDs. As for *b2r* functions we used the *Arithmetic Decision Diagrams* (ADDs) available in the CUDD package. ADDs are designed to represent and efficiently manipulate *b2r* functions returning reals represented as C 64 bit double. All arithmetical operations on ADDs used in Algorithm 1 are available in the CUDD package. The only ones that we had to implement ourselves were the $\max$ and $\min$ functions on ADDs. We developed them with a suitable traversal of the ADD to be minimized (or maximized). See [42] for a full description of our symbolic implementation of Algorithm 1.

XI. Experimental Results

In order to assess effectiveness of our Nash verifier we present experimental results on its usage on an n player mechanism $\mathcal{M}$ generalizing the mechanism presented in Section IV. This is a meaningful and scalable case study that well serves our purposes.

A. Mechanism Description

We are given a set $\mathcal{J} = \{0, \ldots m-1\}$ of m *jobs* and a set $\mathcal{T} = \{0, \ldots q-1\}$ of q *tasks*. Function $\eta : \mathcal{J} \to \mathcal{P}(\mathcal{T})$ defines for each job j the set of tasks $\eta(j)$ needed to complete j.

Each agent $i \in [n]$ is supposed to work (*proposed protocol*) on a given sequence of (not necessarily distinct) tasks $\mathcal{T}_i = \langle \tau(i,0), \ldots \tau(i, \alpha(i)-1)\rangle$ starting from $\tau(i,0)$ and returning to $\tau(i,0)$ after task $\tau(i, \alpha(i)-1)$ has been completed. An agent may *deviate* from the proposed protocol by delaying execution of a task or by not executing the task at all. This models many typical scenarios in cooperative services.

An agent incurs a *cost* by working towards the completion of its currently assigned task. Once an agent has completed a task it waits for its reward (if any) before it considers working to the next task in its list. As soon as an agent receives its reward it considers working to the next task in its list.

A job is completed if for each task it needs, there exists at least one agent that has completed that task. In such a case, each of such agents receive a *reward*. Note that even if two (or more) agents have completed the same task, all of them get a reward.

Agent i can be modeled as follows. Its set of states is $X_i = Y_i \times Z_i$, where: $Y_i = \{0, \ldots (\alpha(i)-1)\}$ and $Z_i = \{0, 1, 2\}$. State variable $\mathbf{y}_i$ ranges on Y_i, state variable $\mathbf{z}_i$ ranges on Z_i.

State variable $\mathbf{z}_i$ models the working state of agent i. Namely, $\mathbf{z}_i = 0$ if agent i is not working; $\mathbf{z}_i = 1$ if agent i has done its assigned work (that is it has completed its currently assigned task); $\mathbf{z}_i = 2$ if agent i currently completed task has been used to complete a job. In the last case agent i gets its reward for having completed a task used by a job. State variable $\mathbf{y}_i$ keeps track of the task currently assigned to agent i. That is, $\mathbf{y}_i = p$ iff agent i is supposed to complete task $\tau(i,p)$ in its sequence of assigned tasks.

Agent i's set of initial states is $I_i = \{(0,0)\}$ whereas agent i's set of actions is $A_i = \{0,1\}$ with variable $\mathbf{a}_i$ ranging on it. Variable $\mathbf{a}_i$ models agent i's choices. Namely, $\mathbf{a}_i = 1$ if agent i will work. and $\mathbf{a}_i = 0$ otherwise.

The state space of $\mathcal{M}$ is $\boldsymbol{S} = \langle Y_1, Z_1, \ldots Y_n, Z_n \rangle$. The set of initial states of $\mathcal{M}$ is $\boldsymbol{I} = \langle I_1, \ldots I_n \rangle$ We denote with $\mathbf{s}$ the vector (of present state variables) $\langle \mathbf{y}_1, \mathbf{z}_1, \ldots \mathbf{y}_n, \mathbf{z}_n \rangle$ and with $\mathbf{a}$ the vector (of action variables) $\langle \mathbf{a}_1, \ldots \mathbf{a}_n \rangle$.

Let $\Phi(i)$ be the set of pairs (j, p) such that the p-th task of agent i task sequence is needed for job j. Formally, $\Phi(i) = \{(j,p) \mid (p \in \{0, \ldots \alpha(i) - 1\} \wedge (j \in \mathcal{J}) \wedge (\tau(i,p) \in \eta(j))\}$.

Let $\Gamma(t)$ be the set of pairs (w, r) such that the r-th task of agent w task sequence is t. Formally, $\Gamma(t) = \{(w,r) \mid (w \in [n]) \wedge (r \in \{0, \ldots \alpha(w) - 1\}) \wedge (t = \tau(w,r))\}$.

Let $\varphi_j(\mathbf{s})$ be a boolean function which is true iff job j is completed. That is, if for each task t in job j there exists an agent w such that w has completed task t. Formally, $\varphi_j(\mathbf{s}) = \wedge_{t \in \eta(j)} \vee_{(w,r) \in \Gamma(t)} ((\mathbf{y}_w = r) \wedge (\mathbf{z}_w = 1))$.

Let $\gamma_i(\mathbf{s})$ be a boolean function which is true iff agent i is currently assigned to a task needed for a currently completed job. Formally, $\gamma_i(\mathbf{s}) = (\vee_{(j,p) \in \Phi(i)} ((\mathbf{y}_i = p) \wedge (\varphi_j(\mathbf{s})))$.

Finally, the underlying behavior B_i of agent i is defined as follows: $B_i(\mathbf{s}, \mathbf{a}_i, \mathbf{y}_i', \mathbf{z}_i') =$

$((\mathbf{z}_i = 0) \wedge (\mathbf{a}_i = 0) \wedge (\mathbf{y}_i' = \mathbf{y}_i) \wedge (\mathbf{z}_i' = \mathbf{z}_i)) \vee$
$((\mathbf{z}_i = 0) \wedge (\mathbf{a}_i = 1) \wedge (\mathbf{y}_i' = \mathbf{y}_i) \wedge (\mathbf{z}_i' = 1)) \vee$
$((\mathbf{z}_i = 1) \wedge \gamma_i(\mathbf{s}) \wedge (\mathbf{z}_i' = 2) \wedge (\mathbf{y}_i' = \mathbf{y}_i))) \vee$
$((\mathbf{z}_i = 1) \wedge \neg\gamma_i(\mathbf{s}) \wedge (\mathbf{z}_i' = 1) \wedge (\mathbf{y}_i' = \mathbf{y}_i))) \vee$
$((\mathbf{z}_i = 2) \wedge (\mathbf{y}_i' = (\mathbf{y}_i + 1) \bmod \alpha(i)) \wedge (\mathbf{z}_i' = 0))$.

The proposed protocol T_i for agent i is $T_i(\mathbf{s}, \mathbf{a}_i) = (\mathbf{a}_i = 1)$, that is agent i is supposed to carry out the assigned task as soon as it can. Reward h_i for agent i is defined as follows:

$$h_i(\mathbf{s}, \mathbf{a}_i) = \begin{cases} -1 & \text{if } ((\mathbf{z}_i = 0) \wedge (\mathbf{a}_i = 1)) \\ +4 & \text{if } (\mathbf{z}_i = 2) \\ 0 & \text{otherwise} \end{cases}$$

B. Experimental Settings

In order to run our Nash verification experiments we instantiate the above class of mechanisms as follows. First of all, we take the number of agents (n) to be greater than or equal to that of tasks (q). Second, we take the number of jobs (m) to be equal to the number of tasks (q). Third, we define $\eta(j)$ (i.e. the set of tasks needed to complete job j) as follows: $\eta(j) = \{j, (j+1) \bmod q\}$. That is, each job requires two tasks and each task participates in two jobs. We take as task sequence for agent i the sequence $\mathcal{T}_i = \langle (i-1) \bmod q, \ldots q-1, 0, \ldots ((i-1) \bmod q) - 1 \rangle$. In other words, all agents consider tasks with the same order (namely $\langle 0, \ldots q-1 \rangle$). The only difference is that agent i will start its task sequence from task $(i-1) \bmod q$. For each agent i we set, $\beta_i = 0.5$ and $\boldsymbol{\beta} = \langle\ \beta_1, \ldots \beta_n\ \rangle$. With the above settings we have only two parameters to be instantiated: n (number of agents) and m (number of jobs).

C. Experimental Results

Table XI-C shows our experimental results on verification of the ε-f-Nash property for the mechanism described in Sects. XI-A, XI-B. Column *Byzantines* in Table XI-C gives the number of Byzantine agents (f). In all experiments we take $\varepsilon = 0.01$ and accuracy $\delta = 0.005$. With such settings the value of k in line 3 of Algorithm 1 turns out to be 15 in all our experiments. Column *Nash* in Table XI-C shows the result returned by Algorithm 1, namely, *PASS* if the mechanisms is ε-f-Nash or $(\varepsilon + \delta)$-f-Nash, *FAIL* otherwise. The meaning of the other columns in Table XI-C should be self-explicative.

From Table XI-C we see that we can effectively handle moderate size mechanisms. Such mechanisms correspond indeed to quite large games. In fact, given a finite horizon k, an n player mechanism can be seen as a game whose outcomes are n-tuple $\langle \sigma_1, \ldots, \sigma_n \rangle$ of strategies of length k, where σ_i is the strategy played by agent i. If the underlying behavior of agent i allows it two actions for each state, then there are 2^k strategies available for agent i. This would yield a game whose normal form has 2^{kn} entries. In the mechanism used in Table XI-C, even without considering Byzantine players, each agent can choose at least among $fib(k)$ (the k-th Fibonacci number) strategies (many more if we consider Byzantine players). With horizon $k = 15$ and $n = 8$ players this leads to a normal form game with at least $fib(k)^n = 610^8 \approx 10^{22}$ entries.

XII. Conclusions

We present a symbolic model checking algorithm for verification of Nash equilibria in finite state mechanisms modeling MAD distributed systems. Our experimental results show the effectiveness of the presented algorithm for moderate size mechanisms. For example, we can handle mechanisms which corresponding normal form games would have more than 10^{22} entries.

Future research work include: improvements to the presented algorithm in order to handle larger mechanisms, verification of Nash equilibria *robust* with respect to agent *collusions*.

Acknowledgments

This work has been partially supported by MIUR grant TRAMP DM24283 and by NSF grant CRS-PDOS 0509338.

References

[1] Amitanand S. Aiyer, Lorenzo Alvisi, Allen Clement, Mike Dahlin, Jean-Philippe Martin, and Carl Porth. Bar fault tolerance for cooperative services. In *Proc. of SOSP'05*, pages 45–58. ACM Press, 2005.

[2] Anuchit Anuchitanukul and Zohar Manna. Realizability and synthesis of reactive modules. In *Proc. of CAV'94*, pages 156–168, 1994.

[3] Anish Arora, Paul C. Attie, and E. Allen Emerson. Synthesis of fault-tolerant concurrent programs. In *Symposium on Principles of Distributed Computing*, pages 173–182, 1998.

[4] S. Balemi, G. J. Hoffmann, P. Gyugyi, H. Wong-Toi, and G. F. Franklin. Supervisory control of a rapid thermal multiprocessor. *Joint issue Automatica and IEEE Trans. Autom. Control on* Meeting the Challenge of Computer Science in the Industrial Applications of Control, 1993.

[5] P. Ballarini, M. Fisher, and M. Wooldridge. Automated game analysis via probabilistic model checking: A case study. In *Proc. of MoChArt'05*, volume 149 of *ENTCS*, pages 125–137. Elsevier, 2006.

[6] Christopher Batten, Kenneth Barr, Arvind Saraf, and Stanley Trepetin. pStore: A secure peer-to-peer backup system. Technical Memo MIT-LCS-TM-632, Massachusetts Institute of Technology Laboratory for Computer Science, October 2002.

[7] Massimo Benerecetti and Fausto Giunchiglia. Model checking security protocols using a logic of belief. In *Proc. of TACAS'00*, pages 519–534. Springer-Verlag, 2000.

TABLE I
EXPERIMENTS RUN ON A 64-BIT DUAL QUAD CORE 3 GHZ INTEL XEON LINUX PC WITH 8 GB OF RAM

Agents	Jobs	Byzantines	Nash	CPU (sec)	Mem (MB)	Max BDD	Present State Bits	Action Bits	Model Size (KB)
5	2	1	PASS	1.49e+01	7.99e+01	5.88e+05	15	5	4.99e+01
5	2	2	FAIL	2.39e+01	7.41e+01	5.42e+05	15	5	4.99e+01
6	2	2	PASS	2.09e+02	8.18e+01	6.15e+05	18	6	6.90e+01
6	2	3	FAIL	2.99e+02	8.60e+01	4.33e+05	18	6	6.90e+01
6	3	3	PASS	1.68e+03	2.77e+02	5.62e+06	24	6	1.83e+02
6	3	4	FAIL	1.14e+03	2.17e+02	5.85e+05	24	6	1.83e+02
7	3	3	PASS	1.91e+04	1.85e+03	2.29e+07	28	7	2.46e+02
7	3	4	FAIL	2.22e+04	2.28e+03	5.64e+07	28	7	2.46e+02
8	3	2	PASS	8.03e+04	4.66e+03	5.53e+07	32	8	3.18e+02
8	3	3	N/A	>1.27e+05	>8.00e+03	>3.45e+07	32	8	3.18e+02

[8] Massimo Benerecetti, Fausto Giunchiglia, Maurizio Panti, and Luca Spalazzi. A logic of belief and a model checking algorithm for security protocols. In *Proc. of FORTE'00*, pages 393–408, 2000.

[9] Rafael H. Bordini, Michael Fisher, Carmen Pardavila, Willem Visser, and Michael Wooldridge. Model checking multi-agent programs with casp. In *Proc. of CAV'03*, pages 110–113, 2003.

[10] R. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. on Computers*, C-35(8):677–691, Aug 1986.

[11] Levente Buttyán and Jean-Pierre Hubaux. *Security and Cooperation in Wireless Networks - Thwarting Malicious and Selfish Behavior in the Age of Ubiquitous Computing (version 1.5)*. Cambridge University Press, 2007.

[12] Zining Cao. Model checking for real-time temporal, cooperation and epistemic properties. In *Intelligent Information Processing III*, volume 228 of *IFIP International Federation for Information Processing*, 2007.

[13] Steve Chien and Alistair Sinclair. Convergence to approximate nash equilibria in congestion games. In *Proc. of SODA'07*, pages 169–178. Society for Industrial and Applied Mathematics, 2007.

[14] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, 1999.

[15] A. Clement, H. Li, J. Napper, J-P Martin, L. Alvisi, and M. Dahlin. BAR primer. In *Proc. of DSN'08*, June 2008.

[16] Bram Cohen. Incentives build robustness in bittorrent. In *Proc. of IPTPS'03*, Feb 2003.

[17] Landon P. Cox and Brian D. Noble. Samsara: honor among thieves in peer-to-peer storage. In *Proc. of SOSP'03*, pages 120–132. ACM, 2003.

[18] CUDD Web Page: http://vlsi.colorado.edu/∼fabio/, 2004.

[19] Constantinos Daskalakis, Aranyak Mehta, and Christos Papadimitriou. Progress in approximate nash equilibria. In *Proc. of EC'07*, pages 355–358. ACM, 2007.

[20] Luca de Alfaro and Rupak Majumdar. Quantitative solution of omega-regular games. *J. Comput. Syst. Sci.*, 68(2):374–397, 2004.

[21] Kfir Eliaz. Fault tolerant implementation. *Review of Economic Studies*, 69(3):589–610, July 2002.

[22] E. Allen Emerson and Edmund M. Clarke. Using branching time temporal logic to synthesize synchronization skeletons. *Sci. Comput. Program.*, 2(3):241–266, 1982.

[23] H. Everett. Recursive games. *Contributions to the theory of games, vol. III - Annals of Mathematical Studies*, 39, 1957.

[24] A. Fabrikant, C. Papadimitriou, and K. Talwar. The complexity of pure nash equilibria. In *Proc. of STOC'04*, pages 604–612. ACM, 2004.

[25] Joan Feigenbaum, Rahul Sami, and Scott Shenker. Mechanism design for policy routing. In *Proc. of PODC'04*, pages 11–20, New York, NY, USA, 2004. ACM.

[26] D. Fudenberg and J. Tirole. *Game theory*. MIT Press, aug 1991.

[27] Richard D. McKelvey, Andrew M. McLennan and Theodore L. Turocy. Gambit: Software Tools for Game Theory, Version 0.2007.01.30 http://gambit.sourceforge.net/, 2007.

[28] Peter Gammie and Ron van der Meyden. Mck: Model checking the logic of knowledge. In *Proc. of CAV'04*, pages 479–483, 2004.

[29] Gear web page: http://jabc.cs.uni-dortmund.de/modelchecking/.

[30] E. Grädel, W. Thomas, and T. Wilke, editors. *Automata, Logics, and Infinite Games: A Guide to Current Research [outcome of a Dagstuhl seminar, February 2001]*, volume 2500 of *LNCS*. Springer, 2002.

[31] T. A. Henzinger. Model checking game properties of multi-agent systems (abstract). In *Proc. of ICALP'98*, page 543. Springer-Verlag, 1998.

[32] I. Walukiewicz. Pushdown processes: Games and model checking. In Rajeev Alur and Thomas A. Henzinger, editors, *Proc. of CAV'96*, volume 1102, pages 62–74. Springer Verlag, 1996.

[33] M. Kacprzak, A. Lomuscio, and W. Penczek. Verification of multi-agent systems via unbounded model checking. In *Proc. of AAMAS'04*, July 2004.

[34] L. Lamport, R. E. Shostak, and M. C. Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, 1982.

[35] M. Leucker. Model checking games for the alternation free μ-calculus and alternating automata. In Harald Ganzinger, David McAllester, and Andrei Voronkov, editors, *Proc. of LPAR'99*, volume 1705 of *LNAI*, pages 77–91. Springer, 1999.

[36] H. Li, A. Clement, E. Wong, J. Napper, I. Roy, L. Alvisi, and M. Dahlin. BAR gossip. In *Proc. of OSDI'06*, November 2006.

[37] Mark Lillibridge, Sameh Elnikety, Andrew Birrell, Mike Burrows, and Michael Isard. A cooperative internet backup scheme. In *Proc. of ATEC'03*, pages 29–41. USENIX Association, 2003.

[38] Alessio Lomuscio and Franco Raimondi. Model checking knowledge, strategies, and games in multi-agent systems. In *Proc. of AAMAS'06*, pages 161–168. ACM, 2006.

[39] C. Stirling M. Lange. Model checking games for branching time logics. *Journal of Logic and Computation*, 12:623–639, 2002.

[40] Ratul Mahajan, Maya Rodrig, David Wetherall, and John Zahorjan. Sustaining cooperation in multi-hop wireless networks. In *Proc. of NSDI'05*, pages 231–244. USENIX Association, 2005.

[41] Petros Maniatis, David S. H. Rosenthal, Mema Roussopoulos, Mary Baker, TJ Giuli, and Yanto Muliadi. Preserving peer replicas by rate-limited sampled voting. In *Proc. of SOSP'03*, pages 44–59. ACM, 2003.

[42] F. Mari, I. Melatti, I. Salvo, E. Tronci, L. Alvisi, A. Clement, and H. Li. Model checking nash equilibria in mad distributed systems. Technical report, Computer Science Department, University of Rome "La Sapienza", http://modelcheckinglab.di.uniroma1.it/techreps/bar-techrep-2008-07-24.pdf, 2008.

[43] J. C. Mitchell, M. Mitchell, and U. Stern. Automated analysis of cryptographic protocols using murphi. In *Proc. of SP '97*, page 141. IEEE Computer Society, 1997.

[44] J. C. Mitchell, V. Shmatikov, and U. Stern. Finite-state analysis of SSL 3.0. In *Seventh USENIX Security Symposium*, pages 201–216. USENIX, San Antonio, 1998.

[45] Noam Nisan and Amir Ronen. Algorithmic mechanism design (extended abstract). In *Proc. of STOC'99*, pages 129–140. ACM, 1999.

[46] NuSMV Web Page: http://nusmv.irst.itc.it/, 2006.

[47] P. J. G. Ramadge and W. M. Wonham. The control of discrete event systems. *Proc. of the IEEE*, 77(1):81–98, 1989.

[48] Fred B. Schneider. *What good are models and what models are good?* ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1993.

[49] Jeffrey Shneidman and David C. Parkes. Specification faithfulness in networks with rational nodes. In *Proc. of PODC'04*, pages 88–97. ACM, 2004.

[50] Wolfgang Thomas. On the synthesis of strategies in infinite games. In *Symposium on Theoretical Aspects of Computer Science*, pages 1–13, 1995.

[51] Wolfgang Thomas. Infinite games and verification (extended abstract of a tutorial). In *Proc. of CAV'02*, pages 58–64, 2002.

[52] Enrico Tronci. Optimal finite state supervisory control. In *Proc. of CDC'96*. IEEE, 1996.

[53] Enrico Tronci. Automatic synthesis of controllers from formal specifications. In *Proc. of ICFEM'98*, page 134. IEEE, 1998.

A Theory-Based Decision Heuristic for DPLL(T)

Dan Goldwasser
CS, Haifa University
Haifa, Israel
dgoldwas@cs.haifa.ac.il

Ofer Strichman
Information System Engineering,
IE, Technion, Haifa, Israel
ofers@ie.technion.ac.il

Shai Fine
IBM Haifa Research Lab,
Haifa, Israel
shai@il.ibm.com

***Abstract*—We study the decision problem of disjunctive linear arithmetic over the reals from the perspective of computational geometry. We show that traversing the *linear arrangement* induced by the formula's predicates, rather than the DPLL(T) method of traversing the Boolean space, may have an advantage when the number of variables is smaller than the number of predicates (as it is indeed the case in the standard SMT-Lib benchmarks). We then continue by showing a branching heuristic that is based on approximating T-implications, based on a geometric analysis. We achieve modest improvement in run time comparing to the commonly used heuristic used by competitive solvers.**

I. Introduction

Quantifier-free Linear arithmetic over the reals (QF_LRA in the terminology of the SMT community, or LRA for short) is arguably the most important decidable first-order theory in verification, other than propositional logic, and a subject for research [8] and annual competitions in the SMT community [3]. All competitive SMT solvers, including Yices [8], Z3 [6], ArgoLib [16], MathSAT [1] and CVC-3 [4], to name a few, decide LRA by instantiating the DPLL(T) framework [21], [12] with general simplex, as introduced by Dutertre and de Moura in [8], [9].

DPLL(T) is a generalization of DPLL for solving a decidable first-order theory T, assuming the existence of a decision procedure DP_T for a conjunction of T predicates. It first appeared in abstract form in a paper by Tinelli [21] and later materialized into the award-winning SMT solver Barcelogic [12]. It is based on an interplay between a SAT solver and DP_T. Consider first the following basic procedure, which existed prior to DPLL(T) in systems such as CVC [20] and an early version of MathSat [1]:

1) Encode each predicate with a new propositional variable.
2) Solve the resulting abstract formula with a SAT solver. If it is unsatisfiable – abort and declare the formula unsatisfiable.
3) Otherwise check with DP_T if the assignment, denoted α, is consistent in T. That is, whether the conjunction of the formula's predicates, each in the polarity assigned to it by α, is T-satisfiable. If yes – abort and declare the formula satisfiable.
4) Otherwise, add to the propositional abstraction a *lemma* in the form of a propositional clause, which rules out α (this will force the SAT solver to backtrack and find another assignment).
5) Return to step 2.

DPLL(T) improves this procedure in several dimensions. First, it calls DP_T after every partial assignment. This means that it cannot just abort with a 'Satisfiable' answer when DP_T returns TRUE. One possibility is that it would simply return the control to the SAT solver, but instead it applies *theory propagation*, which means that it finds predicates that are implied by the theory. Such predicates are said to be T-implied. For example, if $x = y$ and $y = z$ are two predicates assigned TRUE by the SAT solver, the theory solver can deduce that the predicate $x = z$ must be TRUE as well, and report this information to the SAT solver (assuming such a predicate exists. Typically solvers in this framework refrain from adding new predicates). We will give a more formal description of DPLL(T) in Sect. II.

Since theory propagation is a measure of efficiency, not of correctness, the question of how much such propagation should be done depends on the efficiency of the algorithm that deduces this information and perhaps also on the investigated formula. In the case of LRA *exhaustive theory propagation*, i.e., learning all possible T-implications, and sometimes even learning one such implication, is not cost-effective. We do not attempt to solve this problem in this article, but rather to show a method in which some information from the theory can still be obtained in a cost-effective manner. Specifically, we show a method to get *approximated* T-implications and how to integrate them in the solving process without jeopardizing soundness. The approximated information is affecting the decision heuristic: the *decision variable* is still chosen using the SAT solver's normal considerations, while the variable's value is decided using theory related considerations. This is in contrast to the current practice in which decisions are made solely by the SAT solver, and are affected by the theory only indirectly, via the lemmas added by the solver.

We can learn about the potential of theory propagation in the case of LRA, that is, the average number of implications given a partial assignment, from a theoretical result in computational geometry by Haussler and Welzl [13]. Geometrically, each linear constraint is a hyperplane, and the geometrical representation of these hyperplanes in d dimensions, where d is the number of variables in the input linear system, is called a *hyperplane linear arrangement* [10]. Fig. 1 demonstrates a linear arrangement in two dimensions. Each *cell* (i.e., a convex polytope) in a linear arrangement contains exactly the infinite set of points that evaluate all the linear predicates in the same way. Hence, there is a 1-1 (but not onto) mapping from cells to

978-1-4244-2735-2/08/$25.00 ©2008 IEEE

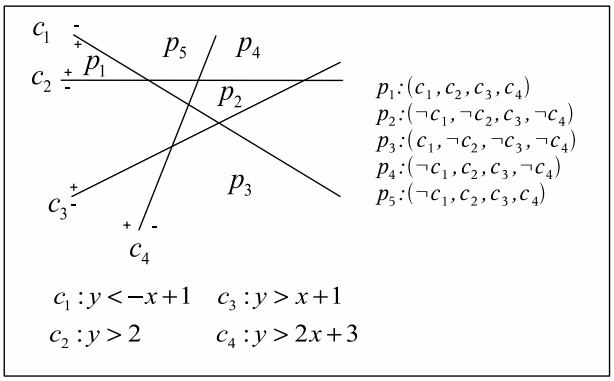


Fig. 1. A two dimensional linear arrangement of hyperplanes induced by a set of constraints. Each cell P_i can be mapped to a full assignment of these constraints – a mapping of some of the cells appears on the right side of the figure.

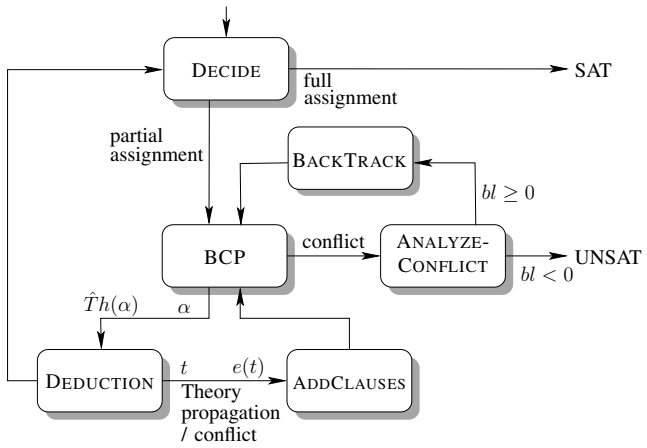


Fig. 2. The main components of DPLL(T). Theory propagation is implemented in DEDUCTION.

2^n, where n is the number of constraints. The number of cells is bounded by $O(n^d)$, which, note, can be much smaller than 2^n. Hence, many combinations of predicates do not correspond to a cell. This is exactly the case that the T-solver declares an assignment as being inconsistent.

In a series of papers, Haussler and Welzl [13] and Clarkson [5], suggested that for a $n \times d$ linear system, if we randomly select r constraints such that $d \leq r < n$, and build a linear arrangement, then with high probability – a probability of $(1-1/r^c)$, where c is a constant – the interior of any cell in the new arrangement is intersected by at most $O((n \log r)/r)$ of the remaining $n - r$ constraints. (This result refers to the number of variables d as a constant. See [11] for a more detailed explanation, and for an explicit proof of this property).

The relevance of this result to theory propagation is clear: if the current partial assignment is T-consistent, it corresponds to a cell, and the value of any unassigned linear constraint that does not intersect this cell is implied. The value r in our case is simply the number of assigned predicates in the current partial assignment.

The rest of this paper is structured as follows. We begin by describing DPLL(T) and general simplex in Sect. II-A. Section III presents a geometric interpretation of the search space and how it can be used for approximating implications that can help the search process. Section IV describes our experimental results. We conclude in Sect. V with some thoughts on possible future uses of the observations we make.

II. PRELIMINARIES

A. The DPLL(T) framework

State-of-the-art SMT solvers follow the DPLL(T) framework [21]. The components of the algorithm are those of DPLL and a decision procedure DP_T for a conjunctive fragment of a theory T. The name DPLL(T) emphasizes that this is a framework that can be instantiated with a different theory T and a corresponding decision procedure. We assume that the reader is familiar with the basics of DPLL in the description that follows.

In the version of DPLL(T) presented in Algorithm 1 (see also Fig. 2, which is copied from [15]), a procedure called DEDUCTION is invoked in line 13 after no more implications can be made by BCP. DEDUCTION performs theory propagation: it finds T-implied literals and communicates them to the DPLL part of the solver in the form of a constraint t, also called a *lemma*. Hence, in addition to implications in the Boolean domain, there are also implications due to the theory T.

What are the restrictions on these new clauses? They have to be implied by the input formula φ and restricted to the atoms in φ (or some finite superset thereof). Let α denote the current assignment and $\hat{T}h(\alpha)$ the conjunction of T-literals corresponding to this assignment. If $\hat{T}h(\alpha)$ is unsatisfiable, the lemma $e(t)$ (where $e(t)$ denotes t after each predicate is replaced with its propositional encoder) has to block α. If $\hat{T}h(\alpha)$ is satisfiable, we require t to fulfill one of the following two conditions in order to guarantee termination:

1) The clause $e(t)$ is an asserting clause under α. This implies that the addition of $e(t)$ to the current propositional formula and a call to BCP leads to an assignment to the encoder of some literal.
2) When DEDUCTION cannot find an asserting clause t as defined above, t and $e(t)$ are equivalent to TRUE.

The second case occurs, for example, when all the Boolean variables are already assigned, and thus the formula is found to be satisfiable. In this case, the condition in line 15 is met and the procedure continues from line 5, where DECIDE is called again. Since all variables are already assigned, the procedure returns "Satisfiable".

As we wrote in the introduction, theory propagation has no influence on correctness, rather only on efficiency, and therefore the question of how much to infer on the theory side and propagate depends on the theory and the benchmark set. It turns out, empirically, that exhaustive theory propagation in the case of LRA is not cost-effective (see, e.g., [9]). Moreover, even checking for consistency of the current partial assignment

is too costly in practice. Instead, competitive solvers only do light-weight theory propagation and defer the consistency check to when a full assignment is achieved, as we will describe later in Sect. II-B.

Algorithm 1 The DPLL(T) framework.

```
1:  function DPLL(T)
2:      AddClauses (cnf(e(φ)));
3:      if BCP () = "conflict" then return "Unsatisfiable";
4:      while (TRUE) do
5:          if ¬Decide () then                    ▷ Full assignment
6:              return "Satisfiable";
7:          repeat
8:              while (BCP () = "conflict") do
9:                  btrack-level := Analyze-Conflict ();
10:                 if btrack-level < 0 then
11:                     return "Unsatisfiable";
12:                 else BackTrack(btrack-level);
13:             t :=Deduction (Th(α));
14:             AddClauses (e(t));
15:         until t ≡ true
```

There are other variations to DPLL(T) that are used in competitive solvers, including procedures for strengthening the lemmas and more aggressive invocations of DP_T (after every partial assignment rather than only after BCP). These optimizations are not relevant to the current work, however.

B. General Simplex

The standard de-facto decision procedure DP_T for the conjunctive fragment of linear arithmetic over the reals is *general simplex*, as was introduced in [8]. This procedure determines the satisfiability of a conjunction of linear constraints (hence, unlike the original simplex, it does not aim to optimize the value of a linear objective function). General simplex is now implemented in most competitive SMT solvers due to its superior performance in the context of SMT.

Let $A\vec{x} \leq \vec{b}$ be the input linear system, where A is a $n \times d$ coefficient matrix, $\vec{x}$ is a vector of d variables, and $\vec{b}$ is a vector of constants. General simplex begins by transforming this system into *general form*, which consists of two types of linear constraints: equalities of the form $\sum_i a_i x_i = 0$ and constrains of the form $x_i \geq l_i$ or $x_i \leq u_i$, where l_i and u_i are constants. The transformation is done as follows: given a constraint of the form $\Sigma a_{ij} x_j \leq b_i$, it replaces it with the two constraints

$$\begin{aligned} &\textstyle\sum a_{ij} x_j - s_i = 0 \\ &s_i \leq b_i \ , \end{aligned}$$

where s_i is a new variable, which is called a *bound variable*. The new bound variables constitute the initial set of what is called the *basic* variables, whereas the other variables constitute the initial set of *nonbasic* variables. The basic variables are also called the *dependent* variables, reflecting the fact that their value is determined by the values of the nonbasic variables. The partitioning of the variables to these two sets change throughout the algorithm.

In addition to these two sets, the algorithm also maintains an assignment β to all variables. Two invariants are maintained during the run of the algorithm:

1) The assignment β satisfies all equalities (i.e., it satisfies $A\vec{x} = 0$), and
2) β satisfies those *bound variables* (the new s_i variables) that are currently in the nonbasic set.

Initially the assignment is 0 to all variables. This satisfies the first invariant trivially, and the second one because all the bound variables are basic. Then, the algorithm searches for a basic variable that violates one of its bounds. If there is no such variable the instance is declared satisfiable, since the current assignment satisfies both the equations and all the bound variables. Otherwise, suppose that the assignment to the basic variable x_i violates its upper bound, and hence has to be reduced. Simplex searches for a nonbasic variable with a positive coefficient that its current value is higher than its lower bound (or such a variable with a negative coefficient that its current value is lower that its upper bound). If there is such a variable x_j, it means that we can reduce the value of x_i by changing the value of x_j. If not – the instance is declared unsatisfiable. Suppose that there exists such a variable x_j (we say then that x_j is *suitable*). The next step is to change the current assignment and perform *pivoting*, which is essentially the same operation that is done in Gaussian elimination. Pivoting between these two variables means that they exchange places (x_i becomes nonbasic whereas x_j becomes basic), the coefficient matrix is updated accordingly, the assignment to x_i is reduced to meet its upper bound, and the assignment to the other variables are updated so the first invariant is maintained. More details about the pivot operation can be found in [8]. This is not essential for understanding the rest of the paper, however, and was only brought here for completeness. The important point is that through a series of such pivoting operations simplex updates its assignment β until it satisfies the input linear system, or declares the system unsatisfiable. Our method uses the assignment β and the pivot operations to approximate T-implications, as will be described later on.

Pseudocode for general simplex appears in Alg. 2. In Fig. 3, assuming the system comprises a conjunction of the predicates $c_1, \ldots, c_5$, general simplex's initial assignment corresponds to the origin (0 to all variables), which is marked as v_1 in the figure. As more pivoting operations take place the assignment is updated, and the points move closer to the target cell P_1.

Recall that in the context of DPLL(T) the linear solver is used incrementally: linear predicates are added or erased as the search progresses. While for most theories, competitive implementations of DPLL(T) check for T-consistency of every partial assignment (and perform theory propagation as described in Sect. I), this is not cost-effective in the case of LRA, at least as long as no better alternative to general-simplex is found. Instead, competitive solvers use a lightweight checking procedure called Assert – see Alg. 3. This procedure can only detect inconsistencies of bounds, for

Algorithm 2 General Simplex

1: **function** GENERAL SIMPLEX
2: Transform the system into the general form Ax=0 and $\bigwedge_{i=0}^{m} l_i \leq s_i \leq u_i$
3: Set *B* to the set of additional variables $s_1, ..., s_m$
4: Construct a tableau for *A*
5: Determine a fixed order on the variables
6: If there is no basic variable that violates its bounds, returns "Satisfiable" Otherwise, let x_i be the first basic variable in the order that violates its bounds
7: Search for the first suitable nonbasic variable x_j in the order for pivoting with x_i. If there is no such variable, return "Unsatisfiable".
8: Perform the pivot operation on x_i and x_j.
9: Go to step 5.

example if both $x_i \leq 5$ and $x_i \geq 6$ are asserted. In addition it updates the assignment of nonbasic variables so the second invariant is maintained.

Algorithm 3 Procedure Assert-Upper detects simple T-inconsistencies in the current assignment to the predicates, and maintains an assignment which satisfies the bounds of the nonbasic variable.

1: **function** ASSERT UPPER ($x_i < c_i$)
2: **if** $c_i \geq u_i$ **then return** "satisfiable";
3: **if** $c_i < l_i$ **then return** "unsatisfiable";
4: $u_i := c_i$;
5: **if** x_i is a nonbasic variable and $\beta(x_i) > c_i$ **then**
6: update-assignment(x_i, c_i);

III. GEOMETRIC REPRESENTATION

A. Background on linear arrangements

Given a linear arithmetic formula φ, we denote by $C(\varphi)$ the set of linear predicates appearing in φ. Each linear constraint $c \in C(\varphi)$ is represented as a hyperplane in R^d, partitioning R^d into two halfspaces: in c^+ all points satisfy c, and in c^- all points do not satisfy c. An intersection of $C(\varphi)$ halfspaces form cells, which are convex regions in R^d. As we saw in Sect. I these cells can be mapped into an assignment to the predicates in $C(\varphi)$.

For example, consider the cell marked P_1 in Fig. 3. This region is the intersection of the positive halfspaces of φ's constraints and hence corresponds to the assignment (c_1, c_2, c_3, c_4).

The space of feasible assignments to the predicates can be described with a hyperplane linear arrangement, which is a well known data structure that is used in computational geometry. An arrangement captures the decomposition of a d-dimensional space into connected cells, induced by a set of hyperplanes in R^d. Each cell in the hyperplane arrangement is associated with a dimension: vertices (i.e., hyperplane intersection points) have a dimension of zero, while the convex regions

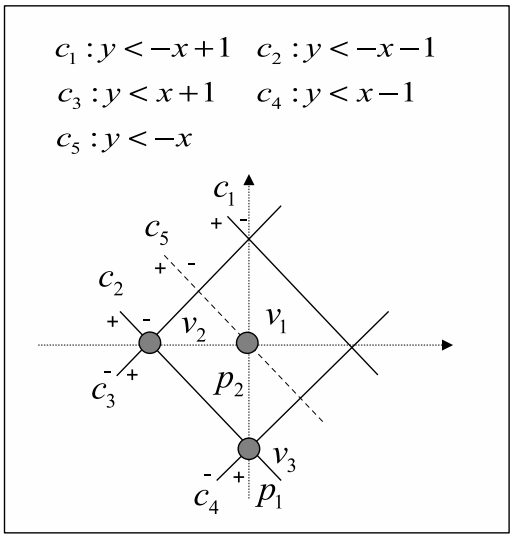


Fig. 3. A linear arrangement corresponding to five linear constraints. The axes and the points v_1, v_2, v_3 are not part of the arrangement, but useful for understanding the progress of simplex given the same constraints. The points $v_1, \ldots, v_3$ represent a possible progress of the assignment maintained by simplex. v_1 is the initial assignment, which is always the origin.

formed by the intersection of halfspaces have a dimension equal to the total number of variables appearing in $C(\varphi)$.

The number of cells is bounded by $O(n^d)$, where $n = |C(\varphi)|$ and d is the total number of distinct variables appearing in $C(\varphi)$.[1] This implies that the complexity of enumerating theory-consistent assignments is exponential in the number of dimensions whereas the complexity of enumerating values in the Boolean space is exponential in the number of constraints.[2] The difference in the two spaces plays a crucial role during the DPLL(T) search: the greater the ratio is, the greater the chance that a propositional assignment is inconsistent in T (in other words, it does not correspond to any cell in the arrangement).[3] Since this difference depends on the values of d and n, we checked these values in various SMT-LIB benchmarks – see Table I. The results show that the number of predicates is greater than the dimension, hence the linear search space is smaller than the propositional one in these benchmarks.

B. Geometric representation of T-implications

A partial propositional assignment is *T-consistent* if it can be mapped into a cell and *T-inconsistent* otherwise. Viewed geometrically, the value of an unassigned predicate p is *T-implied* by a partial assignment, if the cell induced by the partial assignment is contained within any of the halfspaces defined by p.

Example 1: In Fig. 3 the subset (c_2, c_4) is T-consistent and forms the cell P_1. If c_1 is the current decision variable, decid-

[1] The 'O' notation is not precise here, because the constant actually depends on d. We follow here the convention used by Halperin in [14], which used this convention based on an assumption that d is small relatively to n. The bound is in fact $\sum_{i<d} \binom{n}{d-i}$.

[2] We discuss possible implications of this gap in complexity in Sect. V.

[3] As a result one may even tune the search procedure according to this ratio.

TABLE I
PROPORTION OF PREDICATES VS. VARIABLES (DIMENSIONS) IN THE SMT-LIB BENCHMARKS

Benchmark	Predicates:Dimension
QF RDL SCHEDULING	10.9:1
QF RDL SAL	6.7:1
QF LRA SC	3.9:1
QF LRA START UP	6.9:1
QF LRA UART	6.1:1
QF LRA CLOCK SYNCH	3.3:1
QF LRA SPIDER BENCHMARKS	3.2 :1
QF LRA SAL	6.1:1
MathSAT benchmarks (difference logic)	44.5:1
SEP benchmarks (difference logic)	17:1

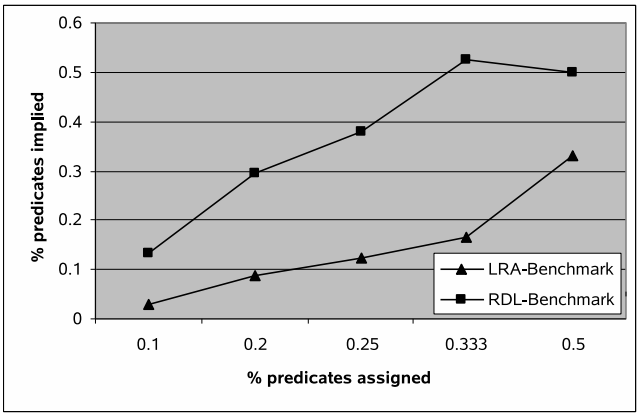


Fig. 4. The proportion of assigned vs. implied values at different points during the search. Two benchmarks were checked: a Linear Arithmetic (LRA) benchmark – invar.induct – containing 633 constraints and 163 variables, and a Difference Logic (RDL) benchmark – abz5_1000.smt – containing 1011 constraints and 102 variables.

ing on $\neg c_1$ would lead to a conflict, expressed geometrically as an empty intersection of c_1^- and P_1. Hence, c_1 is T-implied by the partial assignment. Indeed, P_1 is completely contained within one of c_1's halfspaces.

Now consider P_2 as the current partial assignment. The value of c_5 is not implied as both its halfspaces have a non-empty intersection with P_2.

As the DPLL search advances and the partial assignment grows (i.e., more linear constraints are asserted), more values are likely to be T-implied. The reason is that more predicates imply a smaller cell (or even an empty cell if the partial assignment is T-inconsistent), and hence the chances of an unassigned predicate to intersect this cell is smaller. We tested this observation empirically: Fig. 4 describes the ratio between the partial assignment size and the number of predicates implied by it for two benchmarks. The number of T-implications was measured by randomly selecting 100 different partial propositional assignments of equal size and averaging the number of T-implied values by each such partial assignment. Indeed, it is clear that the probability of an unassigned predicate to be T-implied grows with the partial propositional assignment.

C. Identifying T-implications

There are two natural ways to identify T-implications that we are aware of. Given a system of constraints S corresponding to the current partial assignment and an unassigned predicate p, the first method (called *plunging* in [7]) is to solve S together with the negation of p. If the system is unsatisfiable, it means that S implies p. This is a generic method that is relevant for all decidable theories. The second method is to consider the vertices of the cell corresponding to S: if they fall on both halfspaces of p, then the value of p is not T-implied.[4] For example in Fig. 3 the vertices of the cell P_2 fall on both halfspaces of c_5. Both of these methods turn out to be too expensive in practice: the first because it corresponds to solving a full linear system for each predicate, and the second because there can be an exponential number of vertices to consider for each cell.

[4]This can be applied directly only to closed cells. For open cells a different test has to be made, such as plunging.

A possible way to alleviate the computational problem of checking all vertices is to approximate the geometric representation of the T-solver's state. This, however, prevents us from using this information to identify implications since an approximated implication may affect the soundness of the algorithm; we need therefor to restrict the use of such information, allowing the DPLL search to recover in case the value was not in fact implied. Our system solves this problem by using this information as 'hints' to the decision heuristic as to the value of the current decision variable. In other words, we only use it to affect the decision, not to create new implications. The choice of decision variable is still made by the SAT solver, but when the T-solver has an approximated estimation of the value of this variable, it passes this information to the DPLL solver which then assigns it to the decision variable. Hence, wrong information results in slower solving, not incorrect result. How can T-implications be approximated? we can, for example, generate a small number of points inside the cell corresponding to the current partial assignment (or better of, a small number of vertices of this cell), and then guess the value of an unassigned predicate according to the halfspace of the predicate in which these points fall. If they fall on both sides, the decision on the value can be made by the SAT solver. For example, consider once again the constraint c_5 and the region P_1 appearing in Fig. 3. The partial assignment $(c_1, ..., c_4)$, which corresponds to P_1 implies $c_5 = true$. But identifying this value for c_5 can be done by generating only one of P_1's vertices and checking whether it falls in the positive or negative halfspace of c_5.

An obvious requirement is the ability to generate such a point with little computational cost. Unfortunately, generating points which are known to fall within a cell defined by the

intersection of constraints proved to be a non-trivial issue.[5]

The method our system uses is simple but not very accurate. It relies on the assignment β that is maintained by simplex. Recall that due to efficiency considerations, in competitive DPLL(T) solvers simplex is not fully invoked after each partial (propositional) assignment, and rather only the ASSERT procedure (Alg. 3) is invoked. This means that β does not necessarily correspond to a point in the cell associated with the current partial assignment. It is also possible that there is no such cell at all, if the current assignment is T-inconsistent. Thus, although using the assignment adds no additional complexity, we can only use it, as before, to approximate implications rather than infer them.

We can attempt to improve this approximation by trying to make β more accurate. This can be done by invoking the pivot operation for some bounded number of times k, or until a definite conclusion is reached (i.e., β is in the cell or the current system is T-inconsistent). Thus, k is an *accuracy parameter*. However, additional pivot operations can also make the number of satisfied constraints go down, since the pivot operation is not monotonic in this sense. Hence our implementation takes the assignment β that satisfies the largest number of bound predicates along the way.

D. Taking decisions at T-inconsistent points

Recall that the current partial assignment is not necessarily T-consistent because in practice most solvers perform complete T-consistency checks at selected intervals or even only when the assignment is full. This is the strategy of the SMT solver Argolib [16], on top of which we implemented our heuristic.

We checked empirically the ratio of times that decisions are taken when the partial assignment is T-consistent. The larger this number is, the less our heuristic will be affected by this problem. We evaluated this number by running an external T-solver for validating the partial assignment at each decision point. It should be noted that the outcome of this validation was ignored, and in no way affected the operation of the solver. We ran this experiment for each of the benchmarks that we used for evaluation, as described in Sect. IV. The average proportion of decisions taken when the partial assignment was T-consistent across all benchmarks was 0.78.

IV. EVALUATION

We tested the performance of our approach on the LRA benchmarks [3] that were used in SMT-COMP'07. We set the timeout to 30 minutes for each benchmark. Our implementation is based on the open-source solver ArgoLib. This tool is based on DPLL(T) and the general simplex algorithm for solving linear arithmetic.

ArgoLib's original decision heuristic is the same as in MiniSAT: it selects a decision *variable* using a VSIDS-like [17]

[5]One of the methods we tried for generating such points was to randomly select r constraints out of the partial assignment and identify the point of intersection of these constrains using Gaussian Elimination. The problem is that unless r is large, most of the points generated in this manner are bound to fall outside the target cell, which makes this method inefficient in practice.

TABLE II
OVERALL RESULTS. THE RESULTS SHOW THE NUMBER OF INSTANCES SOLVED CORRECTLY IN LESS THAN 30 MINUTES AND THE OVERALL TIME SPENT BY THE SOLVER.

	Baseline	**PS**	**0-pivot**
Score	172	175	180
Total time	72229.8	68656.04	64887.43
	2-pivot	**4-pivot**	**14 -pivot**
Score	177	174	173
Total time	70491.21	75261.75	84125.97

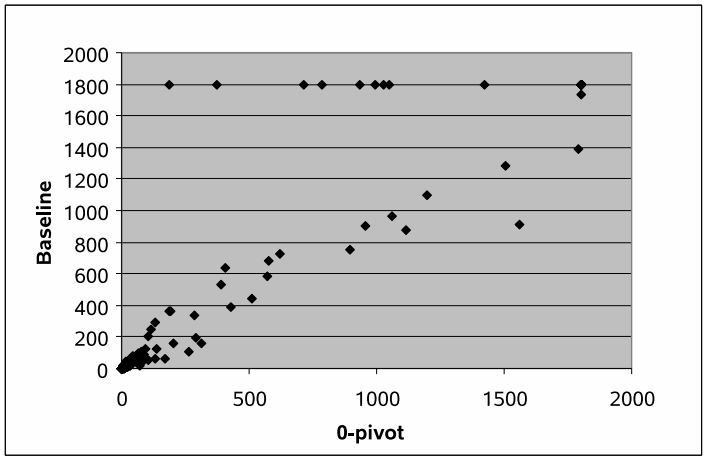


Fig. 5. Baseline vs. 0-pivot (run-time).

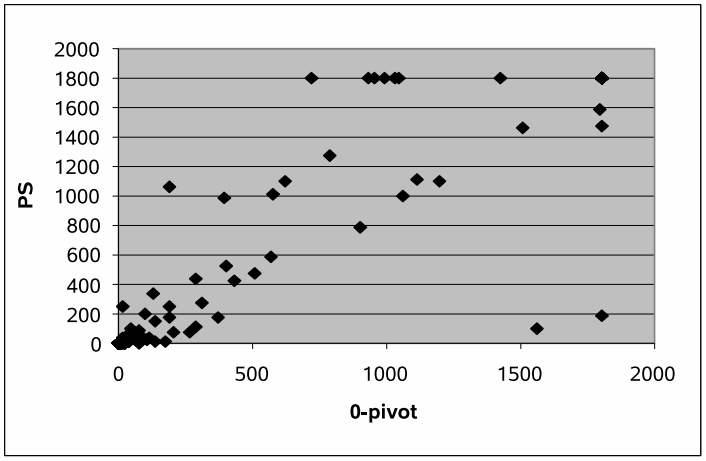


Fig. 6. PS vs. 0-pivot (run-time).

heuristic and sets its value to FALSE. We also implemented and evaluated the system using a decision heuristic in which a decision variable is assigned its last value if it was assigned before, and FALSE otherwise. Such a heuristic was used in CSP solvers [2], introduced to SAT in [19] and lately adopted by RSAT under the name 'progress saving' (PS) [18]. We'll call it the PS strategy.

We evaluated several variants of our approach by choosing different values of the accuracy parameter k. Performance was measured in terms of the total number of instances solved correctly before time-out and the overall running time.

Table II summarizes the results for six different strategies: *Baseline* and *PS* correspond to the original ArgoLib's heuristic and its enhancement with the PS heuristic as define above. Both of these heuristics do not use the theory solver directly to make a decision. The strategy *k-pivot* refers to our heuristic where *k* pivoting steps are made before deciding on a value.

The table shows that our heuristic somewhat outperforms the baseline, regardless of the number of pivoting operations performed. Increasing k turns out to be not cost-effective (there were several benchmarks, however, that it improved run-time). Overall we achieved a modest improvement of 11% in run time and 5% in the number of solved instances over all benchmarks.[6] There was no significant difference between the satisfiable and unsatisfiable instances.

Figures 5–6 show detailed results when comparing 0-pivot with the two SAT-based heuristics. From Fig. 5 it is evident that with minor exceptions, the 0-pivot heuristic has a comparable performance with the baseline heuristic, but it is able to solve most of the instances which cannot be solved by the baseline system, which means that it is more robust. Figure 6 shows that PS, although fails less than the baseline heuristic, still fails in many cases that can be solved with the 0-pivot strategy.

We also checked the accuracy of our approximation. For the first 100 benchmarks in the SMT-COMP'07 benchmark set, we measured the following data, with a time out of 30 min:

1) Total number of decisions: 710782.
2) Number of decisions resulting in T-inconsistency: 299130 (42% of partial assignments). This is much higher than the average of 22% that we reported in the previous section, which can be attributed to the different selection of benchmarks.
3) Number of implied decisions (checked with 'plunging'): 19799 (4.8% of T-consistent partial assignments). There are two possible reasons for this particularly small number in comparison to Fig. 4: First, it may indicate that most decisions are made in a very small decision level and that relatively few predicates are implied with BCP. Such a scenario leads to small partial assignments, and hence a small number of T-implications, when most of the decisions are made. Second, here the statistics refer to one variable per decision – the variable that was chosen by the SAT solver due to propositional considerations, whereas the statistics in Fig. 4 refer to all unassigned variables. It is possible that lemmas bias the SAT solvers's decision variable towards those that are not implied.
4) Number of times k-pivot approximation with $k = 0$ lead to a correct implication, when the partial assignment is consistent: 14447 (72% of T-implications).
5) Number of times k-pivot approximation with $k = 22$ lead to a correct implication, when the partial assignment is consistent: 14456 (73% of T-implications). This result is very surprising: even after that many pivot operations, the improvement in accuracy is very marginal. This explains why $k = 0$ is the best, empirically.
6) Number of times the value chosen by the baseline algorithm (simply FALSE) lead to a correct implication, when the partial assignment is consistent: 12557 (63.4% of T-implications).

There are two main points to observe: first, that the 0-pivot strategy increases the accuracy of the decision from 63.4% to 72% (and, recall, it does so with almost 0 cost). Second, that the fact that in less than 5% of the cases there was an implied value, may possibly indicate that the 0-pivot strategy is also helpful when the decided value is not T-implied. We can speculate why this happens when the instance is satisfiable: the predicate partitions the cell into two parts which are most likely not even. The chosen point has a higher probability to be in the bigger of the two parts, and there is a higher probability that the solution resides in that part.

V. Discussion and Future work

The improvement in run time that we showed is very modest. Yet there are several aspects in this work that are novel and may lead to future research:

1) As far as we know this is the first work that studies the problem of deciding disjunctive linear arithmetic from the perspective of computational geometry;[7]
2) It is the first work in the context of DPLL(T) that lets the theory guide the Boolean search directly, i.e., not by adding new clauses;
3) It is the first work in this context that considers the problem of using conjectured information without losing soundness.

As discussed in Sect. III-A, the number of cells is exponential in the number of variables, whereas the number of Boolean assignments is exponential in the number of predicates. Since the former is typically much smaller than the latter (see

[6] Since the improvement is small, one may wonder if a random choice of the value cannot compete with such results. We ran such a test and the result was much worse than the baseline: 78070.6 sec and 172 solved instances.

[7] An exception is the recently published technical report [11] that we mentioned earlier. Given a CNF-style formula, the authors suggest to check if its negation – in DNF – is valid. Each term in this DNF represents a polygon, and checking whether the whole formula is valid corresponds to checking whether the union of these polygons covers R^d. Doing so efficiently is the subject of the above reference: they consider the polygons one at a time in a random order, and maintain their union incrementally.

Table I), it raises the question whether there is a way to build an efficient SMT solver that exploits this fact. An explicit traversal of the linear arrangement does not seem a reasonable direction, but perhaps there is a way to build efficiently a symbolic representation of the cells in an arrangement – this would enable us to build a solver in which the theory leads the search rather than the SAT solver. In other words, in the current DPLL(T) framework the SAT solver leads the search: SAT suggests an assignment, and the theory solver checks it. Also, only the SAT solver can declare the formula unsatisfiable. This will change if we can find a method in which the linear space is traversed rather than the Boolean one. This will open a new research direction of finding decision heuristics in the linear domain, i.e., choosing which cell should be traversed next.

Acknowledgement

We thank Esther Ezra for her advice on various computational geometry issues.

REFERENCES

[1] G. Audemard, P. Bertoli, A. Cimatti, A. Kornilowicz, and R. Sebastiani. A SAT based approach for solving formulas over boolean and linear mathematical propositions. In *Proc. 18th International Conference on Automated Deduction (CADE'02)*, 2002.

[2] A. Baker. *Intelligent Backtracking on Constraint Satisfaction Problems: Experimental and Theoretical Results*. PhD thesis, Univ. of Oregon, 1995.

[3] Clark Barrett, Leonardo de Moura, and Aaron Stump. Design and results of the 1^{st} satisfiability modulo theories competition (SMT-COMP 2005). *The Journal of Automated Reasoning*, 35:373–390, 2005.

[4] Clark Barrett and Cesare Tinelli. CVC3. In *Computer Aided Verification (CAV)*, pages 298–302, 2007.

[5] K. L. Clarkson. New applications of random sampling in computational geometry. *Discrete Comput. Geom.*, 2(195–22), 1987.

[6] Leonardo de Moura and Nikolaj Bjorner. Z3: An efficient smt solver. In *Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2008.

[7] David Detlefs, Greg Nelson, and James B. Saxe. Simplify: a theorem prover for program checking. *J. ACM*, 52(3), 2005.

[8] Bruno Dutertre and Leonardo de Moura. A fast linear-arithmetic solver for DPLL(T). In *Proc. 18^{th} Intl. Conference on Computer Aided Verification (CAV'06)*, volume 4144 of *Lect. Notes in Comp. Sci.*, pages 81–94. Springer, 2006.

[9] Bruno Dutertre and Leonardo de Moura. Integrating simplex with DPLL(T). Technical Report SRI-CSL-06-01, Stanford Research Institute (SRI), 2006.

[10] Herbert Edelsbrunner. *Algorithms in combinatorial geometry*. Springer-Verlag New York, Inc., New York, NY, USA, 1987.

[11] E. Ezra and S. Fine. On the cover of convex polyhedra in d-space. Technical Report H-0259, IBM Haifa Research Lab, Israel, 2008.

[12] H. Ganzinger, G. Hagen, R. Nieuwenhuis, A. Oliveras, and C. Tinelli. DPLL(T): Fast decision procedures. In *Proc. 16^{th} Intl. Conference on Computer Aided Verification (CAV'04)*, number 3114 in Lect. Notes in Comp. Sci., pages 175–188. Springer, 2004.

[13] D. Haussler and E. Welzl. Epsilon-nets and simplex range queries. *Computational Geometry*, 2:127 – 151, 1987.

[14] Dorit S. Hochbaum, editor. *approximation-algorithms for NP-hard problems*. PWS Publishing Company, 1997.

[15] Daniel Kroening and Ofer Strichman. *Decision procedures – an algorithmic point of view*. Theoretical computer science. Springer-Verlag, May 2008.

[16] Filip Maric and Predrag Janicic. argo-lib: A generic platform for decision procedures. In *IJCAR*, pages 213–217, 2004.

[17] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Proc. Design Automation Conference (DAC'01)*, 2001.

[18] Knot Pipatsrisawat and Adnan Darwiche. Rsat 2.0: Sat solver description. SAT competition'07, 2007.

[19] Ofer Shtrichman. Tuning SAT checkers for bounded model checking. In E.A. Emerson and A.P. Sistla, editors, *Proc. 12^{th} Intl. Conference on Computer Aided Verification (CAV'00)*, Lect. Notes in Comp. Sci. Springer-Verlag, 2000.

[20] A. Stump, C. Barrett, and D. Dill. CVC: a cooperating validity checker. In *Proc. 14^{th} Intl. Conference on Computer Aided Verification (CAV'02)*, 2002.

[21] C. Tinelli. A DPLL-based calculus for ground satisfiability modulo theories. In *Proc. 8-th European Conference on Logics in Artificial Intelligence*, volume 2424, pages 308–319. Springer, 2002.

A Write-Based Solver for SAT Modulo the Theory of Arrays

Miquel Bofill
Universitat de Girona

Robert Nieuwenhuis
Albert Oliveras
Enric Rodríguez-Carbonell
and Albert Rubio
Universitat Politècnica de Catalunya

***Abstract*—The extensional theory of arrays is one of the most important ones for applications of SAT Modulo Theories (SMT) to hardware and software verification.**

Here we present a new T-solver for arrays in the context of the DPLL(T) approach to SMT. The main characteristics of our solver are: (i) no translation of writes into reads is needed, (ii) there is no axiom instantiation, and (iii) the T-solver interacts with the Boolean engine by asking to split on equality literals between indices.

Unlike most state-of-the-art array solvers, it is not based on a lazy instantiation of the array axioms. This novelty might make it more convenient to apply this solver in some particular environments. Moreover, it is very competitive in practice, specially on problems that require heavy reasoning on array literals.

I. Introduction

Over the last few years, traditional generic proof-search methods have been increasingly replaced by more efficient domain-specific procedures or procedures for fragments of certain logics. Despite being more specific, these procedures have seen their way into many real-world applications since most problems can be decomposed, either manually or automatically, into smaller problems to which these concrete procedures can be applied.

Among them, *Satisfiability Modulo Theories* (SMT) tools have received special attention. These procedures decide the satisfiability of (usually ground) formulas modulo some background theory T. The choice of the theory depends on the particular application area: when verifying complex designs, the theory of *Equality with Uninterpreted Functions (EUF)* comes in handy since it allows one to abstract away uninteresting or too complicated details of the system to be verified [1]; if, on the other hand, one is interested in verifying low-level aspects of, e.g., a microprocessor, the theory of *bit-vectors* provides the necessary level of detail [2], [3]; and, just to give another example, for the verification of timed automata a certain fragment of linear arithmetic called *Difference Logic* is the most appropriate choice [4]. Hence, SMT tools deal with problems that may consist of thousands of clauses like:

$p \ \vee \ a{=}f(b{-}c) \ \vee \ read(s,\, f(b{-}c)\,){=}d \ \vee \ a{-}g(c) \ {\leq}7$

containing purely propositional atoms as well as atoms over (combined) theories.

One of the main approaches to SMT has been called DPLL(T) [5], which consists of a general DPLL(X) engine, very similar in nature to a SAT solver, that works in cooperation with a theory solver $Solver_T$. In the most basic version of this approach the engine is in charge of enumerating propositional models of the formula, whereas $Solver_T$ is responsible for checking whether these models are consistent with the theory T (e.g. if T is the theory of linear arithmetic and the current Boolean model contains $x{+}2y \leq 0$, $-y - \frac{1}{2}z \leq 0$ and $x - z > 1$, then $Solver_T$ has to detect that the current assignment is T-inconsistent).

In this paper, we present a new $Solver_T$ for the *Theory of Arrays with Extensionality*. This theory is useful for both software and hardware verification, since it can be used to model the behavior of, e.g., arrays and memories. The signature of the theory consists of two interpreted function symbols: *read*, used to retrieve the element stored at a certain index of the array, and *write*, used to modify an array by updating the element stored at a certain index. More formally, their behavior can be modeled with the following axioms:

$$\begin{array}{l} \forall a : Array, \forall i, j : Index, \forall x : Value \\ \quad i{=}j \quad \Rightarrow \quad read(write(a,i,x),j) = x \\ \quad i{\neq}j \quad \Rightarrow \quad read(write(a,i,x),j) = read(a,j) \end{array}$$

Finally, one still has to consider the extensionality property, stating that two arrays that coincide on all indices are indeed equal. This is enforced by the axiom:

$$\begin{array}{l} \forall a, b : Array \\ \quad (\forall i : Index \ \ read(a,i) = read(b,i)) \quad \Rightarrow \quad a = b \end{array}$$

A possible approach to decide the satisfiability of ground literals in the theory of arrays with extensionality is to consider the necessary instances of the aforementioned axioms and let them be handled by the DPLL(X) engine in cooperation with a $Solver_T$ for EUF. In this setting, the array solver is reduced to a module that only generates clauses. This approach is used in state-of-the art SMT solvers like Yices [6], Z3 [7] and DPT [8] but, until very recently [8], there was no precise description of it in the literature. Others have focused on array problems at the bit-vector level, providing ways of encoding problems involving arrays of bit-vectors into SAT [9], [10].

Our approach does not follow the same lines but is rather based on a careful analysis of the problem that allows us to infer, for each array, which are the relevant indices and which values are stored at these indices. Once this information has

978-1-4244-2735-2/08/$25.00 ©2008 IEEE

been made transparent, checking satisfiability of sets of literals becomes an easy task. Unlike the approach of [11], we do not remove significant *write*'s from the problem, and hence there is no need to use the notion of "equality of arrays except in a certain set of indices" used in [11].

As usual, the extensionality axiom is addressed by forcing that, if two arrays are different, there is an index, called a *disequality witness*, where the two arrays do not coincide. For efficiency reasons, the introduction of these witnesses is delayed so as to reduce the combinatorial explosion caused by the comparison between indices. This explosion is also mitigated in our solver by identifying situations where the set of literals can be proved to admit a model without the need to construct a total partition over the sets of values and indices.

These ideas lead to a very natural and easy-to-understand solver. As a result, all proofs also become noticeably intuitive. Moreover, using the adequate strategy, our solver performs very well on problems that require heavy reasoning on array literals.

The rest of the paper is structured as follows. In Section II we give basic definitions and notation. Then, in Section III we describe our solver in terms of transformation rules and prove their correctness. After that, in Section IV we give some details of the integration of the solver in a DPLL(T) system. Finally, we present experimental results in Section V and we conclude in Section VI.

II. Preliminaries

We will consider a set of index constants denoted by $\mathcal{I}$ and a set of value constants denoted by $\mathcal{V}$, not necessarily disjoint. An array expression is either an array constant or $write(A, i, x)$ where A is an array expression, i is an index constant and x is a value constant. Arrays of the form $write(\ldots write(a, i_1, x_1) \ldots, i_n, x_n)$ will be written as $write(a, \langle i_1, \ldots, i_n\rangle, \langle x_1, \ldots, x_n\rangle)$, and as $write(a, \vec{I}, \vec{X})$ for short. Note that the first elements of $\vec{I}$ and $\vec{X}$ correspond to the innermost *write* and the last elements to the outermost *write*. We are using vector notation for $\vec{I}$ and $\vec{X}$ to emphasize that the elements are ordered and to refer to the elements at position p as i_p and x_p, respectively. Given $\vec{I}$, we will write I to denote the set of elements in $\vec{I}$. As a convention, the array constant a will be seen, when necessary, as $write(a, \emptyset, \emptyset)$.

In what follows, possibly subscripted a, b, c denote array constants and A, B, C denote array expressions. When we write $\equiv$ we mean syntactic equality between expressions and $=_{set}$ denotes equality between sets.

An *array satisfaction (ASAT) problem* is a conjunction of literals of the form:

- $\bot$ (representing an unsatisfiable formula).
- $x = read(A, i)$ or $x \neq read(A, i)$, where A is an array expression, $i \in \mathcal{I}$ and $x \in \mathcal{V}$.
- $A = B$ or $A \neq B$ where A and B are array expressions.
- $x = y$ or $x \neq y$, where $x, y \in \mathcal{I}$ or $x, y \in \mathcal{V}$.

A. The Extensional Theory of Arrays

The extensional theory of arrays is defined by the following axioms:

read-write axioms:

$$\begin{array}{l}\forall a : Array, \forall i, j : Index, \forall x : Value\\ \quad i{=}j \quad \Rightarrow \quad read(write(a, i, x), j) = x\\ \quad i{\neq}j \quad \Rightarrow \quad read(write(a, i, x), j) = read(a, j)\end{array}$$

extensionality axiom:

$$\begin{array}{l}\forall a, b : Array\\ \quad (\forall i : Index \;\; read(a, i) = read(b, i)) \quad \Rightarrow \quad a = b\end{array}$$

The following well-known axioms are not necessary if we consider extensionality, since they are entailed by the previous axioms:

write-write axioms:

$$\begin{array}{l}\forall a : Array, \forall i, j : Index, \forall x, y : Value\\ \quad i{=}j \Rightarrow write(write(a, i, x), j, y){=}write(a, j, y)\\ \quad i{\neq}j \Rightarrow write(write(a, i, x), j, y){=}write(write(a, j, y), i, x)\end{array}$$

Note that, however, the axioms of read-write and write-write do not entail the extensionality axiom.

Example 1: Let us consider the following set of literals:

$$\begin{array}{ccc} write(a, i, x){\neq}b & read(b, i){=}y & read(write(b, i, x), j){=}y\\ a{=}b & & i{=}j\end{array}$$

The two literals on the left imply that $read(b, i) \neq x$. This can be seen by replacing a by b in the first disequality, applying a $read$ at position i at both sides of the disequality and using the first read-write axiom. The two literals on the right imply that $x = y$. This can be deduced if we use that $i = j$ and apply the first read-write axiom at the top-right hand equation. Together with the equation in the middle we get that $read(b, i) = x$ which shows that the set of literals is T-inconsistent. □

B. Models

The models we will consider are given by a mapping I_I from $\mathcal{I}$ to some domain $D_{\mathcal{I}}$, a mapping I_V from $\mathcal{V}$ to some domain $D_{\mathcal{V}}$ and a mapping I_A from array constants to functions $f_A : D_{\mathcal{I}} \longrightarrow D_{\mathcal{V}}$, s.t. if $I_A(a) = f_a$ then $I_A(write(a, i, x)) = f'_a$ where $f'_a(I_I(i)) = I_V(x)$ and $f'_a(k) = f_a(k)$ for all $k \in D_{\mathcal{I}}$ such that $k \neq I_I(i)$. Finally, the interpretation on values is extended to the $read$ operation by the equality $I_V(read(a, i)) = I_A(a)(I_I(i))$.

Let M be a set of equalities and disequalities over $\mathcal{I}$ or $\mathcal{V}$. A position p in $\vec{I}$ is an *M-significant position in $\vec{I}$* if $M \models i_p \neq i_k$ for all $k > p$. The *set of M-significant indices in $\vec{I}$* is denoted by $sig_M(\vec{I})$ and is defined as the set of those indices i_p in I such that p is an M-significant position in $\vec{I}$.

Example 2: Let $M = \{j \neq l, l = k\}$ and $\vec{I} = (i, j, k, l)$. The set $sig_M(\vec{I})$ is $\{j, l\}$. Note that k is not an M-significant index because $M \models k{=}l$ and i is not an M-significant index because, among other reasons, $M \not\models i{\neq}j$. If we now take M and $\vec{I'} = (i, j, k, l, i)$ then i is now an M-significant index in $\vec{I'}$ because of its last occurrence. □

The pairs $(\vec{I}, \vec{X})$ and $(\vec{J}, \vec{Y})$ are said to be M-equivalent, if (i) for all $i, i' \in I$ (similarly for J), we have $M \models i = i'$ or $M \models i \neq i'$, (ii) $M \models sig_M(\vec{I}) =_{set} sig_M(\vec{J})$, and (iii) for all $i_p \in sig_M(\vec{I})$ and $j_q \in sig_M(\vec{J})$, if $M \models i_p = j_q$ then $M \models x_p = y_q$.

III. A New Solver for the Extensional Theory of Arrays

In this section we present, in terms of transformation rules, a solver for deciding the satisfiability of ASAT problems. Our transformation system deals with conjunctions of literals, but they will be represented as $M \mid P$, so as to explicitly distinguish the subset M of literals than can be treated by a simple Union-Find algorithm, i.e., (dis)equalities between indices or values, from the conjunction P of those literals involving arrays. As expected, $M \mid P$ has to be understood as $M \wedge P$. Given a conjunction of literals P, by $P\{a := A\}$ we mean the result of replacing all occurrences of a by A in P.

Definition 1: The system of transformation rules in Figure 1 is called $\mathcal{A}$.

Let us briefly comment on some of the rules. First of all, for rules like **True equality**, **Array inconsistency** or **Disequality witness introduction**, recall that an array constant a can be expressed as $write(a, \emptyset, \emptyset)$.

The **Significance query** rule is used to detect the set of significant indices in an array expression, whereas the **Equality index query** rule is used to detect equal indices in both sides of an equality.

In the **Write introduction** rule, by propagating $b := write(c, i, x)$ in the conclusion we make explicit that b is an array with x at position i. Note that we could keep that literal in order to be able to later recover a model for the original set of array constants. This is also the case for the **Substitution** rule.

Finally, let us mention that the **Disequality witness introduction** rule takes care of extensionality: if we have $write(a, \vec{I}, \vec{X}) \neq write(b, \vec{J}, \vec{Y})$ and we know that the difference is not in the significant indices in $\vec{I}$ (which coincides with significant indices in $\vec{J}$), then we can infer that a and b are different at some position not in $\vec{I}$. The introduction of these witnesses can be delayed at convenience. Similarly to **Write introduction** and **Substitution** we could keep the two substitutions as literals so as to later recover a model for the original problem.

As we will show below in all these rules the premise and the disjunction of the conclusions are equisatisfiable. Hence, they can be used to produce a *derivation tree* rooted by the original array problem and where every non-terminal node has as children the conclusions of an application of a transformation rule. As we will see, the problem will be unsatisfiable if and only if all terminal nodes are $\bot$.

Lemma 1: For all rules in $\mathcal{A}$ the premise and the disjunction of its conclusions are equisatisfiable.

Proof: We show that there is a model for the premise if and only if there is a model for the disjunction of the conclusions. We only detail the proof for the following two rules:

- **Write introduction**. For the left to right implication, let (I_I, I_V, I_A) be a model for $M \mid P \wedge write(a, \vec{I}, \vec{X}) = write(b, \vec{J}, \vec{Y})$. Since by the conditions of the transformation rule, $I_I(i_p) \neq I_I(i_k)$ for all $k > p$, and $I_I(i_p) \neq I_I(j)$ for every $j \in \vec{J}$, we have that $I_A(b)(I_I(i_p)) = I_V(x_p)$. Therefore, extending I_A to interpret the fresh constant c as $I_A(c) = f_c$, where $f_c(k) = (I_A(b))(k)$ for all $k \in D_{\mathcal{I}}$, we have a model for $M \mid P \wedge write(a, \vec{I}, \vec{X}) = write(b, \vec{J}, \vec{Y}) \wedge b = write(c, i_p, x_p)$ and hence also of the conclusion. For the right to left implication, since b does not appear in the conclusion we only have to extend the model so that b is interpreted as $write(c, i_p, x_p)$.
- **Array inconsistency**. In this case we show that there cannot be any model for $M \mid write(a, \vec{I}, \vec{X}) \neq write(a, \vec{J}, \vec{Y})$ if $(\vec{I}, \vec{X})$ and $(\vec{J}, \vec{Y})$ are M-equivalent. Assume we have a model for M. Then $I_I(sig_M(\vec{I}))$ is equal to $I_I(sig_M(\vec{J}))$ and for all M-significant $i_p \in \vec{I}$ and $j_q \in \vec{J}$, if $I_I(i_p) = I_I(j_q)$ then $I_V(x_p) = I_V(y_q)$, which implies that any array interpretation must interpret $write(a, \vec{I}, \vec{X})$ and $write(a, \vec{J}, \vec{Y})$ to the same function, and thus cannot satisfy $write(a, \vec{I}, \vec{X}) \neq write(a, \vec{J}, \vec{Y})$. □

■

Definition 2: An ASAT problem is in *solved form* if either it is $\bot$ or it is $M \mid P$ where M is consistent and all literals in P are of the form:

1) $write(a, \vec{I}, \vec{X}) = write(b, \vec{J}, \vec{Y})$
   with $a \not\equiv b$, and $(\vec{I}, \vec{X})$ and $(\vec{J}, \vec{Y})$ being M-equivalent.
2) $write(a, \vec{I}, \vec{X}) \neq write(b, \vec{J}, \vec{Y})$
   with some M-significant position p in $\vec{I}$ such that $M \not\models i_p = j$ for every $j \in \vec{J}$.
3) $write(a, \vec{I}, \vec{X}) \neq write(b, \vec{J}, \vec{Y})$
   with some M-significant position p in $\vec{I}$ and some M-significant position q in $\vec{J}$ such that $M \models i_p = j_q$ and $M \not\models x_p = y_q$.
4) $x \neq read(a, i)$
   being a an array constant.

Lemma 2: An ASAT problem in solved form is satisfiable if and only if it is not $\bot$.

Proof: The left to right implication is trivial. For the other one, let us consider a problem $M \mid P$ in solved form. First of all, we introduce a new value constant d. Then, we consider $M' = M \cup \{x \neq y \mid M \not\models x = y\}$ that, since M is consistent and equality is a convex theory[1], is also consistent. Essentially, we have extended M by adding that all what is not known to be equal is set to distinct (including all comparisons with the new constant d).

Now, let (I_I, I_V) be a model of M'. We call $f_d : D_{\mathcal{I}} \to D_{\mathcal{V}}$ the constant function that always returns $I_V(d)$. Note that, by

[1]This means that if E is a set of equalities and disequalities, $E \models x_1 = y_1 \vee \ldots \vee x_n = y_n$ iff $E \models x_i = y_i$ for some i in $1 \ldots n$.

UF-Dispatch:

$$\frac{M \mid x \bowtie y \wedge P}{M \cup \{x \bowtie y\} \mid P}$$

with $\bowtie \in \{=, \neq\}$

Read2Write:

$$\frac{M \mid P[x = read(a, i)]}{M \mid P[a = write(b, i, x)]}$$

if a and b are array constants and b is *fresh*.

Read-Write:

$$\frac{M \mid P[read(write(A, i, x), j)]}{M \cup \{i = j\} \mid P[x] \qquad M \cup \{i \neq j\} \mid P[read(A, j)]}$$

Substitution:

$$\frac{M \mid P \wedge a = A}{M \mid P\{a := A\}}$$

if a does not occur in A.

True equality:

$$\frac{M \mid P \wedge write(a, \vec{I}, \vec{X}) = write(a, \vec{J}, \vec{Y})}{M \mid P}$$

if $(\vec{I}, \vec{X})$ and $(\vec{J}, \vec{Y})$ are M-equivalent.

Array inconsistency:

$$\frac{M \mid P \wedge write(a, \vec{I}, \vec{X}) \neq write(a, \vec{J}, \vec{Y})}{\bot}$$

if $(\vec{I}, \vec{X})$ and $(\vec{J}, \vec{Y})$ are M-equivalent.

UF-inconsistency:

$$\frac{M \mid P}{\bot} \qquad \text{if } M \text{ is inconsistent}$$

Write introduction:

$$\frac{M \mid P \wedge write(a, \vec{I}, \vec{X}) = write(b, \vec{J}, \vec{Y})}{M \mid (P \wedge write(a, \vec{I}, \vec{X}) = write(b, \vec{J}, \vec{Y}))\{b := write(c, i_p, x_p)\}}$$

if c is *fresh* and p is an M-significant position in $\vec{I}$ and $M \models i_p \neq j$ for every $j \in \vec{J}$.

Significance query:

$$\frac{M \mid P[write(write(A, \vec{I}, \vec{X}), j, y)]}{M \cup \{i = j\} \mid P[write(write(A, \vec{I}, \vec{X}), j, y)] \quad M \cup \{i \neq j\} \mid P[write(write(A, \vec{I}, \vec{X}), j, y)]}$$

if $M \not\models i = j$ and $M \not\models i \neq j$ for some $i \in \vec{I}$.

Equality index query:

$$\frac{M \mid P \wedge write(a, \vec{I}, \vec{X}) = write(b, \vec{J}, \vec{Y})}{M \cup i_p = j_q \mid P \wedge write(a, \vec{I}, \vec{X}) = write(b, \vec{J}, \vec{Y}) \quad M \cup i_p \neq j_q \mid P \wedge write(a, \vec{I}, \vec{X}) = write(b, \vec{J}, \vec{Y})}$$

if p is an M-significant position in $\vec{I}$, q is an M-significant position in $\vec{J}$ and $M \not\models i_p = j_q$ and $M \not\models i_p \neq j_q$.

Equality values propagation:

$$\frac{M \mid P \wedge write(a, \vec{I}, \vec{X}) = write(b, \vec{J}, \vec{Y})}{M \cup x_p = y_q \mid P \wedge write(a, \vec{I}, \vec{X}) = write(b, \vec{J}, \vec{Y})}$$

if p is an M-significant position in $\vec{I}$, q is an M-significant position in $\vec{J}$ and $M \models i_p = j_q$ and $M \not\models x_p = x_q$.

Disequality witness introduction:

$$\frac{M \mid P \wedge write(a, \vec{I}, \vec{X}) \neq write(b, \vec{J}, \vec{Y})}{M' \mid P\{a := write(c, ni, e_1), b := write(d, ni, e_2)\}}$$

where $M' = M \cup \{ni \neq i \mid i \in \vec{I}\} \cup \{ni \neq j \mid j \in \vec{J}\} \cup \{e_1 \neq e_2\}$
if $a \not\equiv b$, $(\vec{I}, \vec{X})$ and $(\vec{J}, \vec{Y})$ are M-equivalent and c, d, ni, e_1 and e_2 are *fresh*.

Fig. 1. Transformation rule system $\mathcal{A}$

assumption, $I_V(d)$ is different from the interpretation of all other value constants. Now, we define our array interpretation I_A as $I_A(a) = f_d$ for every array constant a. Note that, if in the rules **Substitution**, **Write Introduction** and **Disequality Witness Introduction** we had kept the substitution as a literal of the form $a = A$, where a does not occur elsewhere in P, then we would only need to extend I_A with $I_A(a) = I_A(A)$.

Let us show that (I_I, I_V, I_A) is a model of all literals in P.

1) The literal is of the form $write(a, \vec{I}, \vec{X}) = write(b, \vec{J}, \vec{Y})$, where $a \not\equiv b$, and $(\vec{I}, \vec{X})$ and $(\vec{J}, \vec{Y})$ are M-equivalent. Then, by definition of I_A we have $I_A(a) = I_A(b)$. Now, since if $(\vec{I}, \vec{X})$ and $(\vec{J}, \vec{Y})$ are M-equivalent then they are also M'-equivalent, we have that $I_I(sig_{M'}(\vec{I}))$ is equal to $I_I(sig_{M'}(\vec{J}))$. Moreover, for all M'-significant $i_p \in \vec{I}$ and $j_q \in \vec{J}$ such that $I_I(i_p) = I_I(j_q)$ then $I_V(x_p) = I_V(y_q)$, which implies that $I_A(write(a, \vec{I}, \vec{X})) = I_A(write(b, \vec{J}, \vec{Y}))$.
2) The literal is of the form $write(a, \vec{I}, \vec{X}) \neq write(b, \vec{J}, \vec{Y})$ where there is some M-significant position p in $\vec{I}$ such that $M \not\models i_p = j$ for every $j \in \vec{J}$. Then, since $M \not\models i_p = j$, we have that $M' \models i_p \neq j$, and hence $I_I(i_p) \neq I_I(j)$ for all j in $\vec{J}$. Now, by definition, we have that $I_A(write(a, \vec{I}, \vec{X}))$ is some function f_1 such that $f_1(I_I(i_p)) = I_V(x_p)$ and, since $I_A(b) = f_d$ and $I_I(i_p) \neq I_I(j)$ for all $j \in \vec{J}$, we have that $I_A(write(b, \vec{J}, \vec{Y}))$ is some function f_2 such that $f_2(I_I(i_p)) = I_V(d) \neq I_V(x_p)$, which implies $f_1 \neq f_2$, satisfying the literal.
3) The literal is of the form $write(a, \vec{I}, \vec{X}) \neq write(b, \vec{J}, \vec{Y})$ where there is some M-significant position p in $\vec{I}$ and some M-significant position q in $\vec{J}$ such that $M \models i_p = j_q$ and $M \not\models x_p = y_q$. Then, by definition, we have $M' \models i_p = j_q$ and $M' \models x_p \neq y_q$. Now, similarly to the previous case, we have that $I_A(write(a, \vec{I}, \vec{X}))$ is some function f_1 such that $f_1(I_I(i_p)) = I_V(x_p)$ and $I_A(write(b, \vec{J}, \vec{Y}))$ is some function f_2 such that $f_2(I_I(j_q)) = I_V(y_q)$. Therefore, since $I_I(i_p) = I_I(j_q)$ and $I_V(x_p) \neq I_V(y_q)$, we have $f_1 \neq f_2$, satisfying the literal.
4) The literal is of the form $x \neq read(a, i)$. Then, by definition, we have that $M' \models x \neq d$, and hence $I_V(x) \neq I_V(d)$. Moreover, since $I_A(a) = f_d$, then $I_V(read(a, i)) = f_d(I_I(i)) = I_V(d) \neq I_V(x)$, hence satisfying the literal. □

■

Definition 3: An *$\mathcal{A}$-derivation tree* is a tree whose nodes are ASAT problems and where the children of a node $M \mid P$ are the conclusions of a transformation rule in $\mathcal{A}$ applied to $M \mid P$. We will say that an $\mathcal{A}$-derivation tree is *solved* iff all its leafs are solved forms.

Lemma 3: All paths in an $\mathcal{A}$-derivation tree are finite.

Proof: First of all we show that given an ASAT problem $M \mid P$, the set of indices and values that can occur in any $\mathcal{A}$-derivation tree rooted by $M \mid P$ is finite. This is easy to see since the only rule that introduces new constants is **Disequality witness introduction**, but since every application of this rule removes a disequality literal and no other rule introduces new disequalities, this can only happen a finite number of times. Hence, this also proves that this rule cannot be applied an infinite number of times.

Regarding the rules **Significance query**, **Equality index query** and **Equality values propagation**, they can only be applied a finite number of times. This is because they add a (dis)equality between indices or values to M that was not previously entailed by it. Since there are only a finite number of indices and values, this can only happen finitely often.

Similarly, the **Write introduction** rule can only be applied a finite number of times. The key argument is based on the two following facts: (i) the number of literals appearing in any node of the derivation tree does not increase, since no rule adds new literals and (ii) no rule removes $write$'s from an equality between array expressions. Given these two facts, if we denote by N_I the maximum number of indices that can occur in the derivation tree, we know that for every equality literal in the original problem, we can apply **Write Introduction** at most $2 * N_I$ times, since after $2 * N_I$ applications all indices will appear in both sides of the equality and the rule will no longer be applicable.

Hence, any infinite derivation should end with an infinite sequence of applications of **UF-Dispatch**, **Read-Write**, **Read2Write**, **Substitution** and **True equality**. But this cannot be the case: given an ASAT problem $M \mid P$, we can associate to it the triple of natural numbers $(\#\text{literals in } P, \#reads, |P|)$, being $|P|$ the size of P. It is an easy exercise to check that using a lexicographic ordering all remaining rules decrease that triple of natural numbers.

■

The following lemma is easily proved by comparing the conditions imposed on solved forms and the conditions imposed on the rules.

Lemma 4: If an ASAT problem is not a solved form then a transformation rule in $\mathcal{A}$ can be applied.

Proof: If M is not consistent then we can apply the **UF-inconsistency** rule. Otherwise, we proceed by case analysis according to the kind of literal that is not in solved form.

1) The literal is of the form $a = A$. If a does not occur in A then the **Substitution** rule can be applied. Otherwise, if $A \equiv a$ then **True equality** can be applied. Finally, if A is $write(a, \vec{I}, \vec{X})$ with $\vec{I} \neq \emptyset$, we can apply **Write introduction** for the last index i_n in $\vec{I}$.
2) The literal is of the form $write(a, \vec{I}, \vec{X}) = write(b, \vec{J}, \vec{Y})$, with $\vec{I}$ and $\vec{J}$ non-empty. If there is a pair of indices i_p and i_q in $\vec{I}$ (or in $\vec{J}$) such that $M \not\models i_p = i_q$ and $M \not\models i_p \neq i_q$, we can apply the **Significance query** rule. Otherwise, if there is some M-significant position p in $\vec{I}$ and some M-significant position q in $\vec{J}$ s.t. $M \not\models i_p = j_q$ and $M \not\models i_p \neq j_q$ then we can apply the **Equality index query** rule. Otherwise, if $(\vec{I}, \vec{X})$ and $(\vec{J}, \vec{Y})$ are not M-equivalent it is because either (i) there is some M-significant index i_p in $\vec{I}$ such that $M \models i_p \neq j$ for all j in $\vec{J}$ and we can apply **Write**

introduction (the same happens if we exchange $\vec{I}$ and $\vec{J}$) or (ii) $M \models sig_M(\vec{I}) =_{set} sig_M(\vec{J})$ but there exist an M-significant position p in $\vec{I}$ and an M-significant position q in $\vec{J}$ such that $M \models i_p = j_q$ but $M \not\models x_p = y_p$ and then we can apply **Equality values propagation**. The only possibility left now is that $(\vec{I}, \vec{X})$ and $(\vec{J}, \vec{Y})$ are indeed M-equivalent and then either the literal is in solved form or we can apply **True equality**.

3) The literal is of the form $write(a, \vec{I}, \vec{X}) \neq write(b, \vec{J}, \vec{Y})$. If there is a pair of indices i_p and i_q in $\vec{I}$ (or in $\vec{J}$) such that $M \not\models i_p = i_q$ and $M \not\models i_p \neq i_q$, we can apply the **Significance query** rule. Otherwise, if the literal is not in solved form, then $(\vec{I}, \vec{X})$ and $(\vec{J}, \vec{Y})$ are M-equivalent, and hence we can either apply the **Array Inconsistency** rule when $a \equiv b$, or we can apply the **Disequality witness introduction** otherwise.
4) The literal is of the form $x = read(A, i)$. If A is not a constant then we can apply the **Read-Write** rule. Otherwise, we can apply the **Read2Write** rule.
5) The literal is of the form $x \neq read(A, i)$ and A is not a constant. Then we can apply the **Read-Write** rule.
6) The literal is a (dis)equality between indices or values. Then, we can apply the **UF-Dispatch rule**. □

■

Since the premises and the disjunction of the conclusions of every transformation rule are equisatisfiable, the following lemma holds.

Lemma 5: An ASAT problem $M \mid P$ is satisfiable if and only if at least one leaf of a solved $\mathcal{A}$-derivation tree rooted by $M \mid P$ is not $\perp$.

Now we present our main result.

Theorem 1: The extensional theory of arrays can be decided with $\mathcal{A}$.

Proof: By lemmas 3 and 4, we have that, given a ASAT problem $M \mid P$, a solved $\mathcal{A}$-tree derivation can be obtained. Then by Lemma 5, we can decide if the problem is satisfiable. □

■

Note that we can apply any strategy in the application of the transformation rules.

IV. Integration of the Solver in DPLL(T)

In the previous section, we showed how to check the satisfiability of conjunctions of literals, but SMT deals with arbitrary formulas. For that purpose, in the DPLL(T) approach to SMT a Boolean engine DPLL(X) works in cooperation with a theory solver $Solver_T$. In its most basic version, the engine enumerates all propositional models of the formula and $Solver_T$ checks the models (seen as conjunctions of literals) for consistency over the theory T. As expected, the input formula is declared satisfiable as soon as a T-consistent propositional model is found.

In our implementation of the $Solver_T$ described in this paper, array expressions are stored using a DAG, where the nodes are array constants and the edges are labelled by the index and the value of the corresponding $write$. In this way we can share information and have an efficient way of applying subtitutions of array constants by $write$ expressions. In addition, (dis)equalities between constants are stored in a Union-Find data structure. In what follows, we give some details about usual optimizations on DPLL(T) systems and how they are implemented in our solver.

Incrementality: there is no need to delay consistency checks until a full propositional model has been found. One can check the T-consistency of partial assignments while they are being built, with the aim of detecting T-inconsistencies at an earlier stage. In order to fully exploit this feature, it is interesting to ask $Solver_T$ to be incremental. That is, once an assignment M has been found T-consistent, processing the addition of a literal l should be done faster (in average) than reprocessing the whole assignment $M \cup \{l\}$ from scratch.

For that purpose, for every array (dis)equality literal we have a *watched pair* of indices, one in each side, that is used for the analysis of satisfiability of the literal. If the literal is not in solved form and we know whether these two indices and their associated values are equal or not, and whether they are significant, we move the watched pair to other indices in order to avoid repeated work in future checks.

Splitting on demand: if some of this information is unknown, we allow the solver not to give a conclusive answer, but rather to ask DPLL(X) to split on a certain equality between indices. This refinement, presented in [12] and called *splitting on demand*, allows reusing the case-splitting infrastructure present in DPLL(X) instead of duplicating it inside $Solver_T$. This simplifies the implementation of all splitting rules presented in the previous section.

Theory propagation: if, when checking a non-solved literal, we have all the information about equalities between indices but not about values, we can propagate an equality between values, applying the **Equality values propagation** rule. Similarly, by successive applications of **Read-Write** we can sometimes infer a disequality between values that is propagated by **UF-Dispatch**. If such a (dis)equality already exists in the input formula we can notify it to the DPLL(X) engine. This optimization, introduced in [13], is very effective in reducing the search space.

Backtracking: sometimes an inconsistency can be detected, and then it is beneficial to backtrack to a point where the assignment was still T-consistent, instead of restarting the search. Hence, we need $Solver_T$ to be backtrackable. Our solver annotates some information with timestamps, e.g. the one given by the Union-Find, and some other information is restored using a trail stack.

In addition, $Solver_T$ has to assist DPLL(X) in identifying the backtrack point by providing an *inconsistency explanation*, that is, given a T-inconsistent set of literals M, it has to provide a small subset of M that is also T-inconsistent. As it is well-known, generating short explanations is a determinant factor in the performance of an SMT solver. In our case, an explanation describes basically the conditions of the transformation rule of Figure 1 that has been applied on a given

(dis)equality array literal, together with the explanation of why the relevant indexes and values of both sides of the array literal are there. For this reason, when any of the rules introducing new $write$'s is applied, namely the **Read2Write**, the **Write introduction** or the **Disequality witness introduction** rule, we remember the literal that has generated it. It is crucial to make these explanations for the introduced $write$'s as short as possible.

V. Experiments

In order to evaluate the $Solver_T$ for arrays described in this paper, we implemented it on top of our BARCELOGIC [14] system. We performed experiments on a 2GHz 2GB Intel Core Duo with a time limit of 300 seconds, comparing our implementation with the four systems that competed at SMT-COMP 2007 (the annual SMT competition[2]) in the QF_AUFLIA division, the only one involving quantifier-free formulas with arrays. These systems are: CVC3 1.2 [15], YICES 1.0 and YICES 1.0.10 [6] and Z3 0.1 [7]. We ran all systems on all available benchmarks in SMT-LIB [16], the largest existing library for SMT problems, discarding families of benchmarks consisting only of trivial problems[3]. The remaining benchmark families were:

- *array-benchs* (25 benchs): a variety of verification conditions involving arithmetic and arrays.
- *cvc* (25 benchs): processor verification conditions involving arithmetic and arrays.
- *qlock2* (52 benchs): unbounded version of the queue lock algorithm. All benchmarks result from parameterizing two single problems. They all contain arithmetic and arrays.
- *storecomm* (2030 benchs), *storeinv* (172 benchs) and *swap* (1368 benchs): benchmarks from the paper [17] encoding simple properties about arrays. They do not contain any arithmetic.

As we can see, most benchmarks involve both arrays and arithmetic, hence forcing us to implement some method for combining the $Solver_T$ for arrays with our solvers for arithmetic (both the difference logic one and the one for full linear arithmetic). It is important to note that BARCELOGIC does not implement any sophisticated combination technique. Unlike what it is done in YICES or Z3, where interface equalities are created on the fly and sophisticated techniques are used to reduce their number, BARCELOGIC implements Delayed Theory Combination [18]. This is much simpler but, since we create all interface equalities upfront, it may significantly slow down the search in some cases. The main reason for that is that, since our arithmetic solvers do not admit (dis)equalities, we have to add clauses, for each interface equality $x = y$, expressing that $x = y \leftrightarrow x \leq y \wedge x \geq y$. This problem is specially acute in the *qlock2* family where, in some cases, up to sixty thousand interface equalities had to be created.

[2]See `http://www.smtcomp.org`

[3]We made the experiments with all available benchmarks in April 2008

	YICES 1.0.10		YICES 1.0	
	Total	Max	Total	Max
array_benchs	52	42	69	52
cvc	5	4	4	3
qlock2	49	5	50	6
storecomm	35	0.1	41	0.1
storeinv	1	0.1	1	0.1
swap	970	130	581	60

	Z3 0.1		CVC3 1.2	
	Total	Max	Total	Max
array_benchs	21	8	496 (16)	294
cvc	1	1	114	57
qlock2	114	37	199 (30)	117
storecomm	37	0.1	993	20
storeinv	8	0.3	691 (162)	76
swap	1431	128	13726 (1263)	275

	BARCELOGIC	
	Total	Max
array_benchs	282	162
cvc	59	38
qlock2	652	55
storecomm	48	0.1
storeinv	22	2
swap	275	9

Fig. 2. Experimental results (times are in seconds)

Results are presented in Figure 2, where the column labeled *Total* contains the number of seconds needed to process the whole family. We only count the time for the number of instances solved within 300 seconds and, if not all problems could be handled, we write in parenthesis how many instances could be processed. The column *Max* gives the largest time in seconds needed by a single instance.

From the table it can be seen that BARCELOGIC can solve any instance in less than 3 minutes.In fact, only one benchmark takes more than one minute. Our system is in general much faster than CVC3, but slower than Z3 and YICES. However, apart from the *qlock2* family, where the huge number of interface equalities greatly affects the search space, the difference wrt. Z3 and YICES is similar to the difference we obtain when we run the systems on formulas not involving arrays. Hence, the difference is probably not due to the array solver but rather to other factors such as heuristics and the worse performance of the arithmetic solvers in BARCELOGIC. In fact, for benchmarks containing a big array component, such as the families *storecomm*, *storeinv* and *swap*, our system is comparable, if not better, which shows that our array solver behaves very well in practice. More controlled experiments trying to isolate the perfomance of our solver and compare it with other systems are not possible since, unlike other theory solvers, array solvers in SMT systems usually work

in cooperation with the SAT solver in multiple different ways.

VI. Conclusions

In this paper we have described a new theory solver for extensional arrays. As far as we know, this is the first accurate description of an array solver integrated in a state-of-the-art SMT solver and, unlike most state-of-the-art solvers, it is not based on a lazy instantiation of the array axioms. Moreover, our solver is very intuitive and easy-to-understand: after performing a careful analysis on which indices are relevant in each array, the satisfiability of conjunctions of literals becomes an easy task. We have proved soundness, completeness, and termination of our procedure and shown how it can be integrated to work in a DPLL(T) setting. Finally, we have presented experimental results showing that it performs very well in practice. We want to note that this approach smoothly extends to multidimensional arrays by expressing a position $(i_1, \ldots, i_n)$ in an n-dimensional array as $f(i_1, \ldots, i_n)$, where f is an uninterpreted function symbol.

References

[1] J. R. Burch and D. L. Dill, "Automatic Verification of Pipelined Microprocessor Control," in *6th International Conference on Computer Aided Verification, CAV'94*, ser. Lecture Notes in Computer Science, D. L. Dill, Ed., vol. 818. Springer, 1994, pp. 68–80.

[2] V. Ganesh and D. L. Dill, "A Decision Procedure for Bit-Vectors and Arrays," in *19th International Conference on Computer Aided Verification, CAV'07*, ser. Lecture Notes in Computer Science, W. Damm and H. Hermanns, Eds., vol. 4590. Springer, 2007, pp. 519–531.

[3] R. Bruttomesso, A. Cimatti, A. Franzén, A. Griggio, Z. Hanna, A. Nadel, A. Palti, and R. Sebastiani, "A Lazy and Layered SMT(BV) Solver for Hard Industrial Verification Problems," in *19th International Conference on Computer Aided Verification, CAV'07*, ser. Lecture Notes in Computer Science, W. Damm and H. Hermanns, Eds., vol. 4590. Springer, 2007, pp. 547–560.

[4] R. Alur, "Timed Automata," in *11th International Conference on Computer Aided Verification, CAV'99*, ser. Lecture Notes in Computer Science, N. Halbwachs and D. Peled, Eds., vol. 1633. Springer, 1999, pp. 8–22.

[5] R. Nieuwenhuis, A. Oliveras, and C. Tinelli, "Solving SAT and SAT Modulo Theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T)," *Journal of the ACM, JACM*, vol. 53, no. 6, pp. 937–977, 2006.

[6] B. Dutertre and L. de Moura, " The YICES SMT Solver," Computer Science Laboratory, SRI International, Tech. Rep., 2006, available at `http://yices.csl.sri.com`.

[7] L. de Moura and N. Bjorner, " Z3: An Efficient SMT Solver," Microsoft Research, Redmond, Tech. Rep., 2007, available at `http://research.microsoft.com/projects/z3`.

[8] S. K. A. Goel and A. Fuchs, "Deciding Array Formulas with Fruagal Axiom Instantiation," in *6th International Workshop on Satisfiability Modulo Theories, SMT'08*, Princeton, USA, 2008.

[9] R. Brummayer and A. Biere, "Lemmas on Demand for the Extensional Theory of Arrays," in *6th International Workshop on Satisfiability Modulo Theories, SMT'08*, Princeton, USA, 2008.

[10] P.Manolios, S. K. Srinivasan, and D. Vroon, "Automatic memory reductions for RTL model verification," in *International Conference on Computer-Aided Design, ICCAD'06*, S. Hassoun, Ed. ACM, 2006, pp. 786–793.

[11] A. Stump, C. W. Barrett, D. L. Dill, and J. R. Levitt, "A Decision Procedure for an Extensional Theory of Arrays," in *16th Annual IEEE Symposium on Logic in Computer Science, LICS'01*. IEEE Computer Society, 2001, pp. 29–37.

[12] C. Barrett, R. Nieuwenhuis, A. Oliveras, and C. Tinelli, "Splitting on Demand in SAT Modulo Theories," in *13h International Conference on Logic for Programming, Artificial Intelligence and Reasoning, LPAR'06*, ser. Lecture Notes in Computer Science, M. Hermann and A. Voronkov, Eds., vol. 4246. Springer, 2006, pp. 512–526.

[13] R. Nieuwenhuis and A. Oliveras, "DPLL(T) with Exhaustive Theory Propagation and its Application to Difference Logic," in *17th International Conference on Computer Aided Verification, CAV'05*, ser. Lecture Notes in Computer Science, K. Etessami and S. Rajamani, Eds., vol. 3576. Springer, 2005, pp. 321–334.

[14] M. Bofill, R. Nieuwenhuis, A. Oliveras, E. Rodríguez-Carbonell, and A. Rubio, "The Barcelogic SMT Solver," in *20th International Conference on Computer Aided Verification, CAV'08*, ser. Lecture Notes in Computer Science, A. Gupta and S. Malik, Eds. Springer, 2008.

[15] C. Barrett and C. Tinelli, "CVC3," in *19th International Conference on Computer Aided Verification, CAV'07*, ser. Lecture Notes in Computer Science, W. Damm and H. Hermanns, Eds., vol. 4590. Springer, 2007, pp. 298–302.

[16] C. Tinelli and S. Ranise, "SMT-LIB: The Satisfiability Modulo Theories Library," http://www.smtlib.org.

[17] A. Armando, M. P. Bonacina, S. Ranise, and S. Schulz, "New results on rewrite-based satisfiability procedures," *ACM Transactions on Computational Logic, TOCL*, 2008, to appear.

[18] M. Bozzano, R. Bruttomesso, A. Cimatti, T. A. Junttila, S. Ranise, P. van Rossum, and R. Sebastiani, "Efficient Satisfiability Modulo Theories via Delayed Theory Combination." in *17th International Conference on Computer Aided Verification, CAV'05*, ser. Lecture Notes in Computer Science, K. Etessami and S. Rajamani, Eds., vol. 3576. Springer, 2005, pp. 335–349.

Scaling up the formal verification of Lustre programs with SMT-based techniques

George Hagen
Department of Computer Science
The University of Iowa, USA
ghagen@cs.uiowa.edu

Cesare Tinelli
Department of Computer Science
The University of Iowa, USA
tinelli@cs.uiowa.edu

Abstract

We present a general approach for verifying safety properties of Lustre programs automatically. Key aspects of the approach are the choice of an expressive first-order logic in which Lustre's semantics is modeled very naturally, the tailoring to this logic of SAT-based k-induction and abstraction techniques, and the use of SMT solvers to reason efficiently in this logic. We discuss initial experimental results showing that our implementation of the approach is highly competitive with existing verification solutions for Lustre.

1. Introduction

Lustre is a synchronous dataflow language commonly used in a number of manufacturing industries to model or implement reactive embedded systems [16]. Because of its declarative nature and simplicity, it is well suited to model checking and verification, in particular with regards to safety properties [17]. Existing formal methods tools for Lustre, however, do not seem to have kept pace with the latest developments in software model checking and verification, and are somewhat lacking in scope and scale. Currently, those can be obtained only by a translation into the specification languages of other tools—such as NuSMV, SAL, Prover, PVS, ACL2, and so on.

This work aims at bridging the current gap. Its main ingredient is the choice of an expressive logic with efficient automated reasoners for modeling Lustre programs and proving safety properties for them. Instead of temporal reasoning based on *propositional* logic we rely on temporal reasoning based on a first-order logic with built-in theories for equality over uninterpreted symbols and linear mixed real and integer arithmetic.

An approximation of Lustre's denotational semantics can be modeled simply and naturally in this logic, with two distinctive advantages: i) infinite-state as well as finite-state Lustre programs can be encoded uniformly and *compactly* in the logic; ii) reasoning about these programs can be done directly in terms of Lustre's own dataflow model. This contrast with other approaches which are either mostly limited to finite-state programs or need a translation into some other specification language.

The idea of adopting logics of this sort in software model checking is not new [10, 1, 14, e.g.], and has been applied to the verification of relatively simple Lustre programs already in [13]. Recent advances in *Satisfiability Modulo Theories (SMT)* [4], however, make it now possible to apply this idea to a much wider range of real-world Lustre applications. Our use of SMT technology builds on previous research in SAT-based temporal induction (aka, k-induction) [21, 5], extended and adapted to the SMT case as in [10, 12], as well as on research in predicate abstraction and refinement.

Our main contribution is the combination of a selection of these techniques and their tailoring to Lustre. In particular, we extend SMT-based k-induction to incorporate abstraction and refinement, something that to our knowledge has not been reported before. The main outcome of our work is a new verification system that is highly competitive with current automated solutions for the verification of safety properties of Lustre programs.

In the following, we first sketch our approach, focusing on major aspects such as path compression and abstraction refinement. We then discuss an initial experimental evaluation of our implementation against current alternatives. This evaluation suggests that our new system considerably advances the state of the art, as it greatly outperforms current Lustre-specific verifiers. The study also shows that a direct approach like ours compares very favorably to translating Lustre programs into the language of a premier SMT-based model checker, SAL 3.0 [9], and using that checker.

Related Work. Our work is closest to Franzén's [13] which extends SAT-based k-induction to produce a safety property verifier for Lustre programs with unbounded integers. There, the extension is achieved by adding to the MiniSat

978-1-4244-2735-2/08/$25.00 ©2008 IEEE

SAT solver some simple ILP procedures for handling integer constraints. The focus of [13] was building the extended SAT solver. In contrast, we rely on much more powerful off-the-shelf embeddable SMT solvers that can also handle linear rational arithmetic constraints, and work more on improving the k-induction procedure.

Our path compression method of invariant strengthening and termination checking can be seen as a special case of the one described by de Moura *et al.* [10]. That work, however, also describes invariant strengthening techniques based on quantifier elimination, which we do not use.

A dependency-based abstraction method analogous to ours is described by Chan *et al.* [6] for the dataflow language RSML. There, however, abstractions appear to be generated statically as a preprocessing step, and not refined dynamically. Our abstraction method is more similar to the one developed by Babic and Hu [2], which uses a SAT solver as its reasoning engine. The main idea of our method however—treating local variables as input ones and refining based on spurious counterexamples—goes back to Kurhshan [19] and appears in various forms in several works on hardware verification. Our use of unsatisfiable cores in our refinement heuristic recalls a similar approach by Chauhan *et al.* [7] for SAT-based model checking.

2. Preliminaries

Lustre is a declarative programming language for manipulating data flows, or *streams*, infinite sequences $(v_0, v_1, \ldots)$ of values of conventional data types, such as (bounded) integers, floating point numbers, and Booleans.

In essence, a Lustre function, called *node* in Lustre, is the declarative specification of a stream transformer, mapping a finite set of streams to another finite set of streams. Operationally, a node has a cyclic behavior: at each cycle i, it takes as input the value of each input stream at position or *instant*, i and returns the value of each output stream at instant i. Lustre nodes have a limited form of memory in that, when computing the output values they can also look at input and output values from previous instants, up to a finite limit statically determined by the program itself.

Figure 1 contains a simple example of a two-node Lustre program. The program models a thermostat whose target temperature, initially set to 70 degrees, can be changed by 1 degree at a time by pressing an "up" or a "down" button.

Typically, the body of a Lustre node consists in a set of *definitions*, equations of the form $x = t$ (as seen in the nodes in Figure 1) where x is a variable denoting an output or a locally defined stream and t is an expression, in a certain stream algebra, whose variables name input, output, or local streams. More generally, x can be a tuple of stream variables and t an expression evaluating to a tuple of the same type (as in the last equation of the node

```
node thermostat (actual_temp, target_temp, margin: real)
 returns (cool, heat: bool) ;
let
 cool = (actual_temp - target_temp) > margin ;
 heat = (actual_temp - target_temp) < -margin ;
tel

node therm_control (actual: real; up, down: bool)
 returns (heat, cool: bool) ;
var target, margin: real ;
let
 margin = 1.5 ;
 target = 70.0 -> if down then (pre target) - 1.0
                  else if up then (pre target) + 1.0
                  else pre target ;
 (cool, heat) = thermostat (actual, target, margin) ;
tel
```

Figure 1. A simple Lustre program.

`therm_control` in Figure 1).

Most of Lustre's operators are pointwise liftings to streams of the usual operators over stream values. For example, let $x = (m_0, m_1, \ldots)$ and $y = (n_0, n_1, \ldots)$ be two integer streams. Then, $x + y$ denotes the stream $(m_0 + n_0, m_1 + n_1, \ldots)$; a constant like 5, say, denotes the constant integer stream $(5, 5, \ldots)$. Two important additional operators are a unary shift-right operator pre ("previous"), and a binary initialization operator $\rightarrow$ ("followed by"). The first is defined as $\mathsf{pre}(x) = (v, m_0, m_1, \ldots)$ with the value v left unspecified. The second is defined as $x \rightarrow y = (m_0, n_1, n_2, \ldots)$.

In Figure 1, the definition of the real-valued stream `target` constrains it to have value 70 at instant 0. At each instant $i > 0$ the current value of `target` is its previous value (at instant $i - 1$) minus 1 if the current value of the Boolean stream `down` is `true`; it is its previous value plus 1 if the current value of `up` is `true`; it is the previous value otherwise.

The right-hand side of an equation defining a stream can contain non-recursive calls to other nodes. Syntactical restrictions on the equations in a Lustre program guarantee that its streams are well defined.

2.1. From Lustre to SMT

To reason about the values that the streams of a Lustre program have at a given instant it would be convenient to have a logic in which Lustre's basic data types and their operations are built-in. Automated reasoning in such a logic, however, is not (yet) feasible. In alternative, we can consider an *idealized* version of Lustre with unbounded integers instead of machine integers and infinite-precision rationals instead of floating point numbers. Idealized Lustre programs can be modeled faithfully in a typed first-order logic with uninterpreted function symbols and built-in inte-

gers and rationals[1]. We will call it *Idealized Lustre Logic* and denote by $\mathcal{IL}$. The main lure of $\mathcal{IL}$ is that the validity in it of quantifier-free formulas with *linear* numerical terms is not only decidable but decidable rather efficiently by means of SMT techniques [4]. The main limitations are the exclusion of programs containing non-linear operations, and the inability to model behavior due overflows, underflows and rounding errors.

In this work, we adopted $\mathcal{IL}$ as the underlying logic. Future work will focus on extending our approach efficiently to programs with non-linear expressions and to a logic modeling real Lustre more accurately.

2.2. A logical model of Lustre

In $\mathcal{IL}$, a stream x of values of type τ can be represented simply as an uninterpreted function of type $\mathbb{N} \to \tau$. Then, equations between streams can be reduced to universally quantified equations between stream values by a straightforward homomorphic translation. For instance, the integer stream equation $x = y + z$ can be translated into $\forall n{:}\mathbb{N}.\, x(n) = y(n) + z(n)$, where x, y, and z become functions symbols of type $\mathbb{N} \to \tau$.[2] Similarly, the equation $x = y \to y + \mathsf{pre}\ z$ can be translated into $\forall n{:}\mathbb{N}.\ x(n) = ite(n = 0,\, y(0),\, y(n) + z(n-1))$ where ite is the if-then-else operator in the logic.

Let N be any single-node (idealized) Lustre program with stream variables $\mathbf{x} = \langle x_1, \ldots, x_p, y_1, \ldots, y_q \rangle$ where $x_1, \ldots, x_p$ are input variables and $y_1, \ldots, y_q$ are non-input (i.e., local and output) variables. The semantics of N is completely captured in $\mathcal{IL}$ by the universal quantification over the variable n of the following system of equations

$$\Delta(n) = \left\{ \begin{array}{rcl} y_1(n) & = & t_1[\mathbf{x}(n), \mathbf{x}(n-1), \ldots, \mathbf{x}(n-d)] \\ \vdots & \vdots & \vdots \\ y_q(n) & = & t_q[\mathbf{x}(n), \mathbf{x}(n-1), \ldots, \mathbf{x}(n-d)] \end{array} \right.$$

where each t_i is the translation of the expression defining y_i in N. Multi-node programs can be modeled the same way by first recursively in-lining in the main node all calls to subnodes. For the program in Figure 1, $\Delta(n)$ would look like the following (with m abbreviating `margin`, and so on):

$$\left\{ \begin{array}{l} m(n) = 1.5 \\ t(n) = ite(n = 0, 70.0, ite(d(n), t(n-1) - 1.0, \ldots)) \\ c(n) = (a(n) - t(n)) > m(n) \\ h(n) = (a(n) - t(n)) < -m(n) \end{array} \right.$$

We call an *(instantaneous) configuration (for N)* any well-typed tuple of values for $\mathbf{x}(n)$. Then, the program N can be understood as a system of constraints over instantaneous configurations. The notation for $\Delta(n)$ is suggestive of the fact that the value of $y_i(n)$, the $(p+i)$-th component of the configuration at instant n, is a function of (some of the values in) the configurations at instants $n, n-1, \ldots, n-d$. The value d is the maximum "nesting depth" of the pre operator in the body of N. For simplicity, from now on *we will restrict our presentation to programs with $d \leq 1$*. The extension to the general case, implemented in our verifier, is fairly straightforward.

A *trace* for the program N is a tuple $\mathbf{s} = \langle s_1, \ldots, s_{p+q} \rangle$ of streams of the same type as $\mathbf{x}$. A *path* (of length l) is a finite sequence (of l) consecutive configurations on a trace. The possible input/output behaviors of N are exactly the traces $\mathbf{s}$ that satisfy the formula $\forall n{:}\mathbb{N}.\, \Delta(n)$ when $\mathbf{x}$ is interpreted as $\mathbf{s}$. We call such traces a *legal traces*. A configuration is *reachable* if it occurs in some legal trace; it is *initial* if it is the first configuration on a legal trace. A path is *initial* if it is the initial segment of a legal trace.

As argued in [17], with Lustre programs one is mostly interested in verifying *safety properties*. Being able to checking *liveness properties* is not so crucial because Lustre is typically used to model real-time systems. In such systems, progress must occur within predefined temporal limits and so it expressible as a safety property. More precisely, we are interested in properties invariant in the following sense.

Definition 1 *A property P of configurations is invariant (for N) if it holds for all reachable configurations.*

More generally, one can consider properties of paths, not of single configurations. We do not do that here, just for simplicity. We will consider only *quantifier-free properties*, properties expressible by a quantifier-free formula of $\mathcal{IL}$. While this an actual restriction, invariant properties checked in common practice are indeed quantifier-free.

Checking invariant properties of Lustre programs can be done by lifting and adapting to the logic $\mathcal{IL}$ a number of SAT-based model checking techniques. The main ones used in this work is k-induction.

3. SMT-based k-Induction

Let N be again a single node Lustre program, and let $\Delta(n)$ be the equational system modeling N in $\mathcal{IL}$. Let P be a property of N's configurations expressible by a quantifier-free formula $P(n)$ of $\mathcal{IL}$ over $\mathbf{x}(n)$. If t is any integer term of $\mathcal{IL}$, we denote by Δ_t the formula obtained from $\Delta(n)$ by replacing every occurrence of n with t. Similarly, for P_t.

3.1. Basic Procedure

Similarly to previous work on k-induction-based verification of transition systems [21, 5, 11, 10], we can prove

[1] Having tuples as well is convenient but not necessary because they can be treated as syntactic sugar.

[2] To simplify the notation we use the same metasymbol to denote both a Lustre variable and the corresponding function symbol in the translation.

that P is invariant for N if we succeed in proving the validity of the following two statements for some concrete $k \geq 0$:

$$\Delta_0 \wedge \Delta_1 \wedge \cdots \wedge \Delta_k \models_{\mathcal{IL}} P_0 \wedge P_1 \wedge \cdots \wedge P_k \quad (1)$$

$$\Delta_{\mathsf{n}} \wedge \Delta_{\mathsf{n}+1} \wedge \cdots \wedge \Delta_{\mathsf{n}+(k+1)} \wedge P_{\mathsf{n}} \wedge P_{\mathsf{n}+1} \wedge \cdots \wedge P_{\mathsf{n}+k} \models_{\mathcal{IL}} P_{\mathsf{n}+(k+1)} \quad (2)$$

where $\models_{\mathcal{IL}}$ denotes logical entailment in $\mathcal{IL}$ and n is an uninterpreted integer constant.

Both entailments can be decided by current SMT solvers for $\mathcal{IL}$. To verify P then, one asks the solver to prove both cases for some initial choice of k, retrying with a larger k until either the base case (1) is proven invalid or both the base case and the induction step (2) are proven valid. In the first situation, P is not invariant for N, and a counterexample path can be generated from an $\mathcal{IL}$-model of $\Delta_0 \wedge \cdots \wedge \Delta_k \wedge \neg(P_0 \wedge \cdots \wedge P_k)$ provided that the SMT solver is able to return models. In the second situation, P has been shown to hold for a set of configurations including all reachable configurations, which implies it is invariant.

This procedure is *sound*; it will never mistake a variant property for an invariant one. Standard arguments [12] can be used to show that in general the procedure is not—and cannot be made—*complete* for general Lustre programs; specifically, the procedure may keep increasing k indefinitely for some invariant properties. Nevertheless, a number of improvements are possible to increase the procedure's *accuracy*, the set of invariant properties it can prove. Further improvements can also be applied to accelerate convergence to an answer. Two main enhancements to the basic k-induction procedure that we have found effective in practice are described in the next two sections.

The procedure's efficiency is also increased by exploiting several features of SMT solvers based on the *lazy approach* [4]. Similarly to previous SAT-based k-induction work [11, e.g.], we take full advantage of the fact that such solvers are on-line, incremental and backtrackable, return models, compute unsatisfiable cores, and can remember learned lemmas. Details on this and on other useful enhancements based on program slicing and other static preprocessing of the formulas in (1) and (2) can be found in [15].

3.2. Path Compression

A major enhancement of the basic procedure is represented by *path compression*, first introduced for k-induction in [5, 21]. In our setting, path compression is achieved by strengthening the left hand side of (1) and (2) so as to eliminate from consideration paths that contain repeated configurations or, more generally, configurations that are equivalent in an appropriate sense. A rather general notion of path compression for SMT-based k-induction is presented in [10]. Performance considerations usually prevent one from using that notion in its full generality. We use the one defined below, which seems better suited to the current capabilities of SMT solvers for $\mathcal{IL}$.

Let us assume for convenience that the argument of each application of the operator pre in the program N is just a stream variable.[3] We call those variables the *state variables* of N. Given a configuration $\mathbf{v}$ for N, the *state of* $\mathbf{v}$ is the subtuple of $\mathbf{v}$ corresponding to N's state variables. A path is *compressed* if no two configurations in it have the same state and none of them, except possibly the first, are initial.

We strengthen the premise of (2) by adding a quantifier-free formula $C_{\mathsf{n},k}$ over N's state variables that is satisfied by all and only those traces whose configurations from position n to $\mathsf{n}+k$ form a compressed path:

$$\Delta_{\mathsf{n}} \wedge \Delta_{\mathsf{n}+1} \wedge \cdots \wedge \Delta_{\mathsf{n}+(k+1)} \wedge P_{\mathsf{n}} \wedge P_{\mathsf{n}+1} \wedge \cdots \wedge P_{\mathsf{n}+k} \wedge C_{\mathsf{n},k} \models_{\mathcal{IL}} P_{\mathsf{n}+(k+1)} \quad (2')$$

It is immediate that using (2') instead of (2) preserves the k-induction procedure's accuracy. More interestingly, it also preserves its soundness, as proved in [15].

In addition to facilitating the SMT solver's task of proving $P_{\mathsf{n}+(k+1)}$, the restriction to compressed paths also yields a *complete* k-induction procedure whenever the length of compressed initial paths is (statically) bounded above. Completeness then is achieved, while preserving soundness, by checking (1) and (2') for consecutive values of k starting at 0, and verifying before repeating the whole loop with a larger k whether there are any compressed initial paths of length $k+1$. In turn, this is done by checking that the entailment

$$\Delta_0 \wedge \cdots \wedge \Delta_k \models_{\mathcal{IL}} \neg C_{0,k+1} \quad (3)$$

holds. This is analogous to a loop check in Bounded Model Checking: if the test succeeds, no counterexamples will be found from $k+1$ on—because if they existed they could be compressed to counterexamples of length $k' < k+1$, against the fact that (1) held for k'. Hence, one can stop the induction loop and conclude that P is invariant for N.

3.3. Structural Abstraction

Abstraction is often helpful in scaling up the verification of properties of transition systems. One tries to prove a safety property for a given system S by proving it for a conservative abstraction $S^\sharp$ of S that is easier to deal with. This process may be followed by some refinement of the abstraction when the property does not hold for it. The same principle applies to Lustre programs as well, with some added advantages provided by Lustre's declarative nature.

[3] We can always reduce ourselves to this case by first applying simple observational-equivalence-preserving transformations to N.

Traditionally, *predicate abstraction*, a popular abstraction approach in software model checking [3], takes an infinite-state system S and a set of *abstraction predicates*, and generates a conservative (typically finite-state) abstraction of S. A main challenge in predicate abstraction is coming up with a good set of abstraction predicates.

With Lustre, a predicate completely capturing the behavior of a single node program N is readly provided by the equational system (predicate) Δ introduced in Section 2.1. This suggests a conceptually and practically much simpler yet effective form of abstraction, which we call *structural abstraction* following [2], based on the syntactical structure of N. In our case, it consists simply in removing equations from Δ. Conversely, refinement is achieved by adding back equations from Δ to the current abstraction $\Delta^\sharp \subseteq \Delta$.

At the Lustre level, this is essentially a form of *localization abstraction* [19] as it corresponds to turning some non-input streams of N into input ones, and removing their defining equation from the program. Clearly, the set of legal traces for the resulting program $N^\sharp$ contains all the legal traces for N. Therefore, every property invariant for $N^\sharp$ is also invariant for N. A property P that is invariant for N but not for $N^\sharp$ will have *spurious counterexamples*, legal traces for $N^\sharp$ that falsify P but are not legal traces of N.

We integrated the following counterexample-driven refinement strategy into our k-induction procedure. Instead of instances of the original system Δ, we use instances of its current abstraction $\Delta^\sharp$, initially consisting of the definitions of the variables of N that occur in the property to be checked. Whenever the base step (1) or the induction step (2) produces a spurious counterexample, we refine $\Delta^\sharp$ as explained below and redo the step, repeating this process until no spurious conter-examples are returned for that step.

Note that we refine the predicate $\Delta^\sharp$ itself, not its single instances $\Delta^\sharp_0, \ldots, \Delta^\sharp_k$ or $\Delta^\sharp_{\mathsf{n}}, \ldots, \Delta^\sharp_{\mathsf{n}+(k+1)}$ used respectively in (1) and (2). Doing the latter is usually too fined grained, producing rather slow convergence.

We observe that it is possible to combine abstraction with path compression and loop detection. However, the combination must be done carefully since it can create subtle soundness issues (again, see [15]).

The Refinement Step

We experimented with several choices for refining $\Delta^\sharp$ and we settled on the following one, which seems to be the most effective in practice. To start, observe that Δ induces a (possibly cyclic) directed graph G over its terms, where for each equation $y = t$ in Δ, y is linked to t and t is linked to every stream variable occurring in it.

Refinement is done essentially by starting from a non-input variable y, and doing a depth-first traversal of G—taking care of breaking the loops in the graph—until some input variable x is reached. The set of all equations along the current path in G from y to x that are not already in $\Delta^\sharp$ are then added to $\Delta^\sharp$. If this set is empty, the next path from y to an input variable is considered, until $\Delta^\sharp$ changes or all paths have been explored. In the latter case, we say that y is *completely defined* in $\Delta^\sharp$; then, some other input variable must be chosen for refinement. Progress is guaranteed by eventually choosing a non-input variable that does not yet occur as a left-hand side in $\Delta^\sharp$. When there are no more such variables, $\Delta^\sharp$ has been completely refined into Δ.

The rationale of this *path refinement* strategy is to increase in a refinement step the number of paths in the graph G connecting a non-input variable to the input variables it ultimately depends on. This is more likely to produce a refinement that rules out spurious counterexamples than, say, a breadth-first refinement strategy over G, which may need to go through various levels of G before it starts adding constraints to $\Delta^\sharp$ that connect input and non-input variables.

The remaining question is which non-input variables to choose for refinement. For that, we rely on the help of the SMT solver. Let $\Delta^\sharp_{0,k}$ abbreviate $\Delta^\sharp_0 \wedge \cdots \wedge \Delta^\sharp_k$ and let $P_{0,k}$ abbreviate $P_0 \wedge \cdots \wedge P_k$. When the solver reports that

$$\Delta^\sharp_{0,k} \models_{\mathcal{IL}} \neg P_{0,k}$$

(the version of (1) using $\Delta^\sharp$) does not hold, it also returns a *counter-model* M for it. Specifically, M is a set of equalities of the form $x(m) = v$ where x is a stream variable, m is a numeral in $\{0, \ldots, k\}$, and v is a concrete value of the proper type (an integer number, a rational number, or a Boolean value), such that $M \models_{\mathcal{IL}} \Delta^\sharp_{0,k} \wedge \neg P_{0,k}$.

To check that M can be extended to a counterexample for the original program N, it is enough to check the $\mathcal{IL}$-satisfiability of the set $\Delta_0 \wedge \cdots \wedge \Delta_k \wedge M$. If this set is $\mathcal{IL}$-unsatisfiable, $\Delta^\sharp_n$ needs to be refined. To direct the refinement to relevant program variables we ask the SMT solver for a minimal (or just smaller) subset M' of M that is enough to cause the unsatisfiability. Then, we pick for refinement a stream variable x that occurs in an equation $x(m) = v$ of M' and is not completely defined in $\Delta^\sharp$. This helps the refinement process focus on parts of the definition graph G whose absence from $\Delta^\sharp$ was decisive in generating the counter-model M.

4. Experimental Results

We implemented the k-induction framework above in *Kind*, a new automated inductive prover for idealized Lustre. Kind is written in OCaml and relies on an external SMT solver for reasoning in the logic $\mathcal{IL}$. Currently, Kind supports the SMT solvers CVC3 and Yices. The version discussed here uses Yices 1.0.9. Kind can be run in one of two major modes: *induction* and *bmc*. The first implements

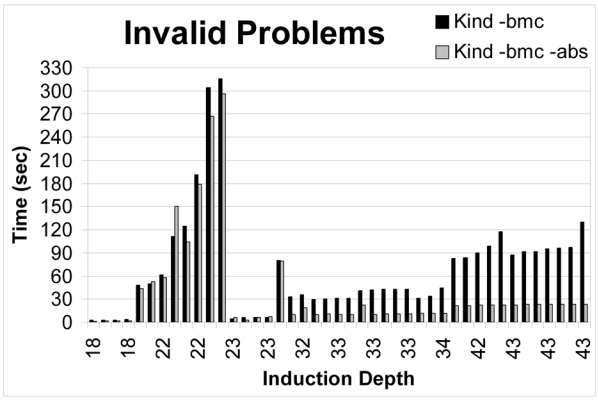


Figure 2. Runtimes for Kind with and without abstraction, on hard invalid problems.

the k-induction decision procedure presented in Section 2.1. The second skips all operations related to the induction step of the procedure, making Kind essentially behave like a bounded model checker.

We performed extensive experiments with Kind, testing the performance of various combinations of enhancements over the basic induction procedure, and comparing the best ones against other verification systems.[4] The tests were run on a dedicated cluster of four 3.0 GHz Intel Pentium 4 machines, each with 1 Gb of physical memory, under RedHat Enterprise Linux 4.0. Timeouts were set to 900 seconds.

We based our test set on a number of sources. Most test problems were provided by colleagues in academia and industry. Others were adapted from published case studies. To increase the number of problems, we introduced a small number of modifications into the existing ones, simulating likely user errors. We grouped together all variants of the same original problem generated this way and removed any *duplicates* from the group, considering two variants duplicates if they produced too similar results (same valid/invalid answer and approximately the same runtimes) for all systems and configurations tried. This was done to prevent biasing the test set against or in favor of any one system.

The final test set had 1047 problems, each with a single property, classifiable into 6 groups: two groups with problems about memory controllers, one involving protocols, one based on simulations (often involving vehicles), one based on larger simulations, with several hundred lines of Lustre code apiece, and one with mostly toy problems involving counters. We partitioned the problems into: (i) a set of 447 *invalid problems*, problems whose property was disproved with a (real) counterexample; (ii) a set of 376 *valid problems*, problems whose property was declared proved by at least one systems; (iii) a set of 224 *unsolved problems*, which will not be considered below. Of the 823 solved problems, 20 valid and 18 invalid problems used rational streams, the rest used only integer and/or Boolean streams.

The most advanced system in the comparison was part of the SAL 3.0 toolset [9]. Even if it has an induction mode, strictly speaking, and contrary to Kind, SAL is not a full-blown induction prover. In induction mode, SAL first performs a BMC-like test up to a user specified limit l on the number k of unrollings, and then follows that with a *single* induction test if the BMC test found no counterexamples. For a fairer comparison, we replicated this behavior in Kind as well. Then we ran both SAL and Kind in an iterative deepening fashion on each problem, starting with $l = 0$ and repeatedly re-starting the system with a greater value of l until a conclusive answer was returned. We eventually chose to increment l by 3 at each restart because it seemed to produce the best results for both systems. For uniformity and space constraints, we report only results where iterative deepening was applied when Kind was run in induction mode.

Of the two main enhancements to the basic k-induction procedure presented in this paper, we focus on the evaluation of structural abstraction. The other enhancement, path compression, gave better overall results for Kind in induction mode, in terms of number of solved problems and run time. In particular, it allowed Kind to determine the validity of 8 problems that were not solvable by any of the other systems. Hence, all the results presented in this section are with Kind running with path compression on, unless specified otherwise. The other input options provided by Kind, and not discussed here, were each given the same value across the board.

4.1. Abstraction vs. no abstraction

When verifying safety properties, it is customary to attempt first a quick run of bounded model checking, to see if a counterexample can be found. If that fails, the more expensive full verification check is then performed. Following this practice, it makes sense to look at Kind's results when run in bmc mode on the invalid problems and in interative-deepening inductive more on the valid ones.

Our expectation was that abstraction would not be effective with easy problems, because of its significant overhead. Hence, the purpose of our evaluation was to verify whether abstraction is beneficial with complex problems without producing an unacceptable slowdown with simple ones. Our results partially confirm this thesis.

As a simple complexity measure, let us classify a problem as *easy* if Kind could solve it within 2 seconds *without* abstraction, and *hard* otherwise.

Of the 447 invalid problems, 404 were easy in this sense

[4] Kind's executable, the benchmarks discussed here as well as more detailed results are available from `http://combination.cs.uiowa.edu/Kind/`.

and were cumulatively solved in 32s. Each of them had a counterexample path of length at most 13. The remaining problems took from 2s to 316s each to solve, for a total of 2989s, with counterexample lengths ranging from 18 to 43. With abstraction, Kind solved each of the easy problems within 2s as well, but took 73s overall, with a slowdown factor of 2.3. However, it solved the hard problems in 1694s with an overall speed up factor of 1.8. Runtimes in seconds for the hard invalid problems are plotted in Figure 2. As can be clearly seen, only in one case does abstraction produce a significant slowdown. In most of the other cases it is instead quite effective, with speed-ups in excess of 300% for several problems.

In contrast, abstraction was not effective for the valid problems. To start, almost all of them, 370 over 376, were easy and were solved in a total of 35s without abstraction, with induction depths (the value of k) of 12 or less. The 6 hard problems took from 2s to 101s, with one timeout. The total time for them was 217s, with induction depths ranging from 11 to 17. With abstraction, Kind solved each of the easy problems within 2s except six (solved in less than 4s), with a total time of 115s and a slowdown factor of 3.3. It also solved all of the hard problems, but it was actually 1.6 times slower on them (340s) than Kind without abstraction.

4.2. Kind vs. other Lustre checkers

To evaluate Kind against the state of the art in the automated verification of Lustre programs, we compared it with all the publicly-available Lustre verifiers we were aware of: Lesar, built by Lustre's developers at Verimag [20], Nbac, also developed at Verimag [18], and Rantanplan and Luke, both developed at Chalmers University [13, 8]. We were unable to run the latest version of Nbac, and we used instead the version provided in the Lustre 4 distribution. Unfortunately, this version appears to be unsound[5] and so we had to discard it from this evaluation.

Lesar is a state-based verification tool included with the Lustre 4 distribution. It is primarily a BDD-based symbolic model checker, with some limited support for integers through abstraction and a polyhedral library. Due to this, it is incomplete for non-Boolean problems.

Rantanplan could be considered the closest precursors to Kind since it is based on k-induction and SMT techniques. As mentioned in related work, its induction procedure and its SMT support are however less sophisticated, in particular it does not perform abstraction and refinement. Finally, it does not support programs with rational streams.

Luke is another k-induction verifier, inspiring much of the work in Rantanplan, but it is based on propositional logic and was developed mostly for educational purposes. It accepts programs with integers by treating them as bounded integers of a user-specified size.

[5]It classified as valid a number of problems for which at least two of the other systems were able to find a counterexample.

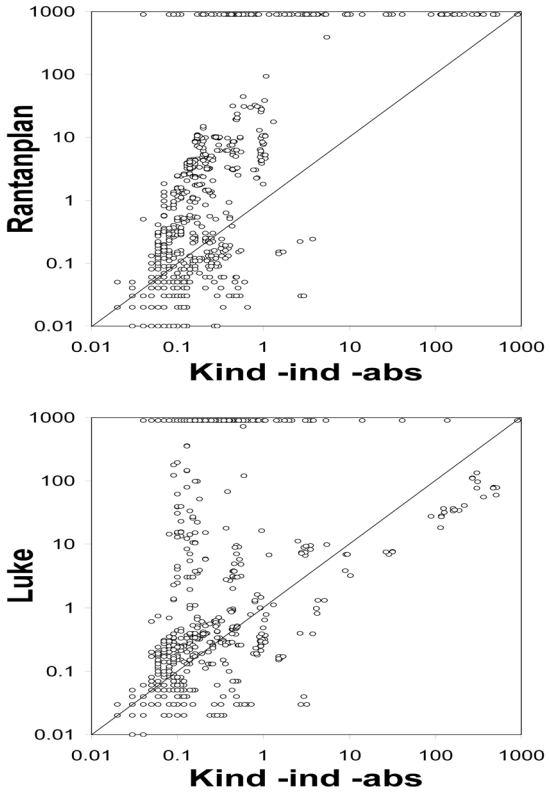


Figure 3. Kind vs. Rantanplan and Luke on valid and invalid problems.

Since none of these systems have something comparable to Kind's bmc mode, we report the results obtained by Kind when run in iterative-deepening induction mode on both valid and invalid problems. In that mode, Kind solved all of the valid problems but timed out on 8 of the invalid ones.

We ran Lesar from the distribution in its standard configuration (with the -poly argument). Lesar solved correctly fewer than 10% of the valid problems, often quickly producing an incorrect answer for the rest. Since Lesar is incomplete but does not return counterexamples, measuring its performance on the invalid problems is not meaningful.

Rantanplan and Luke were more comparable to Kind. Their performance against our system is summarized in the scatter plots of Figure 3, which show runtimes in seconds, on a log-log scale. Timeout points also include problems that a system solved incorrectly due to its incompleteness, or could not solve because they were out of its scope or caused a runtime error (such as stack overflow).

We tried several of the configurations for Rantanplan suggested in [12], with generally similar results, and chose

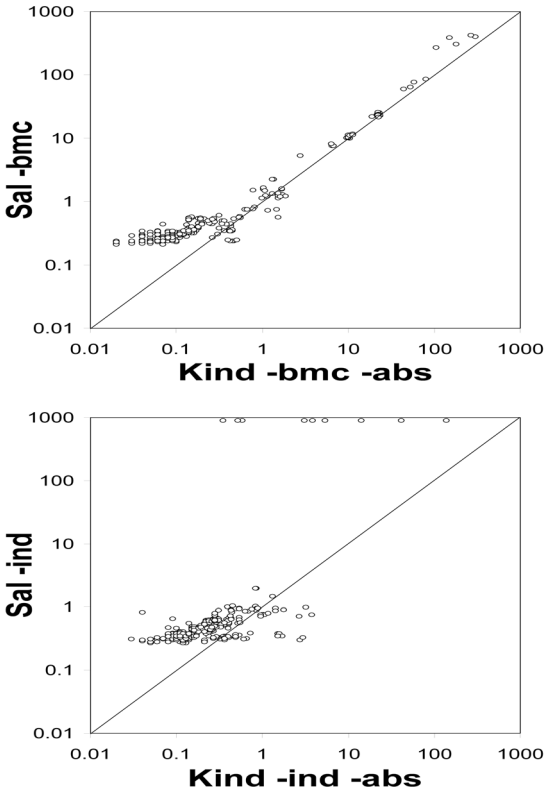


Figure 4. Kind vs. SAL, respectively on invalid and valid problems.

one that seemed one of the best overall performers: deletion filtering with the Pooh checker used in online integration and GLPK as the offline infeasibility checker. Rantanplan solved 84% of the 823 solved problems, (against Kind's 99%) and was overall an order of magnitude slower than Kind on them. Of the 133 problems not solved by Rantanplan, 38 could not be solved due to presence of rational streams, 2 produced incorrect counterexamples, 26 caused run-time errors, and the rest timed out.

Consistently to what observed in [12], Luke performed better than Rantanplan on the invalid problems. However, it solved only 71% of the 823 solved problems (mostly finite-state problems), being overall 30% faster than Kind on those. Of the 242 problems not solved by Luke, 38 could not be solved due to presence of rational streams, 19 produced incorrect counterexamples due to the incompleteness of the bounded integer support, and the rest timed out.

4.3. Kind vs. SAL

Even if Kind was consistently and significantly better than the available Lustre-native verifiers, it could be argued that those systems are, for various reasons, not cutting-edge anymore. We looked then for a publicly available state-of-the-art tool that could verify safety properties of both finite- and infinite-state reactive systems. The eventual choice of the SAL toolset was motivated as follows.

The toolset contains a model checker (sal-inf-bmc, hereafter referred to as SAL) in many ways similar to Kind: it can be run in either bmc or induction mode, has a form of path compression, is SMT-based, and uses Yices. The main differences are that it does not use abstraction and is not Lustre-specific. Instead, it uses its own input language, based on the traditional two-state model. This allowed us to verify our hypothesis that adapting existing techniques to work directly on a logical model of Lustre leads to better performance.

To run SAL on Lustre problems we utilized a well-tuned Lustre-to-SAL translation developed at Rockwell Collins along the lines of the translations described in [22]. Rockwell Collins has been developing and continually improving translators from Lustre to various model checkers, including SAL, over several years for the verification of production-level Lustre models of its avionics software.

We ran SAL and Kind in their respective bmc mode on the invalid problems, and in iterative-deepening induction mode on the valid ones. Among SAL's different command-line configurations only its analogous to path compression (the -acyclic option) produced any significant differences in overall performance with respect to the default configuration. Specifically, enabling path compression was better in induction mode and worse in bmc mode for SAL. The scatter plots of Figure 4 summarize SAL's results in those configurations against Kind's results with path compression enabled only in induction mode and abstraction enabled in both modes. As the plots show, the two systems are comparable but Kind's performance dominates.

Both systems solved all the invalid problems, with Kind being more than 50% faster overall. Moreover, Kind solved all valid problems while SAL timed out on 10 of them. On the valid problems solved by both systems, Kind was more than 50% faster. Interestingly, on the latter problems, Kind without abstraction, a configuration more closely comparable to SAL, was actually more than 500% faster.

Overall, these results seems to support our hypothesis that a Lustre-specific k-induction tool would offer a performance premium over a translation-based approach. On the other hand it must be added that, looking at its source code, SAL does not appear to exploit the advanced features of Yices as much as Kind does, which might be a big factor in Kind's better performance.

5. Conclusion and Further Work

We have presented a general approach for the verification of Lustre programs built around SMT-based k-induction.

The approach has a wider scope than that of existing verifiers for Lustre since it can handle finite- and infinite-state programs with integer and rational streams. It scales much better to realistic programs thanks to the compactness of its logical translation and a tight integration with efficient SMT solvers. Also, it lends itself quite naturally to a form of structural abstraction and refinement that adds further scalability when model checking complex properties or programs. The approach seems general enough to be applicable to other synchronous dataflow languages as well.

We conclude by mentioning a couple of additional directions for further work. Our path refinement strategy adds to the current abstraction entire equations from the concrete system Δ. More fine-grained refinements would be obtained by statically preprocessing Δ and breaking complex equations it into simpler ones by introducing additional local streams. We would like to investigate and evaluate automated strategies for doing this.

Our abstraction technique can support modular verification of multinode programs when individual nodes are annotated with sufficiently expressive invariants. We plan to investigate ways of avoiding or delaying the in-lining of subnode calls when invariants are available. We also plan to investigate ways to produce node invariants automatically during the verification of a property, with the goal of speeding up later re-verification attempts of the same property caused by changes in the code or the property's formulation—something that occurs often in real world situations.

Acknowledgments. This work was partially supported by an equipment grant from Intel Corporation. We thank Koen Claessen for some discussions on k-induction and its implementation in Luke; Bruno Duterte for his assistance with Yices; Anders Franzén for providing many of the problems in our test set. Special thanks go to Steve Miller and Mike Whalen for providing additional problems and translating all of them to SAL. We are also grateful to the anonymous reviewers for some useful suggestions on improving the paper's presentation.

References

[1] J. M. A. Armando and L. Platania. Bounded model checking of software using SMT solvers instead of SAT solvers. In *Proceedings of SPIN'06*, volume 3925 of *LNCS*, pages 146–162. Springer, 2006.

[2] D. Babic and A. J. Hu. Structural abstraction of software verification conditions. In *Proceedings of CAV 2007*. pages 366–378. Springer, 2007.

[3] T. Ball, R. Majumdar, T. D. Millstein, and S. K. Rajamani. Automatic predicate abstraction of C programs. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 203–213, 2001.

[4] C. Barrett, R. Sebastiani, S. A. Seshia, and C. Tinelli. *Handbook on Satisfiability*, chapter Satisfiability Modulo Theories. IOS Press, 2008. (To appear.).

[5] P. Bjesse and K. Claessen. SAT-based verification without state space traversal. In *Proceedings of FMCAD 2000*, pages 372–389, Springer, 2000.

[6] W. Chan, R. J. Anderson, P. Beame, and D. Notkin. Improving efficiency of symbolic model checking for state-based system requirements. In M. Young, editor, In *Proceedings of ISSTA 98*, pages 102–112. ACM, 1998

[7] P. Chauhan *et al.* Automated abstraction refinement for model checking large state spaces using SAT based conflict analysis. In *Proceedings of FMCAD'02* , pages 33–51. 2002.

[8] K. Claessen. Luke webpage, 2006. `http://www.cs.chalmers.se/Cs/Grundutb/Kurser/form/luke.html`.

[9] L. de Moura, S. Owre, H. Rueß, J. Rushby, N. Shankar, M. Sorea, and A. Tiwari. SAL 2. In *Proceedings of CAV 2004*. Springer, 2004.

[10] L. de Moura, H. Rueß, and M. Sorea. Bounded model checking and induction: From refutation to verification. In *Proceedings of CAV 2003*, volume 2725 of *LNCS*, 2003.

[11] N. Een and N. Sorensson. Temporal induction by incremental SAT solving. *Electronic Notes in Theoretical Computer Science*, 89(4), 2003.

[12] A. Franzén. Combining SAT solving and integer programming for inductive verification of Lustre programs. Master's thesis, Chalmers University of Technology, 2004.

[13] A. Franzén. Using satisfiability modulo theories for inductive verification of Lustre programs. *Electronic Notes in Theoretical Computer Science*, 114, 2005.

[14] M. K. Ganai and A. Gupta. Accelerating high-level bounded model checking. In *Proceedings of ICCAD'06*, pages 794–801. ACM, 2006.

[15] G. Hagen. *Verifying safety properties of Lustre programs: an SMT-based approach*. PhD thesis, The University of Iowa, 2008. (Forthcoming).

[16] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data-flow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, 1991.

[17] N. Halbwachs, F. Lagnier, and C. Ratel. Programming and verifying real-time systems by mans of the synchronous data-flow language LUSTRE. *IEEE Transactions in Software Engineering*, 18(9):785–793, 1992.

[18] B. Jeannet. Dynamic partitioning in linear relation analysis: Application to the verification of synchronous programs. *Formal Methods in System Design*, 23:5–37, 2003.

[19] R. P. Kurshan. *Computer-aided verification of coordinating processes: the automata-theoretic approach*. Princeton University Press, 1994.

[20] G. Pace, N. Halbwachs, and P. Raymond. Counterexample generation in symbolic abstract model-checking. *International Journal on Software Tools and Technology Transfer (STTT)*, 5(2):158–164, 2004.

[21] M. Sheeran, S. Singh, and G. Stålmarck. Checking safety properties using induction and a SAT-solver. In *Proceedings of FMCAD 2000*, pages 108–125. Springer, 2000.

[22] M. Whalen, D. Cofer, S. Miller, B. Krogh, and W. Storm. Integration of formal analysis into a model-based software development process. In *12th International Workshop on Industrial Critical Systems (FMICS 2007)*, 2007.

Word-Level Sequential Memory Abstraction for Model Checking

Per Bjesse
Advanced Technology Group
Synopsys Inc.

***Abstract*—Many designs intermingle large memories with wide data paths and nontrivial control. Verifying such systems is challenging, and users often get little traction when applying model checking to decide full or partial end-to-end correctness of such designs. Interestingly, a subclass of these systems can be proven correct by reasoning only about a small number of the memory entries at a limited number of time points. In this paper, we leverage this fact to abstract certain memories in a sound way, and we demonstrate how our memory abstraction coupled with an abstraction refinement algorithm can be used to prove correctness properties for three challenging designs from industry and academia. Key features of our approach are that we operate on standard safety property verification problems, that we proceed completely automatically without any need for abstraction hints, that we can use any bit-level model checker as a back-end decision procedure, and that our algorithms fit seamlessly into a standard transformational verification paradigm.**

I. Introduction

Bit-level model checking methods have for a long time been the standard approach for safety property verification of hardware systems. However, recently there has been a surge of interest in methods that model systems at the *word level*. Such methods differs from traditional bit-level methods by not forcing a fragmentation of a netlist into individual single bit inputs and registers, and by modeling data paths at the level of complex gates such as adders and multipliers. By retaining some of the high-level structure of the hardware systems that are analyzed, and by leveraging decision procedures for the richer word-level logic, the solving process can sometimes be sped up dramatically. Prime examples of such decision procedures include SMT-solvers [14], and approaches based on reduction to satisfiability checking like `UCLID` [5] and `BAT` [11]. These methods are all *formula-based* approaches in that they decide unquantified formulas in some logic, and as such require the verification problem for unbounded safety properties to be reduced to a number of bounded time formula checks.

One of the largest stumbling blocks for traditional model checking is the presence of large memories intermingled with complex control logic. This will typically result in very hard or intractable model checking problems. However, proving the correctness of these systems may require reasoning about much smaller number of memory entries.

In this paper, we present an analysis that leverages this fact together with word-level symbolic memory information to abstract model checking problems with large memories to problems that contain fewer memory slots. Our analysis is specifically geared to proving properties, but it could potentially be useful for finding bugs as well.

We have three prime design requirements for our analysis: (1) we do not want the user to have to provide any annotations or follow some particular design style for our analysis to be applicable, (2) we do not want to commit to the use of some particular model checking technology such as the use of SMT-solvers or the use of a specialized logic, and (3) we want our abstraction to be a computationally cheap netlist-to-netlist transformation so that we allow the use of the standard algorithms for iterated simplification and property checking in a *transformation-based verification environment* [3].

Our design requirements have several important implications. First of all, as we want our approach to be independent of the back-end model checking technology, our transformation will be orthogonal to the standard formula-based word-level approaches. Second, by leveraging word-level information to abstract the problem on the netlist level rather than changing the model checking procedure itself, we will be able to benefit from the differing characteristics of any of the multitude of model checking procedures available today, including both methods based on BDD-based fixpoint computations and SAT-based methods.

Our main idea is to abstract the design by remodeling memories to only represent a fixed, smaller number of memory slots. Writes to slots that are not represented are dropped by our abstracted memory, and reads to unrepresented slots return nondeterministic values. As the abstracted memories can produce all traces that the original memories can generate, our transformation is sound (but not necessarily complete).

In order to automatically come up with the slots we want to represent, we will be using an abstraction refinement loop, where counterexamples for the abstracted system are analyzed to determine what slots that need representing currently are missing.

II. Preliminaries

In the remainder of this presentation, we will be concerned with checking safety properties of synchronous hardware. We assume that a standard modeling of designs are used where all properties and constraints have been compiled into the netlist. All inputs to the netlist are thus unconstrained. The failure of some property is signaled by a dedicated single bit output *safe* assuming the value FALSE. A proof of safety thus needs

978-1-4244-2735-2/08/$25.00 ©2008 IEEE

to demonstrate that no input trace from an initial state of the netlist can reach a state where *safe* fails.

The representation that we operate on is a word-level netlist Directed Acyclic Graph (DAG) representation. We obtain the DAG representation of a verification problem using a standard front-end compilation process that utilizes a Hardware Description Language (HDL) compiler that retains as much word-level information as possible. Our representation is rich enough to represent full synthesizable System Verilog.

All nodes in the DAG representation represents a bitvector of some fixed width k. We sometimes annotate nodes with superscripts denoting their width; $\texttt{sig}^4$ signifies a four bit signal called `sig`.

The sources in the DAG are input variables, current state register variables, and bitvector constants. The sinks of the graph are the one-bit output *safe* together with the next-state registers nodes. If the current-state node for a register is named `r`, we follow the naming convention that the next-state node will be named `r'`. Each current state register has some particular ternary bitvector as its initial state; these vectors characterize the set of initial state bitvectors for the register as the set of vectors resulting from all possible instantiations of the X-values.

The computational semantics of our netlist problems is very simple. The netlist is implicitly clocked: one time instance of computation propagates values from the current state registers and inputs, and generates next state values and a value for the output *safe*. Our model checking problem consists of deciding whether there exists some binary initial state and sequence of input vectors that will take the design in zero or more steps to some state where *safe* assumes the value FALSE.

The internal nodes in our graph representation for a design verification problem have the form

- $\texttt{nd}^k = \texttt{op}_l(\texttt{op1}^i, \texttt{op2}^j, \ldots)$
- $\texttt{nd}^k = \texttt{extract}(k, l, \texttt{op}^m)$
- $\texttt{nd}^k = \texttt{concat}(\texttt{op1}^i, \texttt{op2}^j, \ldots)$
- $\texttt{nd}^k = \texttt{mux}(\texttt{cond}^1, \texttt{optrue}^k, \texttt{opfalse}^k)$
- $\texttt{nd}^k = \texttt{read}(\texttt{op}^i, \texttt{addr}^j)$
- $\texttt{nd}^k = \texttt{write}(\texttt{op}^k, \texttt{addr}^i, \texttt{data}^j)$

The `op` nodes in the network are particular combinational functions of their inputs. The function computed by these nodes range from boolean operators and comparison operators to arithmetic functions such as $+$ or $*$. The `extract` nodes project out k bits from the bitvector `op` starting at bit l, and the `concat` node concatenates its operands into a larger signal. The `mux` node has a single bit node `cond` as input that selects whether to return `optrue` or `opfalse`.

The nodes `read` and `write` are used for modeling memories. Their semantics are simple. If a `read` node has width w, it returns the result of projecting out the bits $[\texttt{addr}{*}w \ldots (\texttt{addr}+1){*}w-1]$ from the bitvector `op`; if a write node's `data` operand has width w, it returns the bitvector that would result from overwriting the region $[\texttt{addr} * w \ldots (\texttt{addr}+1) * w - 1]$ of the bitvector `op` with `data`. The address space of read and write nodes is not restricted in any particular fashion; out-of-bounds reads return nondeterministic values for nonexisting bits, whereas out-of-bounds writes do nothing. Also note that there are no dedicated special register nodes that represent large memories, or restrictions on what signals read and write nodes can be applied to: RT-level memories are compiled to bitvector state registers that are no different than the registers used for modeling other design artifacts. By appropriate use of control logic together with multiple read and write nodes, the DAG representation supports arbitrarily complex memory interfaces with large number of read and write ports. Moreover, by nesting reads, writes, and other nodes, we can implement complex policies on update and read orders resulting from same-cycle reads and writes.

III. A MOTIVATING EXAMPLE

As an example, consider the netlist problem in Figure 1. This example has three word-level inputs (`raddr`, `waddr` and

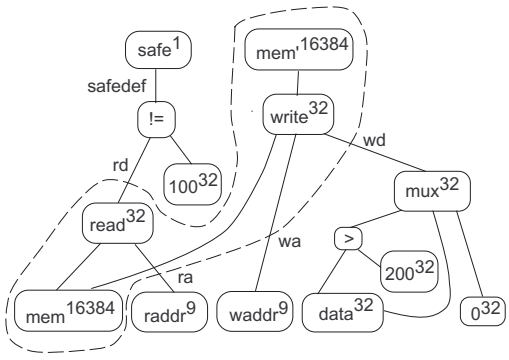


Fig. 1. A simple word-level netlist verification problem.

`data`), and one word level register `mem` which is initialized to the constant 0^{16384}. At each clock cycle the system reads the content of address `raddr` from `mem`. It also writes the input `data` to the address `waddr` if `data` is greater than 200; otherwise it writes 0. The property we check is that the value read from `mem` never is equal to 100. Clearly, this is true for any execution trace for the system. Moreover, as we will see, we can prove this statement by just reasoning about the contents over time of a single slot of the memory—the last slot read.

The circuit modeled in Figure 1 can conceptually be partitioned into two parts. The first part contains the large memory `mem`, and communicates with the rest of the design through two inputs and two outputs: A write address port `wa`, a write data port `wd`, a read address port `ra`, and a read data port `rd`. This part of the design resides inside the dashed line in Figure 1. The second part is the rest of the design.

We can abstract the memory part by removing the original implementation inside the dashed line, and instead instantiating a much simpler memory that contains two registers, $\texttt{sel}^9$ and $\texttt{cont}^{32}$ (see Figure 2). The `cont` register will represent the content of one memory slot selected by the `sel` register. The slot selected by `sel` is chosen during initialization of the circuit, and stays the same during the subsequent system

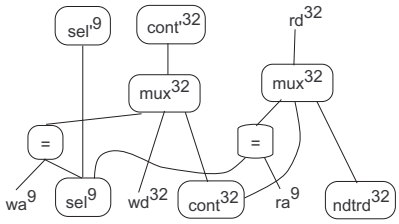


Fig. 2. Reimplementation of the memory part of the netlist from Figure 1.

execution. Specifically, in the new abstracted memory implementation in Figure 2, the register $\texttt{sel}^9$ has an unconstrained initial state, and a next state function that just propagates the current state value to the next state. The register `cont` is initialized to the value zero (remember: all slots in `mem` are zero initialized). In the new memory implementation, reads from address `raddr` return the value of `cont` if $\texttt{sel} = \texttt{raddr}$, otherwise they return a value read from the environment on an unconstrained input `ndtread`. Writes to the memory updates `cont` if $\texttt{sel} = \texttt{waddr}$, otherwise it retains its value.

The new memory implementation overapproximates the original memory as every I/O trace for the original partition can be generated by the modified memory. Every netlist resulting from substituting this simpler memory implementation for the original memory subsystem will hence overapproximate the original netlist.

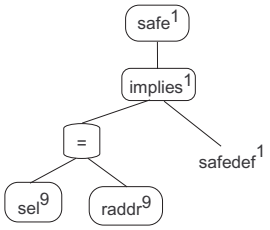


Fig. 3. The modified *safe* output .

Unfortunately, if we attempt to prove the modified system correct, we will immediately get a spurious counterexample where (1) the value of `raddr` in the final cycle is different from the value chosen as the initial value for `sel` and (2) `ndtread` has the value 100. The slot that was chosen for representation by the initialization of `sel` is hence not in sync with the address that is read in the counter example, and an erroneous value is read from the environment.

We can remove this spurious trace by changing the definition of correctness so that we only check the safety of the system when the address that is read the current clock cycle is the one that is represented. A simple way of doing this is to rewrite the combinational output signal *safe* from $\texttt{read}(\texttt{mem}, \texttt{raddr}) \neq 100$ to the signal $\texttt{sel} = \texttt{raddr} \rightarrow \texttt{read}(\texttt{mem}, \texttt{raddr}) \neq 100$, as shown in Figure 3. The resulting modified system N_{abs} is easily shown to be safe.

While it may not be immediately obvious, our remodeling of the memory together with the revised notion of correctness captured by our rewriting of the *safe* output cone is sound: To see this, we can view our rewritten netlist as having been produced in two stages.

In the first stage, we produce a netlist N' identical to the original netlist N with the exception that we have added the `sel` register to the netlist, and rewritten the *safe* output only. The resulting netlist is a sound abstraction of N: Assume that there exists a trace of the original system where at some execution time point $\texttt{read}(\texttt{mem}, \texttt{raddr})$ is 100 for some particular value v of `raddr`. Then the run of N' that is identical to the original trace, but initializes `sel` to v forces the modified safety output $\texttt{sel} = \texttt{raddr} \rightarrow \texttt{read}(\texttt{mem}, \texttt{raddr}) \neq 100$ to fail in the last step of the trace too (remember, we have not changed the memory implementation; we have only changed the property and added one register). As all counterexamples are preserved, this first stage of the transformation is hence sound.

In the second stage, we rewrite the memory in the netlist N' produced by the first transformation so that it only represents the slot chosen by `sel`. This system overapproximates N' as the reimplementation of the memory subsystem simulates the original memory. The second transformation hence also retains soundness, and any proof for the resulting system will guarantee safety of the original netlist N.

In order to systematize the technique used to solve the example in Figure 1, we need to solve three different problems: (1) we need to automatically extract abstractable memories from our netlist, (2) we need to substitute simpler, lossy, memory implementations for the original memories, and (3) we need to come up with an automatic way to extract information on what slots we need to represent in order to infer correctness. Next, we show how these problems can be solved.

IV. Identification of Remodellable Memories

The first part of our analysis will be to scan the DAG under analysis for what we term *remodellable memories*. These are memories that we can abstract. Informally, remodellable memories are addressed in a uniform way, only communicate with the rest of the design in such a way that another implementation easily can be substituted, and have a next state function of a particular simple structure. (Our implementation handle several simple generalizations of remodellable memories, as outlined in Section VI, but we will not delve into these generalizations here in order to keep the exposition simple.)

In order to formally define our notion of a remodellable memory, let us first introduce some notation. Given a register variable `mem`, the set of *pure memory nodes* for `mem` are recursively defined as follows:

- the node `mem` is a pure memory node for `mem`.
- $\texttt{write}(\texttt{op}^k, \texttt{addr}^i, \texttt{data}^j)$ is a pure memory node for `mem` if $\texttt{op}^k$ is a pure memory node for `mem`.
- $\texttt{mux}(\texttt{cond}^1, \texttt{optrue}^k, \texttt{opfalse}^k)$ is a pure memory node for `mem` if $\texttt{optrue}^k$ and $\texttt{opfalse}^k$ are pure memory nodes for `mem`.

The set of *pure read nodes* for a state variable mem is comprised of all netlist read nodes $\texttt{read}(\texttt{op}^i, \texttt{addr}^j)$ for which $\texttt{op}^i$ is a pure memory node for mem. Analogously, the set of *pure write nodes* for mem is comprised by all netlist nodes $\texttt{write}(\texttt{op}^k, \texttt{addr}^i, \texttt{data}^j)$ for which $\texttt{op}^k$ is a pure memory node for mem.

We can now define a remodellable memory to be a register state variable mem that fulfills the following properties:

1) all read and write nodes that have mem in their support read and write data of the same width w using address nodes of the same width a. Moreover, the bitwidth of mem is an integer multiple m of w, and $2^a \leq m * w$ so that all memory accesses occur inside the memory.
2) mem is either initialized to the boolean constant $000\ldots0$ or $111\ldots1$. (We lift this requirement in our implementation, as outlined in Section VI.)
3) the next state function for mem is a pure memory node for mem, and no other next state function is a pure memory node for mem.
4) every fanout path from mem is made up of a sequence of pure memory nodes terminating either in (1) the next state node for mem, or (2) a pure read node.

Requirement 1 ensures that the memory is treated as a bitvector of slots that are read and written in a uniform way. Requirement 2 ensures that all of these slots have the same initial state, which guarantees that the selected slots we will model all have the same initial state. The remaining requirements ensure that the memory register only occur in the fanin of other state registers and outputs through read nodes, and that the next-state function for the memory is a simple multiplexor tree that chooses between different write nodes updating the memory.

Given a netlist problem encoded as a word-level DAG, we use straight forward linear traversal algorithm to extract the set of remodellable memories and compute their associated sets of read and write nodes. Note that while some memories that could be used in a design are not remodellable as per our definition, our experience is that the majority of large memories used in datapath computations in practical hardware designs fall into this category. Our definition of an remodellable memory seem to provide a good balance between being able to cover most interesting memory implementations, while being simple enough to provide relatively straight forward memory abstraction algorithms.

V. Memory Abstraction

A. Remodellable Memory Abstraction

We now systematize the approach used in our motivating example.

In the case of the design from Section III, we abstracted the memory over the current value of $\texttt{raddr}^9$. While this worked well for this particular system, for many systems, memory accesses from a number of previous time instances have to be performed correctly to guarantee that the systems correctness can be checked in the current cycle. For example, if we are checking that a complete multi-part message is always forwarded correctly, then we may have to perform a sequence of reads correctly over time. We will therefore abstract a remodellable memory over a set containing *abstraction pairs* (v^l, t), where (1) each v^l is some signal from the netlist and (2) t is an integer time delay. We require that all abstraction nodes v^l have the same width l as the address nodes of read and writes operating on the memory. The design from Section III is abstracted over the singleton set $\{(\texttt{raddr}^9, 0)\}$.

In the remainder of this section, we will assume that the design that we want to memory abstract contains a single remodellable memory mem (the extension of the procedure to deal with several remodellable memories at a time is straightforward). We also assume that we have been given a set of abstraction pairs; we show how these pairs are generated in Section V-D

The given remodellable memory is abstracted over the provided abstraction pairs as follows:

1) Introduction of Represented Slots: Assume that the data read and written from the remodellable memory mem has width m, and that we are abstracting mem with n abstraction pairs. We introduce represented slots by performing the following three steps for each abstraction pair $p = (v_i^l, d_i)$.

First, we introduce a state variable $\texttt{sel}_i^l$ for (v_i^l, d_i), with an uninitialized initial state function, and a next state function that just propagates the previous value of $\texttt{sel}_i^l$. This *selection register* will contain the concrete slot number that is represented for this abstraction pair during system runs. We also create a register $\texttt{cont}_i^m$. This register is initialized in a way that corresponds to the initialization of mem: If mem has an all-zero initial value, then so does $\texttt{cont}_i^m$; otherwise mem has an all-one initial value, and we initialize $\texttt{cont}_i^m$ correspondingly. (Remember, mem is remodellable so all of its slots are initialized in a uniform way.)

Second, for each pure memory node for mem we create a (v_i^l, d_i)*-abstracted node* as follows:

- the node mem is (v_i^l, d_i)-abstracted to $\texttt{cont}_i^m$.
- the node $\texttt{write}(\texttt{op}^k, \texttt{addr}^l, \texttt{data}^m)$ is (v_i^l, d_i)-abstracted to $\texttt{mux}(\texttt{sel}_i^l = \texttt{addr}^l, \texttt{data}^m, s_0^m)$ if $\texttt{op}^k$ is (v_i^l, d_i)-abstracted to s_0^m.
- the node $\texttt{mux}(\texttt{cond}^1, \texttt{optrue}^k, \texttt{opfalse}^k)$ is (v_i^l, d_i)-abstracted to $\texttt{mux}(\texttt{cond}^1, s_0^m, s_1^m)$ if $\texttt{optrue}^k$ and $\texttt{opfalse}^k$ is (v_i^l, d_i)-abstracted to s_0^m and s_1^m, respectively.

Third, the next-state function for $\texttt{cont}_i^m$ is taken to be the (v_i^l, d_i)-abstracted node of the next-state function for mem; this is possible as the definition of a remodellable memory guarantees that the next state function for mem is a pure memory node for mem.

2) Reimplementation of Read Nodes: We can now reimplement all read nodes for mem in the following fashion. Assume that the ith read node has the form $\texttt{read}(\texttt{op}^k, \texttt{addr}^l)$. For each abstraction pair (v_j^l, t_j), assume that $\texttt{op}^k$ has been (v_j^l, t_j)-abstracted to s_j^m with the associated selection register $\texttt{sel}_j^l$. Introduce a fresh input variable $\texttt{ndtread}_i^m$. We reimplement the read node using a multiplexor tree that returns the contents

of the first selected slot with a matching address. If the address does not match any selected slot, we return a nondeterministic value read from the environment on the input $\texttt{ndtread}_i^m$.

Example 1: The multiplexor tree generated for the case of two represented slots will have the form

$$\texttt{mux}(\texttt{addr}^l = \texttt{sel}_0^l, s_0^m, \texttt{mux}(\texttt{addr}^l = \texttt{sel}_1^l, s_1^m, \texttt{ndtread}_i^m))$$

3) Modifying the Verification Condition: The final step of the translation is to modify the output *safe* so that the property only is checked when the abstraction signals have had the selected values at the appropriate previous time instances. Let us define $prev^d(s)$ as a temporal formula that is true at time t in the execution of a system precisely if $t \geq d$ and the combinational signal s evaluates to true at time $t-d$. Assume there are n abstraction pairs (v_i^l, d_i) for s. Our new *safe* output can now be generated by synthesizing a checker for the temporal formula

$$(\bigwedge_{i=0}^{n-1} prev^{d_i}(\texttt{sel}_i^l = v_i^l)) \rightarrow safedef$$

where *safedef* is taken to be the combinational node feeding the old *safe* output. This checker can be implemented in a very simple manner using a number of register chains that delays previous values of some netlist node comparisons.

As the original memory `mem` is remodellable, it can only occur in the fanin of other state variables through read nodes. We have reimplemented all the read nodes, so the netlist can now be reduced by removing the original memory `mem` and all logic that depends on it.

B. Correctness

Our notion of correctness is the following: If we have n abstraction pairs (v_i^l, d_i) for a remodellable memory in a design that we are abstracting, and the largest time delay in any abstraction pair is d, then *safe* will always hold in the original system if (1) the transformed signal *safe* in the memory-abstracted system always holds, and (2) *safe* holds in the original system for d cycles from the initial states.

To see that this always holds, we can follow the reasoning used to show the soundness of the transformation of our example from Section III. Specifically, we again remark that the composite system that we produce can be seen as constructed by a two-step process.

In the first step we (1) introduce the selector registers $\texttt{sel}_i^l$ that will be used to control what slots are kept by the abstracted memory (but we do not abstract the memory), and (2) generate a new *safe* signal by synthesizing a checker for the temporal formula

$$(\bigwedge_{i=0}^{j-1} prev^{d_i}(\texttt{sel}_i^l = v_i^l)) \rightarrow safedef$$

where *safedef* is the fanin node for the old *safe* output. To see that this transformation is sound, assume that there is a counterexample trace π in the original system of length l and that $d = max(d_0 \ldots d_{n-1})$. If $l < d$, then we will detect the bug using the bounded check of the original system. If $l \geq d$, some particular initial values for the selectors $\texttt{sel}_i^l$ will make the antecedent of the implication hold in the last time step of π (remember, the initial states for the selector variables are unconstrained, and these variables hold their initial state forever). Furthermore, π will make the consequent of the implication fail in the final time instance. The trace π with some particular initialization for the new selector variables will hence be a valid counterexample for the modified system, and the first transformation is hence sound.

In the second step, the memory in the system resulting from the first step is abstracted to only represent the slots selected by the $\texttt{sel}_i^l$ registers. Due to our construction of the new memory implementation, all reads and writes to slots chosen for representation by the selector variables in our modified memory will happen correctly during all system execution traces. The set of traces generated by this abstraction of the remodellable memory structure is hence a superset of the set of traces of the original memory, so any property that holds on the modified composite system holds on the original system. The second step is hence also sound, as is the overall flow.

C. Abstraction Size

Let us analyze the size of the generated abstracted netlist. Assume that the data read and written from given memory `mem` has width w, all address nodes have width a, there are i abstraction pairs, j read nodes, and that the maximum delay value in any abstraction pair is d. Each abstraction entry generates a selector entry `sel` and a memory storage slot `cont` of width a and w respectively. Furthermore, each read node generates a new nondeterministic input of width w, and the synthesized monitor for the output condition can be implemented with d total single bit delay elements. As a result, the remodeled system will have shrunk the number of necessary registers for the memory encoding from $w * 2^a$ bit-level registers to $i*(a+w)+j*w+d$ bit-level state variables. In the case of the netlist in Section III the memory in the original system has more than 16000 registers and our abstracted implementation will use 75 state variables—a decrease by two orders of magnitude.

D. Generation of Abstraction Pairs

Section V-A shows how a given problem can be abstracted with respect to a particular set of abstraction pairs. However, we have not discussed how to generate abstraction pairs for a particular design problem. One alternative would be to rely on the user for this information. However, this would be error prone, effort intensive, and decrease the utility of our approach significantly. To get around these problems, we instead apply a *counter-example guided abstraction refinement framework* [7] (see Figure 4).

For every remodellable memory in the design, we maintain a current set of abstraction pairs that is monotonically growing. Initially this set is empty for each memory (this initial abstraction generates a system where no slots are represented and each read from a memory returns a nondeterministic result).

Each iteration of our verification flow proceeds as follows:

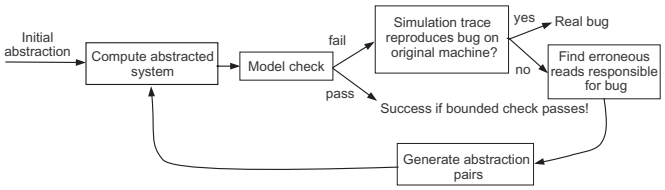


Fig. 4. Our abstraction refinement loop

Given the current set of abstraction pairs, we generate a new abstracted model of the system using the techniques from Section V-A and use bit-level model checking to check correctness. If the property holds on the abstracted system, we check bounded correctness of the original system for the necessary number of cycles using standard SAT-based bounded model checking, and declare the system correct if this check passes. If the bounded check of the original system fails, the original system is faulty.

If the model check detects a counter example on the abstracted system, we try to refine our abstraction. The inputs of the abstracted system are a superset of the inputs of the original system, so we can replay the counter example from the abstraction on the original system. If the bug replays on the original system, we are done and report an error to the user. However, if the bug does not replay, we have to refine the abstraction to remove the error trace.

The only difference between the original and abstracted system that could introduce a spurious counterexample is the memory encoding. Some abstracted read node at some time instance must hence return the contents of an unrepresented slot. By inspecting the simulation run on the original system and comparing the values of the pre- and post-abstraction read nodes, we can identify erroneous reads over time in the execution of the abstracted system.

Not all of the erroneous reads will have an impact on the checked property. We find a minimal set of reads and associated time points by initially forcing correct values for all erroneous reads in the abstract system simulation, and iteratively shrinking this corrected set in a greedy way until we find a local minima of forced reads.

Now that we know what erroneous reads need to be corrected and at what time distances from the error cycle, we need to choose the abstraction signal for each time point. We use the following heuristic: If the fraction of read nodes for a memory relative to the number of memory slots for is smaller than 20%, then we abstract on the address signals of the failing reads at each time point. However, if it is not, then we search for a register that 1) is of the same size as the address width of the memory, 2) is in the cone-of-influence of the memory, and 3) contains the address value being read by the erroneous read node at the time instance that the read is performed. We search for the first such register we can find, and if we succeed we use this register node as the abstraction signal at the correct time point rather than the forced read address signal.

The rationale for our abstraction signal choice in the presence of a large number of memory read nodes is that if we are to succeed, we have to hope that there is a register entry that contains the important slot to focus on somewhere else in the design. This is crucial for being able to deal with certain designs such as the content addressable memory in Section VII, where every entry in the memory is read in each cycle, but only a small number of reads at a given time matter.

Example 2: Assume we have a counter example of length 15 for an abstracted version of a design with a single remodellable memory `mem` with 32 slots and two read nodes. If we detect that the read node $\texttt{read}(\texttt{mem}^{1024}, \texttt{raddr}_1{}^5)$ needs to have a correct value at cycle 13 (one time step before the failure cycle) to remove the bug trace, then we add the abstraction pair $(\texttt{raddr}_1{}^5, 1)$ to the current abstraction set for `mem`. However, if `mem` had 28 read nodes, then we would search for a register $\texttt{reg}^5$ that at cycle 13 contained the concrete address that the read failed for and abstract on $(\texttt{reg}^5, 1)$ instead. If no such register exists, we revert to the an unabstracted modeling of `mem`.

VI. Practical Considerations

We use several improvements to the basic algorithms presented in Section V to improve the accuracy and practicality of our framework.

Real designs often contain more than one remodellable memory. We have therefore extended our algorithms to be able to deal with several remodellable memory simultaneously. Moreover, for simplicity of exposition, our presentation in Section V is centered around memories that either are initialized to the constant value $00\ldots0$ or $11\ldots1$ value. In order to deal with real designs it is also necessary to deal with uninitialized memories, and memories with nonuniform initialization. We have also generalized our algorithms to deal with memories with potential out-of-bound reads and writes. These extensions are performed by generating more complex logic for the new implementation of read nodes and selected memory locations.

If a remodellable memory is small but still needs many abstraction pairs, it may be more expensive to abstract it than to bit blast it into a register bank implementation. One example of when this may happen is when some design artifact is represented as an array of single bits. To deal with this problem, we remove memories from our set of remodellable memories when the size of the abstraction of a given memory approaches 75% of the size of the original memory implementation.

In our presentation of the remodellable memory abstraction we introduce one fresh word-level input for each read node to model reads from unrepresented slots. An alternative encoding, that is more efficient for designs with a large number of reads, is to selectively employ a dual rail encoding [15] where appropriate, and return the value X for unrepresented reads. This encoding incurs a one bit overhead for each word-level register in the recursive fanout of the read nodes, as opposed to a fresh width k input per read node. We dynamically evaluate which encoding will work best, and choose the one the generates the smallest amount of registers.

VII. Experimental Results

We have applied our memory abstraction algorithm to two industrial designs, and one state-of-the-art academic design. These designs are representatives from classes of problems that users see very little return-on-investment from traditional model checking technology on today, due to the presence of wide datapaths and the intermingling of data with control. Unfortunately, these designs also represent fundamental building blocks of many larger hardware systems in important domains like telecommunications, networking and graphics processing. Procedures that allow proofs of full or partial correctness of such systems are hence of very high importance.

In our experiments, we combine the memory abstraction with our property preserving word-level bitwidth reduction [4] before generating the bit-level model for analysis. This reduction runs in less than one second for the examples we study, often provides dramatic further reductions, and is guaranteed to never increase the netlist size on designs that are not amenable to it.

Our first example is an industrial FIFO with 75 slots, each containing 32 bits of data. We check the end-to-end correctness safety property that data that gets written into the FIFO gets read out correctly if it has not been overwritten. The original design has about 2500 registers, which are all necessary for a direct full proof, and the standard black-box checker contains a simple reference array which stores correct values to check outputs against. We have not managed to prove this property using any available bit-level method, even when run for several days. The memory abstraction immediately reduces the design to contain one slot after one ten second iteration of abstraction refinement, which shrinks the design to 276 registers. Our bitwidth reduction pass then shrinks the design further to a 58 register equivalent implementation, which BDD-based model checking proves correct in about 20 minutes. Interestingly, although the final reduced design is very small, it is quite complex—over 19000 forward image computations are necessary to traverse its full state space. We also applied our method to a version of the design with a more space efficient white-box checker, which nondeterministically chooses a slot in the FIFO to observe and then monitors correctness of transactions to that particular slot. Verification runtimes do not change when we use this modified checker.

The second example is an industrial Content Addressable Memory (CAM) with three data ports. The CAM has 48 slots, each storing a 20 bit data packet. We check that data that have been written into the CAM are reported as present when the memory is queried, as long as it has not been evicted. The pre-abstraction netlist has 1111 registers, all of which is necessary for a direct proof. No method in our arsenal manages to provide an unbounded proof for this property. One five second round of abstraction refinement reduces the CAM to one slot, which reduces the netlist size to 156 registers. The bitwidth reduction in turn decrease the size of the model to 26 registers, which can be proven using BDD-based model checking instantaneously.

Our third example is an academic high performance router [13]. The router has four ports that connect to adjacent routers, plus inject and eject ports. Messages passed into the router are broken up into *flits*—packets that include both control data and message payload. The design uses two virtual channels per port, each with an associated flit buffer and an array containing information on whether buffer entries are valid. We check the simple partial correctness property that if the router is in a neutral state, then it will forward a message broken up into a header packet, some payload, and a footer packet correctly from the inject port to the north port in a certain number of cycles. This property is provable on the unabstracted model using SAT-based induction [16], but it takes more than 6900 seconds to find the correct induction depth and prove the goal. We detect twenty remodellable memories in the router, which has 7516 registers before the abstraction. Two rounds of abstraction refinement, which takes about 200 seconds, detects that all the validity memories have to be modeled explicitly, whereas we only need to represent two reads at two timepoints from one of the buffer memories. After bitwidth reduction, we end up with a design containing 2196 registers. Induction proves this reduced system in 133 seconds.

VIII. Related Work

Our circuit transformation hinges on the fact that we can compile general RT-level circuit descriptions into a representation that encodes memory manipulations using dedicated read and write nodes. This word-level approach to describing memory manipulating circuitry was pioneered by Burch and Dill in their work on microprocessor verification using quantifier-free logic [6].

The most closely related work to our memory abstraction is the efficient memory modeling that relies on the boundedness of unquantified formula queries used by Ganai and coauthors [9], [10], and by `BAT` [11]. Our technique is fundamentally different from these approaches, as we *abstract* a sequential *netlist* rather than *simplify* a bounded *formula*. As a case in point, we observe that the standard efficient memory modeling would have provided no reduction for the FIFO and the router from Section VII: From BDD-based model checking of the abstracted design, we know that the FIFO has a backward depth that exceeds 100 time steps (and would hence need more than depth 100 induction). We also know that the router needs depth 20 induction for a proof. These unfolding depths exceed the number of entries in the memories in these designs, so efficient memory modeling can provide no benefits as it must model at least as many slots as the induction depth necessary for a proof. In contrast, we can abstract the FIFO to one slot, and remove several memories completely from the router.

In the domain of software model checking, Armando and coauthors have devised a method for abstracting linear programs operating on arrays [2]. Their idea is to use an abstraction refinement framework to iteratively identify a subset of array elements that needs to be modeled in order to prove a

given property. As is the case in our procedure, array entries that are not modeled return nondeterministic values when read. The main difference between Armando and coauthors work and ours, is that our approach can deal with problems where the array indexes that need to be represented vary in each execution sequence of the problem. In fact, none of the examples in Section VII can be verified by abstracting on some particular nontrivial subset of fixed slots.

In previous work, Symbolic Trajectory Evaluation (STE) [15] has been used to verify correctness of designs such as CAMs that contain large memories through the use of a technique called *symbolic indexing* coupled with a specialized abstraction refinement framework [1]. STE verifies bounded length temporal properties for hardware, and has been successful in internal verification tools at a number of semiconductor companies; however, in commercially available tools, unbounded model checking of safety properties has become the dominant technology. Our work allows the benefits of memory abstraction in a framework that the vast majority of users of formal verification technology is familiar with, without requiring them to learn a new logic, rewrite properties, or change their verification methodology.

Our approach for the memory abstraction can be seen as a combination of (1) the selective abstraction of certain memory slots, and (2) an abstraction refinement loop [7]. In previous work, McMillan has used abstract interpretation together with a proof technique he calls *temporal case splitting* to decompose the refinement checking of complex designs like implementations of Tomasulo's algorithm [12] into refinement checking of several abstract models that each only models a small number of memory slots. McMillan performs his abstraction by manually annotating designs with information on how to decompose the checks. In contrast, we focus on solving safety property verification problems, and have structured our approach as a completely automatic algorithm that requires no user input.

There are other attempts to leverage word-level information during symbolic model checking of safety properties. In particular, research has been done on improving the effectiveness of model checking for systems with wide data paths involving arithmetic [8]. These techniques do not deal with symbolic memories in any special way, and their use is thus orthogonal to our memory abstraction. As a result, we can use such word-level model checking tools as our back-end decision procedure and leverage whatever improvements they offer.

In previous work, we have investigated the use of static analysis to improve bounded and unbounded safety property verification of industrial designs [4]. While this work (1) allows us to reduce the size of the verification problems in Section VII significantly and (2) consistently produce significant speed-ups for bounded checks after the reduction, the bitwidth reduction alone does not always manage to shrink designs to the point where we can decide the properties in the unbounded case. In contrast, the combination of memory abstraction and bitwidth reduction allows us to get to full unbounded proofs.

IX. Conclusion

In this paper we have introduced a method that uses word-level netlist information to identify a particular kind of memories that we refer to as remodellable. Such memories interact with their environment using dedicated read and write nodes only, are initialized in a uniform way, and are accessed uniformly. We have presented an abstraction for netlists containing such memories that allows proofs for certain types of properties for which the proof can be done by reasoning about a significantly smaller number of memory slots and time instances than what would be needed in a standard bit-level model check. Our abstraction is sound but not complete. In order to avoid having to rely on abstraction information from the users of our algorithm, we have coupled the abstraction with a counter-example driven abstraction refinement framework that analyze spurious counter examples to incrementally refine the abstraction.

Key features of our approach is that (1) we seamlessly fit into a standard transformation-based verification system for safety property verification, (2) our algorithms are completely automatic, (3) we require no input on abstraction from users, (4) we can use any bit-level model checker as the decision procedure in our abstraction refinement framework.

As demonstrated by our experimental results in Section VII, we can prove correctness properties for industrial and academic high-performance designs that are out of range for standard bit-level model checkers. The designs we analyze have wide datapaths, manipulate large swaths of memory, and have nontrivial control structure. Approaches that rely on explicit modeling of all memories, and the flow of information between each and every memory cell hence end up with decision problems that can be extraordinarily more expensive to decide than our reduced models.

We believe that the work presented here, together with our previous work on utilizing word-level information to perform bitwidth reduction, will provide a way to leverage model checking on some classes of designs that today are intractable for users of industrial tools. Hopefully, the present research represents just the tip of the iceberg in terms of ways to utilize higher-level information to speed up safety property verification.

Acknowledgments

Many thanks to Tamir Heyman, who participated in discussions and helped with the work necessary to integrate our analysis with the front-end flow. The authors would also like to thank Jason Baumgartner, Aarti Gupta, John Matthews, and the anonymous reviewers for their helpful feedback on previous versions of this paper.

References

[1] S. Adams, M. Björk, T. Melham, and C.-J. Seger. Automatic abstraction in symbolic trajectory evaluation. In *Proc. of the Formal Methods in CAD Conf.*, 2007.

[2] A. Armando, M. Benerecetti, and J. Mantovani. Abstraction refinement of linear programs with arrays. In *Proc. Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, 2007.

[3] J. Baumgartner, T. Gloekler, D. Shanmugam, R. Seigler, G. V. Huben, H. Mony, P. Roessler, and B. Ramanandray. Enabling large-scale pervasive logic verification through multi-algorithmic formal reasoning. In *Proc. of the Formal Methods in CAD Conf.*, 2006.

[4] P. Bjesse. A practical approach to word level model checking of industrial netlists. In *Proc. of the Computer Aided Verification Conf.*, 2008.

[5] R. Bryant, S. Lahiri, and S. Seshia. Modeling and verifying systems using a logic of counter arithmetic with lambda expressions and uninterpreted functions. In *Proc. of the Computer Aided Verification Conf.*, 2002.

[6] J. L. Burch and D. L. Dill. Automatic verification of pipelined microprocessor control. In *Proc. of the Computer Aided Verification Conf.*, 1994.

[7] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *Proc. of the Computer Aided Verification Conf.*, 2000.

[8] E. Clarke, M. Khaira, and X. Zhao. Word-level symbolic model checking: avoiding the Pentium FDIV error. In *Proc. of the Design Automation Conf.*, 1996.

[9] M. K. Ganai, A. Gupta, and P. Ashar. Efficient modelling of embedded memories in bounded model checking. In *Proc. of the Computer Aided Verification Conf.*, 2004.

[10] M. K. Ganai, A. Gupta, and P. Ashar. Verification of embedded memory systems using efficient memory modeling. In *Proc. of Design, Automation, and Test in Europe.*, 2005.

[11] P. Manolios, S. Srinivasan, and D. Vroon. BAT: The bit-level analysis tool. In *Proc. of the Computer Aided Verification Conf.*, 2007.

[12] K. L. McMillan. Verification of an implementation of Tomasulo's algorithm by compositional model checking. In *Proc. of the Computer Aided Verification Conf.*, 1998.

[13] L.-S. Peh and W. Dally. A delay model and speculative architecture for pipelined routers. In *Proc. Intl. Symposium on High-Performance Computer Architecture*, 2001.

[14] S. Ranise and C. Tinelli. Satisfiability modulo theories. *Trends and Controversies - IEEE Intelligent Systems Magazine*, December 2006.

[15] C.-J. H. Seger and R. E. Bryant. Formal verification by symbolic evaluation of partially ordered trajectories. *Formal Methods in System Design*, 6(2):147–190, March 1995.

[16] M. Sheeran, S. Singh, and G. Stålmarck. Checking safety properties using induction and a SAT-solver. In *Proc. of the Formal Methods in CAD Conf.*, 2000.

Combining Predicate and Numeric Abstraction for Software Model Checking

Arie Gurfinkel and Sagar Chaki
Software Engineering Institute, Carnegie Mellon University
{arie,chaki}@sei.cmu.edu

***Abstract*—Predicate (PA) and Numeric (NA) abstractions are the two principal techniques for software analysis. In this paper, we develop an approach to couple the two techniques tightly into a unified framework via a single abstract domain called NUMPREDDOM. In particular, we develop and evaluate four data structures that implement NUMPREDDOM but differ in their expressivity and internal representation and algorithms. All our data structures combine BDDs (for efficient propositional reasoning) with data structures for representing numerical constraints. Our technique is distinguished by its support for complex transfer functions that allow two way interaction between predicate and numeric information during state transformation. We have implemented a general framework for reachability analysis of C programs on top of our four data structures. Our experiments on non-trivial examples show that our proposed combination of PA and NA is more powerful and more efficient than either technique alone.**

I. INTRODUCTION

Predicate abstraction (PA) [2] and Abstract Interpretation (AI) with numeric abstract domains, called Numeric abstraction (NA) [5], are two mainstream techniques for automatic program verification. However, the two techniques have complimentary strengths and weaknesses. Predicate abstraction reduces program verification to propositional reasoning via an automated decision procedure, and then uses a model checker for analysis. This makes PA well-suited for verifying programs and properties that are control driven and (mostly) data-independent, e.g., the code fragment in Fig. 1(a). However, in the worst case, reduction to propositional reasoning is exponential in the number of predicates. Hence, PA is not as effective for data-driven and (mostly) control-independent programs and properties, such as the code fragment shown in Fig. 1(b) In summary, PA works best for propositional reasoning, and performs poorly for arithmetic.

On the other hand, Numeric abstraction restricts all reasoning to conjunction of linear constraints. For instance, NA with Intervals is limited to conjunctions of inequalities of the form $c_1 \leq x \leq c_2$, where x is a variable and c_1,c_2 are numeric constants. Instead of relying on a general-purpose decision procedure, NA leverages a special data structure – Numeric Abstract Domain. The data structure is designed to represent and manipulate sets of numeric constraints efficiently; and provides algorithms to encode statements as transformers of numeric constraints. Thus, in contrast to PA, NA is appropriate for verifying properties that are (mostly) control-independent, but require arithmetic reasoning, e.g., the code fragment in Fig. 1(b). On the flip side, NA performs poorly when propositional reasoning (i.e., supporting disjunctions and negations) is required, e.g., for the code fragment in Fig. 1(a).

```
assume(i==1 || i==2);
switch(i)
  case 1: a1=3; break;
  case 2: a2=-4; break;
switch (i)
  case 1: assert(a1>0);
  case 2: assert(a2<0);
  default: assert(0);
```

(a)

```
if(3 <= y1 <= 4)
   x1 = y1 - 2;
   x2 = y2 + 2;
else if(3 <= y2 <= 4)
   x1 = y2 - 2;
   x2 = y2 + 2;
assert(5 <= (x1+x2) <= 10);
```

(b)

Fig. 1. Two example programs.

In practice, precise, efficient and scalable program analysis requires the strengths of both predicate and numeric abstraction. For instance, in order to verify the code fragment in Fig. 2(a), propositional reasoning is needed to distinguish between different program paths, and arithmetic reasoning is needed to efficiently compute strong enough invariant to discharge the assertion. More importantly, the propositional and numeric reasoning must interact in non-trivial ways. Therefore, a combination of PA and NA is more powerful and efficient than either technique alone. Achieving an effective combination of PA and NA is the subject of our paper.

Any meaningful combination of PA and NA must have at least two features: (a) propositional predicates are interpreted as numeric constraints where appropriate, and (b) abstract transfer functions respect the numeric nature of predicates. The first requirement means that, unlike most AI-based combinations, the combined abstract domain cannot treat predicates as uninterpreted Boolean variables. The second requirement implies that the combination must support abstract transformers that allow the numeric information to affect the update of the predicate information, and vice versa.

Against this background we make the following contributions. We present the interface of an abstract domain, called NUMPREDDOM, that combines both PA and NA, and supports a rich set of abstract transfer functions that enables the updates of numeric and predicate state information to be influenced by each other. We propose four data-structures — NEXPOINT, NEX, MTNDD and NDD — that implement NUMPREDDOM. The data structures (summarized in Table I) differ in their expressiveness and in the choice of representation for the numeric part of the domain. Our target is PA-based software analysis. Thus, all of the data-structures allow for efficient (symbolic) propositional reasoning. We present experimental results on non-trivial examples and compare and contrast

978-1-4244-2735-2/08/$25.00 ©2008 IEEE

Name	Value	Example	Num.
NEXPOINT	$2^{2^P} \times N$	$(p \vee q) \wedge (0 \leq x \leq 5)$	EXP
NEX	$2^P \mapsto N$	$(p \wedge 0 \leq x \leq 3) \vee (q \wedge 1 \leq x \leq 5)$	EXP
MTNDD	$2^P \mapsto N$	$(p \wedge 0 \leq x \leq 3) \vee (q \wedge 1 \leq x \leq 5)$	SYM
NDD	$2^P \mapsto 2^N$	$(p \wedge (x = 0 \vee x = 3) \vee (q \wedge (x = 1 \vee x = 5)))$	SYM

TABLE I
Summary of implementations of NUMPREDDOM; P = predicates; N = numerical abstract values; **Value** = type of an abstract element; **Example** = example of allowed abstract value; **Num** = numeric part representation (explicit or symbolic).

```
assume(x1==x2);
if (A[y1 + y2] == 3)
  x1 = y1 - 2;
  x2 = y2 + 2;

else
  A[x1 + x2] = 5;
if (A [x1 + x2] == 3)
  x1 = x1 + x2;
  x2 = x2 + y1 - 2;
assert(x1==x2);
```

(a)

assume$(x_1 = x_2)$;
((*assume*(p);
  $x_1 := y_1 - 2 \wedge q :=$ *choice*$(\mathtt{f}, \mathtt{f})$;
  $x_2 := y_2 + 2 \wedge$
    $q :=$ *choice*$(x_1 + 2 = y_1 \wedge p, \mathtt{f}))) \vee$
(*assume*$(\neg p)$;
  $q :=$ *choice*$(\mathtt{f}, \mathtt{t})$));
((*assume*(q);
  $x_1 := x_1 + x_2$;
  $x_2 := x_2 + y_1 - 2) \vee$ *assume*$(\neg q)$);
assert$(x_1 = x_2)$

(b)

Fig. 2. A program (a), and its abstraction (b) with $V_P = \{p, q\}$, $V_N = \{x_1, x_2, y_1, y_2\}$, where $p \triangleq ((A[y_1 + y_2] = 3)$, and $q \triangleq (A[x_1 + x_2] = 3)$.

between the four data-structures on the basis of these results. Our experiments show that the proposed combination is more powerful and more efficient than either PA or NA alone.

The rest of the paper is structured as follows. We survey related work in Section II and review background material in Section III. In Section IV, we present the interface of NUMPREDDOM. In Section V, we describe the particularities of each of our NUMPREDDOM implementations. Finally, experimental results and conclusions are presented in Section VI.

II. RELATED WORK

The problem of combining PA and NA involves combining their abstract domains, and is well studied in AI [9]. A typical solution is to combine the domains using a domain combinator such as direct, reduced [8], [9], or logical [12] products. The result can be further extended with disjunctions (or unions) using a disjunctive completion [9]. The domains we develop in this paper are variants of (disjunctive completion of) reduced product between domains of PA and NA.

One approach for combining abstract domains is to combine results of the analyses – e.g., by using light-weight data-flow analyses, such as alias analysis and constant propagation – to simplify a program prior to applying predicate abstraction. Thus, the invariants discovered by one analysis are assumed by the other. For instance, Jain et al.[14] present a technique to compute numeric invariants using NA which are then used to simplify PA. However, this approach only works when the verification task can be cleanly partitioned into arithmetic and propositional reasoning. For example, it is ineffective for verifying the program in Fig. 2(a), where purely numeric reasoning is too imprecise to produce any useful invariants.

Another approach is to run the analyses over different abstract domains in parallel within a single analysis framework, using the abstract transfer functions of each domain as is. The analyses may influence each other, but only through conditionals of the program. This approach is often taken by large-scale abstract interpreters [5], that use different abstract domains to abstract distinct program variables. Recently, a similar approach has been incorporated into software model-checker BLAST [10], [4], [3] to combine predicate abstraction with various data-flow analyses. In principle, this can be adapted to combining PA and NA. The expressiveness of this combination is comparable to NEXPOINT – our simplest combined domain.

From the approaches that tightly combine predicate and numeric abstractions the work of Bultan et al. [7] is closest to ours. They present a model-checking algorithm to reason about systems whose transition relation combines propositional and numeric constraints. Their algorithms are based on a data structure that uses BDDs [6] for propositional reasoning and the Omega library for arithmetic reasoning. While this data structure is similar to NEX, we support more complicated transfer functions, and provide an interface to replace the Omega library with an arbitrary numeric abstract domain.

Our domains MTNDD and NDD use BDDs for a purely symbolic representation of abstract values. Thus, they are similar to Difference Decision Diagrams (DDDs) [15] that represent propositional formulas over difference constraints. However, unlike DDD, we do not restrict the domain of numerical constraints. This makes our implementation more general, at the cost of strong canonicity properties of DDDs.

The contribution of our work is in adapting, extending, and evaluating existing work on combining propositional and arithmetic reasoning to the needs of software model-checking. To our knowledge, none of the tight combinations of the two abstract domains have been evaluated in the context of PA-based software model-checking. A preliminary version of this work has appeared in [13].

III. BACKGROUND

In this section, we define notation and our view of abstract domains.

Expressions and Statements. Let V denote the set of program variables, and E denote the set of expressions over V. A program is built out of statements S of the form: (1) an assignment $l := e$, where l is a variable in V and e is an expression in E, and (2) *assume*(e), where e is in E. Assume operations are used to model conditional branches. We write $||s||$ to denote the collecting semantics, or strongest post-condition transformer, as a function from E to itself. For example, $||x{:=}x{+}1||(x > 3) = (x > 4)$, $||x{:=}5||(x = 3 \wedge y = 6) = (x = 5 \wedge y = 6)$ and $||assume(x > 4)||(y = 6) = (x > 4 \wedge y = 6)$. Atomic statements can be composed in several ways: (a) sequentially, written $s_1; s_2$, meaning s_1 followed by s_2; (b) with alternative choice, written $s_1 \vee s_2$, meaning non-deterministic choice between s_1 and s_2, and (c) in parallel,

Interface: $\text{ABSDOM}(V)$

γ	$: A \to E$	α	$: E \to A$
meet	$: A \times A \to A$	join	$: A \times A \to A$
isTop	$: A \to \textbf{bool}$	isBot	$: A \to \textbf{bool}$
leq	$: A \times A \to \textbf{bool}$	widen	$: A \times A \to A$

$$\alpha\text{Post} : S \to (A \to A)$$

Requires:

let $a, b, c \in A, e \in E, x = \gamma(a), y = \gamma(b), z = \gamma(c)$ **in**

$\text{true} \Rightarrow e \Rightarrow \gamma(\alpha(e))$ $\quad (\alpha\text{Post}(s)(a) = b) \Rightarrow ||s||(x) \Rightarrow y$

$\text{leq}(a, b) \Rightarrow (a \Rightarrow b)$ $\quad (\text{meet}(a, b) = c) \Rightarrow (x \wedge y \Rightarrow z)$

$\text{isTop}(a) \Rightarrow (\text{true} \Rightarrow a)$ $\quad (\text{join}(a, b) = c) \Rightarrow (x \vee y \Rightarrow z)$

$\text{isBot}(a) \Rightarrow (a \Rightarrow \text{false})$ $\quad (\text{widen}(a, b) = c) \Rightarrow (x \vee y \Rightarrow z)$

Fig. 3. Interface of an abstract domain: E denotes expressions, S denotes statements, and A denotes abstract values.

Name	Notation	Abstract Elements
Intervals	$\text{BOX}(V)$	$\{c_1 \le v \le c_2 \mid c_1, c_2 \in \mathcal{N}, v \in V\}$
Octagons	$\text{OCT}(V)$	$\{\pm v_1 \pm v_2 \ge c \mid c \in \mathcal{N}, v_1, v_2 \in V\}$
Polyhedra	$\text{PK}(V)$	linear inequalities over V
Predicates	$\text{PRED}(V)$	propositional formulas over V

TABLE II
Common abstract domains; V is a set of numeric/propositional variables; $\mathcal{N}$ domain of numeric constants.

written $s_1 \wedge s_2$, meaning parallel synchronous execution of s_1 and s_2. The usual rules and restrictions of legal compositions apply. For example, we do not allow for a parallel composition $x := 5 \wedge x := 6$ since both statements change x, etc.

Abstract Domain. We assume that the reader is familiar with abstract interpretation and only give the necessary details. For a detailed overview, please consult [9]. In this paper, we view an abstract domain operationally as an abstract data type that satisfies the interface $\text{ABSDOM}(V)$ shown in Fig. 3. For simplicity, we assume that the concrete domain is the set of expressions E, and not, for example, program states. We use A to denote the set of all the elements of $\text{ABSDOM}(V)$. The interface consists of functions: α and γ to convert between expressions and abstract elements in A; meet and join correspond to conjunction (intersection) and disjunction (union), respectively; leq corresponds to implication (subset); isTop and isBot check for validity (universality), and unsatisfiability (emptiness), respectively; widen is a widening operator [9] that over-approximates a disjunction and guarantees convergence when applied to any (possibly infinite) sequence of abstract elements; and, αPost approximates the semantics of a program statement as an abstract transformer, i.e., a function from A to A.

Examples of several abstract domains are shown in Table II. The first three domains, collectively called Numeric, are used to represent and manipulate arithmetic constraints. The last one represents propositional formulas over a set of predicates.

Syntax for Abstract Transformers. For ease of understanding, our syntax for abstract transformers mirrors that of concrete program statements. Let $\text{NDOM}(V)$ be a numeric domain over variables V. The syntax for assign transformers of $\text{NDOM}(V)$ is $x_1 := e_1 \wedge \cdots \wedge x_n := e_n$, where all x_i are in V, and all e_i are linear arithmetic expressions. The syntax for conditional transformers of $\text{NDOM}(V)$ is *assume*(e).

For the predicate domain $\text{PRED}(P)$ over a set of predicates P, an abstract transformer is represented by a *Boolean* assignment of the form $p := \textit{choice}(t, f)$, where $p \in P$ is a predicate, and t and f are Boolean expressions over P. Informally, t represents the condition forcing p to be true, and f the condition forcing p to be false. For example, $p := \textit{choice}(p, \neg p)$ leaves p unchanged (p is true iff it was true before), $p := \textit{choice}(\text{false}, \text{false})$ changes p non-deterministically (nothing forces p to be only true or false), and $p := \textit{choice}(p \wedge q, \text{false})$ leaves p as true if q is true, otherwise p is changed non-deterministically. Formally, the semantics of a Boolean assignment is a forward image (post) over the relation $(p' \wedge \neg f) \vee (\neg p' \wedge \neg t)$, where p' is the value of p in the next state. Boolean assignments can be composed in parallel using conjunction of their relations, as usual. For numeric abstraction, abstract transformers are computed by the domain itself. For predicate abstraction, the transformer is constructed using a theorem prover [11].

BDDs. Reduced Ordered Binary Decision Diagrams (BDDs) [6] are a canonical representation of propositional formulas. A BDD is a DAG whose nodes correspond to propositional variables, and paths to all satisfying assignments of a formula. We use $\mathbf{0}$ and $\mathbf{1}$ to denote BDDs for true and false, respectively. For a BDD u, we use $\text{varOf}(u)$ for the root variable, $\text{bddT}(u)$ for the then-branch, and $\text{bddE}(u)$ for the else-branch of u, respectively. BDDs have efficient support for conjunction (bddAnd), disjunction (bddOr), negation (bddNot), if-then-else (bddIte), existential quantification (bddExists), and variable renaming (bddPermute). Many of these can be implemented using $\text{bddApply}(f, u, v)$, where u, v are BDDs, and f is a binary operator (i.e., conjunction, disjunction, etc.) that is defined only for constants.

IV. NUMPREDDOM: INTERFACE

In this section, we describe the interface of NUMPREDDOM and its supported transfer functions. NUMPREDDOM deals with propositional formulas over predicates and numeric constraints. A numeric constraint can be treated as both a numeric and a predicate term. For example, in the formula $p \wedge (x \ge 0) \wedge (y \ge 0)$, p is definitely a predicate, but so can be $(x \ge 0)$ and $(y \ge 0)$. Let V_P be a set of predicates, V_N a set of numeric variables, and e be a conjunctive expression. The *propositional projection* of e onto V_P, denoted by $\text{proj}_P(V_P, e)$, is a conjunction of predicates from V_P that is implied by (i.e., over-approximates) e. Similarly, the *numeric projection* of e onto V_N, denoted by $\text{proj}_N(V_N, e)$, is a conjunction of numeric constraints over V_N that is implied by e. Some examples of the projections are:

$$\text{proj}_P(\{p\}, p \wedge (x \ge 0) \wedge (y \ge 0)) = p$$
$$\text{proj}_P(\{x \ge 0\}, p \wedge (x \ge 0) \wedge (y \ge 0)) = (x \ge 0)$$
$$\text{proj}_N(\{y\}, p \wedge (x \ge 0) \wedge (y \ge 0)) = y \ge 0$$

Note that the exact definitions of proj_P and proj_N are implementation dependent. We implement them via approximations based on syntactic reasoning. However, more precise semantic constructions via the use of theorem provers is also possible. Such implementation choices affect the efficiency vs. precision trade off, but not correctness.

Interface: NUMPREDDOM(V_N, V_P) **extends** ABSDOM

α_P	$: E \to A$	α_N	$: E \to A$
unprime	$: A \to A$	reduce	$: A \to A$
exists	$: 2^{V_P} \times A \to A$	$\alpha\mathsf{Post}_N$	$: S \to (A \to A)$

Fig. 4. The interface of NUMPREDDOM: V_N and V_P are numeric and propositional variables, respectively. E, S, and A are as in Fig. 3.

The interface NUMPREDDOM is shown in Fig. 4. It extends, i.e., has all the functions of, the basic abstract domain ABSDOM shown in Fig. 3. The interface NUMPREDDOM has two types of variables: numeric, V_N, and propositional, V_P. Moreover, the domain is extended with "primed" variables $V'_P \triangleq \{p' \mid p \in V_P\}$. Additional functions provided by the interface are: α_N, α_P are restrictions of the abstraction function α to conjunction of numeric and propositional expressions, respectively; exists existentially quantifies propositional variables from an abstract value and must satisfy the over-approximation condition: $(\exists V \cdot \gamma(a)) \Rightarrow \gamma(\mathsf{exists}(V, a))$; unprime renames all "primed" variables into the corresponding unprimed ones; $\alpha\mathsf{Post}_N$ lifts an abstract numeric only transformer to the combined domain. Finally, the interface has a special operation, called reduce, that refines an abstract value by sharing information between propositional and numeric parts of the value. Note that it is possible to apply reduce at any time during analysis to increase precision of the result. However, since excessive calls to reduce are expensive, we have factored it out in the interface.

The abstraction function $\alpha(e)$ is defined recursively using α_P and α_N as follows: if e is a term, then

$$\alpha(e) \triangleq \mathsf{meet}(\alpha_P(\mathsf{proj}_P(V_P \cup V'_P, e)), \alpha_N(\mathsf{proj}_N(V_N, e)))$$

else if $e = e_1 \wedge e_2$, then

$$\alpha(e) \triangleq \mathsf{meet}(\alpha(e_1), \alpha(e_2))$$

else if $e = e_1 \vee e_2$, then

$$\alpha(e) \triangleq \mathsf{join}(\alpha(e_1), \alpha(e_2))$$

NUMPREDDOM is distinguished by its support for a rich set of abstract transformers. The grammar for the supported transformers is shown in Fig. 5. We now describe each type of transformer, illustrate in what situations it is required, and provide a common implementation when applicable.

Numeric. Written as $x_1 := e_1 \wedge \cdots \wedge x_k := e_k$, where the variables in x_i and e_i are in V_N. It is handled by $\alpha\mathsf{Post}_N$ of each implementation of NUMPREDDOM. It is a basic building block for abstracting arithmetic transformations.

Assume. Written as *assume*(e), where e is an arbitrary expression, and interpreted as $\lambda X \cdot \mathsf{meet}(\alpha(e), X)$. It is used to approximate program conditionals with a combination of predicate and numeric conditions. For example, in the presence of aliasing, the C program statement *assume*$(*p > 0)$ can be approximated by:

$$\textit{assume}((p = \&x \wedge x > 0) \vee (p = \&y \wedge y > 0) \vee (p \neq \&y \wedge p \neq \&x))$$

with predicates $V_P = \{p = \&x, p = \&y\}$ and numeric variables $V_N = \{x, y\}$.

$$\begin{array}{rll}
\tau ::= & \tau_N \mid \tau_a \mid \tau_c \mid \tau_P \mid \tau_{NP} \mid & \text{(base case)} \\
& \tau; \tau \mid & \text{(sequence)} \\
& \tau \vee \tau & \text{(non-det.)} \\
\tau_{NP} ::= & (e?\tau_N) \wedge \tau_P & \text{(numeric + predicate)} \\
\tau_P ::= & p := \textit{choice}(e, e) \mid & \text{(predicate)} \\
& \tau_P \wedge \tau_P & \\
\tau_c ::= & e?\tau_N & \text{(conditional)} \\
\tau_a ::= & \textit{assume}(e) & \text{(assume)} \\
\tau_N ::= & x := v \mid & \text{(numeric)} \\
& \tau_N \wedge \tau_N &
\end{array}$$

Fig. 5. BNF grammar for abstract transformers supported by NUMPREDDOM; p is a predicate; x a numeric variable; e an expression over predicates and numeric terms; v a numeric expression.

Conditional. Written as $e?\tau$, where e is an arbitrary expression, and τ is a purely numeric transformer. It is interpreted as: $\lambda X \cdot \alpha\mathsf{Post}_N(\tau)(\alpha\mathsf{Post}(\textit{assume}(e))(X))$. It is most useful in a combination with other transformers. For example, it is used to abstract an assignment $*p := e$ through a pointer as:

$$(p = \&x \ ?\ x := e) \vee (p = \&y \ ?\ y := e)$$

with $V_P = \{p = \&x, p = \&y\}$ and $V_N = \{x, y\}$ and variables in e.

Predicate. Written as: $p_1 := \textit{choice}(t_1, f_1) \wedge \cdots \wedge p_n := \textit{choice}(t_n, f_n)$, where p_i are in V_P and t_i and f_i are expressions over V_P and V_N. It is interpreted using conjunction and existential quantification:

$$\textbf{let } R = \alpha(\textstyle\bigwedge_i (p'_i \wedge \neg f_i) \vee (\neg p'_i \wedge \neg t_i)) \textbf{ in}$$
$$\lambda X \cdot \mathsf{unprime}(\mathsf{exists}(\{p_1, \ldots, p_n\}, \mathsf{meet}(X, R)))\,.$$

This transformer is the basic building block for predicate abstraction. It depends on both predicate and numeric information. For example, suppose that $V_P = \{y > 0, p = \&x, p = \&y\}$ and $V_N = \{x\}$. Then the assignment $*p{:=}x$ is abstracted as:

$$(y > 0) := \textit{choice}((p = \&x) \wedge (y > 0), (p = \&y) \wedge (x > 0))\,.$$

Numeric and Predicate. Written as a parallel composition of conditional numeric and predicate transformers: $(e?\tau_N) \wedge \tau_P$, where e is an arbitrary expression, τ_N is a purely numeric transformer, and τ_P is a predicate transformer. It is interpreted with the help of the following equivalence: $(e?\tau_N) \wedge \tau_P \equiv \textit{assume}(e); \tau_P; \tau_N$. That is, since the purely numeric transformer does not depend on the predicates, this parallel composition is reduced to a sequential one. This transformer is used to abstract statements that influence both predicates and numeric constraints simultaneously. For example, let $V_P = \{y = 1\}$ and $V_N = \{x, v, w\}$. Then, the parallel statement $y := x \wedge x := (y = 1)?v : w$ is abstracted as:

$$(y = 1) := \textit{choice}(x = 1, x \neq 1) \wedge (y = 1)?x = v : x = w\,.$$

Note that the predicate $y = 1$ is both influenced by numeric constraints on x and influences the next value of x.

Sequential and Non-Deterministic. Written as $\tau_1; \tau_2$ and $\tau_1 \vee \tau_2$, respectively. Interpreted using function sequencing and join operator, respectively:

$$\alpha\mathsf{Post}(\tau_1; \tau_2) = \lambda X \cdot \alpha\mathsf{Post}(\tau_2)(\alpha\mathsf{Post}(\tau_1)(X))$$
$$\alpha\mathsf{Post}(\tau_1 \vee \tau_2) = \lambda X \cdot \mathsf{join}(\alpha\mathsf{Post}(\tau_1)(X), \alpha\mathsf{Post}(\tau_2)(X))$$

As a complete example, a combined predicate and numeric abstraction of the program in Fig. 2(a) is shown in Fig. 2(b). Both predicates p and q are necessary to separate different paths through the control flow, and predicate q gets its value from a combination of constraints on numeric variables and predicate p.

In summary, the critical operations in the NUMPREDDOM interface are exists, $\mathsf{unprime}$, proj_N, proj_P, α_N, α_P, γ, leq, meet, join, widen, $\alpha\mathsf{Post}_N$ and reduce.

V. NUMPREDDOM: IMPLEMENTATIONS

In this section, we describe four implementations of NUMPREDDOM. We use N to denote the set of abstract values of the underlying numeric domain over V_N. We write $N.\mathsf{op}$ and $P.\mathsf{op}$ to mean the abstract operation op over numerics and predicates respectively. We write $\sqsubseteq$, $\sqcap$, $\sqcup$ and ∇ to mean leq, meet, join and widen when the abstract domain is clear from context; and, write $X.\mathsf{top}$, $X.\mathsf{bot}$ to mean $X.\alpha(\mathsf{true})$ and $X.\alpha(\mathsf{false})$, respectively. Our domains, NEXPOINT, NEX and MTNDD, share the following definition of reduce: $\mathsf{reduce}(v) \triangleq (\alpha(\gamma(v)))$. Therefore, we only define reduce specifically for (NDD) implementation. In addition, all four implementations share the same definition of proj_N and proj_P based on syntactic simplification of expressions to a normal form.

A. NEXPOINT: *Numeric Explicit Points*

The set of abstract values of NEXPOINT is $2^{2^{V_P}} \times N$. A NEXPOINT value is a pair (p, n) where p is a BDD and n is a numeric abstract value. In particular, $\text{NEXPOINT}.\mathsf{top} = (P.\mathsf{top}, N.\mathsf{top})$ and $\text{NEXPOINT}.\mathsf{bot} = (P.\mathsf{bot}, N.\mathsf{bot})$. The exists and $\mathsf{unprime}$ operations are performed on the BDDs. The other operations are performed pointwise. Thus, we have the following definitions:

$$\begin{aligned}
\alpha_N(e) &\triangleq (P.\mathsf{top}, N.\alpha(e)) \\
\alpha_P(e) &\triangleq (P.\alpha(e), N.\mathsf{top}) \\
\gamma(p, n) &\triangleq P.\gamma(p) \wedge N.\gamma(n) \\
\mathsf{op}((p, n), (p', n')) &\triangleq (P.\mathsf{op}(p, p'), N.\mathsf{op}(n, n')) \\
\mathsf{leq}((p, n), (p', ' n)) &\triangleq p \sqsubseteq p' \wedge n \sqsubseteq n' \\
\alpha\mathsf{Post}_N(s) &\triangleq \lambda(p, n) \,.\, (p, N.\alpha\mathsf{Post}(s)(n)) \,,
\end{aligned}$$

where $\mathsf{op} \in \{\mathsf{meet}, \mathsf{join}, \mathsf{widen}\}$. Recall that reduce is defined as $\alpha(\gamma(v))$. Suppose we have two predicates $q \triangleq (x = 0)$ and $r \triangleq (y = 0)$, where x is also a numeric variable. Then, $\mathsf{reduce}(q \vee r, x = 3 \wedge y \geq 0) = (\neg q \wedge r, x = 3 \wedge y = 0)$. Similarly, $\mathsf{reduce}(q \vee r, x = 3 \wedge y < 0) = \text{NEXPOINT}.\mathsf{bot}$.

B. NEX: *Numeric Explicit Sets*

Each abstract value of the NEX domain is a function $2^{V_P} \mapsto N$. We represent an abstract value as a set of pairs $\{(p_1, n_1), \ldots, (p_k, n_k)\} \subseteq 2^{2^{V_P}} \times N$, where each p_i is a BDD, each n_i is a numeric abstract value, and the following conditions hold:

$$\forall 1 \leq i \leq k \,.\, p_i \neq P.\mathsf{bot} \wedge n_i \neq N.\mathsf{bot} \qquad \textbf{(C1)}$$

$$\forall 1 \leq i < j \leq k \,.\, n_i \neq n_j \quad \textbf{(C2)} \quad \wedge \quad p_i \sqcap p_j = P.\mathsf{bot} \quad \textbf{(C3)}$$

Intuitively, a NEX value is a "union" of NEXPOINT values that are distinguished by their numeric components. Thus, NEX improves upon the precision of NEXPOINT by replacing imprecise numeric join with union. In particular, $\text{NEX}.\mathsf{top} = \{(P.\mathsf{top}, N.\mathsf{top})\}$ and $\text{NEX}.\mathsf{bot} = \emptyset$. Conditions **C1**–**C3** ensure that the data structures are as "tight" as possible: **C1** guarantees that the representation of any abstract value does not include any "empty" components, **C2** ensures that any two elements (p_1, n_1) and (p_2, n_2) are distinguished by their numeric components, and **C3** — that the elements of a NEX value are "mutually disjoint".

To understand the NEX operations, we first introduce a normalizing procedure called norm. Given any set $v \subseteq 2^{2^{V_P}} \times N$ satisfying **C3**, norm returns a set $u \subseteq 2^{2^{V_P}} \times N$ that satisfies **C1**–**C3** by performing the following: (i) replacing any $(p_1, n) \in v$ and $(p_2, n) \in v$ with $(p_1 \sqcup p_2, n)$, and (ii) removing every $(p, n) \in v$ such that $p = P.\mathsf{bot} \vee n = N.\mathsf{bot}$. Thus, $\mathsf{norm}(v)$ is a NEX value that is semantically equivalent to v. norm has linear complexity since it makes single pass over its input. The exists and $\mathsf{unprime}$ are performed on the BDDs, followed by normalization. The other operations are defined as follows:

$$\alpha_N(e) \triangleq \langle (P.\mathsf{top}, N.\alpha(e)) \rangle \qquad \alpha_P(e) \triangleq \langle (P.\alpha(e), N.\mathsf{top}) \rangle$$

$$\gamma(\langle (p_1, n_1), \ldots, (p_k, n_k) \rangle) \triangleq \bigvee_{1 \leq i \leq k} P.\gamma(p_i) \wedge N.\gamma(n_i)$$

Let $v = (p, n)$ be a NEXPOINT value and $v' = \{(p'_1, n'_1), \ldots, (p'_k, n'_k)\}$ be a NEX value. We say that $v \sqsubseteq v'$ iff $p \sqsubseteq \bigsqcup_{\{i \mid n \sqsubseteq n'_i\}} p'_i$. For any two NEX values $v = \{(p_1, n_1), \ldots, (p_k, n_k)\}$ and v', $\mathsf{leq}(v, v')$ iff $\forall 1 \leq i \leq k \,.\, (p_i, n_i) \sqsubseteq v'$.

$$\mathsf{meet}(v, v') \triangleq \mathsf{norm}(\{(p \sqcap p', n \sqcap n') \mid (p, n) \in v \wedge (p', n') \in v'\})$$

$$\mathsf{join}(v, v') \triangleq \mathsf{norm}(\mathsf{NEXJoin}(v, v'))$$

$$\mathsf{widen}(v, v') \triangleq \mathsf{norm}(\mathsf{NEXWiden}(v, v'))$$

The algorithm $\mathsf{NEXJoin}$ is defined recursively as follows: (i) $\mathsf{NEXJoin}(\emptyset, v) = v$, (ii) $\mathsf{NEXJoin}(v, \emptyset) = v$, and (iii) $\mathsf{NEXJoin}(\{(p, n)\} \cup X, \{(p', n')\} \cup X') = \{(p \sqcap p', n \sqcup n')\} \cup \mathsf{NEXJoin}(\{(p \sqcap \neg p', n)\}, X') \cup \mathsf{NEXJoin}(\{(p' \sqcap \neg p, n')\}, X) \cup \mathsf{NEXJoin}(X, X')$. The key idea behind $\mathsf{NEXJoin}$ is to ensure that its output satisfies **C3** by splitting $p \sqcup p'$ into three mutually disjoint fragments: $p \sqcap p'$, $p \sqcap \neg p'$ and $p' \sqcap \neg p$. The algorithm $\mathsf{NEXWiden}$ is identical to $\mathsf{NEXJoin}$ except that it uses widen instead of join. The meet, join and widen operations have quadratic complexity. Finally, the operation $\alpha\mathsf{Post}_N$ is defined as follows:

$$\alpha\mathsf{Post}_N(s) \triangleq \lambda v \,.\, \mathsf{norm}(\{(p, N.\alpha\mathsf{Post}(s)(n)) \mid (p, n) \in v\})$$

C. MTNDD: *Multi-Terminal Numeric Decision Diagrams*

MTNDD is a symbolic alternative to NEX. MTNDD values are also functions of type $2^{V_P} \mapsto N$. However, an MTNDD value is represented as a BDD over predicate and numeric terms. This automatically maintains conditions **C1**–**C3** of NEX.

Conceptually, an MTNDD value is a Multi-Terminal BDD [1] whose terminals are numeric abstract values from N. In practice, we simulate MTBDDs with BDDs. We associate

```
1: BDD MJoinOp (BDD u, BDD v)
2:   if (u = 1 ∨ v = 1) return 1
3:   if (u = 0) return u
4:   if (v = 0) return v
5:   if (isNum(u) ∧ isNum(v))
6:     nu := N.α(toExpr(u))
7:     nv := N.α(toExpr(v))
8:     return toBdd(N.γ(nu ⊔ nv))
9:   return null
```

Fig. 6. Implementation of MJoinOp.

a BDD variable with each predicate and numeric term, and restrict variable ordering to ensure that predicate variables always appear before numeric ones. For any term t that is both predicate and numeric (i.e., $\mathsf{proj}_P(t) = t = \mathsf{proj}_N(t)$), we allocate two distinct variables: one predicate, one numeric. Although there are infinitely many numeric terms, only finitely many are used in any analysis. Thus, we allocate variables for numeric terms dynamically.

We use algorithms toBdd and toExpr to convert between BDDs and expressions in the usual way. Note that for NEXPOINT and NEX, this was achieved via $P.\alpha$ and $P.\gamma$. For a BDD v, we use $\mathsf{isNum}(v)$ to determine whether the root variable of v is a numeric term. MTNDD.top and MTNDD.bot are represented by BDDs $\mathbf{1}$ and $\mathbf{0}$, respectively. Abstraction and concretization functions simply convert between expressions and BDDs. Thus:

$$\alpha_P(t) \triangleq \mathsf{toBdd}(t) \quad \alpha_N(t) \triangleq \mathsf{toBdd}(N.\gamma(N.\alpha(t)))$$
$$\gamma(v) \triangleq \mathsf{toExpr}(v)$$

The unprime operation is the same as its BDD version. The MTNDD.exists operation is similar to bddExists except that MTNDD.join is used instead of bddOr.

The operations meet, join, widen, leq, and $\alpha\mathsf{Post}_N$ are implemented as operators to bddApply. They work by (a) recursively traversing input BDD(s) until they are reduced to BDDs over numeric terms; (b) converting numeric BDDs to abstract values and applying the corresponding numeric operation; and (c) encoding the result back as a BDD. For example MTNDD.join is implemented using $\mathsf{bddApply}(\mathsf{MJoinOp}, u, v)$, where MJoinOp is shown in Fig. 6. Note that the constraint on the variable ordering ensures that whenever a root of a BDD v is numeric, the rest of v is numeric as well.

Since MTNDD operations are implemented using bddApply, their complexity is linear in the size of their input BDDs. Due to sharing between various BDDs, the memory (and hence time) requirement of MTNDD is expected to be better than NEX.

D. NDD*: Numeric Decision Diagrams*

NDD is our most expressive domain, with elements in $2^{2^{V_P} \times N}$. An NDD value is a BDD representing a propositional formula over predicate and numeric terms. With each term t, we associate a BDD variable. The association takes negation into account. Any two terms t_1 and t_2 that complement each other, i.e., $t_1 = \neg t_2$, are associated with the opposite phases of the same BDD variable. For example, whenever $x > 0$ is mapped to a BDD variable v, $x \leq 0$ is mapped to $\neg v$. We use $\mathsf{term}(v)$ to denote the term corresponding to v. We extend the notation to BDDs and write $\mathsf{term}(u)$ to mean the term of the root variable of BDD u. Each term t is allocated a single BDD variable, independently of whether t is a predicate, a numeric term, or both. Thus, propositionally inconsistent expressions are always reduced to $\mathbf{0}$, unlike in the previous three implementations. For example, if $p \triangleq (x > 0)$ is a predicate, then $p \wedge (x \leq 0)$ is reduced to $\mathbf{0}$.

```
1: BDD ctxApply (BDD u, Op g, N c, Set V)
2:   r := g(u, c)
3:   if (r ≠ null) return r
4:   b := varOf(u); e := term(u)
5:   tt = ctxApply(bddT(u), g, e ⊓ c, V)
6:   ff = ctxApply(bddE(u), g, ¬e ⊓ c, V)
7:   if (b ∈ V)
8:     return bddOr(tt, ff)
9:   else
10:    return bddIte(b, tt, ff)
```

Fig. 7. Implementation of ctxApply.

For the most part, NDD operations are done using corresponding BDD operations. The NDD.top and NDD.bot are represented by BDDs $\mathbf{1}$ and $\mathbf{0}$, respectively. Abstraction and concretization functions α_P, α_N, and γ are exactly the same as in MTNDD — they simply convert between expressions and BDDs. Functions unprime, exists, meet, and join are implemented as bddPermute, bddExists, bddAnd, and bddOr, respectively. The widen operation is implemented by conversion to MTNDD. Additionally, bddNot is used to over-approximate negation. That is, whenever v over-approximates an expression e, $\mathsf{bddNot}(v)$ over-approximates $\neg e$.

All of the above operations work on propositional structure of the abstract value. Effectively, they treat numeric constraints as uninterpreted propositional symbols. Their complexity is linear in the size of the input. The operations reduce, leq, and αPost treat numeric terms differently. For these operations, we introduce a function ctxApply, whose implementation is shown in Fig. 7. The function ctxApply recursively traverses a BDD, collecting the context of the current path in c, and existentially eliminating variables in V. The complexity of this operation is linear in the number of paths in a BDD.

The reduce operation is implemented as $\mathsf{ctxApply}(u, \mathsf{reduceOp}, N.\mathsf{top}, \emptyset)$, where $\mathsf{reduceOp}(u, c)$ returns $\mathbf{0}$ when $N.\mathsf{isBot}(c)$ or $u = \mathbf{0}$, returns $\mathbf{1}$ when $u = \mathbf{1}$, and returns **null** otherwise. Essentially, it replaces every unsatisfiable cube in a BDD with $\mathbf{0}$. In particular, for any unsatisfiable BDD v, $\mathsf{reduce}(v)$ is $\mathbf{0}$. For leq, we use the fact that for any two formulas u, and v, u implies v (i.e., u is less than v) iff $u \wedge \neg v$ is unsatisfiable. Since both satisfiability and negation are available, we implement leq as $(\mathsf{reduce}(\mathsf{meet}(u, \mathsf{bddNot}(v))) = \mathbf{0})$.

The implementation of $\alpha\mathsf{Post}_N(s)$ is similar to reduce. It uses ctxApply to apply a transformer of s to every path

	Precision	Succinct	PA	NA	Prop Op	Num Op
NEXPOINT	-	+ +	+	+	++	+ +
NEX	+	-	+	+	-	+ +
MTNDD	+	-	+	-	+	-
NDD	+ +	+	+	-	++	- -

TABLE III
Summary of the implementations; **Precision** = precision of abstract values; **Succinct** = succinctness of the representation; **PA** = applicability to predicate abstraction; **NA** = applicability to numeric abstraction; **Prop Op** = complexity of propositional operations (meet, join, etc.); **Num Op** = complexity of numeric operations.

of an input BDD. For a purely numeric statement s, we first define a function $\mathsf{NDDPost}(s)(u, c)$ such that it returns $\mathbf{0}$ if $N.\mathsf{isBot}(c)$ or $u = \mathbf{0}$, returns $N.\alpha\mathsf{Post}(s)(c)$ if $u = \mathbf{1}$, and returns **null** otherwise. Second, let $NumV$ be the set of all numeric BDD variables. Then, $\alpha\mathsf{Post}_N(s)(u) \triangleq \mathsf{ctxApply}(u, \mathsf{NDDPost}(s), N.\mathsf{top}, NumV)$.

In this domain, predicate and numeric terms share BDD variables. Thus, parallel composition $\tau_N \wedge \tau_P$ of a numeric (τ_N) and a predicate (τ_P) transformers cannot be reduced to a sequential composition (as in Section IV). Part of the BDD that is affected by τ_P may be needed for application of τ_N. To solve this, we implement the transformer by first applying τ_P partially by storing its result in "shadow" variables, then applying τ_N while eliminating variables changed by τ_P, and finally restoring the state from the shadow variables. Let τ_P be of the form $\bigwedge_i p_i := \mathit{choice}(t_i, f_i)$, let R be the relational semantics of τ_P (as defined in Section IV), and $V = NumV \cup \{p_i\}_i$ be the set of all numeric variables and all variables changed by τ_P. Then, $\alpha\mathsf{Post}(\tau_N \wedge \tau_P)(u)$ is defined as:

$$\mathsf{unprime}(\mathsf{ctxApply}(u \sqcap R, \mathsf{NDDPost}(\tau_N), N.\mathsf{top}, V))$$

The $u \sqcap R$ part corresponds to partial application of τ_P, ctxApply applies τ_N and eliminates all current-state variables in V, and unprime copies shadow variables into current state. For example, let V_P be $\{(x = 3), (x = 4)\}$, V_N be $\{x\}$, τ_N be $x := x + 1$, and τ_P be $(x = 4) := \mathit{choice}(x = 3, \mathsf{f})$. Assume that u is $(x = 3) \wedge (x \geq 3)$. Then, applying τ_P partially results in $(x = 3) \wedge (x \geq 3) \wedge (x = 4)'$; applying τ_N and eliminating $(x = 3)$ produces $(x \geq 4) \wedge (x = 4)'$, and renaming yields $(x \geq 4) \wedge (x = 4)$.

E. Summary

To summarize our four implementations, we compare them with respect to six different criteria: precision, i.e., ability to represent different abstract values; succinctness, i.e., conciseness of representation; performance of the data structure when used solely for predicate (PA), or numeric abstraction (NA); and efficiency of propositional (i.e., meet, join), and numeric operations. The results are shown in Table III.

NDD is the most precise domain. Furthermore, since it uses BDDs to encode the propositional structure of the value, it is more succinct than NEX and MTNDD that do not share storage between predicate and numeric parts of the abstract value. Succinctness of NEXPOINT is a side-effect of its imprecision.

All of the data-structures reduce to BDDs when there are no numeric terms present. Thus, they are all equally well suited for predicate abstraction. NEXPOINT and NEX represent numeric abstract value explicitly, and, therefore, are efficient for numeric abstraction. Both MTNDD and NDD encode numeric values symbolically, and introduce additional overhead.

NDD is the best data structure for propositional operations since those are implemented directly using BDDs. At the same time, it is the worst for numerical operations — those use ctxApply, whose complexity is linear in the number of paths in a diagram. Again, the efficiency of NEXPOINT is a by-product of its imprecision.

As shown by our informal comparison, there is no clear winner between the four choices. In the next section, we evaluate the data structures empirically in the context of software model-checking.

VI. EMPIRICAL EVALUATION AND CONCLUSION

We implemented a general reachability analysis engine for C programs in Java on top of the four implementations of NUMPREDDOM. We used the APRON package for numeric reasoning (in our experiments we used the Polyhedra domain), a Java implementation of BDDs, and CVCLITE for building the PA part of the abstraction and for analyzing counterexamples. We experimented with two types of examples: (a) synthetic examples designed to compare and contrast our four implementations of NUMPREDDOM with each other, and (b) examples derived from more realistic benchmarks. For the synthetic examples, we only compare NEX, MTNDD and NDD since NEXPOINT is less expressive. All our experiments were carried out on a 2.4 GHz machine with 4 GB of RAM.

Synthetic Examples. NEX and MTNDD join numeric constraints, but NDD maintains an exact union. Thus, we conjecture that NDD performs poorly when numeric joins are exact. To validate this hypothesis we experimented with the template shown in Fig. 8(a). Our experiments support this hypothesis. NEX and MTNDD scale beyond `C = 10000` (NEX performs better than MTNDD since it does not have the extra overhead of manipulating BDDs). NDD blows up even for `C = 400`.

Our second conjecture was that when a problem requires a propositionally complex invariant, the sharing capability of NDD will place it at an advantage to NEX and MTNDD. To test this conjecture we experimented with the template in Fig. 8(b). Our experiments support this hypothesis as well. NDD requires seconds for `C = 10` while NEX and MTNDD both require several minutes with NEX being the slowest.

Realistic Examples. For a more realistic evaluation, we used a set of 22 benchmarks (3 from a suite by Zitser et al. [16], 2 from OpenSSL version 0.9.6c, 9 based on a controller for a metal casting plant, 2 based on the Micro-C OS version 2.72, and 6 based on Windows device drivers). We analysed them using our four implementations of NUMPREDDOM, and also with PA and NA. The results are summarized in Table IV. The total time taken by each individual experiment is shown

```
(a) int x = 0;
    while (x < C) ++x;
    assert(x == C);
```

```
(b) n = 1;
    if(x0 < 0) n = 0; ...
    else if(xC < 0) n = 0;
    if(x0 < 0) assert(n == 0); ...
    else if(xC < 0) assert(n = 0);
```

Fig. 8. Two templates for synthetic examples.

Domain	Num	Total	γ	join	αPost	Apply
Numeric	9	2.52	0.43	0.41	0.44	0.38
Predicate	17	333.38	0.05	0.03	0.20	0.06
NEXPOINT	21	42.30	0.38	1.13	4.04	8.50
NEX	22	45.17	0.59	2.22	3.99	7.20
MTNDD	22	94.05	0.02	3.71	2.11	56.10
NDD	22	42.15	0.03	0.02	1.96	17.81

TABLE IV
Time requirements for various operations on realistic examples. Numeric = purely numeric analysis; Predicate = purely predicate analysis; **Num** = no. of examples analysed; Apply = applying abstract transformers. All times are in seconds.

in Fig. 9. For the experiments, we have implemented a simple abstraction-refinement scheme based on the analysis of an UNSAT-core of the WP of an infeasible counterexample. First, the scheme adds all of the numeric variables in the UNSAT core; second, the predicates in the core are added when the first step fails to eliminate the spurious counterexample. Since the goal of the experiments is to explore the difference between our data structures, we only report on the time taken by the last iteration of abstraction-refinement, and do not include the time needed to find a suitable abstraction. Not all examples could be analyzed by every domain. In particular, only 9 could be analyzed numerically, and 17 using predicates. In the case of PA, the maximum number of predicates was 10; in the case of NA, the maximum number of numeric variables was 17; in the combined domains, these were at 8 (with 6 for NDD) and 17, respectively. Thus, combining PA and NA requires less predicates, with fewer predicates required for the most expressive combination.

In Table IV, we show the number of examples analyzed, as well as time used by basic abstract operations. The total time includes *all* of the analysis, including predicate abstraction with CVCLITE. Note that the last 4 columns of the table correspond to operations inside the reachability computation (they do not add up to total time). The experiments indicate that a combination of PA and NA is more expressive, and more importantly, more efficient, then either one in isolation. In particular, all of the combined domains could not only solve more problems that PA, but were 6-7 times faster. For this evaluation, NDD performs the best (NEXPOINT solves only 21/22 problems), which is probably explained by lack of deep loops in the benchmarks. The two extremes are NEX and NDD: NEX transformers are efficient to apply, but its join is rather slow, while the opposite is true for NDD.

In summary, we have presented an approach to couple PA and NA tightly into a unified analysis framework via a single abstract domain called NUMPREDDOM. In particular, we develop and evaluate four data structures that implement NUMPREDDOM but differ in their expressivity and internal representation and algorithms. We have implemented a general framework for reachability analysis of C programs on top of our four data structures. Our experiments on non-trivial examples show that our proposed combination of PA and NA is more powerful and more efficient than either technique alone. Employing these data structures in an industrial setting requires extending automated abstraction-refinement to them. We used a simple refinement strategy for our preliminary experiments. In the future, we plan to further explore the spectrum of possibilities in this area.

REFERENCES

[1] R. Bahar, E. Frohm, C. Gaona, G. Hachtel, E. Macii, A. Pardo, and F. Somenzi. "Algebraic Decision Diagrams and Their Applications". *FMSD*, 10(2/3), 1997.

[2] T. Ball and S. K. Rajamani. "Automatically Validating Temporal Safety Properties of Interfaces". In *Proc. of SPIN*, 2001.

[3] D. Beyer, T. A. Henzienger, and G. Theoduloz. "Configurable Software Verification: Concretizing the Convergence of Model Checking and Program Analysis". In *CAV*, 2007.

[4] D. Beyer, T. A. Henzinger, and G. Théoduloz. "Lazy Shape Analysis". In *CAV*, 2006.

[5] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. "A Static Analyzer for Large Safety-Critical Software". In *PLDI*, 2003.

[6] R. Bryant. "Graph-based Algorithms for Boolean Functions Manipulation". *IEEE Transactions on Computers*, 35(8), 1986.

[7] T. Bultan, R. Gerber, and W. Pugh. "Composite Model Checking: Verification with Type-Specific Symbolic Representations". *TOSEM*, 9(1), 2000.

[8] P. Cousot and R. Cousot. "Systematic Design of Program Analysis Frameworks". In *POPL'79*, 1979.

[9] P. Cousot and R. Cousot. "Abstract Interpretation Frameworks". *JLC*, 2(4), 1992.

[10] J. Fischer, R. Jhala, and R. Majumdar. "Joining dataflow with predicates". In *FSE*, 2005.

[11] S. Graf and H. Saïdi. "Construction of Abstract State Graphs with PVS". In *CAV*, 1997.

[12] S. Gulwani and A. Tiwari. "Combining Abstract Interpreters". In *PLDI*, 2006.

[13] A. Gurfinkel and S. Chaki. "Combining Predicate and Numeric Abstraction for Software Model Checking (EXTENDED ABSTRACT)". In *LFM*, 2008.

[14] H. Jain, F. Ivancic, A. Gupta, I. Shlyakhter, and C. Wang. "Using Statically Computed Invariants Inside the Predicate Abstraction and Refinement Loop". In *CAV*, 2006.

[15] J. B. Møller, J. Lichtenberg, H. R. Andersen, and H. Hulgaard. "Difference Decision Diagrams". In *CSL*, 1999.

[16] M. Zitser, R. Lippmann, and T. Leek. "Testing Static Analysis Tools Using Exploitable Buffer Overflows from Open Source Code". In *FSE*, 2004.

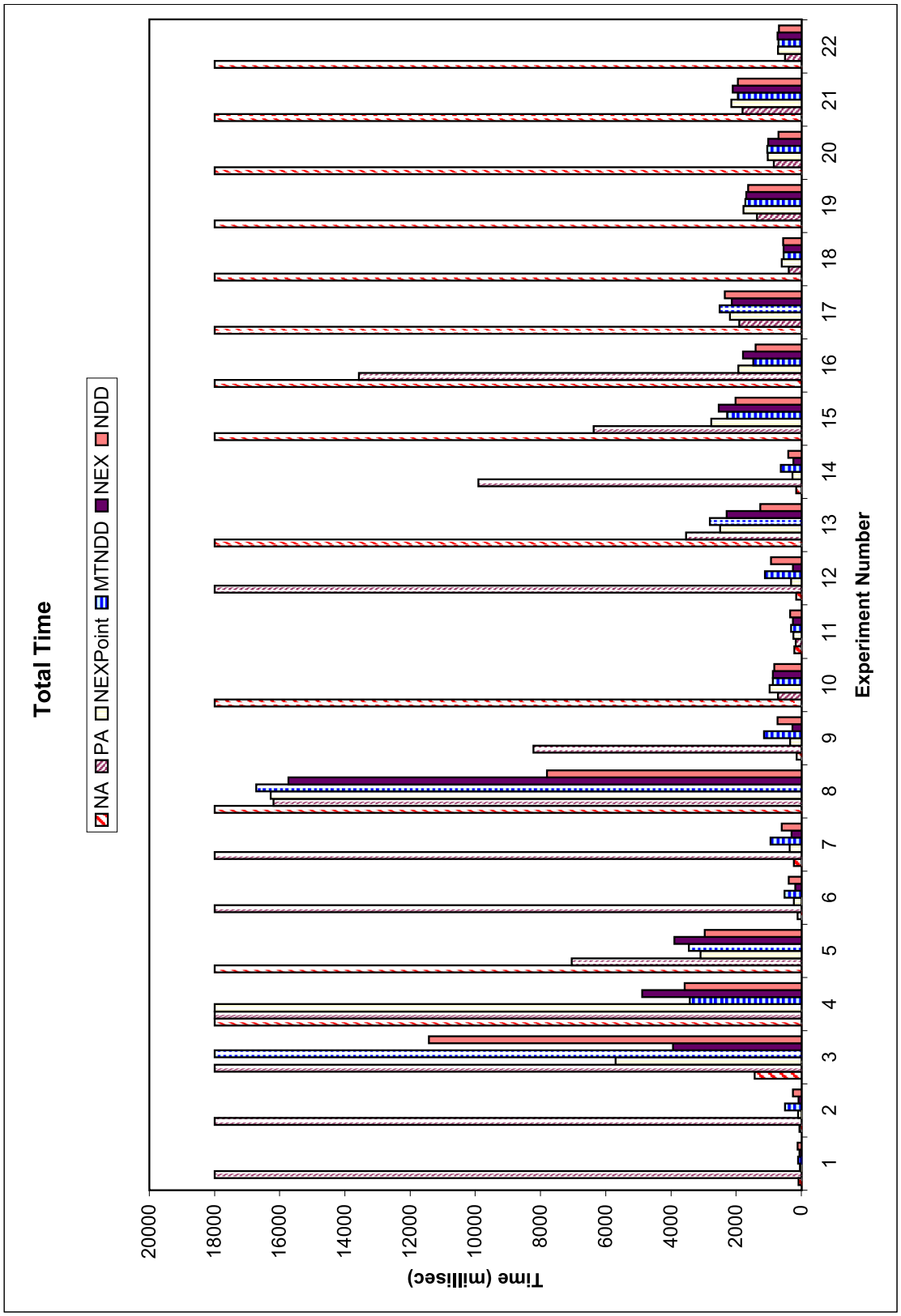


Fig. 9. Bar-chart showing total time taken by each experiment.

A Refinement Approach to Design and Verification of On-Chip Communication Protocols

Peter Böhm, Tom Melham
Oxford University Computing Laboratory
Oxford, OX1 3QD, England
Email: {peter.boehm,tom.melham}@comlab.ox.ac.uk

***Abstract*—Modern computer systems rely more and more on on-chip communication protocols to exchange data. To meet performance requirements these protocols have become highly complex, which usually makes their formal verification infeasible with reasonable time and effort.**

We present a new refinement approach to on-chip communication protocols that combines design and verification together, interleaving them hand-in-hand. Our modeling framework consists of *design steps* and *design transformations* formalized as finite state machines. Given a verified design step, transformations are used to extend the system with advanced features. A design transformation ensures that the extended design is correct if the previous system is correct.

This approach is illustrated by an arbiter-based master-slave communication system inspired by the AMBA High-performance Bus architecture. Starting with a sequential protocol design, it is extended with pipelining and burst transfers. Transformations are generated from design constraints providing a basis for correctness-by-design of the derived system.

I. Introduction

Modern computer systems rely more and more on highly complex on-chip communication protocols to exchange data. The enormous complexity of these protocols results from meeting high-performance requirements. Communication can be pipelined, data may be distributed into burst parts, and burst transfers may be split. Protocol control can be distributed and there may be non-atomicity or speculation. Moreover, some components may have separate clocks or adjustable clock frequencies, requiring asynchronous communications. These complexities arise in many important domains, such as multicore architectures, system-on-chip, or network-on-chip designs. Although the effort of chip manufacturers to validate or even formally verify their designs has increased, the complexity of communication protocols usually makes their formal verification infeasible with reasonable time and effort.

In this paper, we present a new approach to the design and formal verification of on-chip communication protocols based on refinement steps. The approach can be summarized as follows. The modeling framework is based on two central concepts, *design steps* and *design transformations*. We start with a core design step for a basic protocol that can be formally verified with reasonable effort. This is then extended with advanced features step-by-step to meet performance demands, handle asynchronous communication, or improve fault-tolerance. These extensions are realized by mathematical transformations of a previous design step, rather than constructing a new design. The correctness of an extended design is obtained from the correctness of the previous version and either the transformation itself (*correctness-by-design*) or a refinement or simulation relation between the two versions.

Verification by stepwise refinement is in general, of course, not new and there is a rich literature going at least back to Dijkstra [1], [2] and Wirth [3]. Our contributions are the application of this methodology to protocol verification, the design of a generic modeling framework specifically for this application, and the development of technical details of the specific optimising transformations we are investigating.

As a case study, we illustrate this refinement approach with an arbiter-based master-slave communication system inspired by the AMBA High-performance Bus architecture (AHB) [4]. The initial design step is a basic, sequential communication system. This system is extended with pipelined transfers and burst support. The corresponding transformations are partly derived automatically based on a behavioural constraint specification. This provides a basis for a correctness-by-design argumentation which is used within the demonstration of the correctness of the extended systems.

A. Related Work

Hardware verification based on refinement checking or simulation relations has a long history. Abadi and Lamport [5] show the existence of refinement mappings in their widely-cited article. McMillan [6] proposes a compositional rule for hardware verification based on local refinements which can be efficiently model checked. Aagaard *et al.* [7] present a framework for microprocessor correctness statements based on simulation relations.

Dill *et al.* [8] propose the protocol description language and verifier Murφ and its application to a industrial cache coherency as well as a link-level protocol. Eiríksson [9] and more recently German [10] emphasize the need for integration of formal verification into the (industrial) design process.

Most existing work on communication protocol verification addresses the correctness of specific protocols. For example, Roychoudhury *et al.* [11] present a formal specification of the AMBA protocol. They use an academic protocol version and verify design invariants using the SMV model checker. Amjad [12] verifies latency, arbitration, coherence, and deadlock freedom properties of a simplified AMBA model. Schmaltz *et al.* [13] present initial work on a generic network on chip

978-1-4244-2735-2/08/$25.00 ©2008 IEEE

model as a framework for correct on-chip communication. They identify key constraints on architectures and show protocol correctness provided they are satisfied.

All these approaches rely on a post-hoc protocol verification, which is a key difference from the methodology presented here. Even the framework in [13] relies on a post-hoc verification of protocol properties against their constraints.

Chen *et al.* [14] propose a modular, refinement based approach to verify transaction based hardware implementations against their specification models. They use a cache coherency protocol to illustrate their methodology. Müffke [15] presents a framework for the design of communication protocols. He provides a dataflow-based language for protocol specification and decomposition rules for interface generation relating dataflow algebra and process algebra. Aside from noting that correct and verified protocol design is still an unsolved problem, Müffke does not address the verification aspect in general. Claiming that the generated interfaces are correct by construction in terms of their specification he neither addresses the protocol correctness itself nor the verification of the implementation against the specification.

The basic idea of our approach is similar to Intel's integrated design and verification (IDV) system [16]. The IDV system justifies its transformations by a *local proof* using simple equivalence checking. We expect, however, that transformations tailored for high-performance on-chip communication protocols will need more intricate refinement steps than can be handled by equivalence checking.

II. Modeling Framework

In this section we present a generic modeling framework for the design and verification of arbiter-based, master-slave communication protocols. The two basic concepts are *design steps* and *design transformations*. The design process starts with an initial design step that is transformed into subsequent design steps with extended features or functionality.

A. Design Step

A design step is used to represent a verified protocol version during the design process. It has three parts: a specification of the communication system, a definition of a transaction model, and a predicate defining message transfer correctness.

1) Communication System: All communication systems under consideration have three kinds of basic components: a finite numbers of *masters* and *slaves* plus a single *arbiter*. Masters are attached to a host system and they have to react to host requests for data transfers that use the communication system. Slaves have an attached memory system and they respond to requests from masters by accessing it. The arbiter is responsible for granting bus access to a master. They are all specified as finite state machines. Additionally, there is a stateless communication bus between the masters and slaves.

We have a collection of N_M identical masters where $M_i = (I_M, O_M, Q_M, q_M^0, \delta_M, \omega_M)$ for $1 \leq i \leq N_M$. Using conventional notation for finite state machines, I_M and O_M denote the sets of inputs and outputs, Q_M the finite set of states, and q_M^0 the initial state. The state-transition function is given by $\delta_M : (Q_M \times I_M) \rightarrow Q_M$ and the output function is denoted by $\omega_M : (Q_M \times I_M) \rightarrow O_M$. Analogously, we have a collection of N_S identical slaves where $S_i = (I_S, O_S, Q_S, q_S^0, \delta_S, \omega_S)$ for $1 \leq i \leq N_S$ and an arbiter A where $A = (I_A, O_A, Q_A, q_A^0, \delta_A, \omega_A)$.

The communication system is then specified by the parallel execution of those finite state machines, thus again a finite state machine. As we have to refer to the communication bus frequently, we also add the bus as a separate component.

Definition 1 (Communication System CS) *A communication system CS is given by the finite state machine*

$$CS = (I, O, Q, q^0, \delta, \omega, cbus)$$

where I and O denote the sets of external inputs and outputs, $Q = ({Q_M}^M \times {Q_S}^S \times A)$ denotes the state space and q^0 is the initial state. The state-transition function is given by $\delta : (Q \times I) \rightarrow Q$ and $\omega : (Q \times I) \rightarrow O$ is the output function. The bus $cbus$ is given by a collection of signals obtained by combinatorial logic from the outputs of masters and slaves.

Signals are modeled as functions from discrete time to bit vectors of length n, i.e. $\mathbb{N} \rightarrow \mathbb{B}^n$, and sig^t denotes the value of a signal sig at time t.

The transition function of the system applies the transition functions of the components simultaneously in every step.

2) Transaction Model: In order to talk about transfers, we need a concept of a *transaction* which represents a unit of communication between a master and the memory modeled by the slaves. Our concept of a transaction is called *abstract transfer*. It is defined as a function $tr : \mathbb{N} \rightarrow T$ where T is the state space of a single transaction. We refer to the i-th transfer by $tr(i)$. This model encapsulates timing information as the definition of start and end times of a transfer. It is specific to a concrete instantiation and we present examples later.

3) Correctness Predicate: The third part of a design step is the correctness predicate $corr$. In general, this has to relate the execution behaviour of the communication system to a memory model. As every slave has a memory attached, the collection of all slaves provides a global memory spread over the communication system. A global memory model gm is a function over a cycle-accurate time domain $\mathbb{N}$ and an address space Ad to a data element of some data space D, i.e. $gm : (\mathbb{N} \times Ad) \rightarrow D$. The operations allowed can be categorized into *reads* and *writes*.

Given such a model, the correctness predicate has to enforce correctness properties for read transfers (RD), i.e. a transfer initiated by a host of some master requesting a read operation on the communication system, and write transfers (WR), i.e. a transfer initiated by a host requesting a write operation. Additionally, an optional predicate (OPT) may formulate specific protocol properties not 'visible' in standard, single transfers such as pipelining. Thus, the correctness predicate $corr(CS, tr) \subset (Q \times T)$ has the following structure:

$$corr(CS, tr) = RD \wedge WR \wedge OPT$$

Finally, we can summarize a design step as a triple consisting of the afore-mentioned parts:

Definition 2 (Design Step DS) *A design step is given by a triple $(CS, tr, corr)$. We call a design step* valid *iff $corr(CS, tr)$ holds.*

Note that the definition of a valid design step allows meaningless constructions as $corr(CS, tr) = True$. We assume that the correctness predicate argues about the correct message transmission according to the global memory model in a meaningful way. We will see examples of meaningful correctness predicates in Sections III and IV.

B. Design Transformation

A transformation is an operator on a design step to move from one valid design to another. It has three parts: a function θ modifying a given communication system CS, a corresponding abstract transfer model tr_θ, possibly relying on the 'input' transfer model, and a new correctness predicate $corr_\theta$.

Definition 3 (Design Transformation $trans$) *A design step transformation $trans$ is given by the triple $(\theta, tr_\theta, corr_\theta)$.*

The validity of the new correctness predicate $corr_\theta$ is obtained from the transformation function θ and the correctness of the previous design $corr$.

$$\theta \wedge corr(CS, tr) \implies corr_\theta(\theta(CS), tr_\theta)$$

As it stands, the state machine framework just described is rather obvious and unspecific. Our ultimate goal, however, is to elaborate this clear-cut starting point with extra formal structure that captures the generic capabilities and constraints of a broad, but specific, target class of high-performance communications architecture. The case study presented in the remainder of the paper is an example of the kind of experimental investigation that will inform the derivation of this framework.

III. Core Design Step

In this section, we use the notion of a design step to specify the initial protocol design of our case study. The system is a protocol design inspired by the AHB supporting sequential transfers with possible slave-side wait-states for memory operations. It is the initial design step, though its correctness has to be proven initially. As it supports sequential transfers, we prefix the components with *s*, thus $SDS = (SCS, str, scorr)$. This design step is extended with pipelining in Section IV-A and support for burst transfers in Section IV-B.

A. Communication System and Abstract Tranfer

To specify the communication system SCS, we define the the finite state machines of the three kinds of components, and the communication bus. We start with the latter.

Definition 4 (Sequential Communication Bus) *The value of the master-slave communication bus at time t of the sequential system is defined as the following tuple:*

$$(rdy^t, trans^t, wr^t, addr^t, wdata^t, rdata^t) \in \mathbb{B} \times \mathbb{B} \times \mathbb{B} \times \mathbb{B}^{ad} \times \mathbb{B}^d \times \mathbb{B}^d$$

where the components are:

- *rdy is the bus ready signal.*
- *$trans$ is the signal indicating either an idle transfer (0) or a data transfer (1).*
- *wr says if a data transfer is a read (0) or a write (1) transfer.*
- *$addr$ denotes the address bus of width ad. The address consists of a local address part (the lower sl bits) specifying the memory address within a slave and a device address (the upper $ad - sl$ bits) specifying the currently addressed slave.*
- *$wdata$ denotes the write data bus of width d.*
- *$rdata$ denotes the read data bus of width d.*

We refer to the last two components as the data bus *and to the second to fourth components as the* control bus*.*

Before specifying the remaining components, we first define an abstract sequential transfer. A transfer is split into two main parts: first an address phase $aphase$ and then a succeeding data phase $dphase$. During the former, a master puts the control data on the control bus and the addressed slave has to sample the data at the end of that phase. During the latter, the addressed slave performs its memory operation and either the master provides the data to be written or the slave delivers the read data at the end of that phase. The end of each phase is indicated by an active bus ready signal, i.e. $rdy = 1$.

Definition 5 (Abstract Sequential Transfer) *The i-th abstract sequential transfer $str(i)$ is defined in terms of a grant value $gnt \in [1 : N_M]$, a single bit $isdata \in \mathbb{B}$ indicating a idle or data transfer, and three cycle-accurate time points. The first time point $tg \in \mathbb{N}$ is the time when the bus is granted to the master gnt. The second time point $ta \in \mathbb{N}$ is the time when the address phase ends. The third time point $td \in \mathbb{N}$ denotes the time when the data phase of transfer i ends.*

$$str(i) = (gnt, isdata, tg, ta, td) \in [1 : N_M] \times \mathbb{B} \times \mathbb{N}^3$$

The components are defined as

$$\begin{aligned} gnt &= arb.grant^{tg} \\ isdata &= \begin{cases} 0 & : \textit{idle transfer} \\ 1 & : \textit{otherwise} \end{cases} \\ tg &= \begin{cases} 0 & : i = 0 \\ str(i-1).td & : \textit{otherwise} \end{cases} \\ ta &= \min\{t > tg \mid rdy^t\} \\ td &= \min\{t > ta \mid rdy^t\} \end{aligned}$$

where $arb.grant$ denotes the arbiter configuration component specifying the currently granted master.

From the abstract transfer definition, we can summarize three key protocol characteristics for this core design step: (i) every transfer consists of an address and a data phase, (ii) the end

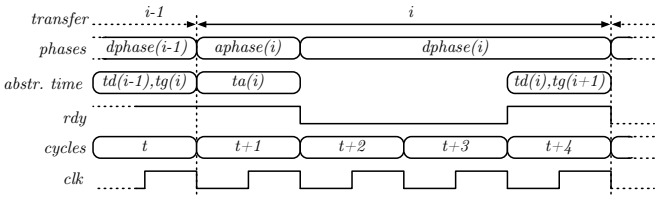


Fig. 1. Sequence of Sequential Transfers

of each phase is defined by the bus signal rdy, and (iii) the bus is granted to some master at every time instant. These characteristics and the definition of an abstract sequential transfer are illustrated in Fig. 1. To ease notation, we define a function $i(u,t)$. It denotes the next transfer such that tg is greater or equal to t and the bus is granted to master u.

$$i(u,t) = \min\{j \mid t \leq str(j).tg \wedge u = str(j).gnt\}$$

In case no such minimum is defined, we say $i(u,t) = -\infty$.

Next we define the transition function $s\delta$. We will specify the state-transition function by the functions for the components, i.e. $s\delta_M$, $s\delta_S$, and $s\delta_A$. In the following we assume that $N_M = 2^k$ for some $k \in \mathbb{N}$ and $N_S = 2^{ad-sl}$.

The arbiter grants the bus to master M_i with $1 \leq i \leq N_M$ by activating $grant[i-1]$ of its output $grant \in \mathbb{B}^{N_M}$. In case no master requests the bus, the arbiter grants the bus to some fixed default-master. In the scope of this project, we abstract the arbiter to a combinatorial circuit relying on an abstract function af generating a new grant vector given the current one and the current request vector. The arbiter updates the $grant$ bit vector at the end of an address phase, i.e. at ta. To store the information whether an active rdy signal represents the end of an address phase or the end of a data phase, we use a flag $aphase$.

If the $grant$ vector is updated at the end of the address phase, the old vector is still required during the data phase to select the correct $wdata$ output from a transmitting master. It is stored as a delayed grant value in $dgrant$. The state of the sequential arbiter at a given time t is then:

$$(grant^t, dgrant^t, req^t, aphase^t)$$
$$\in [1:N_M] \times [1:N_M] \times \mathbb{B}^M \times \mathbb{B} = SQ_A$$

The state-transition function $s\delta_A$ and the output function $s\omega_A$ are obtained directly from the description above.

The slave is similarly straightforward. Its task is to perform read or write accesses to an attached memory system mem. The slave obtains as inputs all bus components except for $rdata$. The $addr$ signal is reduced to the local address $saddr = addr[sl-1:0] \in \mathbb{B}^{sl}$. The upper part of the address bus ($addr[ad-1:sl]$) is used to generate a select input signal sel by an address decoder.

In case a slave is currently addressed for a data transfer, indicated by an active sel and $trans$ input at the end of the address phase, it has to sample the control bus data. Afterwards, the actual memory access is performed during the data phase. During that access the memory system can activate a memory busy signal $mem.busy$. The memory delivers the requested data when $mem.busy$ is low for the first time after the start of the request. We assume that the memory system is busy for only a finite number of cycles. At the end of the memory request, the slave activates the rdy output. It also provides the read data on the $rdata$ output for a read access.

The sequential slave has to generate the rdy signal indicating the end of the address phase, ta, in addition to the rdy signal indicating td. As the address data can be sampled during one cycle, a unit delay register rdy' is used to delay an active rdy single by one cycle. Then, if rdy' and sel are active, the rdy_{out} output is enabled to generate the rdy signal of the bus at ta. Moreover, in case of an $idle$ transmission, the slave just produces an other rdy_{out} signal in the next cycle (specifying td). The configuration of a sequential slave SS_v for $1 \leq v \leq N_s$ at a given time t is defined as the tuple:

$$(state^t, wr^t, addr^t, wdata^t, mem^t)$$
$$\in \{idle, req\} \times \mathbb{B} \times \mathbb{B}^{sl} \times \mathbb{B}^d \times (\mathbb{B}^{sl} \to \mathbb{B}^d) = SQ_S$$

where mem denotes the local memory. Similarly to the arbiter, the slave is realized according to the above description.

The master provides the interface between the communication system and an attached host system. It handles host requests to transfer data. Thus the master has inputs from the host denoted $startreq \in \mathbb{B}$, indicating a transfer request, and host data signals denoted $hwr \in \mathbb{B}$, $haddr \in \mathbb{B}^{ad}$, and $hwdata \in \mathbb{B}^d$ for the respective transfer data. In case the master is not granted the bus, it has to perform a bus request to the arbiter. Additionally, in case there is no data to transmit but the master is granted the bus, it has to initiate an idle transfer in order to meet the protocol requirements. The inputs to master M_u are the $grant[i-1] \in \mathbb{B}$ signal from the arbiter and the signals $rdy \in \mathbb{B}$, $rdata \in \mathbb{B}^d$ from the bus.

As outputs the master provides the signals $trans \in \mathbb{B}$, $wr \in \mathbb{B}$, $addr \in \mathbb{B}^{ad}$, and $wdata \in \mathbb{B}^d$ used to generate the corresponding bus signals. It provides a request signal $req \in \mathbb{B}$ to the arbiter and a busy signal $busy \in \mathbb{B}$ as well as a signal $hrdata \in \mathbb{B}^d$ to the host. The purpose of the $busy$ signal is the following: the correct transmission of a host request is shown if the master is not busy while the transfer is initiated ($startreq^t \implies \neg busy^t$). We call such a host request *valid* and define a predicate $validreq^t = startreq^t \wedge \neg busy^t$.

The configuration of the master consists of a $state$ component, a flag $vreq$ indicating a pending request, and a set of sampling registers. Thus the state of a sequential master SM_u for $1 \leq u \leq N_m$ at a given time t is defined as the tuple:

$$(state^t, vreq^t, lwr^t, laddr^t, lwdata^t, lrdata^t)$$
$$\in \{idle, aph, dph\} \times \mathbb{B} \times \mathbb{B} \times \mathbb{B}^{ad} \times \mathbb{B}^d \times B^d = SQ_M$$

where the components are:

- $state$ denotes the automaton state: $idle$ is the idle state, aph the state denoting the address phase, and dph the state denoting the data phase, respectively.
- $vreq$ denotes that a valid request is currently processed.
- $lwr, laddr, lwdata$ denote the local sampling register for the corresponding host data inputs.

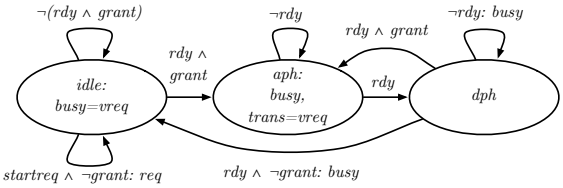


Fig. 2. Sequential Master Control Automaton

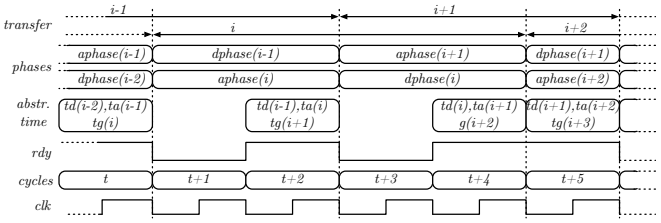


Fig. 3. Sequence of Pipelined Transfers

- $lrdata$ denotes the register used to sample the $rdata$ bus.

The control automaton is shown in Fig. 2 illustrating the update of the $state$ component and the $trans$ output generation. The bus outputs different from $trans$ are simply the respective local components. The local sampling registers are updated in a straightforward way, i.e. $lrdata$ is updated at the end of the data phase in case of a read transfer and the others are updated on a valid host request. The valid request flag $vreq$ is updated according to the following specification:

$$vreq^{t+1} = \begin{cases} 1 & : startreq^t \wedge (idle^t \wedge \neg busy^t) \\ & \quad \vee (data^t \wedge\ grant^t \wedge rdy^t) \\ 0 & : data^t \wedge rdy^t \\ & \quad \wedge \neg(startreq^t \wedge grant^t) \\ vreq^t & : \text{otherwise} \end{cases}$$

This concludes the required parts for the state-transition function $s\delta_M$ and the output function $s\omega_m$.

B. Correctness Predicate

The core design step should realize a simple single-port, read-write memory model. The correctness property $scorr$ relates this memory model to the communication system presented in the previous section. This property says that the communication system behaves like a memory model gm from the view of a host system but with respect to the time points given by tr. Let

$$gm^t(x[ad-1:0]) = SS_{x[ad-1:sl]}.mem^t(x[sl-1:0])$$

and let $1 \leq u \leq N_M$ denote some master index. Then $scorr(SCS, str)$ is defined by:

$$\begin{aligned} validreq_u^t &\implies i(u,t) \neq -\infty \wedge str(i(u,t)).isdata \wedge \\ \neg hwr_u^t &\implies hrdata_u^{td+1} = gm^{tg}(haddr_u^t) \quad \wedge \\ hwr_u^t &\implies gm^{td}(x) = \begin{cases} hwdata_u^t & : x = haddr_u^t \\ gm^{tg}(x) & : \text{otherwise} \end{cases} \end{aligned}$$

Theorem 1 (Valid Core Design Step) *The initial design step SDS is valid, i.e. $scorr(SCS, str)$ holds.*

The proof of this main theorem is based on a series of local correctness properties and communication system invariants. It is detailed in [17].

IV. Transformations

In this section we present transformations to extend the initial design step with pipelining and support for burst transfers.

A. Pipelining

In order to obtain a design with pipelined communication, we devise a design transformation based on duplication of masters. The idea is to execute the address phase of transfer i in parallel with the data phase of the previous transfer $i-1$. This is possible due to the separated control and data buses.

Recall that a transformation consists of a transformation function, a new abstract transfer definition, and a new correctness predicate. We write $ptrans = (pipe, ptr, pcorr)$ for the pipelining transformation.

As we do not introduce new bus signals, the communication bus has the same signals as for the core system.

The basic idea of the abstract transfer model is analogous to the sequential abstract transfer but we have to represent parallel phases. This is achieved by modifying the definition of the bus grant time tg. Note that this is the only change required to that model. Fig. 3 shows a sequence of transfers.

Definition 6 (Abstract Pipelined Transfer) *The i-th abstract pipelined transfer $ptr(i)$ is given by the tuple $(gnt, isdata, tg, ta, td)$ where the components are defined as:*

$$tg = \begin{cases} 0 & : i = 0 \\ ptr.ta & : otherwise \end{cases}$$
$$x = tr_{seq}(i).x \quad for\ x \in \{gnt, isdata, ta, td\}$$

The arbiter is obtained by ignoring the $aphase$ flag of the sequential arbiter and updating the $grant$ vector each time the rdy signal is active. Since the $grant$ vector is updated every time rdy is active, we have to introduce a third grant component denoted $ddgrant$ which is updated with the data from $dgrant$ in exactly the same way as $dgrant$ with $grant$.

The slave is obtained by removing the unit delay rdy'. The pipelined slave only generates a rdy signal at td.

To conclude the definition of the $pipe$ function, we need to specify the transformation for the master. Our goal is to obtain a master supporting pipelined transfers from the master only supporting sequential transfers. The presented transformation is based on the idea of using two sequential masters to construct a single master (see Fig. 4). Since pipelining denotes a process of executing tasks in parallel, using duplicated masters internally is straightforward. But we have to restrict the behaviour of the parallel system by excluding some bad executions, e.g. where both sequential masters are in the address phase, as this obviously leads to a conflict.

The required behavioural constraints are obtained by modifying the inputs to the sequential masters. This is realized by

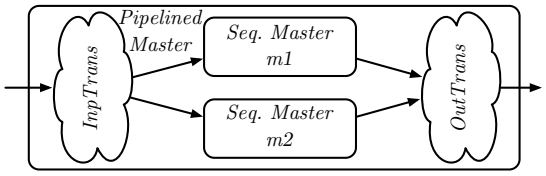


Fig. 4. Basic Concept of Pipelining Transformation

a logic denoted $ITrans$ splitting the inputs into two sets of inputs. Additionally, we need to combine the outputs of the two internal masters to generate the outputs of a single master. This combinatorial function is called $OTrans$.

In the following, we denote the current configuration of the two sequential masters with $sm1 \in SQ_M$ and $sm2 \in SQ_M$. Thus $pm = (sm1, sm2) \in PQ_M$. If we execute two sequential masters in parallel, the state space of the control automaton $pm.state$ is equal to the Cartesian product of the state components of the two sequential masters, i.e. $pm.state \in (sm.state \times sm.state)$.

The key question is which behaviours have to be excluded from the purely interleaved execution of both control automata. We define the desired behaviour in term of constraints regarding the control automaton. We have three kinds of constraints: *state*, *transition*, and *signal* constraints. The first specifies static state constraints, i.e. not reachable states, whereas transition constraints restrict the active, outgoing transitions of a current state. Signal constraints restrict the output signals of the control automaton. We aim at the following constraints:

- State constraints: (i) no *phase contention*, i.e.

$$pm.state \notin \{(aph, aph), (dph, dph)\}$$

and (ii) $sm2$ is only used for pipelining, i.e.

$$pm.state \in \{(idle, aph), (aph, aph), (dph, aph)\} \implies pm.state = (dph, aph)$$

- Transition constraints: *valid steps* of the automaton, i.e.

$$(s,t) \rightsquigarrow (s',t') \wedge s \neq s' \implies t \neq t' \vee t = idle$$
$$(s,t) \rightsquigarrow (s',t') \wedge t \neq t' \implies s \neq s' \vee s = idle$$

- Signal contraints: $sm2$ never requests the bus, i.e. $\neg req_2$

In order to satisfy these constraints, we analyse the product automaton of the two sequential masters. The control automaton of a single master is depicted in Fig. 2. We aim at deriving the constraints to the inputs in an algorithmic way. Using a graph-based algorithm, specifying edges which are not valid according to the constraints above, we obtain predicates to restrict the inputs of the second sequential master.

Let $pmi = (rdy, grant, rdata, startreq, hinp)$ denote the inputs of a master with $hinp = (hwr, haddr, hwdata)$. Then we define the function $ITrans$ as the tuple $(pmi, (rdy, grant_2, rdata, startreq_2, hinp))$ where

$$grant_2 = grant \wedge (sm1.state = aph)$$
$$startreq_2 = startreq \wedge grant_2$$

The outputs are obtained in a straightforward way. The sequential master which is currently in the aph or dph state provides the corresponding bus outputs. Additionally, the req signal to the arbiter is only triggered by $sm1$.

The only non-obvious computation is the $busy$ signal. Obviously, the pipelined master has to be busy if both sequential masters are busy. Additionally, the second sequential master is not allowed to request the bus. Therefore we have to enable the busy signal in cases where the first master is busy and the second master would have to request the bus after a $startreq$ signal, i.e. if the bus is not granted anymore.

Finally, we obtain the transformation for the step function of the master. Let $pm = (sm1, sm2) \in PQ_M$ and $pmi \in PI_M$ denote the inputs to the master. Given that $(smi1, smi2) = ITrans(pmi)$, the transformation $pipe_M$ is defined by:

$$pipe_M(s\delta_M)\ (pm, pmi) = (s\delta_M(sm1, smi1), s\delta_M(sm2, smi2))$$

To complete the specification of the pipeline transformation, we have to define the correctness predicate $pcorr$ and argue about its validity. The correctness predicate of the pipelined system is basically the same as for the core system but we add an optional property stating that the transfer is pipelined. Let gm be defined as in $scorr$ (see Section III-B).

$$validreq_u^t \implies i(u,t) \neq -\infty \wedge ptr.isdata \wedge$$
$$\neg hwr_u^t \implies rdata_u^{td+1} = gm^{tg}(haddr_u^t) \quad \wedge$$
$$hwr_u^t \implies gm^{td}(x) = \begin{cases} hwdata_u^t & : x = haddr_u^t \\ gm^{tg}(x) & : \text{otherwise} \end{cases} \quad \wedge$$
$$[ta : td] = [ptr(i(u,t)+1).tg : ptr(i(u,t)+1)).ta]$$

The main part of its correctness proof is based on a correctness-by-design argumentation of the pipelined master.

B. Burst Support

Next we specify a transformation providing burst transfers applicable to either the core or the pipelined design step. It is general enough that there are only few cases where one has to distinguish whether a sequential or a pipelined design is extended. Again, we will start with the specification of the transformation function and the abstract transfer model. We write $btrans = (bst, btr, bcorr)$ for this transformation.

A burst transfer of size $bsize$ is a transfer starting at address $addr$ transferring all data address $addr$ to $addr + bsize - 1$. We support burst transfers of arbitrary but fixed length up to a maximum of $2^b - 1$ such that $bsize \in \mathbb{B}^b$. The length of a specific transfer is specified during the host request.

In the following, we denote the configuration of a burst master BM_u at time t by bm_u^t, of a sequential master SM_u by sm_u^t, and of a pipelined master PM_u by pm_u^t, respectively.

Next, we define the new abstract transfer model. The changes are more complex than the changes from sequential to pipelined transfers. The basic idea is to add a component bst to indicate a burst transfer together with a field $bsize$ specifying the size. The end of the address phase is now not only a single time point but a partial function assigning an address phase end time to every *sub-transfer* $n \in [0 : bsize(i) - 1]$.

Definition 7 (Abstract Burst Transfer) *The i-th abstract burst transfer $btr(i)$ is defined as the tuple*

$$(gnt, isdata, tg, ta, td, bst, bsize) \\ \in [1 : N_M] \times \mathbb{B} \times \mathbb{N} \times (\mathbb{N} \to \mathbb{N}) \times \mathbb{N} \times \mathbb{B} \times \mathbb{B}^b$$

where the components are:

$$gnt = \{s,p\}tr(i).gnt$$
$$isdata = \{s,p\}tr(i).isdata$$
$$tg = \begin{cases} 0 & : i = 0 \\ btr(i-1).ta(btr(i-1).bsize) & : i > 0 \wedge pipe \\ btr(i-1).td & : i > 0 \wedge seq \end{cases}$$
$$ta(n) = \begin{cases} \min\{t > tg \mid rdy^t\} & : \neg bst \vee n = 0 \\ \min\{t > ta(n-1) \mid rdy^t\} & : bst \wedge \\ & \quad 1 \leq n \leq bsize - 1 \\ \textit{undefined} & : otherwise \end{cases}$$
$$td = \begin{cases} \min\{t > ta(0) \mid rdy^t\} & : \neg bst \\ \min\{t > ta(bsize-1) \mid rdy^t\} & : bst \end{cases}$$
$$bst = bm_{gnt}^{ta(0)}.bst$$
$$bsize = bm_{gnt}^{ta(0)}.bsize$$

The main difference between this definition and the previous one is the case split on actual burst transfers. For a non-burst transfer the above definition resolves to one of the previous transfer definitions depending on the system we are extending.

The arbiter is obtained from the corresponding previous arbiter by adding two additional registers for every master to sample burst request data. The interface to the master is also extended with two inputs $bst \in \mathbb{B}$ and $bsize \in \mathbb{B}^b$ indicating a request for a burst transfer of size $bsize$. Upon an active req signal from some master, a flag indicating a burst request is set and the $bsize$ input is sampled if the bst input active. When the bus is granted to a master with a pending burst request, the arbiter keeps the $grant$ vector stable for $bsize$ may transfers.

The slave is the same as in the corresponding sequential or pipelined system, thus the transformation is the identity.

Again, the more interesting part is the transformation of the master. The basic idea is again simple: we add a counter for the *sub-transfers* and simulate a sequence of $bsize$ many standard transfers. Thereby, the arbiter correctness ensures that the bus is granted during the complete burst transfer. We specify the transformation in terms of an input transformation $ITrans$ and an output transformation $OTrans$.

The interface from host to master has to be extended with two new signals: $hbst \in \mathbb{B}$ signals that the current transfer is a burst request of size $hbsize \in \mathbb{B}^b$. We also introduce a signal to the host called $bdataupd \in \mathbb{B}$. For write transfers, it signals that the data in $lwdata$ has to be updated und we require the host to update the $wdata$ input when the signal is active. For read transfers, it indicates that new data can be read from the $hrdata$ output in the next cycle. We define $bdataupd$ in detail as part of $OTrans$.

We also have to extend the configuration of the master with three new components to handle burst requests: $bst \in \mathbb{B}$ and $bsize \in \mathbb{B}^b$ are used to sample the corresponding host signals. As the local address register has to be incremented before every burst sub-transfer except the very first, we introduce a flag $bfst \in \mathbb{B}$ indicating the first one. The configuration of a master BM_u is given by $bm_u = \{s,p\}m_u \times (bst, bsize, bfst)$.

In contrast to $ITrans$ for the pipelined master, this transformation is a state-representing extension. Thus, we describe $ITrans$ as a finite state machine, i.e. by a state-transition function $bmi\delta$ and a output function $bmi\omega$. Let $bmi = (rdy, grant, rdata, startreq, hinp)$ denote the inputs to a master with $hinp = (hwr, haddr, hwdata, hbst, hbsize)$ and let $\{S,P\}M$ denote the master which is extended. Moreover, let $done = (bsize = 0)$ and

$$bupd = \begin{cases} rdy \wedge \neg bfst & : PM \\ rdy \wedge (state = dph) & : SM \end{cases}$$

Then $bmi\delta\ ((bst, bsize, bfst), bmi)$ is given by the tuple $(bst', bsize', bfst')$ where

$$bfst' = \begin{cases} 1 & : startreq \wedge \neg\{s,p\}m.busy \\ & \quad \wedge\ hbst \wedge \neg(grant \wedge rdy) \\ 0 & : rdy \wedge grant \\ bfst & : \text{otherwise} \end{cases}$$
$$bst' = \begin{cases} hbst & : startreq \wedge \neg\{s,p\}m.busy \\ 0 & : bupd \wedge done \\ bst & : \text{otherwise} \end{cases}$$
$$bsize' = \begin{cases} hbsize & : startreq \wedge \neg\{s,p\}m.busy \\ bsize - 1 & : bupd \wedge bst \wedge \neg done \\ bsize & : \text{otherwise} \end{cases}$$

Given $nextbst$ as a shorthand for $bupd \wedge bst \wedge \neg done$, the output is given by $bmi\omega\ (bm, bmi) = (bstartreq, bwr, baddr)$ where

$$bstartreq = startreq \vee nextbst$$
$$bwr = \begin{cases} \{s,p\}m.lwr & : nextbst \\ hwr & : \text{otherwise} \end{cases}$$
$$baddr = \begin{cases} \{s,p\}m.laddr + 1 & : nextbst \\ haddr & : \text{otherwise} \end{cases}$$

Given $ITrans$, we can define the state-transition function of a master supporting burst transfers. Let xm for $x \in \{s,p\}$ denote the state components of some burst master bm which are also part of the state of the master to be extended, i.e. $bm = (xm, bst, bsize, bfst)$. Then the state-transition function $bst_M(\{s,p\}\delta_M)$ is define as:

$$bst_M(x\delta_M)\ (bm, bmi) \\ = (xm', bmi\delta\ ((bst, bsize, bfst), bmi))$$

where

$$xm' = x\delta_M\ xm\ (rdy, grant, rdata, bstartreq, \\ bwr, baddr, hwdata) \\ (bstartreq, bwr, baddr) = bmi\omega\ (bm, bmi)$$

Next we specify the output transformation. Except for the $busy$ and $bdataupd$ outputs, the outputs remain the same

as for the previous masters. The signal to update the burst data is given by $bdataupd^t = bupd^t \wedge bst^t \wedge \neg done^t$ where $done$ denotes $bsize = 0$ as before. The $busy$ signal has to be adapted for the case a burst access is in progress such that it remains active during that transfer. We obtain $busy^t = \{s,p\}m.busy^t \vee (bst^t \wedge \neg done)$. For all other outputs, $OTrans$ is just the identity.

Finally, we must specify the correctness predicate. We spilt cases on burst transfers: in case of a non-burst transfer, the correctness predicate is the one from the 'source' system.

In case of a burst transfer, the correctness predicate specifies a sequence of correct single transfers. Let $1 \leq n \leq bsize - 2$ and $1 \leq m \leq bsize - 1$, then it is given by:

$$
\begin{array}{l}
validreq_u^t \wedge hbst_u^t \implies \\
\quad i(u,t) \neq -\infty \wedge btr(i(u,t)).isdata \wedge \\
\neg hwr_u^t \implies rdata_u^{ta(1)+1} = gm^{tg}(haddr_u^t) \wedge \\
\qquad\qquad rdata_u^{ta(n+1)+1} = gm^{tg}(haddr_u^t + n) \wedge \\
\qquad\qquad rdata_u^{td+1} = gm^{tg}(haddr_u^t + bsize - 1) \wedge \\
hwr_u^t \implies gm^{td}(x) = \begin{cases} hwdata_u^t & : x = haddr_u^t \\ hwdata_u^{ta(m-1)} & \\ & : x = haddr_u^t + m \\ gm^{tg}(x) & : \text{otherwise} \end{cases}
\end{array}
$$

The validity is again shown using the correctness of the components. Using the input and output transformation of the master, which are correct by construction, a burst transfer simulates a sequence of 'normal' transfers.

V. Conclusion

We have illustrated our general approach with a common and accessible protocol example (AMBA) and cover two very general features: pipelining and burst mode. A broad variety of on-chip busses support both features. Our initial focus on AMBA also provides a useful comparison with previously published work on this protocol [11], [12] that relied on post-hoc verification. We believe our general approach and also the specific idea we presented for realizing transformations as input and output restrictions will provide a basis for many other optimising transformations.

All definitions have been formalized and theorems have been proven using the Isabelle/HOL theorem prover [18]. Local properties of the finite state machine have been shown using the symbolic model checker NuSMV [19] used via an oracle-based interface [20].

These initial results provide encouraging evidence for the practicability of our approach, but are only a starting point. Our larger scientific hypothesis is that we are able to derive much more complex, verified protocols with a similar refinement approach and modelling framework. Our target is not AMBA, but high-performance protocols approaching the complexity and subtlety of today's industrial implementations.

Much remains to be done to test this hypothesis. We will need more transformations, and systematic methods of applying them supported by automatic proof. We will need to invent a greater range of refinement relations to link different abstraction levels. These will relate atomic, high-level transfers to non-atomic communications spread over space and time at the cycle-accurate, register-transfer level (see [14] for example). Finally, we will need to tackle circuit issues arising from the implementation layer—for example, we expect to develop methods to argue about low-level timing constraints or distributed, asynchronous clocks. As part of this, we expect to identify and characterise the right level of abstraction at which our refinement towards implementation will stop—and that this will be somewhat below the conventional unit-delay or half clock-cycle levels usually assumed in much of today's formal verification research.

VI. Acknowledgements

This work is funded by the Engineering and Physical Sciences Research Council and a donation from Intel Corporation.

References

[1] E. W. Dijkstra, "A constructive approach to the problem of program correctness," *BIT*, vol. 8, pp. 174–186, Feb. 1968.
[2] ——, *A Discipline of Programming*. Prentice-Hall, 1976.
[3] N. Wirth, "Program development by stepwise refinement," *Commun. ACM*, vol. 14, no. 4, pp. 221–227, 1971.
[4] *AMBA Specification Revision 2.0*, ARM, 1999.
[5] M. Abadi and L. Lamport, "The existence of refinement mappings," *Theor. Comput. Sci.*, vol. 82, no. 2, pp. 253–284, 1991.
[6] K. L. McMillan, "A compositional rule for hardware design refinement," in *CAV '97*. Springer, 1997, pp. 24–35.
[7] M. Aagaard, B. Cook, N. A. Day, and R. B. Jones, "A framework for microprocessor correctness statements," in *CHARME '01*. Springer, 2001, pp. 433–448.
[8] D. L. Dill, A. J. Drexler, A. J. Hu, and C. H. Yang, "Protocol verification as a hardware design aid," in *ICCD'92*, 1992, pp. 522–525.
[9] A. T. Eiríksson, "Integrating formal verification methods with a conventional project design flow," in *DAC '96*. ACM, 1996, pp. 666–671.
[10] S. M. German, "Formal design of cache memory protocols in IBM," *Formal Methods in System Design*, vol. 22, no. 2, pp. 133–141, 2003.
[11] A. Roychoudhury, T. Mitra, and S. Karri, "Using formal techniques to debug the amba system-on-chip bus protocol," *Design, Automation and Test in Europe Conference and Exhibition, 2003*, pp. 828–833, 2003.
[12] H. Amjad, "Model checking the AMBA protocol in HOL," University of Cambridge, Computer Laboratory, Tech. Rep. UCAM-CL-TR-602, Sep. 2004. [Online]. Available: http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-602.pdf
[13] J. Schmaltz and D. Borrione, "Towards a formal theory of on chip communications in the ACL2 logic," in *ACL2 '06*. ACM, 2006, pp. 47–56.
[14] X. Chen, S. M. German, and G. Gopalakrishnan, "Transaction based modeling and verification of hardware protocols," in *FMCAD '07*. IEEE Computer Society, 2007, pp. 53–61.
[15] F. Müffke, "A better way to design communication protocols," Ph.D. dissertation, University of Bristol, May 2004.
[16] C. Seger, "The design of a floating point unit using the integrated design and verification (IDV) system," in *DCC '06: Participants' Proceedings*, M. Sheeran and T. Melham, Eds., March 2006.
[17] P. Böhm and T. Melham, "Design and verification of on-chip communication protocols," Oxford University Computing Laboratory, Research Report RR-08-05, April 2008. [Online]. Available: http://web.comlab.ox.ac.uk/publications/publication1538-abstract.html
[18] T. Nipkow, L. C. Paulson, and M. Wenzel, *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, ser. LNCS. Springer, 2002, vol. 2283.
[19] A. Cimatti, E. M. Clarke, E. Giunchiglia, F. Giunchiglia, M. R. Marco Pistore, R. Sebastiani, and A. Tacchella, "NuSMV 2: An open source tool for symbolic model checking," in *CAV '02*. Springer, 2002, pp. 359–364.
[20] S. Tverdyshev, "Combination of Isabelle/HOL with automatic tools," in *FroCoS '05*, ser. LNCS, vol. 3717. Springer, 2005, pp. 302–309.

Symbolic Program Analysis using Term Rewriting and Generalization

Nishant Sinha
NEC Research Labs, Princeton, USA
nishants@nec-labs.com

***Abstract*—Symbolic execution [28] is a popular program verification technique, where the program inputs are initialized to unknown symbolic values, and then propagated along program paths with the help of decision procedures. This technique has two main bottlenecks: (a) the number of program execution paths to be explored may be exponential, and, (b) the state representation (map from variables to terms) may blow-up. We propose a new program verification technique that addresses the problems by (a) performing a *work list* based analysis that handles *join* points, and (b) simplifying the intermediate state representation by using term rewriting. In addition, our technique tries to compact expressions generated during analysis of program loops by using a term generalization technique based on anti-unification [40], [42]. We have implemented the proposed method in the F-SOFT verification framework using the Maude term rewriting engine. Preliminary experiments show that the proposed method is effective in improving verification times on real-life benchmarks.**

I. INTRODUCTION

Symbolic execution (SE) for program verification was proposed by King [28] and has been adopted as a popular verification technique [27], [41], [1], e.g., in the JavaPathFinder [2] tool, and forms the basis of systematic dynamic testing techniques [21], [13]. The main idea is to assign unknown symbolic values to input program variables and then propagate these values along paths of the program control flow graph using, e.g., depth-first-search (DFS). As compared to other verification techniques, SE combines the power of explicit and symbolic techniques in a unique manner for program verification: the technique can explore large program depths (which may be a bottleneck for symbolic techniques like *bounded model checking* [7]), while symbolically exploring all possible inputs simultaneously (as opposed to explicit state techniques [24] which enumerate the inputs). The symbolic state is represented as a pair (C, σ), where C is the path condition denoting the conjunction of all guards occurring in the current path, and σ is a mapping from program variables to their symbolic values (terms). A SE engine relies on two main components: a *term substitution* computation for evaluating program expressions and updating symbolic values due to assignments, and, a *decision procedure*, e.g., an SMT solver [20], [17], to check if the current symbolic values can be propagated into a conditional statement.

Two main bottlenecks arise when applying symbolic execution to verify large programs. Firstly, since the algorithm enumerates all program paths iteratively, there may be an exponential number of paths to be explored (known as the *path explosion* problem [1]), e.g., due to a sequence of conditional statements or function calls. Secondly, the terms representing the symbolic values of program variables eventually blow-up after several substitution operations. Moreover, symbolic execution of loops may lead to deep execution paths, which may cause further blow-up. Although modern incremental SMT solvers, e.g., [17], are able to handle such blow-up for path conditions to a certain extent by using specialized algorithms, deep program exploration reduces their performance significantly. Moreover, they do not help simplifying the state representation in any way. In this paper, we propose a new program analysis method to address the above shortcomings.

In order to avoid path explosion during analysis, our technique performs program traversal using a *work list* based approach, as in several data-flow analysis algorithms [37]. This approach necessitates handling *join* nodes by *merging* information obtained from the incoming paths. We combine the incoming symbolic information by using a `choose` function symbol, which allows us to select non-deterministically among multiple symbolic values. In contrast to approaches based on abstract interpretation [15] which perform approximation at join points, using `choose` avoids losing any path-specific information, while maintaining a functional representation of the state.

Term blow-up in the state representation is the main bottleneck for SE. Apart from the blow-up due to repeated substitutions, there is additional blow-up due to large `choose`-terms obtained by merging symbolic values from multiple paths. In order to ameliorate this blow-up, we propose to use *term rewriting* [4] methods to simplify intermediate terms. This technique relies on the simple, but crucial, insight that at most join points, the incoming symbolic values are guarded by disjoint path conditions. Hence, the `choose`-terms computed at those join points can be simplified to `ite` (*if-then-else*) terms. Storing symbolic values as structured `ite`-terms enables us to exploit their structure to obtain significant simplification. Consider, for example, the two program snippets in Figure 1(a) and (b). In (a), we obtain $(x \mapsto \texttt{ite}(z > 1, v_1, v_2)$ after analyzing the first conditional statement. On analyzing further, we obtain $\{y \mapsto \texttt{ite}(z = 3, k + \texttt{ite}(z > 1, v_1, v_2), \bot_y)\}$ where $\bot_y$ is an unknown value. Clearly, the terms representing symbolic values would blow up quickly from such analysis. Rewriting based `ite`-term simplification leads to significant term compaction in such cases. For example, using the rewrite rules (cf. Section III-A) we obtain $\{y \mapsto \texttt{ite}(z > 1, k+v_1, \bot_y)\}$. Similarly, in Figure 1(b), we get $\{b \mapsto t_1\}$. Simplifying

978-1-4244-2735-2/08/$25.00 ©2008 IEEE

$if(z > 1)\ x := v_1;$
else $x := v_2;$
...
$if(z = 3)\ y := x + k;$
(a)

$p := \&a;$
...
$b := (p = \&a)?t_1 : t_2;$
(b)

Fig. 1. Two motivating examples for ite-based simplification.

ite-terms is especially useful for cases like (b), where an imprecise pointer analysis may lead to large ite terms.

Compacting ite terms, therefore, is the focus of our simplification procedure. An equational axiomatization of the ite theory (the set of valid ite equations) was first provided by McCarthy [33] and later by Bloom and Tindell [8] and others [23], [35]. Sethi provided a more semantic algorithm (to overcome the locality of syntactic transformations) for simplifying ite-terms with equality predicates using two basic transformations: *implied* and *useless* tests [44]. Nelson and Oppen proposed a method for simplification of formula [38], involving rules for ite-simplification based on McCarthy's axiomatization. Burch and Dill [26] used a variant of Sethi's *implied* test to check satisfiability of ite-terms over the logic of equality and uninterpreted symbols (EUF) using a specialized case-splitting algorithm. Burch [11] proposed an improvement in terms of a simplification algorithm for ite-terms, again based on the *implied* test. Similar rewrite rules have been proposed to extend BDD-like [10] representations to the EUF logic and the logic of difference constraints, e.g., a rewrite system for normalizing decision diagrams with nodes containing equality over variables was given in [22].

In this paper, we propose a unified methodology for ite-simplification in terms of a (terminating) rewrite system specified in the general framework of rewriting logic [36], [32]. Our rewrite system combine the rules from the equational axiomatizations of the ite theory [33], [8], as well as more semantic rules for simplifying ite-terms [44], [11]. Such a formalization, to the best of our knowledge, has not been provided earlier. Moreover, our rules generalize and extend the previous set of rules to allow further simplification. A procedure for simplifying ite terms, in turn, needs simplification rules for terms involving other theories for exhaustive simplification. To this effect, we provide additional rules for simplifying Presburger arithmetic constraints, which commonly occur in program verification.

At loop heads, however, we cannot reduce choose-terms to ite-terms in most cases. Consider, for example, a simple loop $for(i := 0; i < 100; i := i + 2);$. The states obtained on symbolic analysis of this loop for variable i will be of form $\{i \mapsto \texttt{choose}(\langle(true, 0), (true, 2), (true, 4), ...\rangle)\}$ (explanation of precise syntax is in Section III), which will blow-up as the number of loop iterations become larger. Here, the values $0, 2, 4, \ldots$ occur under the same path condition $true$ and hence we cannot obtain an ite term. We propose a technique for compactly representing such terms based on the idea of *anti-unification*. Anti-unification [42], [40] (the dual of term *unification* [4]) is a method to generalize a set of terms by identifying the common structure in the term set, while *merging* the differences among them by introduction of new variables. Symbolic analysis of loops typically produces similar expressions at the loop head, which can be, abbreviated to a term *parameterized* by the loop counter. For example, on generalizing the above choose term, we obtain a *parameterized* term of form $\texttt{choose}((1 \leq j \leq k, 2 \times j))$, where j is a new index variable, k is the number of loop iterations performed at a given point, and, $1 \leq j \leq k$ and $2 \times j$ denote the parameterized condition and term values respectively.

The task of simplifying first-order logic terms is not trivial. While each of the simplification rules maybe relatively simple, there are a large number of them and they need to be applied effectively, in the correct order. Hence, using a rewrite engine is apt for the purpose of simplification. Moreover, our rule set is modifiable, i.e., rules can be added or removed in order to specialize the analysis to the verification task at hand. Note that having a dedicated simplification engine for this purpose is impractical, since it must be modified for every rule addition/deletion. Finally, we note that term rewriting based on rewrite logics [36], [32] provides a seamless framework for simplifying terms over a combination of theories, which is crucial for a program analysis algorithm.

We have implemented the proposed approach in the F-SOFT verification framework for C programs [25]. Preliminary experiments show that our approach scales up to real-life examples and is competitive with an optimized DFS-based symbolic execution procedure inspite of the additional overhead of our method in computing joins on symbolic values.

Related Work. Several approaches have been proposed to perform forward symbolic execution and backward weakest preconditions effectively. Expression renaming [19], [30] was proposed to avoid blowup during weakest precondition computation by using SSA representation. Although SSA representation assists SMT solvers to a certain extent, it does not allow semantic simplifications of symbolic values that SSA variables may assume. By preserving term structure in the symbolic state being propagated, our technique is able to perform term simplifications using rewriting, *before* flattening terms to clauses. These simplifications are hard to obtain at the clausal level inside an SMT solver, as is demonstrated by our experiments. Moreover, calls to SMT solver are obviated in many cases due to rewriting. Arons et al. [3] and Calysto [5], [6] reuse SAT query results by caching and structural term analysis. Simplification and caching are complementary optimizations: simplification can reduce SAT query times even when caching fails and vice-versa. Our approach is closest in spirit to that of Calysto [6]; however, simplification and generalization of terms are unique to our approach. The remainder of this paper is organized as follows. Section II provides background for rewriting and modeling C programs for verification. We describe our symbolic analysis algorithm and simplification rules in Section III. In Section IV, we present our simplification algorithm based on term anti-unification. Section V discusses the experimental results and conclusions.

II. PRELIMINARIES

The reader is requested to refer to [16] for an excellent introduction to rewriting. We only cover the main ideas relevant

to this paper here. A *signature* Σ is a set of function symbols, where each $f \in \Sigma$ is associated with a non-negative integer n, called the *arity* of f. The constant symbols have arity zero. A symbol can have one or more arities. We assume that Σ is many-sorted, i.e., different symbols belong to different sorts (types). Let $\mathcal{X}$ denote a set of variables disjoint from Σ. In this paper, variables are of either Boolean or integer sorts and function symbols of either *expression* or *predicate* sorts. Given a signature Σ and a set of variables $\mathcal{X}$, the set of all terms over $\mathcal{X}$ is denoted by $\mathcal{T}(\Sigma, \mathcal{X})$ ($\mathcal{T}$, in short). For example, $f(a, g(x, y))$ is a term over symbols f, g and a, and variables x and y, A term containing no variables is said to be a *ground* term. A term of predicate sort is also referred to as a formula. To avoid confusion, we will use the symbol $\equiv$ for syntactic identity of terms. We use symbols s, t, etc. to denote terms. We will often write binary function symbols (such as $+$, $*$), in infix form, e.g. $(x + y)$ instead of $+(x, y)$. We refer to a term with outermost symbol f as an *f-term* and denote the empty term by λ. We use symbols x, y for variables, a, b for constants, p, q for predicates, and f, g, etc. for other functions.

A substitution $\sigma : \mathcal{X} \rightarrow \mathcal{T}$ is a partial mapping from variables to terms denoted by $\{x_1 \mapsto s_1, \ldots, x_n \mapsto s_n\}$, indicating that the variable x_i maps to the term s_i. We also use the notation $\sigma(x_i) = s_i$. The domain of a substitution σ, denoted by $dom(\sigma)$ consists of the set of variables for which σ is defined. For a term t and a substitution σ, we denote the image of t under σ by $t\sigma$, defined as follows: if $\sigma(x) = s$, then $x\sigma \equiv s$; else if $x \notin dom(\sigma)$, then $x\sigma \equiv x$; and, $f(s_1, \ldots, s_n)\sigma \equiv f(s_1\sigma, \ldots, s_n\sigma)$.

A *term-rewriting* or *rewrite* system [4], [16] is a set of rewrite rules, $l \rightarrow r$, where l and r are each terms, both of which may contain variables which refer to arbitrary terms. An *equation* is a formula of the form $r = s$ where r and s are terms. A *conditional equation* is of form $u_1 = v_1 \wedge \cdots \wedge u_n = v_n \Rightarrow r = s$. A *(conditional) equational system* is any set of equations. Rewriting mainly involves *orienting* equations (fixing a direction for applying them) to obtain rules of form $l \rightarrow r$, finding a subterm t *matching* l (say, $t = l\sigma$), and then *applying* substitution to replace t by $r\sigma$. The goal of such a computation is to obtain a *normal* form, i.e., a term which cannot be rewritten further. Rewrite systems have two desirable properties: *termination* and *confluence*. A rewrite system is terminating if no term can be rewritten infinitely. A rewrite system is confluent if rewriting equal terms always leads to a unique normal form (irrespective of the order of application of rules). A terminating and confluent rewrite system is also said to be *convergent*: all derivations in such systems lead to unique normal forms. Hence, such systems can be used to decide equational theories by checking if two (potentially equal) terms rewrite to the same normal form.

A rewrite logic theory [36], [32] consists of a set of uninterpreted symbols (e.g., car, $cons$, $insert$, $delete$, etc.) constrained equationally by defining a set of equations on those symbols, together with a set of rewrite rules meant to define the evolution of the system. Although equational definitions differ from rewrite rules semantically, both of them are executed as rewrite rules. In this paper, we restrict ourselves to equational definitions, e.g., of the `ite` symbol. Rewrite logics are quite powerful: they can be used to specify semantics of different kinds of languages and models of concurrency as well as different kinds of logic. Datatypes defined using algebraic specifications and the operations (constructors, modifiers, observers) on them are specified using (conditional) equations. For example, Maude [14] is a high performance system that supports both equational and rewriting logic specifications over many-sorted signatures. The ability to specify conditional equations in Maude allows us to write simplification rules involving multiple theories.

A. Modeling C programs

We describe the basic modeling assumptions for verifying C programs in F-SOFT [25]. A program is modeled as a control flow graph (CFG) $G = (V, E)$, where V and E are the set of nodes and edges respectively. A subset of nodes in a CFG are marked as *error* nodes, which correspond to assertion failures when checking for various program properties. Each edge $e_{ij} = (v_i, c_{ij}, v_j)$ represents a guarded transition between nodes v_i and v_j with the guard c_{ij}. Each node has a set of assignments associated with it (parallel assignments to program variables). Each assignment is of the form $x := t$, where x is a program variable and t is a term over program variables. Each program variable is assigned an integer address and pointers as modeled as integers having address values. Pointer dereferences are translated to an `ite`-term based on a points-to analysis [25]; the result of the expression is the value of variable v if the pointer value equals the integer address value assigned to v. Functions are not inlined; matching calls and returns are identified by assigning a unique value to a caller-id variable for each call-return pair. Heap and stack are modeled as finite arrays, which leads to a bounded model[1]. Please see [25] for more details.

III. SYMBOLIC ANALYSIS ALGORITHM

As mentioned earlier, our symbolic analysis algorithm uses a *work list* to propagate symbolic values instead of exploring each path explicitly. We first describe the data representation used in the algorithm.

Symbolic Data. The algorithm represents the symbolic state as a *condition-map* pair (C, σ) (*cm-pair* in short), where C denotes the current *path condition* predicate and σ denotes a map from program variables to their symbolic values (terms $t \in \mathcal{T}$). A *condition-value* pair $cv = (C, t)$ (*cv-pair* in short) is the projection of a cm-pair on a particular program variable, say x, where C is the path condition and the term t denotes the value of x. Intuitively, $x \mapsto (C, t)$ denotes the set of concrete program states where C holds and x has the symbolic value t. For each CFG node n, the algorithm stores a list of cm-pairs, denoted by $Data_n$. This list accumulates symbolic values for a node so that we can process them later in a lazy manner. Given a cm-pair $D = (C, \sigma)$ and a term t, the substitution $t\sigma$ denotes the *evaluation* of t with respect to D. Also, we assume that a procedure $ComputePost$ evaluates the set of assignments

[1] Although our approach can be extended to handle dynamically allocated data structures in the manner of [27], we shall restrict our attention to finite size heaps and stacks in this paper.

at a CFG node n with respect to a cm-pair $D = (C, \sigma)$ and returns a new pair (C, σ') with updated values of variables.

`choose` *and* `ite` *terms.* We assume that the signature Σ contains four special function symbols, `ite` and `itep`, `choose`, `choosep`. The `ite` and `itep` symbols denote the if-then-else expressions over integer and predicate sorts, respectively. A term of form $\texttt{choose}((C_1, t_1), (C_2, t_2), (C_3, t_3))$ on expression sort denotes a non-deterministic choice between the values t_i $(1 \leq i \leq 3)$ given the corresponding condition C_i holds. In other words, it corresponds to the union of sets of states for (C_i, t_i). Similarly, we define `choosep` on predicate sorts. A `choose`-term can have any number of cv-pairs as its argument. For ease of notation, we will represent both `ite` and `itep` by `ite` and `choose` and `choosep` as `choose`, unless not clear from context. Also, we will represent a `choose`-term as an abbreviated list, i.e., $\texttt{choose}(\langle C_i, t_i \rangle)$.

Figure 2 shows the pseudocode of our algorithm. The main procedure is SYMPROGANALYZE (SPA), which processes CFG nodes by iteratively popping a node n from a reverse postorder work list Q [37] (essentially, where parent nodes are popped before child nodes), collecting the incoming symbolic information at n using $ProcessData$ procedure, propagating the values to its successors and finally inserting the successors back into the list for processing. The $ProcessData$ procedure first obtains the current symbolic value D associated with a node n using the $UnifyData$ procedure and then uses $ComputePost$ to update D with the results of assignments at node n. The procedure $UnifyData$ collects the values in $Data_n$ by disjuncting the path conditions and combining the values in form of a `choose` term for each variable. Then SPA determines the successors n' of n to which values must be propagated. This is done by evaluating the edge condition C in the current *state* σ_n $(C\sigma_n)$ and checking if its conjunction with the current path condition C_n is satisfiable using a decision procedure (DP). If satisfiable, $Data_{n'}$ is updated with the new value D' and n' is inserted back into the list Q. After processing n, SPA sets $Data_n$ to an empty set. This avoids propagating the same set of values in the future iterations. Note that we need to eliminate the `choose` symbol from the predicates that we send to a DP. We describe this translation in the next sub-section.

Discussion. The SPA procedure is sound with respect to reachability of error nodes. Given a reachable error node and the corresponding path condition C, a witness input can be generated by using a DP to obtain a model for C. SPA may not terminate in general, due to undecidability of the program verification problem. In practice, we use *under-approximation* of values at loop exiting nodes to ensure termination (see Section IV). In this case, our procedure can only detect errors soundly. The presented procedure can be optimized by checking if the current set of cm-pairs at a node being processed are *subsumed* by the cm-pairs processed at that node earlier. Also, one can use a different priority ordering for the work list that processes all the loop nodes first and then propagates the collected values forward.

Given a cm-pair $cm = (C, \sigma)$ and a list of cm-pairs $l = \langle (C_1, \sigma_1), \ldots, (C_n, \sigma_n) \rangle$, we say that cm is subsumed by l, if the set of states represented by cm is a subset of the union of the set of states represented by elements of l. We can check if cm is subsumed by l, denoted as $subsume(cm, l)$, by using a DP. We assume that the terms have been rewritten so that no unary or binary operator term contains a `choose` or `ite`-term as a subterm. The predicate $subsume(cm, l)$ is defined by structural induction over terms:

(S1) $subsume((C, \sigma), \langle C_i, \sigma_i \rangle) =$
$\bigwedge_{x \in dom(\sigma)} subsume((C, \sigma(x)), \langle C_i, \sigma_i(x) \rangle)$

(S2) $subsume((C, t), \texttt{choose}(\langle C_i, t_i \rangle)) =$
$C \Rightarrow \bigvee_i (C_i \wedge subsume((C, t), t_i)$

(S3) $subsume((C, t), \texttt{ite}(C', t_1, t_2)) =$
$\bigwedge_{i=1}^{2} (C \wedge C') \Rightarrow subsume((C, t), t_i)$

(S4) $subsume((C, t), t') = (x = t \Rightarrow x = t')$, otherwise,
where x is a fresh variable of appropriate sort.

```
proc SymProgAnalyze()
  // Let Q be a reverse postorder work list [37] of CFG nodes
  while Q is not empty do
    Extract the first element n from Q
    if n is an error node then
      // Invoke error handling
    end if
    (C_n, σ_n) := ProcessData(n)
    for all successor n' of node n in CFG do
      // Let C be the constraint on the edge (n, n')
      C'_n := Cσ_n ∧ C_n
      if C'_n is satisfiable then
        D' := (C'_n, σ_n)
        Data_n' := Data_n' ∪ {D'}
        insert(Q, n')
      end if
    end for
    Data_n := {}
  end while
end proc
proc ProcessData(n)
  D := UnifyData(Data_n)
  // Let St_n be the set of statements at the node n
  D' := ComputePost(D, St_n)
  return D'
end proc
proc UnifyData(⟨(C_1, σ_1), ..., (C_n, σ_n)⟩)
  C := (C_1 ∨ ... ∨ C_n)
  σ := {x ↦ choose(⟨(C_i, σ_i(x))⟩) |
  x ∈ ⋃_{i=0}^{n} (dom(σ_i) where σ_i(x) is well-defined.}
  return (C, σ)
end proc
```

Fig. 2. Pseudocode for the algorithm SYMPROGANALYZE (SPA).

A. Simplification by Rewriting

The symbolic values, in particular `choose` terms, generated by the SPA procedure often blow-up quickly due to repeated substitutions and joins. We now describe rules for simplifying `choose`-terms; the main idea is to convert them to `ite`-terms whenever possible and then use a variety of simplification rules on `ite`-terms to avoid the blow-up. The rules are presented in terms of equations $l = r$; they can be directly converted to rewrite rules $l \rightarrow r$. In the following, we use the following function symbols (with appropriate associativity and commutativity assumptions): `EQ` and `NEQ` to denote equality and disequality (respectively) on expressions and integers; and, not , $implies$ denote conjunction, negation and implication over predicates; $<$, $\leq$, $>$, $\geq$ represent the usual relation operators over expressions. The symbol $\odot$ stands for the

common binary operators ($\{+, *, ...\}$) over expressions. E, E', Et, Ee, F and G, are terms of expression sort, P and Q are integers and X, Y are variables of integer sort. An equation suffixed by the `[owise]` construct (borrowed from Maude syntax [14]) denotes the fact that it will be used for rewriting when no other similar equation is applicable.

a) Rules for choose: Here, the variable CV denotes a (possibly empty) list of cv-pairs. We denote an unknown value by $\perp$. (C1-4) show the distributive and redundancy removal rules. Rules (C5-6) convert a `choose`-term to an `ite` term. Note that rule C6 involves checking that that C_1 is disjoint from other conditions.

(C1) $\texttt{choose}(\langle(false, t), CV\rangle) = \texttt{choose}(CV)$

(C2) $\texttt{choose}(\langle(C_1, t), (C_2, t), CV\rangle) = \texttt{choose}(\langle(C_1 \vee C_2, t), CV\rangle)$

(C3) $\texttt{choose}(\langle C_i, t_i\rangle) \odot t = \texttt{choose}(\langle C_i, t_i \odot t\rangle)$

(C4) $\texttt{choose}(\langle(C, \texttt{choose}(\langle C_i, t_i\rangle)), CV\rangle) = \texttt{choose}(\langle(C \wedge C_i, t_i), CV\rangle)$

(C5) $\texttt{choose}(\langle C, t\rangle) = \texttt{ite}(C, t, \perp)$

(C6) $\texttt{choose}(\langle(C_1, t_1) \cdots (C_n, t_n)\rangle) = \texttt{ite}(C_1, t_1, \texttt{choose}(\langle(C_2, t_2) \cdots (C_n, t_n)\rangle$ if all for $2 < i < n$, $C_1 \wedge C_i = false$

b) Rules for ITE: Figure 3(a) shows a representative set of rules for `ite`-term simplification. The equations `I1-6` correspond to equational axioms [33], [8]. The equations `I7-10` generalize the axioms `I5-6`. The equation `I11-12` allows distributing binary and unary operators to enable further simplification. Rules `S1-10` correspond to more semantic rules proposed by Sethi [44], [11]. The rules `S3-4` and `S5-6` are a generalization of the *implied* and *useless* tests respectively [44], which is obtained by using a conditional equation. They are defined using a new operator $simplEnv(t, P)$, denoting simplification of t assuming that an *environment condition* P holds. This is especially useful for simplifying nested `ite`-terms since the guard of an outer `ite`-term acts as a valid environment condition for an `ite` subterm. For simplifying any given term t, we invoke $simplEnv(t, true)$ at the top-level. The $simplEnv$ rules modularize reasoning over `ite`-terms with reasoning on subterms from an arbitrary theory. For example, rule `S6` checks if the condition P implies C is true, which is in turn performed by rewriting with rules for the (arbitrary) theories on which P and C are defined.

`S7-8` are redundant rules (subsumed by `S3-4`); however, they are useful in cases where pattern-matching is sufficient to simplify the term. Rules `S1-4` are new and provide additional simplifications over other rules. The main idea is that even if simplification cannot fully eliminate an `ite`-subterm, e.g., in rule `S5-6`, `ite`-subterms can still be simplified. `S9-10` rules are used when no other rule is applicable. The rules for `itep` of predicate sort are analogous and omitted. We have also omitted rules for simplifying predicates over and and not symbols. A comprehensive set of rules can be found in [9]. We have also omitted rules for integers and Booleans, which are standard.

c) Rules for Presburger Arithmetic: The arithmetic rules `A1-13` essentially normalize all inequalities and disequalities to terms having only EQ and $\leq$ symbols, with integer constants collected on one side of the relation. Rules `A14-18` handle the interaction between EQ and $\leq$ operators on terms. Rules `A19-20` provide simplification for difference logic constraints.

Example. Consider a program fragment where on analysis, we obtain a term of form $\texttt{choose}((x > 1, y\ EQ\ 1), (x \leq 1, y\ EQ\ 0))$. Using the `choose` rules `C5-6`, we obtain $\texttt{ite}(x > 1, y\ EQ\ 1, \texttt{ite}(x \leq 1, y\ EQ\ 0, \perp))$. Using the $simplEnv$ rules (mainly due to `S3`), we get $\texttt{ite}(x > 1, y\ EQ\ 1, y\ EQ\ 0)$ as the final result.

Theorem 1: The rewrite system consisting of rules in Figure 3 is terminating.

Proof. Let the ground symbols be ordered as follows: $simplEnv \gg \{EQ, NEQ, -, +, *\} \gg \texttt{ite} \gg \{not, and, <, >, \geq, implies\} \gg \{\leq\} \gg \texttt{itep} \gg \{true, false\}$. The `ite` rules in Fig. 3(a) can be shown to be *Neotherian* [16] (the corresponding rewrite system without conditions terminates), by using a *multiset path ordering* [4] (say, $\prec$) induced by $\gg$. It remains to show that rewriting due to recursive evaluation of conditions terminates. For this, we need to show that the rules for Presburger arithmetic (excluding `itep`) terminate. Since conditions in these rules only involve predicates over integers, the rewriting in conditions terminates. Now, this rewrite system can be shown to terminate by using a multiset ordering consisting of the following orderings: (i) the reduction ordering $\prec$ (ii) minimum of number of constants on each side of EQ + number of negated non-constant terms, (iii) sum of number of constants in the first argument and number of variables in the second argument of the $\leq$ operator, and (iv) the total sum of non-constant terms. The rewrite system consisting of Presburger arithmetic together with `itep` rules can be shown to be Neotherian based on the reduction ordering $\prec$. We can show that conditional equations for `itep` terminate due to an inductive argument: the rewrite system inside conditions will rewrite terms with fewer `itep` symbols than the rewrite system containing the equation, with the Presburger arithmetic rewrite system as the base case. □

d) SMT Decision Procedure (DP) Interface.: Recall the pseudocode for SPA procedure given in Figure 2. In order to check for path feasibility, the procedure invokes a DP, an SMT solver [20], [17] in our case. Given an SMT DP interface, we need to translate the symbolic value terms to a format that the interface supports. Although the translation for most other terms is standard, we need to translate predicate terms with `choose` as the topmost symbol. Let term t be of form $\texttt{choosep}(\langle C_i, t_i\rangle)$. Note that for each i, t_i is a predicate. Therefore, we can translate t into an equivalent formula $\vee_{i=0}^{n}(C_i \implies t_i)$. In general, the translation is as follows. Since the rewrite rules contain distributive laws for `choosep` and `itep` for all binary and unary operators, so for any given predicate term t, either `choosep` or `itep` is the top symbol, or there exists no subterm with either of the symbols. In the first case, t must be a predicate and can be handled as above. In the second case, if t is of form $\texttt{itep}(P, E_1, E_2)$, then both E_1 and E_2 are predicates and can be translated recursively. A DP is also used for checking for mutual exclusiveness of arguments in `choose`-terms in the rule `C6`.

Optimizations. We do not initialize all the program variables to unknown values in the beginning. Instead, when computing

```
(I1) ite (true, E, E') = E
(I2) ite (false, E, E') = E'
(I3) ite (not N, E, E') = ite (N, E', E)
(I4) ite (N, E, E) = E
(I5) ite (N, ite(N, E, E'), F) = ite (N, E , F)
(I6) ite (N, F, ite(N, E, E') ) = ite (N, F , E')
(I7) ite (N, ite(N1, E, E'), E)
    = ite (N and not N1, E', E)
(I8) ite (N, ite(N1, E, E'), E')
    = ite (N and N1, E, E')
(I9) ite (N, E, ite(N1, E, E'))
    = ite (not N and not N1, E', E)
(I10) ite (N, E', ite(N1, E, E'))
    = ite (not N and N1, E, E')

(I11) ite (N, E, E')⊙F = ite (N, E⊙F, E'⊙F)
(I12) - ite (N, E, E')= ite (N, -E, -E')

(S1) simplEnv (V, V EQ N and C) = N
(S2) simplEnv (ite(C and P, Et, Ee), C and Q)
  = simplEnv(ite (P, Et, Ee), C and Q )
(S3) simplEnv (ite(C, Et, Ee), P)
  = simplEnv(Ee, P) if P implies (not C)
(S4) simplEnv (ite(C, Et, Ee), P)
  = simplEnv(Et, P) if P implies C
(S5) simplEnv (ite(C, X, Y), P) = Y
  if (P and C) implies (X EQ Y)
(S6) simplEnv ( ite(C, X, Y), P) = X
  if (P and not C) implies (X EQ Y)
(S7) simplEnv (ite(C, Et, Ee), C and P)
  = simplEnv(Et, C and P)
(S8) simplEnv (ite(C, Et, Ee), not C and P)
  = simplEnv(Ee, not C and P)
(S9) simplEnv (ite(C, Et, Ee), P )
  = ite (C , simplEnv(Et, P and C),
      simplEnv(Ee, P and not C) ) [owise]
(S10) simplEnv (E , P ) = E [owise]
```

(a)

```
*** Presburger arithmetic rules
(A1) (E NEQ E') = not (E EQ E')
(A2) ((E EQ P) and (E EQ Q)) = false if P NEQ Q .
(A3) (not(E EQ P) and (E EQ Q)) = E EQ Q if P NEQ Q .
(A4) ((E + P) EQ Q) = (E EQ (Q - P))
(A5) (-E EQ P) = (E EQ -P) if E is not an integer
*** normalize to <=
(A6) (E > E') = (E' <= E - 1)
(A7) (E < E') = (E <= E' - 1)
(A8) (E >= E') =  (E' <= E)
(A9) (not (E <= E')) = (E' <=  E - 1)
*** orient constants to right and other terms to left
(A10) (E + P <= E') = (E <= E' - P)
(A11) (P <= E') = (0 <= E' - P)
(A12) (E <= F) = (E - F <= 0) if F is not an integer
(A13) (E <= F + G) = (E - F <= G) if F is not an integer

*** constant propagation
(A14) ((E <= F) and (E EQ P)) = (P <= F) and (E EQ P)
(A15) (E' + E <= F) and (E EQ P)
  = (E' + P <= F) and (E EQ P)
(A16) (-E <= F) and (E EQ P)
  = (-P <= F) and (E EQ P)
(A17) ((E <= P) and (E <= Q)) = E <= P if P < Q
(A18) ((E <= P) and (E <= Q)) = E <= Q [owise]

*** difference logic rules
(A19) ((E + (- F) <= P and F + (- E) <= Q))
  = false if P + Q < 0
(A20) ((E <= P and  - E <= Q))
  = false if P + Q < 0
```

(b)

Fig. 3. Rewrite rules for simplifying ite-terms and Presburger arithmetic terms

a substitution, say $t\sigma$, we lazily initialize [27] a variable that occurs in t but not in σ. The simplification algorithm is called at two places: (i) during construction of choose-term in $UnifyData$ procedure (cf. Fig. 2) and (ii) before calling the DP on a path condition formula. In the first case, we obtain further compaction by simplifying under the environment context of the disjuncted path condition C (by using $simplEnv(ite_x, C)$ where ite_x is the ite-term value for variable x). We first simplify top-level choose-terms and then apply the ite and other rules.

IV. Simplification for loops

As mentioned earlier, the choose terms may blow-up during analysis of a loop since ite-based simplifications cannot be applied in such cases. We propose a technique to simplify these terms based on the technique of *anti-unification* [42], [40]. The essential idea is to find a common pattern in the arguments of a choose expression that allows us to represent the list of arguments by a single *parametric* expression (if possible), together with additional constraints on the parameters. In other words, our goal is to obtain a parameterized generalization of the list of arguments in a choose-term.

Two terms t_1 and t_2 are said to be *anti-unifiable* if there is a term t (called the *anti-unifier*) and two substitutions σ_1, σ_2 such that $t\sigma_1 \equiv t_1$ and $t\sigma_2 \equiv t_2$. Intuitively, an anti-unifier [42], [40] (or a generalization) of a list of terms retains information that is common to all the terms, introducing new variables when information conflicts. We are interested in the most specific generalization, if it exists, which captures maximum similarity between the terms, e.g., the most specific generalization of the terms $f(a, g(b, c))$ and $f(b, g(x, c))$ is $f(z, g(x, c))$. Since the first argument of f can be either a or b, the introduction of a new variable z *abstracts away* this information in the generalized term. The generalization algorithm also produces two substitutions as a result: $\sigma_1 = \{z \mapsto a, x \mapsto b\}$ and $\sigma_2 = \{z \mapsto b\}$. Note that we can always extend σ_1 and σ_2 so that $dom(\sigma_1) = dom(\sigma_2)$ holds. It is known that for any list of terms, a most specific generalization (or anti-unifier) exists [42], [40] (for terms that are completely different, a single variable term is the trivial generalization).

In our case, however, we want to find a *parameterized* generalization of a list of terms, i.e., given $T = \{t_1, \ldots, t_n\}$, we need to compute some term t having a free variable k, such that $t_i \equiv t\sigma_i$, where $\sigma_i = \{k \mapsto i\}$ for all $1 \leq i \leq n$. We call t a *parametric* term and write it as $t(k)$ henceforth[2]. Also, we define a *bounded parametric* (*bp*-) term to be of form $\oplus_{k=l}^{u}(t(k))$, where u and l denote the upper and lower bounds of the parameter k and $\oplus$ denotes an associative operator. This allows us to represent a list of similar terms by a single bp-term, e.g., T is represented as $t' = \oplus_{k=1}^{n}(t(k))$, where $\oplus$ stands for the list *cons* operator. Also, we say that the term list T is *abbreviated* by a bp-term t'. Moreover, if the given list of terms is an argument to some function symbol f, then we use f instead of $\oplus$. For example, $(a \neq 1 \wedge \cdots \wedge a \neq n)$, represented as $\bigwedge^*(a \neq 1, a \neq 2, \ldots, a \neq n)$ in normalized

[2]Parametric terms of predicate sort are similar to indexed predicates [29]

```
proc P-AU (⟨t_1, ..., t_n⟩)
  {t_au, ⟨σ_1, ..., σ_n⟩} := MATCH(⟨t_1, ..., t_n⟩)
  if ∀i. dom(σ_i) is empty then
    return t_au
  end if
  σ := PARAMETERIZE(⟨σ_1, ..., σ_n⟩)
  if dom(σ) ≠ ⋃_{i=0}^{n} dom(σ_i) then
    return λ
  else
    return ⊕_{i=1}^{n} PSUBST(t_au, σ)
  end if
end proc
proc PARAMETERIZE (⟨σ_1, ..., σ_n⟩)
  Let σ := {}
  // Let V := dom(σ_1) (= dom(σ_2) ··· = dom(σ_n))
  for all v ∈ V do
    // Let j and k be fresh variables not occurring elsewhere
    if ∀i. σ_i(v) is a constant (let σ_i(v) = c_i) then
      Solve the list of equations E = {a × i + b = c_i}
      if ∃a_0, b_0 satisfying all equations in E then
        σ(v) := ⊕_{j=1}^{n}(a_0 × j + b_0)
      end if
    end if
    if ∀i. σ_i(v) is a list (let σ_i(v) = ⟨t_i^1, ..., t_i^n⟩) then
      Compute for all i, t_i := P-AU(⟨t_i^1, ..., t_i^n⟩)
      // Let t_i be of form ⊕_{l_i}^{u_i}(t'_i(j_i))
      Compute t(j) := P-AU(⟨t'_1(j_1), ..., t'_n(j_n)⟩)
      Compute ⊕_{k=1}^{n}(u(k)) := P-AU(⟨u_1, ..., u_n⟩)
      Compute ⊕_{k=1}^{n}(l(k)) := P-AU(⟨l_1, ..., l_n⟩)
      if t(j), u(k) and l(k) are non-empty then
        σ(v) := ⊕_{k=1}^{n}(⊕_{j=l(k)}^{u(k)}(t(j)))
      end if
    end if
  end for
  return σ
end proc
```

form [18], is abbreviated as $\bigwedge^*(\oplus_{i=1}^{n}(a \neq i))$, which we write as $(\bigwedge_{i=1}^{n}(a \neq i))$.

We present a procedure (P-AU) that implements this generalization algorithm below. The procedure P-AU first calls MATCH to find the common structure among the terms and compute the substitutions for the differences. The procedure MATCH computes an anti-unifier t_{au} of a list of terms $t_1, \ldots, t_n$ together with the corresponding substitutions $\sigma_1, \ldots, \sigma_n$ respectively. These substitutions are passed on to PARAMETERIZE function, which computes a parametric substitution σ, if it exists, that generalizes the input substitution set. The PARAMETERIZE function either uses templates (e.g., a linear equation) for generalizing base cases or recursively calls P-AU for anti-unifying a list of subterms. Finally, the parameterized substitution σ is applied to the term t_{au} obtained earlier, by using PSUBST. MATCH uses an algorithm for computing a syntactic anti-unifier [42], [40]. If the function symbols in the terms are associative or commutative, the terms are first converted to their normal forms [18] and then the previous algorithm can be used, e.g., the normalized form for a term containing nested $\wedge$s has $\wedge^*$ as the top symbol. Alternatively, an approach to compute anti-unifiers modulo an equational theory based on manipulating regular tree languages can be used [12]. In general, MATCH may return multiple possible substitutions, which must be processed by the procedure P-AU iteratively until an anti-unifier is found. Given a parameterized substitution $\{\sigma = v \mapsto \oplus_{i=l}^{u}(t(i))\}$ and a term t with free variable v, PSUBST(t, σ) returns $\oplus_{i=l}^{u}(t')$ where t' = PSUBST$(t,\{v \mapsto t(i)\})$. For example, PSUBST$(c+v, v \mapsto \oplus_{i=0}^{n}(i+4))$ evaluates to $\oplus_{i=0}^{n}(c+i+4)$.

Example. Given a list of terms $t_1 \equiv (a \neq 1)$, $t_2 \equiv (a \neq 1 \wedge a \neq 2)$, $t_3 \equiv (a \neq 1 \wedge a \neq 2 \wedge a \neq 3)$, P-AU$(\langle t_1, \ldots, t_3\rangle)$ will work as follows. P-AU first calls MATCH, which returns $\wedge^*(v)$ with $\sigma_i(v) = \langle a \neq 1, \ldots, a \neq i\rangle$ for $1 \leq i \leq 3$. Here, each $\sigma_i(v)$ is a list and hence calling PARAMETERIZE leads to three recursive calls to P-AU with argument $\langle a \neq 1, \ldots, a \neq i\rangle$ for $1 \leq i \leq 3$. As a result, we get three terms $\oplus_{j=1}^{1}(a \neq j)$, $\oplus_{j=1}^{2}(a \neq j)$ and $\oplus_{j=1}^{3}(a \neq j)$. The next call to P-AU trivially returns $a \neq j$. The next two calls to P-AU return i (as a result of solution of linear equations) and 1 respectively. Hence PARAMETERIZE returns $\sigma = \{v \mapsto \oplus_{j=1}^{i}(a \neq j)\}$. Finally, PSUBST$(\wedge^*(v),\sigma)$ produces the result $\bigwedge_{j=1}^{i}(a \neq j)$, and the value $\oplus_{i=1}^{n}(\bigwedge_{j=1}^{i}(a \neq j))$ is returned.

The PARAMETERIZE algorithm will succeed only if a list of constants can be anti-unified using a linear equation template of form $a \times k + b$ (otherwise it returns an empty term λ). The algorithm can be extended to allow addition of more templates (e.g., of polynomial form) provided a decision procedure is available for the corresponding theory.

Theorem 2: Let $t = \oplus_{k=0}^{n}(t(k))$ = P-AU$(\langle t_1, \ldots, t_n\rangle)$ ($t \not\equiv \lambda$). Then $\forall 1 \leq i \leq n$. PSUBST$(t(k),\{k \mapsto i\}) \equiv t_i$ (modulo expansion).

Proof. By structural induction on terms.

Before we present the complete algorithm for simplifying `choose`-terms, we need a technique to convert a bp-term obtained after generalization back into a term without bounds. Given a bp-term t of form $\oplus_{i=l}^{u}(t(i))$, we can obtain a cv-pair over *ordinary* terms by iteratively eliminating bounds in t inside-out using a quantifier elimination procedure for Presburger arithmetic, e.g., Fourier-Motzkin elimination [43]. We assume that a procedure ELIM-BP performs this conversion: for a term t, ELIM-BP(t) = (c, t'), where c is the predicate corresponding to bound constraints and t' is an ordinary term. For example, ELIM-BP$(\bigvee_{i=1}^{10}(\bigwedge_{j=1}^{i}(a \neq j)))$ returns a cv-pair $(a < 0 \vee a > 1, true)$ after simplification. We now present the ABBR-CHOOSE algorithm for abbreviating a `choose`-term.

```
proc ABBR-CHOOSE(choose((c_1, t_1), ..., (c_n, t_n)))
c := P-AU(⟨c_1, ..., c_n⟩)
t := P-AU(⟨t_1, ..., t_n⟩)
if both c and t are non-empty then
  (c', c'') := ELIM-BP(c)
  (c_t, t') := ELIM-BP(t)
  return choose((c' ∧ c'' ∧ c_t, t'))
else
  return choose(⟨(c_1, t_1), ..., (c_n, t_n)⟩)
end if
```

Given a `choose`-term, say t, the ABBR-CHOOSE procedure invokes P-AU on the list of conditions and values separately. If P-AU succeeds, then a new `choose`-term with the abbreviated cv-pair is returned. In this case, we obtain a more compact representation of the `choose`-term by adding constraints in terms of new variables. Otherwise, the original expression is returned. Note that `choose`-terms are generated at all join points. However, it is wasteful to try to abbreviate them at nodes other than loop heads (loop exit nodes, in general), since the terms obtained from merging different paths are mostly different. Therefore, we only apply ABBR-CHOOSE at loop heads. Note that ABBR-CHOOSE does not approximate the

choose terms; it tries to compute an equivalent parameterized term, if possible. Although ABBR-CHOOSE may lead to a more compact choose-term at a loop exit after each iteration, the number of iterations needed for exiting a loop can be large or unknown. We propose to avoid this problem by computing an under-approximation as follows.

Under-approximation. We under-approximate by essentially fixing the maximum number of iterations, say cnt, for which we analyze the loop. Given a loop condition predicate p, we say that a cm-pair (c, σ) is *loop-exiting* if $(p\sigma \wedge c)$ is satisfiable. When the number of loop-exiting cm-pairs at a loop-exit node exceeds cnt, we under-approximate by merging and propagating the current set of cm-pairs outside the loop. As a result, our approach can only detect errors soundly; in order to obtain proofs, we either have to continue analysis until the loop terminates or use an over-approximate loop invariant [15]. The abbreviated bp-term may also be used for computing the loop bounds and invariants, but we have not explored this direction further.

V. IMPLEMENTATIONS AND EXPERIMENTS

We implemented the proposed symbolic program analysis (SPA) algorithm in the F-SOFT [25] verification framework for C programs. We use the SMT solver YICES [17] as the decision procedure (DP) and the Maude rewrite engine [14] for implementing rewrite-based simplifications. We also implemented an optimized DFS-based symbolic execution (DFS-SE) algorithm, similar to [2] (without *symbolic state abstraction*), in the F-SOFT framework using YICES. Two parameters, *depth* and *loop* bounds are used to limit the search in DFS-SE. The depth bound denotes the maximum depth of the stack, while the loop bound denotes the maximum number of times a node can occur on the stack. SPA also uses the loop bound parameter for under-approximation (cf. Section IV). For handling path conditions in DFS-SE, we experimented with incremental SMT solving in YICES (+IncSMT) and a combination of rewriting with DP (+Rew). For SPA, we compared the results with and without rewriting based simplifications (+/-Rew). The rules for choose-terms are implemented in memory (and also used in SPA-Rew) while the rest of them are specified in a text file used by Maude. The procedure for generalization of choose-terms is under implementation.

Figure 4 shows the experimental results. Two benchmarks were used in experiments: we checked (i) an air-traffic controller software (TCAS) for reachability of error nodes (p_3,p_4 are reachable, while the rest are not), (ii) a module in a network protocol implementation (M) for pointer validity (30 error nodes). No under-approximation (in terms of depth and loop bounds) was required for verifying TCAS. For M, we fixed the depth bound and loop bound to 30 and 5, respectively. Our method SPA(+Rew) finished analyzing TCAS in 88s. In comparison, SPA(-Rew) timed out after 1800s. This clearly highlights the impact of ite-based simplifications. SPA also detects 28 out of 30 errors in module M are within 15s (SE-DFS+IncSMT takes 9s). In summary, our method (SPA+Rew) is competitive with an optimized DFS-SE implementation, inspite of additional overhead of computing joins and term simplification. Moreover, SPA outperforms DFS-SE without incremental SMT (but with rewriting) in most cases. Further, rewriting path conditions even improves the performance of plain DFS-SE. We also observed that SPA is competitive on these benchmarks with the F-SOFT implementations of other software verification techniques, e.g., based on bounded model checking and predicate abstraction.

In general, the average time for invoking a rewrite engine (with a large number of rules) is no more than that associated with using a DP. Also, rewriting often simplifies a formula to either a constant or simple enough to be checked heuristically for satisfiability. However, we observed that the current set of rules are not effective in simplifying predicate terms of large size, mainly due to the fact that the conditional rules require *recursive* rewriting to decide such terms. Moreover, we perform eager rewriting for simplification, which is not successful in many cases. Therefore, methods to perform a more lazy and selective rewriting must be developed. In summary, we observed that rewriting based simplification assists our program analysis (and SMT solvers) dramatically in scaling up. However, eager rewriting, especially for formulas, neutralizes the improvements obtained to a certain extent.

Conclusions. We presented a symbolic program analysis (SPA) method that uses symbolic terms to perform a precise data-flow analysis on programs, by merging symbolic values from different paths. SPA tries to capture program structure in the symbolic state (terms) computed during analysis and avoids analyzing the same CFG region multiple times. To this goal, SPA first imposes a work-list ordering to follow program structure and combines path information using choose-terms. Second, SPA exploits the term structure to perform on-the-fly simplification and generalization during analysis before flattening to clauses inside an SMT solver. Preliminary experiments show that our technique scales to real-life programs and is competitive than other software verification techniques. Term rewriting based simplification may be viewed as a systematic preprocessing front-end to SMT-solvers, to improve handling of structured terms. Besides, simplification leads to concise summaries during program analysis. SPA can also benefit traditional symbolic execution by computing function summaries [5], [1] for the latter.

Our work is a step in the direction of tightly integrating static analysis methods with automated theorem proving [46], [31] as opposed to generating verification conditions to be discharged by a theorem prover at a later stage. Although we use rewriting techniques mainly for removing term redundancies currently, our framework can be extended to perform saturation-based theorem proving [39] to simplify terms with uninterpreted symbols, arrays, bitvectors, and recursive data structures (see, e.g., [34]). We also plan to make our technique modular by using function summaries [5], [6]. Another direction involves defining systematic strategies for application of rewrite rules [14]. We believe that our framework provides the means for exploiting theorem proving techniques focused towards the goal of program analysis [45], [46] by allowing a synergistic use of different theorem proving components, e.g., decision procedures, unification and rewriting.

Acknowledgements. We thank the anonymous reviewers

Benchmark		SE-DFS						SPA			
		-Rew,-IncSMT		+Rew,-IncSMT		+IncSMT,-Rew		-Rew		**+Rew**	
name	loc	T(s)	DP T(s)	T(s)	DP/Rew T(s)	T(s)	DP T(s)	T(s)	DP T(s)	T(s)	DP/Rew T(s)
tcas(p1)	550	TO	>1782	200	131/40	3	2	TO	> 1720	88	20/67
tcas(p2)	550	TO	>1782	200	131/40	3	2	TO	> 1720	88	20/67
tcas(p3)	550	104	103	3	2.4/0.5	0	0	TO	> 1720	27	7/19
tcas(p4)	550	112	112	3	2.4/0.5	0	0	TO	> 1720	34	10/24
tcas(p5)	550	TO	>1782	200	131/40	3	2	TO	> 1720	88	20/67
M (p1-30)	3500	1433	1389	313	214/65	9	5	TO	> 1761	15	8/6

Fig. 4. Comparison of analysis times of DFS-based symbolic execution, SE-DFS and our method SPA. +Rew = with rewriting, +IncSMT = with incremental SMT. T denotes total time for the check, DP T denotes total time spent calling the decision procedure, Rew T denotes total time spent rewriting. *loc* denotes lines of code after slicing. TO denotes timeout of 1800s. MO denotes memory out (limit 2GB).

for their comments and suggestions, and the members of the verification group at NEC for many useful discussions.

REFERENCES

[1] S. Anand, P. Godefroid, and N. Tillmann. Demand-driven compositional symbolic execution. In *TACAS*, pages 367–381, 2008.

[2] S. Anand, C. S. Păsăreanu, and W. Visser. JPF-SE: A symbolic execution extension to Java Pathfinder. In *TACAS*, pages 134–138, 2007.

[3] Tamarah Arons, Elad Elster, Shlomit Ozer, Jonathan Shalev, and Eli Singerman. Efficient symbolic simulation of low level software. In *DATE*, pages 825–830, 2008.

[4] F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1999.

[5] Domagoj Babic and Alan J. Hu. Structural abstraction of software verification conditions. In *CAV*, pages 366–378, 2007.

[6] Domagoj Babic and Alan J. Hu. Calysto: scalable and precise extended static checking. In *ICSE*, pages 211–220, 2008.

[7] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Yhu. Symbolic model checking without BDDs. In *TACAS*, volume 1579 of *LNCS*, pages 193–207. Springer-Verlag, March 1999.

[8] Stephen L. Bloom and Ralph Tindell. Varieties of "if-then-else". *SIAM Journal on Computing*, 12(4):677–707, 1983.

[9] R. Brummayer and A. Biere. Local two-level and-inverter graph minimization without blowup. In *MEMICS*, 2006.

[10] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Computers*, 35(8):677–691, 1986.

[11] J.R. Burch. Techniques for verifying superscalar microprocessors. *DAC*, pages 552–557, 1996.

[12] J. Burghardt. E-generalization using grammars. *Artif. Intell.*, 165(1), 2005.

[13] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. Exe: automatically generating inputs of death. In *ACM Conference on Computer and Communications Security*, pages 322–335, 2006.

[14] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn L. Talcott, editors. *All About Maude*, volume 4350 of *LNCS*. Springer, 2007.

[15] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252, 1977.

[16] Nachum Dershowitz and David A. Plaisted. Rewriting. In *Handbook of Automated Reasoning*, pages 535–610. 2001.

[17] Bruno Dutertre and Leonardo de Moura. A fast linear-arithmetic solver for DPLL(T). In *CAV*, pages 81–94, 2006.

[18] Steven Eker. Associative-commutative rewriting on large terms. In *RTA*, pages 14–29, 2003.

[19] Cormac Flanagan and James B. Saxe. Avoiding exponential explosion: generating compact verification conditions. In *POPL*, pages 193–205, 2001.

[20] Harald Ganzinger, George Hagen, Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. DPLL(T): Fast decision procedures. In *CAV*, pages 175–188, 2004.

[21] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: directed automated random testing. In *PLDI*, pages 213–223, 2005.

[22] Jan Friso Groote and Jaco van de Pol. Equational binary decision diagrams. In *LPAR*, pages 161–178, 2000.

[23] Irène Guessarian and José Meseguer. On the axiomatization of "if-then-else". *SIAM J. Comput.*, 16(2):332–357, 1987.

[24] Gerard Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley, 2003.

[25] Franjo Ivancic, Zijiang Yang, Malay K. Ganai, Aarti Gupta, Ilya Shlyakhter, and Pranav Ashar. F-soft: Software verification platform. In *CAV*, pages 301–306, 2005.

[26] J. R. Burch and D. L. Dill. Automatic verification of pipelined microprocessors control. In David L. Dill, editor, *CAV*, volume 818, pages 68–80, Standford, California, USA, 1994. Springer-Verlag.

[27] Sarfraz Khurshid, Corina S. Păsăreanu, and Willem Visser. Generalized symbolic execution for model checking and testing. In *TACAS*, pages 553–568, 2003.

[28] James C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.

[29] Shuvendu K. Lahiri and Randal E. Bryant. Indexed predicate discovery for unbounded system verification. In *CAV*, pages 135–147, 2004.

[30] K. Rustan M. Leino. Efficient weakest preconditions. *Inf. Process. Lett.*, 93(6):281–288, 2005.

[31] K. Rustan M. Leino and Francesco Logozzo. Loop invariants on demand. In *APLAS*, pages 119–134, 2005.

[32] Narciso Martí-Oliet and José Meseguer. Rewriting logic as a logical and semantic framework. *Electr. Notes Theor. Comput. Sci.*, 4, 1996.

[33] John McCarthy. A Basis for a Mathematical Theory of Computation. In P. Braffort and D. Hirschberg, editors, *Computer Programming and Formal Systems*, pages 33–70. North-Holland, Amsterdam, 1963.

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

[35] Alan H. Mekler and Evelyn M. Nelson. Equational bases for if-then-else. *SIAM J. Comput.*, 16(3):465–485, 1987.

[36] José Meseguer. Conditioned rewriting logic as a united model of concurrency. *Theor. Comput. Sci.*, 96(1):73–155, 1992.

[37] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[38] G. Nelson and D. C. Oppen. Simplification by cooperating decision procedures. *ACM Trans. Program. Lang. Syst.*, 1(2):245–257, 1979.

[39] Robert Nieuwenhuis and Albert Rubio. Paramodulation-based theorem proving. In *Handbook of Automated Reasoning*, pages 371–443. 2001.

[40] G. D. Plotkin. A note on inductive generalisation. In *Machine Intelligence 5*, pages 153–163. 1970.

[41] C. S. Păsăreanu and W. Visser. Verification of java programs using symbolic execution and invariant generation. In *SPIN*, pages 164–181, 2004.

[42] John C. Reynolds. Transformational systems and the algebraic structure of atomic formulas. In *Machine Intelligence 5*, pages 135–151. 1970.

[43] Alexander Schrijver. *Theory of linear and integer programming*. John Wiley & Sons, Inc., New York, NY, USA, 1986.

[44] Ravi Sethi. Conditional expressions with equality tests. *J. ACM*, 25(4):667–674, 1978.

[45] Natarajan Shankar. Little engines of proof. In *FME*, pages 1–20, 2002.

[46] Ashish Tiwari and Sumit Gulwani. Logical interpretation: Static program analysis using theorem proving. In *CADE*, pages 147–166, 2007.

Machine-code verification for multiple architectures

An application of decompilation into logic

Magnus O. Myreen, Michael J. C. Gordon
University of Cambridge
Computer Laboratory
William Gates Building, 15 JJ Thomson Avenue
Cambridge CB3 0FD, UK
Email: {magnus.myreen, mike.gordon}@cl.cam.ac.uk

Konrad Slind
University of Utah
School of Computing
50 South Central Campus Drive
Salt Lake City UT84112, USA
Email: slind@cs.utah.edu

***Abstract*— Realistic formal specifications of machine languages for commercial processors consist of thousands of lines of definitions. Current methods support trustworthy proofs of the correctness of programs for one such specification. However, these methods provide little or no support for reusing proofs of the same algorithm implemented in different machine languages. We describe an approach, based on proof-producing decompilation, which both makes machine-code verification tractable and supports proof reuse between different languages. We briefly present examples based on detailed models of machine code for ARM, PowerPC and x86. The theories and tools have been implemented in the HOL4 system.**

I. Introduction

This paper concerns verification of programs written in machine code of commercial processors whose semantics is accurately specified. Such processors support a large number of instructions and a multitude of features. Our aim is to be able to verify machine code:

A: without introducing simplifying assumptions, and

B: not requiring expert knowledge of the models, while still

C: allowing reuse of proofs between different architectures.

Current approaches struggle to address challenge C, as they either involve direct reasoning about the next-state function [1] or are based on annotating the code with assertions [2], [3]. Annotating the code with assertions inevitably ties the proof to the specific code and machine model as assertions are mixed with the code and depend on machine-specific resource names.

Our contribution is a method for addressing challenges A, B and C. Our approach adds a thin layer of abstraction to the verification process in order to make verification proofs tractable and reuseable. A fully automatic decompiler is presented, which translates machine code, via automatic deduction, into tail-recursive functions defined in the language of a theorem prover. Given a sequence of machine-code instructions, the decompiler derives a tail-recursive function and proves a theorem stating that the function accurately describes the effect of the given machine code (addresses challenge A). The user can concentrate on proving properties of the generated function, which hides irrelevant details of the underlying machine language specification (challenge B). Properties proved about the generated function are, via an automatically derived theorem, related to the execution of the original machine code. The function describes the executed low-level algorithm and is likely to be very similar (illustrated in Section II-C) to another function describing the same algorithm implemented in a different machine language and thus can facilitate proof reuse (challenge C). The decompiler and all examples, presented in this paper, have been implemented in the HOL4 theorem prover (our theories and tools are available from [4]).

Notation. We write program specifications as *Hoare triples* $\{p\}\, c\, \{q\}$; informal meaning: if p holds for the current state then code c will leave the process in a state satisfying q (formal definition given in Section III-B).

II. Example

This section shows how decompilation aids verification. Subsequent sections describe the decompilation algorithm.

A. Running the automation

Consider the following ARM machine code (and assembly code on the right) which calculates the length of a linked-list. The code sets register 0 to zero; it then compares register 1 (the list pointer) with zero (nil), the last three instructions execute conditionally based on the result of this comparison, if register 1 is not zero, then the last three instructions increment register 0, load register 1 from memory and jumps back to the compare instruction, otherwise they do nothing.

```
 0:  E3A00000         mov r0, #0
 4:  E3510000     L:  cmp r1, #0
 8:  12800001         addne r0, r0, #1
12:  15911000         ldrne r1, [r1]
16:  1AFFFFFB         bne L
```

Given the above list of hexadecimal numbers, our decompiler produces a function f describing the effect of the code.

$$f(r_0, r_1, m) = \mathsf{let}\ r_0 = 0\ \mathsf{in}\ g(r_0, r_1, m)$$

$$g(r_0, r_1, m) = \mathsf{if}\ r_1 = 0\ \mathsf{then}\ (r_0, r_1, m)\ \mathsf{else}\ \mathsf{let}\ r_0 = r_0{+}1\ \mathsf{in}\ \mathsf{let}\ r_1 = m(r_1)\ \mathsf{in}\ g(r_0, r_1, m)$$

978-1-4244-2735-2/08/$25.00 ©2008 IEEE

The decompiler also automatically proves the following theorem relating the execution of the ARM code with the function f (and an automatically generated precondition f_{pre}, given in Section IV-F). For now informally read the following Hoare-triple specification as (defined in Section III-B): given a state where register 0, register 1 and a part of memory is described by (r_0, r_1, m), the program counter is p and precondition f_{pre} holds, then executing the code will leave the processor in a state where register 0, register 1, a part of memory is described by $f(r_0, r_1, m)$ and the program counter is $p + 20$.

$$\begin{gathered}\{\ (\mathsf{r0}, \mathsf{r1}, \mathsf{m})\ \mathsf{is}\ (r_0, r_1, m) * \mathsf{pc}\ p * f_{pre}(r_0, r_1, m)\ \} \\ p : \texttt{E3A00000},\ p{+}4 : \texttt{E3510000}\ \ldots\ p{+}16 : \texttt{1AFFFFFB} \\ \{\ (\mathsf{r0}, \mathsf{r1}, \mathsf{m})\ \mathsf{is}\ f(r_0, r_1, m) * \mathsf{pc}\ (p + 20)\ \}\end{gathered}$$

The precondition $f_{pre}(r_0, r_1, m)$ states the side-conditions which must hold for f to execute properly.

B. Verifying the code

In order to verify that the above code computes the length of a linked-list, we need to formalise the statement "the memory holds a linked-list". Let $list(l, a, m)$ be a recursively-defined predicate which states that an abstract list of 32-bit words l, e.g. $l = [4, 5] = 4{::}5{::}\mathsf{nil}$ (we write '::' for list cons), is represented by a linked-list in memory m with its head at address a. Each element of the list is represented by a word for the next pointer and a word for the data. The words are positioned 4 bytes apart, hence "+4" below.

$$\begin{aligned} list(\mathsf{nil}, a, m) &= a = 0 \\ list(x{::}l, a, m) &= \exists a'.\ m(a) = a' \wedge m(a{+}4) = x \wedge a \neq 0 \wedge \\ &\qquad list(l, a', m) \wedge aligned(a) \end{aligned}$$

Let $length(l)$ be the length of an abstract list l, e.g. $length(4{::}5{::}\mathsf{nil}) = 2$. It is now easy (14 lines of HOL4) to prove (by induction on the abstract list l) that function f from above calculates the length of a linked list and also that $list$ implies the precondition f_{pre} (which implies that f and the machine code terminates).

$$\begin{aligned} &\forall x\, l\, a\, m.\ list(l, a, m) \Rightarrow f(x, a, m) = (length(l), 0, m) \\ &\forall x\, l\, a\, m.\ list(l, a, m) \Rightarrow f_{pre}(x, a, m) \end{aligned}$$

These lemmas about the generated function f can be given to a proof tactic, which automatically proves (using the theorem produced by the decompiler) that the original ARM code calculates $(length(l), 0, m)$, i.e. it proves a specification about the original code which does not involve the automatically generated function f or precondition f_{pre}:

$$\begin{gathered}\{\ (\mathsf{r0}, \mathsf{r1}, \mathsf{m})\ \mathsf{is}\ (r_0, r_1, m) * \mathsf{pc}\ p * list(l, r_1, m)\ \} \\ p : \texttt{E3A00000}\ \ldots\ p{+}16 : \texttt{1AFFFFFB} \\ \{\ (\mathsf{r0}, \mathsf{r1}, \mathsf{m})\ \mathsf{is}\ (length(l), 0, m) * \mathsf{pc}\ (p + 20)\ \}\end{gathered}$$

Hence by proving a property of the abstract function f we have proved a property about the ARM code.

C. Reusing the proof

An interesting aspect of our approach (addressing challenge C from above) is that it facilitates reuse of proofs, even between different architectures. To illustrate this point consider the following x86 code,

```
0:  31C0          xor eax, eax
2:  85F6      L1: test esi, esi
4:  7405          jz L2
6:  40            inc eax
7:  8B36          mov esi, [esi]
9:  EBF7          jmp L1
              L2:
```

and also the following PowerPC code for calculating the length of a linked-list.

```
 0:  38A00000        addi 5,0,0
 4:  2C140000    L1: cmpwi 20,0
 8:  40820010        bc 4,2,L2
12:  7E80A02E        lwzx 20,0(20)
16:  38A50001        addi 5,5,1
20:  4BFFFFF0        b L1
                L2:
```

Since the functional behaviour of all three code examples are essentially the same, the functions describing their behaviour are almost identical. f' is the function extracted for the x86 code and f'' is the same for PowerPC. We write '$\otimes$' for bitwise xor and '$\&$' for bitwise and.

$$f'(eax, esi, m) = \mathsf{let}\ eax = eax \otimes eax\ \mathsf{in}\ g'(eax, esi, m)$$

$$\begin{aligned} g'(eax, esi, m) = \mathsf{if}\ esi\,\&\,esi = 0\ \mathsf{then}\ (eax, esi, m)\ \mathsf{else} \\ \mathsf{let}\ eax = eax{+}1\ \mathsf{in} \\ \mathsf{let}\ esi = m(esi)\ \mathsf{in} \\ g'(eax, esi, m) \end{aligned}$$

$$f''(r_5, r_{20}, m) = \mathsf{let}\ r_5 = 0\ \mathsf{in}\ g''(r_5, r_{20}, m)$$

$$\begin{aligned} g''(r_5, r_{20}, m) = \mathsf{if}\ r_{20} = 0\ \mathsf{then}\ (r_5, r_{20}, m)\ \mathsf{else} \\ \mathsf{let}\ r_{20} = m(r_{20})\ \mathsf{in} \\ \mathsf{let}\ r_5 = r_5{+}1\ \mathsf{in} \\ g''(r_5, r_{20}, m) \end{aligned}$$

Minor differences, such as register names, conditional execution (ARM), variable instruction length (x86) and some instruction reordering (PowerPC example has load before increment), disappear in the functional description of the behaviour of the code. As a result the extracted functions can be proved equal by a short proof, in this case a three line HOL4 proof, using facts $w \otimes w = 0$ and $w\,\&\,w = w$.

$$f = f' = f'' \quad \text{and} \quad f_{pre} = f'_{pre} = f''_{pre}$$

Thus, any result proved for f and f_{pre} also describes the x86 and PowerPC implementations. By applying an automatic proof tactic we immediately obtain the same specification for the x86 code:

$$\begin{gathered}\{\ (\mathsf{eax}, \mathsf{esi}, \mathsf{m})\ \mathsf{is}\ (eax, esi, m) * \mathsf{eip}\ p * list(l, esi, m)\ \} \\ p : \texttt{31C0}\ \ldots\ p{+}9 : \texttt{EBF7} \\ \{\ (\mathsf{eax}, \mathsf{esi}, \mathsf{m})\ \mathsf{is}\ (length(l), 0, m) * \mathsf{eip}\ (p + 11)\ \}\end{gathered}$$

and similarly for the PowerPC code:

$$\{\ (\mathsf{r5}, \mathsf{r20}, \mathsf{m})\ \mathsf{is}\ (r_5, r_{20}, m) * \mathsf{pc}\ p * list(l, r_{20}, m)\ \}$$
$$p : \texttt{38A00000}\ \ldots\ p{+}20 : \texttt{4BFFFFF0}$$
$$\{\ (\mathsf{r5}, \mathsf{r20}, \mathsf{m})\ \mathsf{is}\ (length(l), 0, m) * \mathsf{pc}\ (p + 24)\ \}$$

The decompiler automates the machine-specific proofs and delivers completely automatically to the user a recursive function describing the code. The generated functions are sufficiently abstract to be reusable while at the same time have a strong connection with the original machine code enabling properties of the function to carry over to properties proved of the machine code.

D. Larger examples

Our decompilation technique has been applied to a number of verification examples. The most significant are the verification of two copying garbage collectors (variants of the Cheney garbage collector [5]). The largest examples consist of approximately one hundred machine instructions.

Similar Cheney collectors have been verified by Birkedal et al. [6] (on paper) and McCreight et al. [7] (using Coq). The proofs by McCreight et al. (7775 lines) are much longer than our (approximately 2000-line) proofs, which suggests that decompilation aided our verification effort. Our decompiler is currently implemented in 1123 lines of ML with approximately 2300 lines of supporting proof scripts, and works on top of models of x86, ARM and PowerPC whose HOL4 definitions total more than 10000 lines.

III. Preliminaries

Before presenting the algorithm for decompilation into logic (next section), we overview the models used in our examples, the Hoare triples, and our approach to proving loops.

A. Mechanised models of x86, ARM and PowerPC

Our current implementation supports x86, ARM and PowerPC machine code. The underlying operational models of the machine languages are detailed descriptions of a next-state function $next : state \rightarrow state$, which executes one instruction for each next application.

The machine language models were not developed for use with our tool. The ARM model is a specification of ARMv4 written by Fox [8], which he proved correct with respect to a register-transfer-level specification of an ARM processor (ARM6). The PowerPC model is a translation (manual translation of Coq to HOL4) of Leroy's PowerPC specification, which he used in a proof of an optimising C compiler [9]; we attached an instruction decoder to it in order to transform his assembly level model into a machine code model of PowerPC. The x86 specification is a functional HOL4 version of Sarkar's x86 specification [10], developed originally in Twelf for use in a applications of proof-carrying code. All the models we base this work on are available with our HOL4 proof scripts [4].

B. Hoare triple specifications

This section defines, in higher-order logic, the Hoare triples we use to make specifications concise and manageable. These are a stream-lined version of previously presented Hoare triples for machine code [11], [12].

The Hoare triples view states as sets of state elements. In order to accommodate this view, we define translations from the states used by the models to sets of state elements, e.g. for PowerPC we define the type ppc_elem of state elements with the following type constructors:

$$\mathsf{pReg} : \mathsf{ppc\_reg\_name} \times \mathsf{word32} \rightarrow \mathsf{ppc\_elem}$$
$$\mathsf{pMem} : \mathsf{word32} \times \mathsf{word8\ option} \rightarrow \mathsf{ppc\_elem}$$
$$\mathsf{pStatus} : \mathsf{ppc\_bit\_name} \times \mathsf{boolean\ option} \rightarrow \mathsf{ppc\_elem}$$

and define a translation function $ppc2set$, which translates states from the processor model's format to states in the set-based representation. The following is an example of the output from $ppc2set$: (GPR = general purpose register)

$$\{\ \mathsf{pReg}\ (\mathsf{GPR}\ 0)\ 5, \mathsf{pReg}\ (\mathsf{GPR}\ 1)\ 56, \mathsf{pReg}\ (\mathsf{GPR}\ 2)\ 89, \ldots,$$
$$\mathsf{pMem}\ 0\ \mathsf{none}, \mathsf{pMem}\ 1\ (\mathsf{some}\ 67), \mathsf{pMem}\ 2\ (\mathsf{some}\ 255), \ldots,$$
$$\mathsf{pStatus}\ (\mathsf{CPR0}\ 0)\ (\mathsf{some}\ true), \mathsf{pStatus}\ (\mathsf{CPR0}\ 1)\ \mathsf{none}, \ldots\ \}$$

The Hoare triple uses a separating conjunction $*$ inspired by separation logic. Separation logic defines its $*$ over partial functions; we define ours over sets:

$$(p * q)\ s = \exists u\, v.\ p\ u \wedge q\ v \wedge (u \cup v = s) \wedge (u \cap v = \{\})$$

The separating conjunction splits the state into two disjoint parts. Basic assertions access only part of the state, e.g. asserting that GPR 0 has value x is defined as:

$$(\mathsf{r0}\ x)\ s = (s = \{\mathsf{pReg}\ (\mathsf{GPR}\ 0)\ x\})$$

Basic assertions together with the $*$-operator consume a part of the state, e.g. $(\mathsf{r0}\ x{*}res)\ (ppc2set(s))$ is false if res makes an assertion about the value of GPR 0.

Let $run(n, s)$ be a function which applies the $next$-function n-times to s, i.e.

$$run(0, s) = s$$
$$run(n{+}1, s) = run(n, next(s))$$

The Hoare triple is defined to assert: for any state s for which a portion satisfies p and a separate portion satisfies a code assertion (defined, together with other memory assertion, in Section IV-K), there exists some number of $next$ applications which will take the processor to a state where q satisfies a portion of the state separate from the code.

$$\{p\}\ c\ \{q\}\ =$$
$$\forall s\ r.\ (p * \mathsf{code}\ c * r)\ (ppc2set(s))\ \Rightarrow$$
$$\exists k.\ (q * \mathsf{code}\ c * r)\ (ppc2set(run(k, s)))$$

Thus $\{p\}\ c\ \{q\}$ is a total-correctness specification.

The frame rule can be derived from this definition: it allows any assertion for a disjoint portion of the state to be added to the specification:

$$\{p\}\ c\ \{q\}\ \Rightarrow\ \forall r.\ \{p * r\}\ c\ \{q * r\}$$

Other important rules are composition, which combines two specifications and takes the set-union of their code elements:

$$\{p\}\ c_1\ \{m\}\ \wedge\ \{m\}\ c_2\ \{q\}\ \Rightarrow\ \{p\}\ c_1 \cup c_2\ \{q\}$$

and a rule for moving pure assertions g (an assertion g is 'pure' if it consumes no resources, i.e. $\forall p\ s.\ (p * g)\ s = p\ s \wedge g$) out of the precondition:

$$\{p * (g)\}\ c\ \{q\}\ =\ (g \Rightarrow \{p\}\ c\ \{q\})$$

Note that these rules are just theorems of higher-order logic proved from the definitions of Hoare triples and separating conjunction given above.

C. Proving loops

When proving specifications for code with loops our decompilation algorithm instantiates a special loop-rule for tail-recursive functions. This rule assumes the existence of a termination proof for the tail-recursive function and uses the induction arising from the termination proof in proving the specification for the code implementing the tail-recursion.

Decompilation only generates tail-recursive functions, i.e. functions $tailrec$ with instantiations of G, F and D, where:

$$tailrec(x)\ =\ \mathsf{if}\ G(x)\ \mathsf{then}\ tailrec(F(x))\ \mathsf{else}\ D(x)$$

$tailrec$ can be defined directly (without a termination proof in HOL4) using a trick by Manolios and Moore [13].

However, the decompiler requires $tailrec$ to terminate for certain inputs. Tail-recursions defined by $tailrec$ terminate for input x if and only if some number of applications of F to x make G false, i.e. $\exists n.\ \neg G(pow(n, F, x))$ with pow as:

$$\begin{aligned} pow(0, F, x)\ &=\ x \\ pow(n{+}1, F, x)\ &=\ pow(n, F, F(x)) \end{aligned}$$

We define $pre(x)$ to state that $tailrec(x)$ terminates and also that side-condition P is true for each call to $tailrec$ (Section IV-E illustrates an instantiation of P).

$$\begin{aligned} pre(x)\ =\ &\exists n.\ \neg\, G\,(pow(n, F, x)) \wedge \\ &\forall k.\ (\forall m.\ m < k \Rightarrow G\,(pow(m, F, x))) \Rightarrow \\ &\quad P\,(pow(k, F, x)) \end{aligned}$$

pre satisfies two desirable properties: pre can be unrolled by a rewrite (particularly useful in proofs by induction):

$$pre(x)\ =\ P(x) \wedge (G(x) \Rightarrow pre(F(x)))$$

and the following induction can be derived from its definition:

$$\begin{aligned} \forall \varphi.\ &(\forall x.\ pre(x) \wedge G(x) \wedge \varphi(F(x)) \Rightarrow \varphi(x)) \wedge \\ &(\forall x.\ pre(x) \wedge \neg G(x) \Rightarrow \varphi(x)) \Rightarrow \\ &(\forall x.\ pre(x) \Rightarrow \varphi(x)) \end{aligned}$$

This induction rule leads to the following loop rule, which the decompiler instantiates whenever it encounters a loop (an example instantiation is given in Section IV-E).

$$\begin{aligned} \forall \mathsf{res}\ \mathsf{res'}\ c.\ &(\forall x.\ P(x) \wedge G(x) \Rightarrow \{\mathsf{res}\ x\}\ c\ \{\mathsf{res}\ \ F(x)\}) \wedge \\ &(\forall x.\ P(x) \wedge \neg G(x) \Rightarrow \{\mathsf{res}\ x\}\ c\ \{\mathsf{res'}\ D(x)\}) \Rightarrow \\ &(\forall x.\ pre(x) \Rightarrow \{\mathsf{res}\ x\}\ c\ \{\mathsf{res'}\ tailrec(x)\}) \end{aligned}$$

For the proof of this rule, instantiate φ in the induction principle above with $\lambda x.\ \{\mathsf{res}\ x\}\ c\ \{\mathsf{res'}\ tailrec(x)\}$ and use the composition rule and the inductive hypothesis together with $c \cup c = c$ and $G(x) \Rightarrow tailrec(F(x)) = tailrec(x)$.

IV. Algorithm

This section outlines our algorithm for decompiling machine code into logic. There are six steps:

1) calculate the behaviour of each individual instruction;
2) prove a specification for each instruction;
3) discover the control flow by analysing the specifications;
4) split the code according to the control-flow graph;
5) for each code segment:
   a) derive a specification for one pass through the code,
   b) generate a function describing the code;
   c) for loops, instantiate a loop rule.
6) compose the top-level specifications and repeat step 5 until all of the code is described by one specification.

The following subsections explain these steps when applied to the linked-list example from Section II. Later subsections describe support for procedure calls as well as non-nested loops. Restrictions of our approach are outlined in Section V.

A. Behaviour of instructions

As a first step, each instruction's effect on the underlying machine-language model is evaluated. We use standard techniques from symbolic simulation to construct statements about the next-state function. As an example: the x86 instruction `40`, which is the hexadecimal encoding of `inc eax` (increment the EAX register), produces the following theorem. Here eip is the instruction pointer, a.k.a. program counter, and AF, SF, ZF etc. are status bits called "eflags". Here and throughout option-types[1] are used for values that may hold unpredictable or unmodelled values, e.g. the theorem shows that eflag OF gets an unmodelled value, while eflag ZF is assigned the boolean result of the comparison $eax + 1 = 0$.

$$\begin{aligned} &x86\_read\_reg\ \mathsf{EAX}\ s = eax\ \wedge \\ &x86\_read\_eip\ s = eip\ \wedge \\ &x86\_read\_mem\ eip\ s = \mathsf{some}\ \mathsf{0x40}\ \Rightarrow \\ &\quad x86\_next\ s = \\ &\qquad \mathsf{some}\ (x86\_write\_reg\ \mathsf{EAX}\ (eax + 1) \\ &\qquad\quad (x86\_write\_eip\ (eip + 1) \\ &\qquad\quad (x86\_write\_eflag\ \mathsf{AF}\ \mathsf{none} \\ &\qquad\quad (x86\_write\_eflag\ \mathsf{SF}\ (\mathsf{some}\ (sign\_of(eax + 1))) \\ &\qquad\quad (x86\_write\_eflag\ \mathsf{ZF}\ (\mathsf{some}\ (eax + 1 = 0)) \\ &\qquad\quad (x86\_write\_eflag\ \mathsf{PF}\ (\mathsf{some}\ (parity(eax + 1))) \\ &\qquad\quad (x86\_write\_eflag\ \mathsf{OF}\ \mathsf{none}\ s))))))) \end{aligned}$$

[1] something of type option is either 'some x', meaning 'has value x', or 'none' meaning 'has an unmodelled or unpredictable value'.

B. Instruction specifications

As a second step we derive Hoare-triple specifications for each instruction in the given program, e.g. the move-instruction from the ARM code for length-of-linked-list has the following specification:

$$\{\, \mathsf{r0}\ r_0 * \mathsf{pc}\ p \,\}\ \ p : \texttt{0000A0E3}\ \ \{\, \mathsf{r0}\ 0 * \mathsf{pc}\ (p{+}4) \,\}$$

Two specifications are produced for instructions that execute conditionally, e.g. the ARM instruction for branch-if-not-equal: (sz asserts the value of status bit 'z')

$$\{\, \mathsf{sz}\ z * \mathsf{pc}\ (p{+}16) * \neg z \,\}\ \ p : \texttt{FBFFFF1A}\ \ \{\, \mathsf{sz}\ z * \mathsf{pc}\ (p{+}4) \,\}$$

$$\{\, \mathsf{sz}\ z * \mathsf{pc}\ (p{+}16) * z \,\}\ \ p : \texttt{FBFFFF1A}\ \ \{\, \mathsf{sz}\ z * \mathsf{pc}\ (p{+}20) \,\}$$

C. Control-flow discovery

A heuristic reads the pc assertions in the postconditions and builds a summary of how control can flow. The linked-list example results in the following description:

$$0 \rightarrow 4,\ 4 \rightarrow 8,\ 8 \rightarrow 12,\ 12 \rightarrow 16,\ 16 \rightarrow 20,\ 16 \rightarrow 4$$

The heuristic searches for loops by analysing this graph. It finds that instructions $4, 8, 12, 16$ constitute a loop.

D. Finding the function

Once loops have been detected in the control-flow graph, we start by proving a specification for the inner-most loop. We compose specifications for individual instructions in order to get a specification for one pass of execution through the code. Composing the specifications for the loop in the ARM code of the linked-list example results in two specifications[2][3]; one for the case $r_1 = 0$:

$$\{\, \mathsf{r0}\ r_0 * \mathsf{r1}\ r_1 * \mathsf{m}\ m * \mathsf{pc}\ (p{+}4) \,\}$$
$$\textit{... the arm code ...}$$
$$\{\, \mathsf{r0}\ r_0 * \mathsf{r1}\ r_1 * \mathsf{m}\ m * \mathsf{pc}\ (p{+}20) \,\}$$

and one for the case $r_1 \neq 0$, if $r_1 \in dom(m) \wedge aligned(r_1)$:

$$\{\, \mathsf{r0}\ r_0 * \mathsf{r1}\ r_1 * \mathsf{m}\ m * \mathsf{pc}\ (p{+}4) \,\}$$
$$\textit{... the arm code ...}$$
$$\{\, \mathsf{r0}\ (r_0{+}1) * \mathsf{r1}\ (m(r_1)) * \mathsf{m}\ m * \mathsf{pc}\ (p{+}4) \,\}$$

Notice that the program counter is returned to $p{+}4$ in case $r_1 \neq 0$, indicating that the function describing the code is to loop when $r_1 \neq 0$. The generated function is constructed to mimic the effect of the code:

$$g(r_0, r_1, m) = \mathsf{if}\ r_1 = 0\ \mathsf{then}\ (r_0, r_1, m)\ \mathsf{else}\ g(r_0{+}1, m(r_1), m)$$

[2]Our implementation inserts let-expressions at this stage but we avoid them in our illustrations in order to reduce the size of expressions.

[3]To be completely accurate, from this point onwards all pre- and postconditions in this paper should end in "... * s}". Here s existentially quantifies the values of the status bits, i.e. the Hoare triples abstract the values of the status bits: they state "... and the status bits have some value before and after execution". One can turn off this abstraction and keep track of the exact value of the status bits just as for any other resource.

E. Proving the specification

The generated function is always a tail-recursion (if recursive at all). This means that the function can be defined as an instance of $tailrec$ from Section III-C. Function g from above is defined as $tailrec$ with F, G and D as:

$$G = \lambda(r_0, r_1, m).\ r_1 \neq 0$$
$$F = \lambda(r_0, r_1, m).\ (r_0{+}1, m(r_1), m)$$
$$D = \lambda(r_0, r_1, m).\ (r_0, r_1, m)$$

and the precondition g_{pre} is defined as pre (also from Section III-C) with the same instantiations, and parameter P as:

$$P = \lambda(r_0, r_1, m).\ r_1 \neq 0 \Rightarrow r_1 \in dom(m) \wedge aligned(r_1)$$

P is defined as such in order for $g_{pre}(r_0, r_1, m)$ to imply the side condition appearing in case $r_1 \neq 0$ of Section IV-D.

Let resource assertions res and res' be:

$$\mathsf{res} = \lambda(r_0, r_1, m).\ (\mathsf{r0}, \mathsf{r1}, \mathsf{m})\ \mathsf{is}\ (r_0, r_1, m) * \mathsf{pc}\ (p{+}4)$$
$$\mathsf{res'} = \lambda(r_0, r_1, m).\ (\mathsf{r0}, \mathsf{r1}, \mathsf{m})\ \mathsf{is}\ (r_0, r_1, m) * \mathsf{pc}\ (p{+}20)$$

With these instantiations the conclusion of the loop-rule from Section III-C is exactly the desired result for the loop:

$$\{\, (\mathsf{r0}, \mathsf{r1}, \mathsf{m})\ \mathsf{is}\ (r_0, r_1, m) * \mathsf{pc}\ (p{+}4) * g_{pre}(r_0, r_1, m) \,\}$$
$$\textit{... the arm code ...}$$
$$\{\, (\mathsf{r0}, \mathsf{r1}, \mathsf{m})\ \mathsf{is}\ g(r_0, r_1, m) * \mathsf{pc}\ (p{+}20) \,\}$$

The premises of the loop-rule are trivial consequences (by simple rewriting in HOL4) of the theorems describing one pass through the code, given in Section IV-D.

F. Merging cases recursively

Sections IV-D and IV-E showed how tail-recursive functions $tailrec$ and specifications of the form

$$\{\mathsf{res}\ x * \mathsf{pc}\ (...) * pre(x)\}\ c\ \{\mathsf{res'}\ tailrec(x) * \mathsf{pc}\ (...)\}$$

can be constructed and proved for code with at most one top-level loop, given specifications for each individual instruction of the code c, obtained in Sections IV-A and IV-B.

Specification for nested loops and code around loops can be proved by recursively repeating the above procedure for code enclosing the inner loops. For the linked-list example the decompilation algorithm will repeat Sections IV-D and IV-E based on the specification of the move-instruction (at the top of Section IV-B) and the above specification proved for g (at the end of Section IV-E), which then defines f and proves the theorem shown in Section II-A. The generated f_{pre} is defined as an instance of pre, but returned as the following theorem.

$$\begin{aligned} f_{pre}(r_0, r_1, m) &= g_{pre}(0, r_1, m) \\ g_{pre}(r_0, r_1, m) &= (r_1 \neq 0) \Rightarrow \\ &\qquad r_1 \in dom(m) \wedge aligned(r_1) \wedge \\ &\qquad g_{pre}(r_0{+}1, m(r_1), m) \end{aligned}$$

G. Non-nested loops

The examples above have considered machine-code programs that start executing at the top of the code and exit at the end of the code, with all intermediate loops properly nested. More general forms of control flow are handled by treating the program counter as any other resource, i.e. the program counter becomes part of the function '$(..., \mathsf{pc})$ is $f(...)$', just as any other register value. In such cases, the position q of the code needs to be passed in to the generated function f. As an example, when the following non-nested loops are processed

```
 0:  E2800001   L: add  r0,r0,#1
 4:  E3100001   M: tst  r0,#1
 8:  1AFFFFFC      bne  L  ;; may goto L
12:  E2500002      subs r0,r0,#2
16:  1AFFFFFB      bne  M  ;; may goto M
```

the generated function compares the value of the program counter p with the position of the code q:

$$
\begin{array}{l}
f(r_0, p, q) = \\
\quad \mathsf{if}\ p = q\ \mathsf{then} \\
\quad\quad \mathsf{let}\ r_0 = r_0 + 1\ \mathsf{in}\ f(r_0, q{+}4, q) \\
\quad \mathsf{else\ if}\ r_0\ \&\ 1 \neq 0\ \mathsf{then} \\
\quad\quad f(r_0, q, q) \\
\quad \mathsf{else} \\
\quad\quad \mathsf{let}\ r_0 = r_0 - 2\ \mathsf{in} \\
\quad\quad\quad \mathsf{if}\ r_0 = 0\ \mathsf{then}\ (r_0, q{+}5)\ \mathsf{else}\ f(r_0, q{+}4, q)
\end{array}
$$

The resulting theorem is:

$$
\begin{array}{c}
\{\ (\mathsf{r0}, \mathsf{pc})\ \mathsf{is}\ (r_0, p)\ *\ p \in \{q, q{+}4\}\ \} \\
q : ...\, code\, ... \\
\{\ (\mathsf{r0}, \mathsf{pc})\ \mathsf{is}\ f(r_0, p, q)\ \}
\end{array}
$$

The decompiler instantiates q with p before using the above Hoare triple specification in subsequent proofs.

H. Procedure calls

Procedure calls are, in machine code, implemented using branch-and-link instructions. These branch instructions perform a normal branch and at the same time save a return address, e.g. on ARM the branch-and-link instruction stores the return address in register 14:

$$\{\mathsf{r14}\ x * \mathsf{pc}\ p\}\ p : \texttt{EB000009}\ \{\mathsf{r14}\ (p{+}4) * \mathsf{pc}\ (p{+}48)\}$$

Given the following specification for the procedure's code,

$$
\begin{array}{c}
\{\ (\mathsf{pc}, \mathsf{r14}, \mathsf{res})\ \mathsf{is}\ (p, r_{14}, x) * t_{pre}(p, r_{14}, x)\ \} \\
p : procedure\_code \\
\{\ (\mathsf{pc}, \mathsf{r14}, \mathsf{res})\ \mathsf{is}\ t(p, r_{14}, x)\ \}
\end{array}
$$

we can compose the call with the procedure's specification,

$$
\begin{array}{c}
\{\ (\mathsf{pc}, \mathsf{r14}, \mathsf{res})\ \mathsf{is}\ (p, r_{14}, x) * t_{pre}(p{+}48, p{+}4, x)\ \} \\
p : \texttt{EB000009}\ \cup\ p{+}48 : procedure\_code \\
\{\ (\mathsf{pc}, \mathsf{r14}, \mathsf{res})\ \mathsf{is}\ t(p{+}48, p{+}4, x)\ \}
\end{array}
$$

and by strengthening the precondition t_{pre} to assume that control returns to the callee, let $\mathsf{fst}(x, y) = x$,

$$t'_{pre}(p, q, x) = t_{pre}(p, q, x) \wedge (\mathsf{fst}(t(p, q, x)) = q)$$

we have a specification for the procedure which has the shape of a normal instruction specification (enters at $\mathsf{pc}\ p$ and exits at $\mathsf{pc}\ (p{+}4)$). Thus specifications for procedure calls can be derived from the specification of the called procedure. The function generated by the decompiler includes a reference to function t generated for the procedures body.

$$\mathsf{let}\ (\_, r_{14}, x) = t(p{+}48, p{+}4, x)\ \mathsf{in}\ ...$$

Procedural recursion poses a challenge as an induction is required. In principle, it is possible to support procedural induction by regarding the program counter as any other resource, as was done in the previous section. However, the generated function is far less intuitive using that technique.

I. Support for user-defined resource assertions

Notice that the operations of the decompiler do not depend on the particular properties of the basic resource assertions (r0, r1, m etc.). As a result, specifications involving completely different, user-defined, assertions can be fed into the decompiler for use instead of automatically proved instruction specifications.

As an example consider this Hoare triple describing the `alloc` routine of one the garbage collectors we have verified using decompilation (see Section II-D). Here heap is predicate stating that a garbage collected heap is present in memory. The alloc function's precondition states that the number of (#) reachable elements in the abstract heap h from roots v_1, v_2, v_3, v_4 must be less than the heap limit l. The post condition state that the abstract heap modelling function h is updated with a new element *fresh* h, which points at a cons cell containing (v_1, v_2). The address of the new element is stored in the place of root variable v_1.[4]

$$
\begin{array}{l}
\{\ \mathsf{pc}\ p * \#reachable(v_1, v_2, v_3, v_4, h) < l\ * \\
\quad \mathsf{heap}\ (a, v_1, v_2, v_3, v_4, h, l)\ \} \\
\quad\quad ...\ collector\ code\ ... \\
\{\ \mathsf{pc}\ (p{+}332)\ * \\
\quad \mathsf{heap}\ (a, \textit{fresh}\ h, v_2, v_3, v_4, h[\textit{fresh}\ h \mapsto (v_1, v_2)], l)\ \}
\end{array}
$$

When such specifications can be given as input to the decompiler, in our implementation using a special keyword '`insert` ...' in the given code, the decompiler can look-up and use this specification. The resulting generated function contains the roots v_1, v_2, v_3, v_4 and heap h as variables:

$$\mathsf{let}\ (v_1, h) = (\textit{fresh}\ h,\ h[\textit{fresh}\ h \mapsto (v_1, v_2)])\ \mathsf{in}\ ...$$

the theorem contains '$(\mathsf{heap}, \mathsf{r0}, ...)$ is ...', and the generated precondition will keep track of a sufficient condition under which the heap limit is not exceeded.

J. Code and memory assertions

The code and memory assertions are defined in this section. The code assertion for ARM is the simplest one: it states that

[4]Details of this specification for `alloc` and its proof will be part of the first author's forth coming PhD thesis.

the memory hold a set of (p, w) instructions, where p is the address and w is the 32-bit word (the instruction).

$$\mathsf{code}\ set\ s\ =\ (s = \{\ \mathsf{aMem}\ p\ (\mathsf{some}\ w)\ |\ (p, w) \in set\ \})$$

The memory assertion $\mathsf{m}\ m$ states that memory location a has value $m(a)$ if $a \in dom(m)$ and a is word-aligned (two least significant bits are zero, i.e. $a\,\&\,3 = 0$).

$$\mathsf{m}\ m\ =\ \mathsf{code}\ \{\ (a, m(a))\ |\ a \in dom(m) \wedge a\,\&\,3 = 0\ \}$$

The code assertions for PowerPC and x86 are slightly more complicated due to the fact that their set representation (and the underlying model) considers the memory as byte-addressed, i.e. a 32-bit word consists of four bytes. The code assertion for PowerPC (which is a big-endian architecture):

$$\begin{aligned} word(p, w) = \{\ &\mathsf{pMem}\ (p{+}0)\ (\mathsf{some}\ (w[31{-}24])),\\ &\mathsf{pMem}\ (p{+}1)\ (\mathsf{some}\ (w[23{-}16])),\\ &\mathsf{pMem}\ (p{+}2)\ (\mathsf{some}\ (w[15{-}08])),\\ &\mathsf{pMem}\ (p{+}3)\ (\mathsf{some}\ (w[07{-}00]))\ \} \end{aligned}$$

$$\mathsf{code}\ set\ s\ =\ (s = \bigcup\{\ word(p, w)\ |\ (p, w) \in set\ \})$$

The definition of m for PowerPC uses code just as ARM.

The code assertion for x86 is defined recursively, since x86 instructions are lists of bytes:

$$\begin{aligned} x86\_list(p, []) &= \{\}\\ x86\_list(p, c{::}cs) &= \{\ \mathsf{xMem}\ p\ (\mathsf{some}\ c)\ \} \cup ia\_list(p{+}1, cs) \end{aligned}$$

$$\mathsf{code}\ set\ s\ =\ (s = \bigcup\{\ ia\_list(p, cs)\ |\ (p, cs) \in set\ \})$$

x86's memory assertion takes into account that the architecture is little-endian:

$$\begin{aligned} word(p, w) = \{\ &\mathsf{xMem}\ (p{+}0)\ (\mathsf{some}\ (w[07{-}00])),\\ &\mathsf{xMem}\ (p{+}1)\ (\mathsf{some}\ (w[15{-}08])),\\ &\mathsf{xMem}\ (p{+}2)\ (\mathsf{some}\ (w[23{-}16])),\\ &\mathsf{xMem}\ (p{+}3)\ (\mathsf{some}\ (w[31{-}24]))\ \} \end{aligned}$$

$$\begin{aligned} \mathsf{m}\ m\ s\ =\ (s = \bigcup\{\ word(a, m(a))\ |\ a \in dom(m) \wedge\\ a\,\&\,3 = 0\ \ \}) \end{aligned}$$

These assertions act as a thin layer of additional abstraction.

K. Memory separation

The examples presented so far have only used a single memory assertion $\mathsf{m}\ m$ at a time. However, it is often useful to separate memory into logical segments, e.g. one for the stack s and one for the heap h:

$$\{\mathsf{m}\ s * \mathsf{m}\ h * ...\}\ p : ...\ \{...\}$$

Notice that this specification implicitly assumes that s and h describe disjoint parts of memory, since '$*$' makes assertions 'consume' memory (Section III-B). The functions produced by the decompiler will then use two memory modelling functions s and h, and most importantly an update to the stack s will not affect the heap h (and *vice versa*). This feature is used heavily in some of the garbage collector proofs in order to avoid some proof obligations that arise from possible pointer aliasing between the stack and the heap (in case the stack and the heap happened to overlap, a case we rule out).

Memory separation can easily be implemented by modifying the output from the routines that derive Hoare triple specifications (Section IV-B). A heuristic is fed in to that routine which renames memory modelling functions depending on the registers that access them, e.g. our default heuristic renames memory modelling functions to s, if the stack pointer is used, while all other accesses are to memory called m.

V. Restrictions

Our method is completely automatic and is reasonably light-weight to implement. Restrictions of our approach are discussed below.

A. Deterministic behaviour required

The method is only applicable to programs that have deterministic behaviour, for otherwise the code is not a *function* of its inputs and the decompiler could not produce a function describing the code.

B. Heuristics used for control flow discovery

The decompiler uses heuristics to discover the possible execution paths in the code. The heuristics work well for code where all branches are made to offsets of the current program counter. Branch-and-link instructions are considered to be procedure calls and any instruction moving an address into the program counter from a register or stack location is assumed to perform a procedure return. As a result, our simple heuristic is easily confused by computed branches and calls to code pointers.

VI. Discussion of Related Work

Different techniques for program verification, with respect to accurate models of machine code, are discussed below.

Symbolic simulation is a technique applicable to machine code modelled by an operational semantics. The approach is based on executing the next-state function on states where registers and memory location have been assigned symbolic values (logical variables, e.g. x, y); the result is a new state where resources hold expressions (e.g. $x{+}y$), for which the verifier is to prove properties. This method is emphasised and successfully applied by the ACL2 community [14], [15], e.g. Boyer and Yu used symbolic simulation in pioneering work on verification of the GNU string library compiled by GCC for the Motorola MC68020 [1], and similar techniques were used by Liu and Moore in proofs of Java bytecode programs with respect to an extensive model of the Java Virtual Machine (JVM) [16]. Symbolic simulation has the disadvantages that the expressions produced by simulation, directly on top of the operation semantics, can become fiendishly complex, and loops reqiure user interaction. However, Currie et al. [17] have developed automatic tools, based on symbolic simulation, which prove equivalence between snippets of machine code for embedded devices.

Using a programming logic directly on top of the definition of the semantics of the machine code is an approach which lends itself well to reasoning about loops and control flow that is problematic for symbolic simulation. Shao's group at Yale [18] have used programming logics (inspired by separation logic and rely-guarantee) in verification of (slightly idealised) assembly programs. Foundational proof checking [19] and Typed Assembly Language [20], [21] also belong to this category. However, they aim to check relatively weak safety properties – while this paper's techniques are concerned with proving complete functional correctness.

Using *verification condition generators (VCGs)* one annotates the code with assertions for which the VCGs calculates verification conditions that imply consistency of the assertions with respect to some programming logic. The integrity of the VCG is a concern, as practical VCGs tend to be complex [22], [23]. However, Homeier and Martin [24] showed that VCGs can be verified and Matthews et al. [3] has showed that off-the-shelf theorem provers can be used in a way which gives the benefits of a VCG without actually constructing a full VCG. Hardin et al. [2] have applied the technique described by Matthews et al. to machine code of Rockwell Collins AAMP7G. The main disadvantage of annotating the code with assertions is that the assertions become tied to the specific machine language and/or the particular definition of the semantics and, thus, do not provide the appropriate abstractions required for proof reuse.

Decompilation automatically reverse-engineers an abstraction of machine code. Decompilation is most often used to reverse compilation from a language such as C [25], but can, as we have shown in this paper, be used to produce abstractions in higher-order logic – a language much more amenable to formal reasoning than C. There is generally little work in this area, but work by Filliâtre [26] and Katsumata and Ohori [27] is related to ours. Filliâtre shows how imperative loops can, in type theory, be turned into recursive functions for purposes of verification. Unlike our approach his requires the code to be annotated with invariants and does not apply the method to low-level languages. Katsumata and Ohori have developed a decompiler, from a small subset of idealised Java bytecode to recursive functions, based on ideas from type theory. The decompiler implementing their methodology has not been verified. It seems that the decompiler would need to be trusted, if its output were to be used in verification. In contrast our approach is to produce a proof for each run, hence the decompiler need not be trusted.

ACKNOWLEDGMENTS

We would like to thank Thomas Tuerk, Anthony Fox, Boris Feigin, Max Bolingbroke and John Regehr for research discussions. The first author is grateful for funding from Osk. Huttusen säätiö, Finland, and EPSRC, UK.

REFERENCES

[1] R. S. Boyer and Y. Yu, "Automated proofs of object code for a widely used microprocessor," *J. ACM*, vol. 43, no. 1, pp. 166–192, 1996.

[2] D. S. Hardin, E. W. Smith, and W. D. Young, "A robust machine code proof framework for highly secure applications," in *Proceedings of the Sixth International Workshop on the ACL2 Theorem Prover and Its Applications*, P. Manolios and M. Wilding, Eds., 2006.

[3] J. Matthews, J. S. Moore, S. Ray, and D. Vroon, "Verification condition generation via theorem proving," in *Logic for Programming Artificial Intelligence and Reasoning (LPAR)*, ser. LNCS, M. Hermann and A. Voronkov, Eds., vol. 4246. Springer, 2006, pp. 362–376.

[4] Project sources files available under 'HOL/examples/mc-logic' in the HOL4 distribution at SourceForge: http://hol.sourceforge.net/. 2008.

[5] C. J. Cheney, "A non-recursive list compacting algorithm," *Commun. ACM*, vol. 13, no. 11, pp. 677–678, 1970.

[6] L. Birkedal, N. Torp-Smith, and J. Reynolds, "Local reasoning about a copying garbage collector," in *Principles of programming languages (POPL)*. ACM Press, 2004, pp. 220–231.

[7] A. McCreight, Z. Shao, C. Lin, and L. Li, "A general framework for certifying garbage collectors and their mutators," in *Programming Language Design and Implementation (PLDI)*, J. Ferrante and K. S. McKinley, Eds. ACM, 2007, pp. 468–479.

[8] A. Fox, "Formal specification and verification of ARM6," in *Theorem Proving in Higher Order Logics (TPHOLs)*, ser. LNCS, D. Basin and B. Wolff, Eds., vol. 2758. Springer, 2003.

[9] X. Leroy, "Formal certification of a compiler back-end, or: programming a compiler with a proof assistant," in *33rd symposium Principles of Programming Languages POPL*. ACM Press, 2006, pp. 42–54.

[10] K. Crary and S. Sarkar, "Foundational certified code in a metalogical framework," Carnegie Mellon University, Tech. Rep. CMU-CS-03-108, 2003.

[11] M. O. Myreen and M. J. Gordon, "A Hoare logic for realistically modelled machine code," in *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, ser. LNCS. Springer-Verlag, 2007.

[12] M. O. Myreen, A. C. Fox, and M. J. Gordon, "A Hoare logic for ARM machine code," in *International Symposium on Fundamentals of Software Engineering (FSEN)*, ser. LNCS. Springer-Verlag, 2007.

[13] P. Manolios and J. S. Moore, "Partial functions in ACL2," *J. Autom. Reasoning*, vol. 31, no. 2, pp. 107–127, 2003.

[14] R. S. Boyer and J. S. Moore, "Proving theorems about pure LISP fucntions," *JACM*, vol. 22, no. 1, pp. 129–144, 1975.

[15] J. S. Moore, "Symbolic simulation: An ACL2 approach," in *Formal Methods in Computer-Aided Design (FMCAD)*, 1998, pp. 334–350.

[16] H. Liu and J. S. Moore, "Java program verification via a JVM deep embedding in ACL2," in *Theorem Proving in Higher Order Logics (TPHOLs)*, ser. Lecture Notes in Computer Science, K. Slind, A. Bunker, and G. Gopalakrishnan, Eds., vol. 3223. Springer, 2004, pp. 184–200.

[17] D. Currie, X. Feng, M. Fujita, M. Kwan, S. Rajan, A. J. Hu, and A. J. Hu, "Embedded software verification using symbolic execution and uninterpreted functions," *International Journal of Parallel Programming*, vol. 34, 2006.

[18] The FLINT Group. Yale University. http://flint.cs.yale.edu/.

[19] A. Chlipala, "Modular development of certified program verifiers with a proof assistant," in *International Conference on Functional Programming (ICFP)*. New York, NY, USA: ACM, 2006, pp. 160–171.

[20] J. G. Morrisett, D. Walker, K. Crary, and N. Glew, "From system F to typed assembly language," in *Principles of Programming Languages (POPL)*, 1998, pp. 85–97.

[21] J. Chen, D. Wu, A. W. Appel, and H. Fang, "A provably sound TAL for back-end optimization," in *Programming Language Design and Implementation (PLDI)*. New York, NY, USA: ACM, 2003, pp. 208–219.

[22] C. Flanagan and J. B. Saxe, "Avoiding exponential explosion: generating compact verification conditions," in *Principles of Programming Languages (POPL)*, 2001, pp. 193–205.

[23] K. R. M. Leino, "Efficient weakest preconditions," *Inf. Process. Lett.*, vol. 93, no. 6, pp. 281–288, 2005.

[24] P. V. Homeier and D. F. Martin, "A mechanically verified verification condition generator," *Comput. J.*, vol. 38, no. 2, pp. 131–141, 1995.

[25] Proceedings of the Working Conference on Reverse Engineering. IEEE. 1995–.

[26] J.-C. Filliâtre, "Verification of non-functional programs using interpretations in type theory," *J. Funct. Program.*, vol. 13, no. 4, pp. 709–745, 2003.

[27] S. Katsumata and A. Ohori, "Proof-directed de-compilation of low-level code," in *European Symposium on Programming (ESOP)*. Springer-Verlag, 2001, pp. 352–366.

Scheduling Optimisations for SPIN to Minimise Buffer Requirements in Synchronous Data Flow

Pieter H. Hartel and Theo C. Ruys
University of Twente, The Netherlands
email: {P.H.Hartel,T.C.Ruys}@utwente.nl

Marc C. W. Geilen
Eindhoven University of Technology, The Netherlands
email: M.C.W.Geilen@tue.nl

***Abstract*—Synchronous Data flow (SDF) graphs have a simple and elegant semantics (essentially linear algebra) which makes SDF graphs eminently suitable as a vehicle for studying scheduling optimisations. We extend related work on using SPIN to experiment with scheduling optimisations aimed at minimising buffer requirements. We show that for a benchmark of commonly used case studies the performance of our SPIN based scheduler is comparable to that of state of the art research tools. The key to success is using the semantics of SDF to prove when using (even unsound and/or incomplete) optimisations are justified. The main benefit of our approach lies in gaining deep insight in the optimisations at relatively low cost.**

I. Introduction

Synchronous Data Flow (SDF) is a paradigm for describing Digital Signal Processing (DSP) applications [13]. SDF has a long history dating back to the early 70s. Mainly due to the ever increasing interest in embedded systems, SDF is currently an active area of research. A typical application processes an infinite stream of data samples, which enter the SDF graph at the source node(s), and which exit the graph at the sink node(s). The SDF formalism abstracts away from the actual calculations taking place at the nodes, the contents of the tokens, and the time taken to transfer tokens or to perform calculations.

An SDF graph is a directed, connected graph. Each node in the graph represents a processing step, and the edges transport tokens between nodes. The nodes may fire independently of each other, and concurrently. The term synchronous means that when a node fires, it always consumes the same number of tokens from each input port, and the node always produces the same number of tokens on each output port. Each edge is connected to precisely one producer and precisely one consumer. A node that does not consume tokens is a source node, and a node that does not produce tokens is a sink node. An SDF graph may be cyclic. An SDF graph cannot be used to represent conditionals (this would make the SDF asynchronous). The semantics of an SDF graph can be given using linear algebra.

SDF graphs come in many flavours; we focus on the classical variant as discussed by Lee and Messerschmitt [13].

a) Problem: There are special purpose analysis tools that optimise throughput, latency, buffer requirements, timing and other relevant architectural parameters of an SDF graph as part of the DSP design flow. Even though the optimisation problems are typically NP complete [14], the simple semantics of SDF makes it possible to prove a wealth of useful properties that can be used as optimisations in the analysis. However, designing the algorithms, and experimenting with the optimisations requires a significant amount of effort.

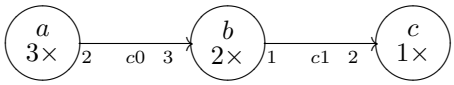


Fig. 1. Simple SDF graph.

b) Contribution: We show that due to the semantic simplicity of the SDF graph it is feasible to use a model checker as an efficient analysis tool for buffer requirements, making it easy to experiment with various optimisations. Such experiments are more difficult to conduct with a special purpose tool than with a powerful general purpose tool. The optimisations themselves are not specific to the model checker but can be applied in any other setting. We build on work from Geilen, Basten and Stuijk [7] (henceforth referred to as GBS) focusing on minimising the buffer space required for the channels. We improve the work of GBS in two ways. Firstly, we explore improvements to the efficiency of *checking* the minimum bounds, both in case the channel buffers share a common area of memory and in the case where each channel buffer has a separate area of memory (see Sections III . . . VI). Secondly, we develop new theory and the algorithms necessary for *finding* the minimum bounds (Section VII) for the common buffer case.

II. Examples

To give the intuition for the semantics of SDF we discuss three examples, the first of which is shown in Figure 1. The SDF graph has three nodes a, b, and c and two edges $c0$ and $c1$. The number at the tail of an edge is the production rate; the number at the head of an edge is the consumption rate. Node a is the source, and node c is the sink. The number in the node (e.g. $3\times$) is the relevant component of the repetition vector as calculated by Equation 4. Figure 1 is actually a chain, which is a directed connected graph of k nodes and $k-1$ edges such that only one path exists from the first to the last node [1, Chapter 4].

Each time node a fires, two tokens are produced and sent on channel $c0$ to node b. Node a must fire at least twice before

978-1-4244-2735-2/08/$25.00 ©2008 IEEE

Fig. 2. Cyclic SDF graph (left) and an inconsistent SDF graph (right).

node b is able to fire, because b consumes 3 tokens. Similarly, b must fire at least twice before c is able to fire. The state of the system records the current number of tokens on each channel. Firing a node causes the system to make a state transition. A periodic schedule is a sequence of state transitions that, starting from an initial state brings the system back into the initial state. The SDF graph of Figure 1 admits infinitely many periodic schedules. The shortest periodic schedules for our example are $(aababc)^*$ and $(aaabbc)^*$. These schedules are actually sequential schedules. In the first schedule the data dependencies inhibit concurrency, in the second schedule a and b may fire concurrently: $(aa(a||b)bc)^*$. Following GBS, in the sequel we will focus on sequential schedules. (Parallel schedules never require less buffer space than sequential schedules.) The minimum buffer capacity for $c0$ required by the second (sequential) schedule is 6 tokens, whereas for the first schedule 4 tokens would suffice on $c0$. Therefore schedule $(aababc)^*$ is the best of the two schedules in terms of the buffer capacity for $c0$.

The second example (Figure 2 left) shows a cyclic graph with two nodes d and e. Unlike the previous example, in which data can flow directly, this example is deadlocked, unless some initial tokens are present. Assume that 2 initial tokens are present on $c2$, as indicated by the two bullets. Then node e can fire twice, producing a total of 4 tokens on $c3$, after which node d can fire, once. This brings the system back in the initial state. This time the only possible schedule is: $(eed)^*$. The minimum buffer capacity required for $c2$ is 2 and 4 for $c3$.

The third example (Figure 2 right) shows an inconsistent SDF graph. The problem is that each time node f fires, it places 2 tokens on $c4$ and only one token on $c5$, whereas node g removes one token from both channels. This means that tokens will continue to accumulate on $c4$, which thus requires an infinite buffer capacity for any periodic (hence non-terminating) schedule; this is infeasible.

III. Semantics

An SDF graph with N nodes and C channels can be characterised by a *topology matrix*, with C rows and N columns, where the entries of the matrix give the production rates (positive) and consumption rates (negative) of the SDF graph. The topology matrix Γ for Figure 1 is:

$$\Gamma = \begin{bmatrix} 2 & -3 & 0 \\ 0 & 1 & -2 \end{bmatrix}$$

The state vector $\vec{s}(i)$ of the system is a non-negative column vector (of height C) representing the number of tokens held in each channel after i nodes have fired. The initial state $\vec{s}(0)$ specifies the number of tokens initially present on the channels, for example:

$$\vec{s}(0) = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \tag{1}$$

A state transition consists of two steps. Firstly a non-deterministic choice is made to select the node that is to be fired. This choice is represented in the column vector $\vec{f}(i)$:

$$\vec{f}(i) = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} \text{ or } \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} \text{ or } \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \tag{2}$$

Secondly, the effect of firing the node on the state is specified by Equation 3, making sure that firing the selected node maintains a non-negative state vector (we ignore self edges, as the required buffer size for a self edge is easy to calculate):

$$\vec{s}(i+1) = \vec{s}(i) + \Gamma\vec{f}(i), \qquad \vec{s}(i+1) \geq \vec{0} \tag{3}$$

The schedule $aababc$ of Figure 1 for example corresponds to the following sequence of state transitions:

$$\vec{s}(0)\ldots\vec{s}(6) = \begin{bmatrix}0\\0\end{bmatrix}\begin{bmatrix}2\\0\end{bmatrix}\begin{bmatrix}4\\0\end{bmatrix}\begin{bmatrix}1\\1\end{bmatrix}\begin{bmatrix}3\\1\end{bmatrix}\begin{bmatrix}0\\2\end{bmatrix}\begin{bmatrix}0\\0\end{bmatrix}$$

Inspecting the top most elements of the state vectors shows that the minimum buffer capacity on $c0$ is 4, and inspecting the bottom elements reveals that a buffer capacity of 2 suffices for $c1$. Depending on how buffer space is allocated to channels we can now draw two conclusions. Firstly, if all buffers share a *common* area of memory, the maximum buffer capacity required is 4, which is reached by states 2 and 4. Secondly if each channel has a *separate* buffer, the maximum buffer capacity is 6, since the maximum capacity of 4 for $c0$ is reached at state 2 and the maximum buffer capacity of 2 for $c1$ is reached at state 5.

We now review those results from the literature about the semantics of SDF that we need in the sequel.

An SDF graph is *consistent* iff $\mathsf{rank}(\Gamma) = N - 1$ [13]. A deadlock free and consistent SDF graph has periodic schedules [8].

The N element *repetition vector* $\vec{r}$ is the least non-trivial integer solution of the equation [13]:

$$\Gamma\vec{r} = 0 \tag{4}$$

The repetition vector for Figure 1 is $\vec{r} = [3\ 2\ 1]^T$.

Assume that for a given channel x the production rate is p, the consumption rate is c, and the initial number of tokens on the channel is t, the *lower bound* on the buffer capacity of the channel for a deadlock free schedule is [2]:

$$\mathsf{lwb}_c(x) = p + c - d + t \bmod d, \qquad \textbf{where } d = \mathsf{gcd}(p, c) \tag{5}$$

Assume that for a given channel x the production rate is p, the channel is connected to the output port of node n, and the component of the repetition vector corresponding to node n is r, the *upper bound* on the buffer capacity of the channel for a deadlock free schedule is [2]:

$$\mathsf{upb}_c(x) = r \times p \tag{6}$$

```
byte c0, c1;  /* Common buffer pool model */
init{ do
/*a*/ ::             c0+=2;
/*b*/ :: (c0>=3) -> c0-=3; c1+=1;
/*c*/ :: (c1>=2) -> c1-=2;
      od }
/* LTL feasible:   [](c0+c1<=4) */
/* LTL infeasible: [](c0+c1<=3) */
```

```
byte c0, c1;     /* Separate buffer model */
byte s0=4, s1=2;
#define max(a,b) (a>b->a:b)
init{ do
/*a*/ ::             c0+=2;        s0=max(c0,s0);
/*b*/ :: (c0>=3) -> c0-=3; c1+=1; s1=max(c1,s1);

/*c*/ :: (c1>=2) -> c1-=2;
      od }
/* LTL feasible:   [](s0+s1<=6) */
/* LTL infeasible: [](s0+s1<=5) */
```

Fig. 3. GBS model of the simple SDF graph with a common buffer pool (above) and separate buffers for each channel (below).

A lower bound on the buffer space for the whole graph is $\Sigma_{1\leq x\leq C}\mathsf{lwb}_c(x)$ and an upper bound is $\Sigma_{1\leq x\leq C}\mathsf{upb}_c(x)$.

With these results, a significant part of the problem of finding a periodic schedule with a minimum buffer size has been solved, because we can check first whether a graph is consistent. If a graph is indeed consistent, calculating the repetition vector gives the number of times each node must fire, and calculating the lower and upper bound on the buffer capacity we have the range in which to search for the minimum buffer size. Unfortunately, in practical cases the upper bound is typically much larger than the lower bound (See Table III). On the other hand, the lower bound is often also the minimum buffer size, which suggests that a good heuristic would be to look for a periodic schedule with the lower bound first. If this fails, a more general search is needed.

IV. Model checking with SPIN

A state based model checker such as SPIN [11] is a tool that explores all possible behaviours of a Labelled Transition System, either to prove the absence of unwanted behaviour (safety properties), or to prove the existence of desired behaviour (liveness properties). As observed by GBS, when given an appropriate model of an SDF graph, the model checker can be used to check whether or not a schedule exists, calculating both the schedule and the minimum buffer size of each channel.

There are several reasons for choosing SPIN for our analysis. Firstly, SPIN is arguably one of the most powerful explicit state model checkers available. Secondly, the SPIN `c_code` extensions allow us to implement the Branch and Bound extensions of Section VII. Finally, as GBS also use SPIN, the comparison between GBS and our work is fair.

A. GBS models with a common buffer pool

We will describe the essence of the GBS models (Figure 3), indicating the direct correspondence between the model and the semantics of Section III. The state of the model consists of the pair of channel counters (i.e counting the number of tokens in each channel) `c0` and `c1`. This pair represents the state vector of Equation 1. The `do ... od` statement causes the system to make a sequence of state transitions, and each guarded command `:: ...` corresponds to firing one of the nodes, provided that the command is enabled (i.e when the guard is true). The guards ensure that the state vector remains non-negative, as specified by condition of Equation 3. The assignments in each guarded command correspond to Equation 3. If more than one guard is true a non-deterministic choice is made to select one of the guarded commands. This selection corresponds to the non-deterministic choice of Equation 2.

The model of Figure 3 (above) is used to check whether the total amount of buffer space (i.e. when one common pool of buffer space is used for all channels) is less than or equal to 4. When presented to SPIN, the Linear Temporal Logic (LTL) property `[](c0+c1<=4)` requests the model checker to find a schedule represented as an infinite sequence of states, where each state satisfies `(c0+c1<=4)`, the buffer capacity invariant. (In SPIN jargon the schedule represents a counter example to the error behaviour specified by the LTL formula). The model can also be used to verify that no periodic schedule exists with a bound less than or equal to 3 (using the second, infeasible property), thus proving that 4 is indeed the minimum size of the common buffer pool.

To avoid clutter, we show a simplified version of the GBS models. In particular all guarded commands `:: ...` in our models should be interpreted as atomic statements, i.e. they should be read as `:: atomic{ ... }`.

B. GBS models with separate buffers

The state space generated by SPIN from the model of Figure 3 (above) coincides with the state space of the SDF semantics as discussed in Section III, and may therefore be considered a good concrete model. The GBS model for the case where instead of one buffer pool, each channel has its own buffer space is shown in Figure 3 (below). The two variables `s0` and `s1` store the maximum number of tokens buffered by `c0` and `c1`. GBS show that the *lower bound* optimisation, which initialises `s0` and `s1` to the lower bound calculated according to Equation 5 is effective. The reason is that if `s0` and `s1` are initialised to 0, a first set of *transient* states must be explored until `s0` reaches 4 and `s1` reaches 2. Then, the values of `s0` and `s1` must be maintained while a second set of *periodic* states is explored that represent the schedule. Since the schedule consists of the periodic set, it is beneficial to avoid the transient set. This is exactly what the GBS optimisation *lower bound* achieves.

The model of Figure 3 (below) can be used to check that the sum of the bounds on two separate buffers is 6 (feasible property), and that no periodic schedules are possible with a sum less than or equal to 5 (infeasible property).

```
byte na, nb, nc;   /* Same for both models */
#define c0 (na*2-nb*3)
#define c1 (nb*1-nc*2)
init{ do
/*a*/ :: (na<3)           -> na++;
/*b*/ :: (nb<2 && c0>=3)  -> nb++;
/*c*/ :: (nc<1 && c1>=2)  -> nc++;
      od }
#define r  (c0==0 && c1==0)
#define p0 ((c0<=3) && (c1<=2))
#define p1 ((c0<=4) && (c1<=1))
```

```
                        /* Common buffer model */
/* LTL feasible:   X ((c0+c1<=4) U r) */
/* LTL infeasible: X ((c0+c1<=3) U r) */
```

```
                        /* Separate buffer model */
/* LTL feasible:   X ((c0<=4 && c1<=2) U r) */
/* LTL infeasible: X (p0 U r) || X (p1 U r) */
```

Fig. 4. Our model (also showing the Limiting optimisations) of the simple SDF graph with a common buffer pool (middle) or separate buffers for each channel (below). The top part is common to both models.

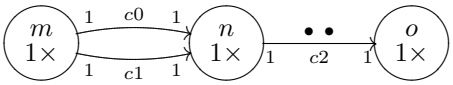


Fig. 5. Avoiding the transient is incomplete.

V. Optimisations

Optimisations avoid searching those parts of the state space that cannot lead to periodic schedules, or that lead to schedules worse than we have already seen. An optimisation that may miss correct periodic schedules satisfying a given buffer constraint is *incomplete*. An optimisation that that may yield incorrect schedules is *unsound*. All types of optimisation may be useful. For example a schedule found by an incomplete optimisation is correct but it may be sub-optimal, and it is often possible to check via some alternative means whether a schedule found by an unsound optimisation is correct or not. We give examples of optimisations, indicating whether the optimisation is effective, sound and/or complete on a benchmark of commonly used SDF graphs, including some realistic applications.

A. Node counters

The GBS *channel* counters contain redundancy that can be avoided by using *node* counters instead (Figure 4). It is easy to calculate the value of a channel counter from the relevant node counters (as shown by the macro definitions for `c0` and `c1`) but it is not possible the other way round. Unlike the channel counters, node counters are in principle unbounded, and should be used in combination with the Limiting optimisation (Section V-D). Sound. Complete. It depends on the SDF graph whether node counters are effective.

B. Avoiding the transient

GBS models produce schedules with a transient and a periodic part. For example the SDF graph of Figure 1 may generate a schedule such as $aa(babaac)^*$ with a transient aa and a periodic part $(babaac)^*$. Often there is a shorter schedule without a transient part. To avoid a schedule with a transient we use an LTL property that ensures that the schedule begins *and* ends in the initial state. This property is of the general form `X (p U r)` with the following interpretation. Assume that in the initial state property `r` (characterising the initial state) is true. The neXt operator `X` moves to the next state. Then we use the Until operator `U` to specify a sequence of states for which the property `p` (the buffer constraint) holds, until finally again the property `r` holds (and also `p` since `r` implies `p`). Using the feasible LTL property of Figure 4 (middle) we can verify that a periodic schedule exists with a bound of 4 on the common buffer pool. To verify that no schedules exist for smaller bounds the infeasible property of Figure 4 (middle) can be used. For this particular benchmark, as we argued in Section III, 4 is provably the lower bound on the common buffer size. Therefore, there is no need to run the model checker to confirm that 4 is indeed the minimum bound. The only benchmarks where the lower bound is not the minimum bound are ade and adebetter (See Section VI for more information about the benchmarks).

Avoiding the transient is sound (because we are not changing schedules) but incomplete as demonstrated by the example of Figure 5, which admits a schedule $o(omn)*$ with a common buffer of size 2, that is found by the GBS model but not by our model. (Avoiding the transient is complete for the separate buffer case.) Effective.

Our model for the separate buffer pool (Figure 4 below) is the same as for the common buffer pool. Unfortunately we need an LTL property that is exponential in size in the number of channels, which is clearly infeasible. Instead, we use a property that consists of as many conjunctions as there are channels, with each conjunct reducing the buffer space for its channel by 1. Incomplete.

C. Priority

When making a non-deterministic choice, SPIN explores the guards top down, so reversing the order of the guards causes different parts of the state space to be explored first. This property of the semantics of SPIN makes it possible to model the priority principle [1, Section 4.1], which gives increasing priority to successive nodes in a chain. Most practical graphs, including the graphs in our benchmark are not chains but cyclic graphs, where the priority principle cannot be applied. Indeed the benchmark results show no significant changes when applying the principle. Sound and Complete, because we are merely changing the order in which schedules appear. Ineffective.

D. Limiting

The number of times a node fires is limited by the repetition vector (Equation 4), because a periodic schedule must invoke each node at least as often as given by the repetition vector. This is shown by the guarded commands of Figure 4, where each guarded command has a condition for the form

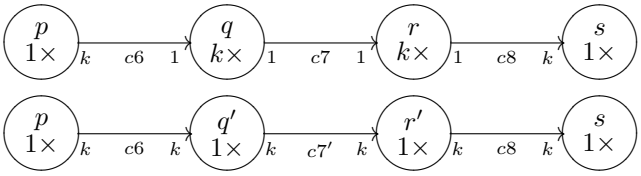


Fig. 6. Chain from the h263 decoder (above) and the same chain (below) without spurious repetition of the nodes q and r, where $k = 2376$.

nx<y. Sound, because we are not changing any schedules. Incomplete as illustrated in Figure 5.

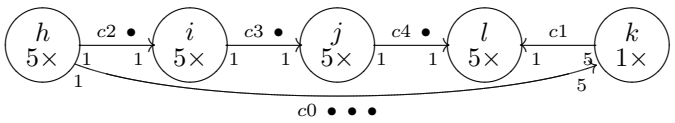
E. Clustering

In realistic data flow graphs the firing rate of some nodes may differ considerably. Figure 6 (above) gives an example of a chain from the h263 decoder, where nodes p and s are fired once against k times for nodes q and r. This difference in firing rate increases the number of interleavings exponentially in k (as the Catalan number of k) and hence also increases the size of the state space. Our clustering optimisation *transforms* a chain into one with smaller differences in the firing rates such as Figure 6 (below). To transform nodes q and r into q' and r' the consumption and production rates of these nodes are multiplied by k, at the same time the entries in the repetition vector of the nodes are divided by k. Let Γ and Γ' be the topology matrix before and after the transformation. It is easy to check that $\mathsf{rank}(\Gamma) = \mathsf{rank}(\Gamma')$ hence the transformation does not affect consistency. However, the transformation is unsound, since $\mathsf{lwb}_c(c7') = k$ whereas $\mathsf{lwb}_c(c7) = 1$.

Once a schedule has been found for the transformed model, it is possible to construct a schedule for the original model. For example given a schedule $(pq'r's)^*$ for the transformed system, we can represent q' in the schedule of the transformed system by q^k in the original system, and likewise for r'; this yields a schedule $(pq^kr^ks)^*$ for the original system. Unfortunately, this schedule requires a buffer of size k for channel $c7$. We can do better than this by interleaving q and r, which yields the following schedule for the original system: $(p(qr)^ks)^*$. Using Equation 3 it is possible to prove (simply by replaying the schedule) that this is indeed a valid schedule for the original system of Figure 6 (above). Using Equation 5 we can also prove that this is an optimal schedule. Sound if we include the transformation of the schedule as suggested above. Incomplete. Effective.

F. Look ahead

Look ahead is an optimisation where each node has knowledge of the behaviour of its immediate successors. Look ahead permits a node to fire only when at least one of its outputs has insufficient tokens for the successor node. Consider the example of Figure 7. Node h may fire when its successor i has insufficient input or when its successor k has insufficient input (or both). The idea behind look ahead by node h is that if both successors do have sufficient input, h is blocked to avoid overfilling $c0$ and $c2$.




```
#define p4 (c0<=5&&c1<=5&&c2<=1&&c3<=1&&c4<=3)
#define p2 (c0<=5&&c1<=5&&c2<=3&&c3<=1&&c4<=1)
#define r  (c0==0&&c1==0&&c2==0&&c3==0&&c4==0)
/* LTL sound:   X (p2 U r) */
/* LTL unsound: X (p4 U r) */
```

Fig. 7. Two chains with a common start node h, and a common end node l in the state where node h has fired three times, i twice and j once. A sound and an unsound version of the LTL property are shown below.

TABLE II
SUMMARY OF THE OPTIMISATIONS. * = IF THE TRANSFORMATION OF THE SCHEDULE IS INCLUDED

Optimisation	Effective	Sound	Complete
Node counters V-A	±	+	+
Avoid transient V-B	+	+	-
Priority V-C	-	+	+
Limiting V-D	+	+	-
Clustering V-E	++	-	+*
Look ahead V-F	+	+	-

The example of Figure 7 has been constructed such that there are two chains (i.e. the upper chain h, i, j, and l and the lower chain h, k, and l). Both the lower chain and the upper chain have to store 5 tokens. However, looking at the production and consumption rates of the upper chain alone it would appear that one token on each channel (hence a total of 3) would suffice on the upper chain.

There are many ways in which to distribute the two extra tokens over the buffer capacity of the upper chain. For example property p4 of Figure 7 forces the excess to be stored in $c4$, and property p2 stores the excess in $c2$. However, only one of these methods (i.e. property p2) is compatible with the optimisation for look ahead. To illustrate this point Figure 7 shows the state of the system where node h has fired three times, i twice and j once. The entire system is now blocked: Nodes k and l are blocked because there are *insufficient* tokens on their input channels and nodes i and j are blocked because there are already *sufficient* tokens on their output channels. Node h is blocked because p4 only allows the excess to be stored in $c4$. If on the other hand we would have used property p2, the network would not have been blocked. Incomplete. In the adebetter benchmark, which has been carefully constructed by Adé to demonstrate the intricacies of SDF scheduling [2], look ahead will increase the minimum buffer capacity by one. Unsound. Effective.

VI. CHECKING THE OPTIMAL BUFFER SIZE

Our benchmark consists of 10 SDF graphs taken from various sources. The benchmarks simple, bipart, cddat, modem, ade, adebetter, inmarsat, and h263 are used by GBS [7], the benchmarks mp3sys and mp3dec are used by Stuik et al [17]. These benchmarks are used by many other authors in the field

TABLE I
STATES STORED BY SPIN FOR THE BEST VERSIONS OF THE 10 BENCHMARKS AND MS EXECUTION TIME FOR STATE OF THE ART RESEARCH TOOLS. (* = ADEBETTER WITHOUT THE LOOK AHEAD OPTIMISATION, † ENTRIES SWAPPED IN GBS [7, TABLE 1])

	simple	bipart	cddat	modem	ade	adebetter	inmarsat	V-E	h263	V-E	mp3dec	mp3sys	V-E
States stored checking feasible schedule with given bound. Separate buffer space													
GBS	11	88	4127	210	497	8602†	2862	66	4758	9	139	19308	1797
V-D, V-F	8	84	614	50	47	129*	1133	52	4758	8	15	16385	125
States stored checking infeasible schedule with given bound. Separate buffer space													
GBS	2	2	2	2	2241	1708†	2	2	2	2	2	2	2
V-D	-	-	-	-	581	721	-	-	-	-	-	-	-
States stored checking feasible schedule with given bound. Common buffer space													
GBS	9	124	755	1231	366	156	180369	163	9511	11	23	55004	424
V-D, V-F	8	94	614	176	96	109	1110	52	4758	8	15	16381	121
States stored checking infeasible schedule with given bound. Common buffer space													
GBS	4	150	3542	853	303	140	aborted	240	aborted	7	12	aborted	925
V-D	3	149	3541	852	127	139	13102300	81	2826250	4	8	19653500	925
milliseconds CPU time±standard deviation checking feasible+infeasible schedule. Separate buffer space													
SDF3	6±4	6±4	8±4	10±4	42±4	8±3	19±4	11±5	22±4	7±5	10±3	55±5	7±4
Hebe	11±3	11±3	11±3	15±3	12±3	12±3	16±3	15±3	12±3	12±4	14±2	11±3	12±3
SPIN	18±8	20±8	20±9	19±8	41±16	41±19	24±9	18±9	33±10	18±9	19±10	70±12	19±8

and are therefore assumed to be representative for SDF graphs.

To avoid creating the same variants of 10 different benchmarks, we wrote a C program that given the topology matrix and the initial token assignment of an SDF benchmark generates the SPIN models necessary to explore the optimisations described in Section V and summarised in Table II.

Table I shows for each benchmark the best results that we obtain in terms of the number of states stored by SPIN to find a feasible schedule, or to prove that such a schedule does not exist. In all cases the same or a better feasible schedule is found, indicating that in the benchmark examples unsound and incomplete optimisations do not cause problems.

The table is divided into five sections. The first two sections report the states stored for models where each channel has a separate buffer. The next two sections apply to models where there is one common buffer pool for all channels. The first and third section presents the results when looking for (the first occurrence of) a feasible schedule, whereas in the second and fourth section we report on the number of states encountered when exploring the state space exhaustively because there is no feasible schedule. The last section reports execution times.

The rows marked GBS in the first column are the best results of GBS taken from their paper [7, Table 1]. We have repeated the experiments of GBS to be able to include GBS results on the mp3dec and mp3sys benchmarks also.

In the entries marked "aborted" we terminate the experiment after 5 minutes of CPU time. In all benchmarks except ade and adebetter a feasible schedule is found with the minimum buffer size, so it does not make sense (indicated by a hyphen) to try to prove that a configuration with less buffer space than the theoretical minimum is infeasible.

The rows *not* marked GBS represent our best results, indicating which optimisation(s) have been deployed (referred to by section number). Without exception, our results are better than GBS, in some cases by several orders of magnitude, for example in case of the inmarsat benchmark. Overall, the most important cause for the improvement is the use of node counters with the Limiting optimisation instead of channel counters.

The benchmarks with large differences in production and consumption rates on the same channel, such as inmarsat, h263 and mp3sys benefit significantly from the clustering optimisation, by up to five orders of magnitude. The columns marked V-E report the data for the clustered versions of these benchmarks. The reason is that the number of interleavings is exponential (the Catalan number) in the number of times each node may fire. The clustering optimisation reduces this to a linear dependency, hence the significant difference.

State of the art research tools do not provide an equivalent to the number of states explored as a metric. Therefore, to compare our results to those tools, we have repeated the first (i.e. Separate buffer, feasible+infeasible schedule) experiment for all benchmarks using SDF3 [18] and Hebe [19], all on the same Linux machine. The SPIN models and SFD3 provide an exact solution, Hebe calculates a good approximation (within 10%) to the minimum buffer size. The SPIN models can only be used to analyse the minimal buffer capacity for deadlock-free execution of SDF graphs, whereas SDF3 and Hebe can also take throughput into account. We have tried to make sure that this does not give our approach an unfair advantage; in fact the authors of SDF3 have helped us to make various modifications to avoid bias as much as possible. The CPU user times measured as an average over 50 runs as well as the sample standard deviation are shown in the last section of Table I. The error margins overlap so much that we conclude that the performance of all three tools is comparable. This shows that it is cost effective to gain insights by experimenting with a range of optimisations using a general purpose tool, before undertaking costly special purpose tool development. For example GBS spent only a few days implementing the minimum buffer size algorithm of the SDF3 tool (which computes the entire buffering-throughput trade-off space), after having spent more time experimenting with SPIN. Ultimately, an ideal tool framework would include

TABLE III

BUFFER SIZES AND STATES STORED RATIOS FOR THE BENCHMARKS FOR THE COMMON BUFFER (TOP) AND THE SEPARATE BUFFER (BOTTOM). $\mathsf{min}_{c,n}$ IS THE MINIMUM BUFFER SIZE REQUIRED BY A FEASIBLE SCHEDULE.

	simple	bipart	cddat	modem	ade	adebett	inmarsV-E	h263V-E	mp3dec	mp3sysV-E
Bounds for the common buffer case based on the analysis of nodes.										
$s = \mathsf{min}_{1\leq x\leq N}\ \mathsf{lwb}_n(x)$	2	10	1	1	5	4	240	3	1	5
$g = \mathsf{max}_{1\leq x\leq N}\ \mathsf{lwb}_n(x)$	4	16	15	10	25	9	720	4752	2	1536
min_n	4	26	16	13	67	18	1008	4754	2	1539
$\mathsf{upb}_n = \Sigma_{1\leq x\leq N}\ \mathsf{lwb}_n(x)$	12	60	60	149	105	72	5472	23762	22	8843
state stored ratio	1.0	1.0	1.9	1.5	3.9	5.0	3.9	1.4	1.0	1.2
SPIN runs	1	2	2	4	10	4	3	2	1	2
Bounds for the separate buffer case based on the analysis of channels.										
$\Sigma_{1\leq x\leq C}\ \mathsf{lwb}_c(x)$	6	28	32	38	49	39	3072	9508	12	2961
min_c	6	28	32	38	83	42	3072	9508	12	2961
$\Sigma_{1\leq x\leq C}\ \mathsf{upb}_c(x)$	8	264	1021	61	209	133	3936	9508	12	27406

a range of techniques [9].

VII. FINDING THE OPTIMAL BUFFER SIZE

Thus far we have explored optimisations to the GBS approach to check whether a given bound on the buffer size is optimal. The check requires running the model checker twice: once to verify that a schedule with the given bound can be found, and a second time to verify that no schedule can be found with a bound of one less. Finding the optimal bound is a more challenging problem for two reasons. Firstly, we must be able to calculate an initial guess for the minimum bound. Secondly, depending on the quality of the guess, we may have to run the model checker several times. To make the problem even more challenging, we will study the case of the common buffer, which as Table I shows, requires considerable more work (i.e. more states to be stored) than the separate buffer case. Therefore in this section we will develop the necessary theory and apply the theory in practical optimisations to find optimal bounds on common buffers for the benchmarks.

A. Theoretical lower bound

The literature provides theoretical results on the lower bound and upper bound on the buffer space required for SDF graphs when each channel buffer resides in a separate area of memory (c.f. $\mathsf{lwb}_c(.)$ and $\mathsf{upb}_c(.)$ in Section III). Unfortunately, we have not been able to find equivalent results for the case where all buffers share a common area of memory. Therefore, we will develop new theory to calculate a lower bound on the total common buffer space required by an SDF graph. The idea for the calculation is to analyse each node n separately by decoupling n from the graph, together with all its direct neighbours and the channels connecting n to the neighbours. We will call this sub graph the decoupled graph of n. For example, decoupling node a in Figure 1 would create a new graph consisting of copies of nodes a, and b, and the connecting channel $c0$. Decoupling node d in Figure 2 would create a new graph consisting of a copy of node d, and *two* copies of node e as well as the connecting channels $c0$, and $c1$. The schedule admitted by a decoupled graph of node n is completely unconstrained, hence the schedule is defined by the following algorithm:

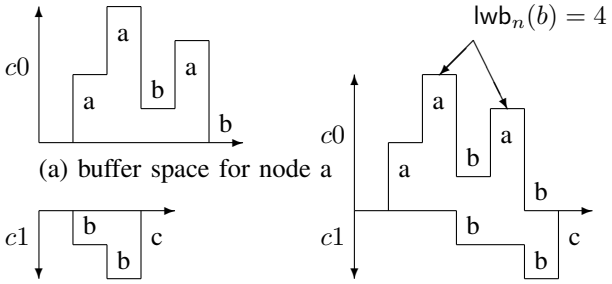


Fig. 8. Common buffer space analysis for SDF graph from Figure 1

1. put the initial tokens on all channels of the decoupled graph of n.
2. repeat
2.1 Fire each node sending tokens to n as often as necessary to satisfy the consumption rates of the inputs to node n.
2.2 Fire node n once.
2.3 Fire each node receiving tokens from n as often as possible.
3. until node n has been fired $\vec{r}(n)$ times.

The lower bound on the total common buffer size $\mathsf{lwb}_n(n)$ of the decoupled graph for node n is then the maximum number of tokens on all channels of the decoupled graph observed during the execution of the algorithm.

For example the total buffer capacity for the decoupled graph of node a from Figure 1 is $\mathsf{lwb}_n(a) = 4$. The maximum is reached after two firings of a as shown in Figure 8(a). Figure 8(b,c) show that $\mathsf{lwb}_n(b) = 4$, and $\mathsf{lwb}_n(c) = 2$.

To prove that $\mathsf{lwb}_n(n)$ is indeed a lower bound on the amount of common buffer space required by node n we analyse the algorithm. Line 2.1 ensures that when node n fires, no more tokens are present on the input channels to node n than strictly necessary to satisfy the consumption rates of n. In a realistic schedule, there may be more tokens present on the input channels than in the decoupled graph, but not less. Line 2.3 ensures that the output channels are emptied as much as possible. In a realistic schedule there may be more tokens that

TABLE IV
BRANCH AND BOUND SEARCH FOR THE OPTIMAL COMMON BUFFER SIZE FOR ADEBETTER. STEP SIZE $s = 4$.

Initial guess g	States stored	Feasible bounds
10	30	none
14	66	none
18	157	none
22	7648	21,20,19,18
total	7901	
19	1568	18
ratio	5.0	

remain in the output channels than in the decoupled graph, but not less. Summarising, both on the input and on the output side of node n no more tokens are present than strictly necessary. Hence $\mathsf{lwb}_n(n)$ gives a lower bound on the amount of common buffer space required by node n.

The complexity of the algorithm to calculate $\mathsf{lwb}_n(n)$ is linear in $\vec{r}(n)$.

B. Optimisations for the minimum bound

Equipped with a lower bound on the size of the common buffer pool for each node we are ready to develop a scheduling algorithm. A good basis for this is the SPIN version of the Branch and Bound algorithm as proposed by Ruys [15], which can be adapted to our needs as follows:

1. Start with an initial guess g for the optimal bound and a step size s, where: $g \leftarrow \mathsf{max}_{1\leq n\leq N}\mathsf{lwb}_n(n)$, and $s \leftarrow \mathsf{min}_{1\leq n\leq N}\mathsf{lwb}_n(n)$.
2. repeat
2.1 Let SPIN find a schedule with an optimal bound $b \leq g$.
2.2 if such a schedule can be found then exit
2.3 else $g \leftarrow g + s$
3. end repeat

Since g is a lower bound on the buffer size, and $s > 0$, the algorithm is guaranteed to terminate. The SPIN models used are basically the same as the GBS models, with the modifications described by Ruys [15] to find the minimum bound $b \leq g$. The appendix provides the complete source code of the simple benchmark. Note that to check whether an optimal bound exists for guess g, we initialise with $g+1$ (see Table IV), to let SPIN find a bound less than $g + 1$, i.e. g.

To analyse how successful the Branch and Bound strategy is, we take as an example the adebetter benchmark. Table IV shows that starting with an initial guess of $g = 10$, after visiting 30 states SPIN terminates because no feasible schedules can be found with a bound lower than 10. Then the guess is increased by step size $s = 4$ to 14, and SPIN is run a second time, again without finding a schedule. This is repeated until $g = 22$. Now SPIN finds a feasible schedule with a bound of $b = 21$, and starts looking for another schedule with a bound lower than 21. Indeed such a schedule is found; with a bound of 20 etc until a schedule with a bound of 18 is found, and no schedule can be found with a bound lower than 18. The total number of states stored (7901) is a measure for the amount of work performed to search for a feasible schedule with the optimal bound.

The choice of the initial guess g, and the step size s is critical for the efficiency of the search. For many benchmarks, the initial guess is a reasonable bound, as we can see by comparing the second row (labelled $g = \mathsf{max}_{1\leq x\leq N}\ \mathsf{lwb}_n(x)$) and the third row (labelled min_n) that shows the true minimum bound in table III for all benchmarks. For completeness the table also shows the step size $s = \mathsf{min}_{1\leq x\leq N}\ \mathsf{lwb}_n(x)$ and a (poor) upper bound calculated as $\Sigma_{1\leq x\leq N}\mathsf{lwb}_n(x)$.

The choice of the step size is motivated as follows. The initial guess represents the needs of the decoupled graph with the largest buffer requirements, and the step size represents the needs of the decoupled graph with the smallest buffer requirements. In the extreme case of an SDF graph with only two nodes, the optimal buffer size can be anywhere between g (when the buffer capacities of the two nodes completely overlap) and $g + s$ (when the buffer capacities are completely disjoint). So if the optimal buffer is not found with the initial guess g it will definitely be found with the next guess $g+s$. In an SDF graph with more than 2 nodes, the step size controls how many more iterations than two could be necessary. There are two reasons why starting with an initial guess that is likely to be too low and increasing the guess is better than starting with an initial guess that is too high. Firstly, there are many schedules with a sub optimal buffer size, such that the search starting from a high initial guess yields many spurious results that are time consuming to find and discount. Secondly, an initial guess that is too low causes many branches in the search space to be pruned quickly.

To indicate how good the search optimisations are, Table IV (bottom) shows that for adebetter with an initial guess $g = 19$ (i.e. one more than the true lower bound) the number of states visited is 1568. This means that to find the best schedule SPIN has to do about 5 times as much work as to check the best schedule. Table III shows these work ratios for each benchmark (row labelled state stored ratio) as well as the relevant bounds. The conclusion is that with our Branch and Bound algorithm finding the minimum schedule on the benchmark is up to five times more expensive than checking the best bound, which we believe is a good result.

VIII. CONCLUSIONS AND FUTURE WORK

Many authors have used model checkers to solve scheduling problems [3] [4] [5] [6] [12] [16] [10], but Geilen, Basten and Stuijk [7] (GBS) were the first to use SPIN for the analysis of SDF graphs. Their results are promising but inconclusive in the sense that some realistic SDF graphs cannot be analysed effectively. Our approach towards *checking* given bounds using unsound and incomplete optimisations generally pays off and in specific cases exponential complexity is reduced to linear complexity by our clustering optimisation. As a result all case studies used can be analysed by SPIN in about the same time as needed by state of the art research tools. This makes SPIN a useful prototype tool for the buffer size analysis of SDF

graphs. In the end the most effective techniques could then be integrated in special purpose tools such as SDF3 and Hebe.

We offer new theory and an efficient Branch and Bound algorithm to *find* minimum bounds, thus solving a problem not considered by GBS. The main advantage of using SPIN as the Swiss army knife of computer science is that no special purpose tools have to be created in order to gain deep insight into NP complete problems by extensive experimentation with optimisations. It would be an interesting challenge to extend the SPIN models, particularly with throughput constraints, or to more liberal dataflow models, such as models with data-dependent rates. Furthermore, we will investigate whether the Branch and Bound optimisations can be further improved, e.g., by using binary search, or by looking ahead in the search path.

ACKNOWLEDGEMENTS

Maarten Wiggers ran our models through his Hebe tool. Gerard Holzmann answered all our SPIN questions. Hylke van Dijk, Angelika Mader, Sander Stuijk, and Maarten Wiggers provided helpful feedback.

REFERENCES

[1] M. Adé. *Data Memory Minimization for Synchronous Data Flow Graphs Emulated on DSP-FPGA Targets*. PhD thesis, Katholieke Universiteit Leuven, Oct 1996.

[2] M. Adé, R. Lauwereins, and J. A. Peperstraete. Data memory minimisation for synchronous data flow graphs emulated on DSP-FPGA targets. In *34th Design Automation Conf. (DAC)*, pages 64–69, Anaheim, California, Jun 1997. IEEE Computer Society.

[3] Gerd Behrmann, Ansgar Fehnker, Thomas Hune, Kim Larsen, Paul Pettersson, and Judi Romijn. Efficient Guiding Towards Cost-Optimality in UPPAAL. In Tiziana Margaria and Wang Yi, editors, *Procs. of the 7th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2001)*, volume 2031 of *LNCS*, pages 174–188, Genova, Italy, April 2001. Springer.

[4] Ed Brinksma and Angelika Mader. Verification and Optimization of a PLC Control Schedule. In Klaus Havelund, John Penix, and Willem Visser, editors, *SPIN Model Checking and Software Verification, Procs. of the 7th Int. SPIN Workshop (SPIN'2000)*, volume 1885 of *LNCS*, pages 73–92, Stanford, California, USA, August 2000. Springer.

[5] Ansgar Fehnker. Scheduling a Steel Plant with Timed Automata. In *Procs. of the 6th Int. Conf. on Real-Time Computing Systems and Applications (RTCSA 1999)*, pages 280–286. IEEE Computer Society, 1999.

[6] Ansgar Fehnker. *Citius Vilius Melius – Guiding and Cost-Optimality in Model Checking of Timed and Hybrid Systems*. PhD thesis, University of Nijmegen, The Netherlands, April 2002.

[7] M. C. W. Geilen, T. Basten, and S. Stuijk. Minimising buffer requirements of synchronous dataflow graphs with model checking. In *42nd Design Automation Conf. (DAC)*, pages 819–824, San Diego, California, Jun 2005. ACM.

[8] A. H. Ghamarian, M. C.W. Geilen, T. Basten, B. D. Theelen, M. R. Mousavi, and S. Stuijk. Liveness and boundedness of synchronous data flow graphs. In *6th Int. Conf. on Formal Methods in Computer Aided Design (FMCAD)*, pages 68–75, San Jose, California, Nov 2006. IEEE Computer Society Press.

[9] R. Govindarajan, G. R. Gao, and P. Desai. Minimizing buffer requirements under Rate-Optimal schedule in regular dataflow networks. *J. VLSI Signal Process. Syst.*, 31(3):207–229, Jul 2002.

[10] S. Haynal and F. Brewer. Representing and scheduling looping behavior symbolically. In *Int. Conf. on Computer Design*, pages 552–555, Austin, Texas, Sep 2000. IEEE Computer Society.

[11] G. J. Holzmann. *The SPIN Model Checker: Primer and Reference manual*. Pearson Education Inc, Boston Massachusetts, 2004.

[12] Kim Larsen, Gerd Behrmann, Ed Brinksma, Ansgar Fehnker, Thomas Hune, Paul Pettersson, and Judi Romijn. As Cheap As Possible: Efficient Cost-Optimal Reachability for Priced Timed Automata. In Gérard Berry, Hubert Comon, and Alain Finkel, editors, *Procs. of the 13th Int. Conf. on Computer Aided Verification (CAV 2001)*, volume 2102 of *LNCS*, pages 493–505, Paris, France, July 2001. Springer.

[13] E. A. Lee and D. G. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Transactions on Computers*, C-36(1):24–35, Jan 1987.

[14] P. K. Murthy, S. S. Bhattacharyya, and E. A. Lee. Joint minimization of code and data for synchronous dataflow programs. *Formal Methods in System Design*, 11(1):41–70, Jul 1997.

[15] T. C. Ruys. Optimal scheduling using branch and bound with SPIN 4.0. In T. Ball and S. K. Rajamani, editors, *10th Int. SPIN Workshop on Model Checking Software*, volume 2648 of *LNCS*, pages 1–17, Portland, Oregon, May 2003. Springer.

[16] Theo C. Ruys and Ed Brinksma. Experience with Literate Programming in the Modelling and Validation of Systems. In Bernhard Steffen, editor, *Procs. of the 4th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'98)*, volume 1384 of *LNCS*, pages 393–408, Lisbon, Portugal, April 1998.

[17] S. Stuijk, M. C. W. Geilen, and T. Basten. Exploring trade-offs in buffer requirements and throughput constraints for synchronous dataflow graphs. In *43rd Design Automation Conf. (DAC)*, pages 899–904, San Francisco, California, Jul 2006. ACM.

[18] S. Stuijk, M. C.W. Geilen, and T. Basten. SDF3: SDF for free. In *6th Int. Conf. on Application of Concurrency to System Design (ACSD)*, pages 276–278, Turku, Finland, Jun 2006. IEEE Computer Society.

[19] M. H. Wiggers, M. J. G Bekooij, P. G. Jansen, and G. J. M. Smit. Efficient computation of buffer capacities for Multi-Rate Real-Time systems with Back-Pressure. In *4th Int. Conf. on Hardware/Software Codesign an System Synthesis (CODES+ISSS)*, pages 10–15, Seoul, Korea, Oct 2006. ACM.

IX. Appendix

The complete source code of the simple benchmark, which, starting from an initial guess of 5 lowers `__best` each time a schedule is found with a better bound. The assignment `first=false` in `UPDATE` can be optimised away at the expense of a longer and less readable model.

```
c_state "int __best = 5" "Hidden"

#define MAX(a,b) (a>b->a:b)
#define SUM      (ch[0]+ch[1])
#define WORSE    (c_expr{(now.maxsum)>=__best})
#define UPDATE   first=false; \
                 maxsum=MAX(maxsum,SUM)

#define PRODUCE(c,n) ch[c] = ch[c] + n
#define CONSUME(c,n) ch[c] = ch[c] - n
#define WAIT(c,n)    ch[c]>=n

byte ch[2], maxsum;
bool first=true;

init{
end:
   do
   :: atomic{
      (!first&&(ch[0]==0&&ch[1]==0))->break;
   }
/* Actor_c */
   :: atomic{
      WAIT(1,2) ->
      CONSUME(1,2);
      UPDATE;
   }
/* Actor_b */
   :: atomic{
      WAIT(0,3) ->
      CONSUME(0,3);
      PRODUCE(1,1);
      UPDATE;
   }
/* Actor_a */
   :: atomic{
      PRODUCE(0,2);
      UPDATE;
   }
   od;
   c_code{\
      if( now.maxsum < __best ) {\
         __best = now.maxsum;\
         printf( ">best now: %d\n",__best);\
         putrail();\
         Nr_Trails--;\
      }\
   };
}

never{ /* !<> WORSE */
accept_init:
   if
   :: (! (WORSE)) -> goto accept_init
   fi;
}
```

The bash script shown below runs SPIN iteratively, starting from the initial `guess`, and incrementing the `guess` by `step`, until a feasible schedule is found as indicated by the presence of a trail file. Note that that the verifier `pann.c` is compiled only once.

```
spin -a ${prom_file}

# add -#N option to pan to initialise __best
sed -e "/default : usage(efd); break;/i\
case '#': __best = atoi(&argv[1][2]);\
          break;" < pan.c > ppan.c

# note: ppan is now the verifier
gcc -o ppan -DSAFETY ppan.c

while [ ! -e "$trail_file" ]; do
   out_file=${prom_file}_${2}_${guess}.log
   echo "now try __k = ${guess}"
   echo "    file=${out_file}"
   time ./ppan -\#${guess} -c0 -E \
               -w24 -m100000 \
        > ${out_file} 2>&1
   guess=$((guess+step))
done
```

For a full explanation of the mechanisms used please consult Ruys [15].

A Temporal Language for SystemC

Deian Tabakov
Rice University Computer Science
6100 Main Str. MS-132
Houston, TX 77005-1892
dtabakov@cs.rice.edu

Moshe Y. Vardi
Rice University Computer Science
6100 Main Str. MS-132
Houston, TX 77005-1892
vardi@cs.rice.edu

Gila Kamhi
Design Technology and Solutions Division
Intel Corporation
Haifa, Israel
gila.kamhi@intel.com

Eli Singerman
Design Technology and Solutions Division
Intel Corporation
Haifa, Israel
eli.singerman@intel.com

***Abstract*—We describe a general approach for defining new temporal specification languages, and adopting existing languages, for SystemC. We define the concept of "underlying trace" describing the execution of a SystemC model, and then define a set of important primitive assertions about the states in the trace. Our framework not only provides additional expressive power for making atomic assertions, but also provides very fine control over the temporal resolution of the language. Using the primitives defined here as clock expression allows sampling at different levels, from transaction-level to the level of individual statements. The advantage of our approach is that it defines important SystemC properties that have been overlooked previously, and also provides a uniform mechanism for specifying the sampling rate of temporal languages.**

I. Introduction

The increasing complexity of current hardware designs and the need for efficient development of systems-on-chip has motivated a gradual migration away from RTL design and toward more abstract approaches. SystemC[1] has become a successful modeling language partly because it allows designers to model systems at several abstraction levels, from the most concrete (gate level) through the most abstract (system level) [16], as well as to interconnect components from different abstraction levels. In addition to being a modeling language, SystemC is also a simulation environment, driven by a *simulation kernel*. The kernel schedules processes and updates the values of signals and channels in a fashion that mimics concurrent execution, even though in reality the processes are run sequentially. This makes SystemC a particularly useful platform for prototyping and testing hardware and hybrid systems early in the design process [8].

SystemC is built as a library extending C++. The core language provides macros for modeling the fundamental components of the system, for example, modules and channels. The object-oriented encapsulation of C++ and its inheritance capabilities help make designs modular, which, in turn. facilitates reuse and makes IP transfer possible [6]. Various libraries provide further functionality. For example, a popular library called TLM (short for Transaction-Level Modeling) defines channels, interfaces, and protocols that streamline and standardize the development of high-level models in which complex communication and protocols are reduced to a single "transaction". These factors have helped propel SystemC as a *de facto* industry-wide standard modeling language, less than a decade after its first release.

The growing popularity of SystemC has motivated an explosion of research efforts aimed at the verification of SystemC models. Most verification efforts for SystemC so far have been focused on *dynamic verification* (also called *testing* and *simulation*). This approach involves executing the model under verification (MUV) in some environment, while running checkers in parallel with the model. The checkers typically monitor the inputs to the MUV and ensure that the behavior or the output is consistent with the expected behavior or output. The complementary approach of *formal verification* either produces a mathematical proof that the MUV correctly implements a specified property, or returns a counterexample, which is a trace that violates the property. Formal verification approaches have received less attention [30], mostly because they are currently applicable only to rather small designs [22], [24], [25] or simple gate-level models [11], [15].

Assertion-based verification (ABV) has recently been gaining acceptance as an essential method for validation of hardware and hybrid models [10]. With ABV, the designer writes properties that capture the design intent in a formal language, e.g., PSL[2] [14] or SVA[3] [31]. The design then can be verified against the properties using dynamic or formal verification techniques. A successful ABV solution requires two components: a formal declarative language for expressing properties, and a mechanism for transforming the specifications into monitors [10]. Many approaches to SystemC specification are limited to simple invariance assertions (using, for example,

Work partially supported by a gift from Intel.

[1]IEEE Standard 1666-2005

[2]IEEE Standard 1850-2007

[3]IEEE Standard 1800-2005

978-1-4244-2735-2/08/$25.00 ©2008 IEEE

C++'s `assert`) [16], but the industry has recently recognized the need for temporal languages that can express properties related to ongoing behavior, such as p must hold until q is true [3].

There have been a few attempts to adapt temporal languages to SystemC. Ecker et al. [13] describe an implementation of a SystemC Assertion Library inspired by Accellera's Open Verification Library. The library defines 11 properties, most of which are invariance properties and a few are simple temporal templates. The library does not provide a mechanism for defining new temporal templates. Große and Drechsler [11], [15] use a C++ representation of bounded LTL formulas, which are then compiled together with the SystemC model. Their approach is limited to gate-level models. Traulsen et al. [29] translate SystemC models into Promela models, which enables them to use the model checker Spin to verify LTL properties. Habibi, Gawanmeh, and Tahar advocate using PSL [17] and SVA [18] for SystemC, but do not propose an adaptation.

Karlsson et al. [20] use Petri-nets [26] to create a formal representation of the SystemC model at the statement level (i.e., each statement is represented by one place and one transition). Ecker et al. [12] propose an extension of PSL and SVA to express properties of `sc_events`. A disadvantage for both of these two approaches is that the formal model has one level of abstraction. Pierre and Ferro's framework [27] samples at the statement level, and then only considers those states which are relevant to the property, thus providing a somewhat more flexible temporal resolution. This approach shares the same drawback as all existing approaches: the state of the library code and the state of the simulation kernel are not taken into consideration.

The 1850 PSL Working Group proposed adding a "SystemC flavor" to PSL [2]. Most of the changes in this flavor of PSL are cosmetic; the only significant addition is allowing SystemC's event expressions to be used as clock expressions in PSL. A more serious industrial effort for developing temporal languages for SystemC is by Jeda Technologies [21]. They describe two languages for SystemC inspired by SVA. NSCa is an adaptation of SVA and is aimed at cycle-level verification. NSCv is a variant aimed at transaction-level verification.

In our opinion, current works on temporal languages for SystemC are lacking in three major respects. First, none of these works addresses the most fundamental issue for temporal languages, which is a precise definition of a trace. The notion of a trace is a key notion in any discussion of temporal semantics. Hardware-oriented languages such as PSL or SVA assume an underlying notion of a clock-cycle-level trace, but for higher-level languages such as SystemC the notion of a clock- cycle-level trace may not be appropriate. Second, a temporal language should be adaptable to different levels of abstraction in the design of the model. One of the strengths of SystemC is modeling at different levels of abstraction; during the design process the model typically gets refined, evolving from a system-level model to a gate-level model. Jeda Technologies' solution of different languages for different levels of abstractions is simply not flexible enough. Third, the existing specification languages are approaching the issue mostly from a hardware perspective and are ignoring the fact that a SystemC model is, fundamentally, a C++ program. There is a large body of work on specification and model checking of Java, C++ and C code (e.g., [4], [5], [7], [9], [19]) and the specification primitives used there should be adapted for SystemC.

In this paper we describe a new approach to defining temporal languages for SystemC. Our starting point is a precise definition of a SystemC trace. Intuitively, a trace is a sequence of states in the execution of the model. Defining this notion precisely for SystemC is quite nontrivial. First, we abstract the simulation kernel and define its state with respect to this abstraction. Second, we recognize that one needs to distinguish between the SystemC model developed by the designer and the set of SystemC libraries used by this model. While the state of the model is fully detailed, the libraries are modeled only at the level of their exposed interfaces. Finally, we define the notion of a trace with respect to these abstractions of the kernel and the libraries.

We then argue that SVA and the SystemC flavor of PSL fail to identify important Boolean properties relevant to the execution of SystemC models. We propose enriching the Boolean layer with a new set of atomic properties, thereby making the specification language more expressive. First, we extend PSL and SVA with a set of atomic propositions that are adequate for expressing properties of C++ programs. For example, we add propositions that enable us to express pre- and post-conditions for functions. Second, we extend PSL and SVA with propositions that enable us to refer to the current phase of the simulation execution, corresponding to our abstraction of the simulation kernel. Finally, we observe that the clock-sampling mechanism available in PSL and SVA offers us a way to express temporal properties at different levels of abstractions. A fine sampling would correspond, say, to subcycle-level abstraction, while coarser sampling would correspond, say, to transaction-level abstraction. Since in PSL and SVA any Boolean expression can be used as a clock expression, the additions to the Boolean layer that we propose here also provide us a much finer control over the temporal resolution. We show that this approach enables us to tailor temporal languages to SystemC in a uniform way; all that is needed is to adapt the underlying syntax for state assertions. The main feature of the resulting framework is the ease with which properties can be expressed at different levels of abstractions, without having to use different languages.

II. Semantics of SystemC Simulation

SystemC was originally developed for the specification of circuit models that could be compiled using a regular C++ compiler and simulated efficiently [22]. As the language evolved it changed its primary goal to enable system-level modeling, that is, modeling of systems that might be implemented in software, hardware, or a combination of the two (e.g., system-on-a-chip). One of the strengths of SystemC is that it can handle different models of computation

and communication, levels of abstraction, and system design methodologies. This is achieved by a layered approach where high-level constructs share an efficient simulation engine [16].

At the base layer SystemC provides an event-driven *simulation kernel*. *Modules* and *ports* represent structural information, and *interfaces* and *channels* abstract communication. The behavior of a module is specified by defining one or more *processes*. Each process can declare a *sensitivity list*: a number of *events* that trigger its execution. A *waiting* process becomes *runnable* when one or more of the events on its sensitivity list has been *notified*. If there are several processes that are runnable, the kernel arbitrarily selects one of them and gives it execution control. The simulation semantics imposes *non-preemptive execution* of processes, that is, once the kernel gives a process execution control the kernel cannot take it back until the process finishes executing or explicitly suspends itself by calling `wait()`.

Like VHDL and Verilog, the SystemC kernel supports the notion of a delta-cycle. The simulation clock does not advance during a delta-cycle, and as a result all processes that execute during the delta-cycle appear to be executing simultaneously. In order to maintain the appearance of parallel execution it is also necessary to postpone the effect of all channel and signal updates and event notifications. To that end, during a delta-cycle the kernel switches from *evaluation* phase to *update* phase to *delta-notification* phase. During the evaluation phase any values written to a channel or a signal are not immediately available, and the value of the channel or signal is not updated until there are no more runnable processes and the kernel enters the update phase. Likewise, during the evaluation phase an event might be notified but the processes sensitive to that event will not become runnable until the delta notification phase. An exception to this rule are *immediate notifications*, which cause dependent processes to be added to the pool of runnable processes during the evaluation phase, rather than waiting until the delta notification phase.

The simulation semantics of SystemC, which is defined in [1], is presented in pseudo code in Figure 1.

The execution of a SystemC application starts with the *Elaboration phase*, during which all modules are instantiated and channels are bound to ports, and some channels may register a `request_update()` (line 7). Then the kernel enters the *Initialization phase* (lines 8–20). During the Initialization phase all channels with pending updates are updated (lines 8–10), all initializable `SC_THREAD`s and `SC_METHOD`s are made runnable (lines 11–15), and pending delta notifications cause their dependent processes to become runnable (lines 16–20). Next the kernel starts a delta-cycle and runs all runnable processes one at a time (*Evaluation phase*, lines 23–26). During this phase pending channel updates are collected in U, and pending event notifications are collected in D. The evaluation phase is followed by an update phase (lines 27–29) where all collected channel update requests are executed and writes to signals take effect. After that the kernel enters the delta-notification phase (lines 30–34) where notified events trigger their dependent processes. Note that immediate notifications may make new processes runnable during the execution of lines 22-26.

If at this point there are runnable processes the kernel loops back to line 22 and starts another evaluation phase and a new delta-cycle. Alternatively, if there are no more runnable processes, the kernel advances the simulation clock to the earliest timed-delay notification (essentially, a notification that

```
1:  PC ← all primitive channels
2:  P ← all processes
3:  R ← ∅ /* Set of runnable processes */
4:  D ← ∅ /* Set of pending delta notifications */
5:  U ← ∅ /* Set of update requests */
6:  T ← ∅ /* Set of pending timed notifications */
7:  /* Start elaboration; collect all update requests in U */
8:  for all chan ∈ U do
9:     run chan.update()
10: end for
11: for all p ∈ P do
12:    if p is initializable and p is not clocked thread then
13:       R ← R ∪ p /* Make p runnable */
14:    end if
15: end for
16: for all p ∈ P do
17:    if p is triggered by an event in D then
18:       R ← R ∪ p /* Make p runnable */
19:    end if
20: end for /* End of Initialization phase */
21: repeat
22:    while R ≠ ∅ do /* New delta-cycle begins */
23:       for all r ∈ R do /* Evaluation phase */
24:          R ← R \ r
25:          run r until it invokes wait() or returns
26:       end for
27:       for all chan ∈ U do /* Update phase */
28:          run chan.update()
29:       end for
30:       for all p ∈ P do /* Delta notification phase */
31:          if p is triggered by an event in D then
32:             R ← R ∪ p /* p is now runnable */
33:          end if
34:       end for /* End of delta-cycle */
35:    end while
36:    if T ≠ ∅ then
37:       Advance the clock to the earliest timed delay t.
38:       T ← T \ t
39:       for all p ∈ P do /* Timed notification phase */
40:          if t triggers p then
41:             R ← R ∪ p /* p is now runnable */
42:          end if
43:       end for
44:    end if
45: until end of simulation
```

Fig. 1. Simulation Semantics of SystemC

is explicitly set to be notified after some delay). All processes sensitive to this event are triggered (lines 39–43) and the kernel loops back to line 21 and starts a new delta-cycle. This process is repeated indefinitely, unless the designer has specified a fixed simulation time or all processes have terminated.

III. Definition of Execution Trace

All (linear) temporal languages are interpreted over execution traces, therefore before we can define the semantics of temporal properties for SystemC we need a precise definition of an execution trace. Traditionally, a trace has been defined as a sequence of states in the execution of the model, but there has been remarkably little discussion in the literature about the definition of SystemC traces.

Some existing approaches adopt a clock-cycle-level temporal resolution, so implicitly a state is a valuation of all variables at the boundaries of clock cycles, and the trace consists of the sequence of such valuations [21]. We believe that such an approach is inadequate for SystemC, because it fails to take into account the unique simulation semantics of SystemC, which allows for a much finer grained temporal resolution. For example, algorithmic-level SystemC models are often timeless, with the simulation being completely driven by events and the simulation clock making no progress during the whole simulation [16], [24]. In fact, the whole simulation can consist of a single delta cycle, if the simulation is driven solely by immediate event notifications. Thus, clock-cycle-level temporal resolution is clearly inappropriate for such models. In this section we give a more refined definition of SystemC traces that accounts for SystemC's simulation semantics. Our definition starts with a precise definition of a system state, which encompasses the state of the kernel, the state of the user model, and the state of the external libraries.

A. Kernel State

1) Kernel Phases: It is not immediately clear why the state of the kernel needs to be included in the execution trace. For example, in the work of Kroening and Sharygina [22] the kernel is abstracted away completely. Each process is modeled as a labeled transition system, and the global system is defined as a product of these local transition systems. The transitions of the global system are defined according to the simulation semantics, which requires that "components must synchronize on shared actions and proceed independently on local actions" [22]. Under this model synchronization occurs when a process encounters a `wait()` or a `notify()` instruction. The observable behavior of their abstraction of execution matches well the execution of a SystemC model. Thus, on the surface, it may seem that taking into account the state of the kernel would only complicate the semantics.

Similar philosophy has been adopted by Karlsson et al. [20], Ecker et al. [12], and Pierre and Ferro [27], which likewise do not model the kernel.

This may sound reasonable at first, but one soon realizes that many important properties require some knowledge of the state of the kernel. A consistency property may be required to hold

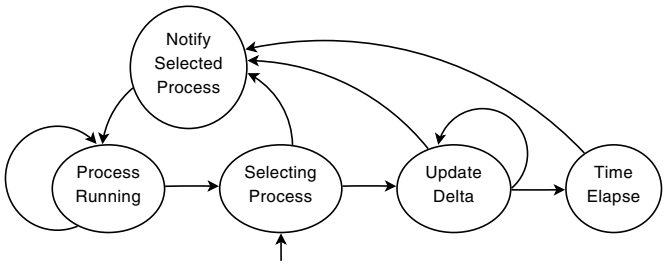


Fig. 2. Kernel states proposed by Moy et al.

all the times, at the evaluation-phase boundary, at the delta-cycle boundary, or at a timed-cycle boundary. If the kernel is abstracted away completely, then there is no way to make these distinctions and specify the consistency requirement properly. We conclude, therefore, that the state of the kernel must be exposed to a certain extent, in order to enable the user to specify properties at different levels of abstraction. (We do not discuss in detail here how such exposure is to be implemented; that may depend on the context. For example, in the context of dynamic verification such exposure can be implemented by extending the kernel with an API for querying the state of the kernel.)

This approach of exposing the state of the kernel to some extent was taken by Moy et al. [24], [25]. Their work formalizes SystemC models in terms of communicating state machines, where the kernel is modeled as a particular state machine (Figure 2). Thus, the state of the kernel is exposed at an abstraction level corresponding to this specific state machine.

Once one accepts the principle of exposing the kernel state, the question remains at what abstraction level to expose the kernel. Moy et al. offer a specific abstraction, but their choice is open to criticism. Their formalism is somewhat less detailed than the simulation semantics in SystemC's Language Reference Manual (LRM) [1]. One could offer other abstractions of the kernel, but without some guiding principle such abstractions are also open to criticism. Our guiding principle is that the abstraction should abstract away the kernel implementation, but expose fully SystemC's simulation semantics, as described in [1] and Figure 1. A coarse abstraction might hide details that may be of importance to some users. Thus, an abstraction at the level of the simulation semantics is as generic as possible, enabling further abstraction if required by specific applications.

We therefore abstract the simulation semantics, as described in Figure 1, by the state machine described in Figure 3. Our abstraction may seem, at first sight, to be somewhat too detailed. A simpler abstraction of the kernel would consider each phase (*Initialization*, *Evaluation*, *Update*, *Delta notification*, *Timed notification*) as a separate state in a state machine. Why does our model have more states? The answer is that the simpler model does not expose the start and the finish of individual processes (all processes are run while the kernel is in the Evaluation phase, and similarly for the Update phase). Our abstraction exposes the transfer of control

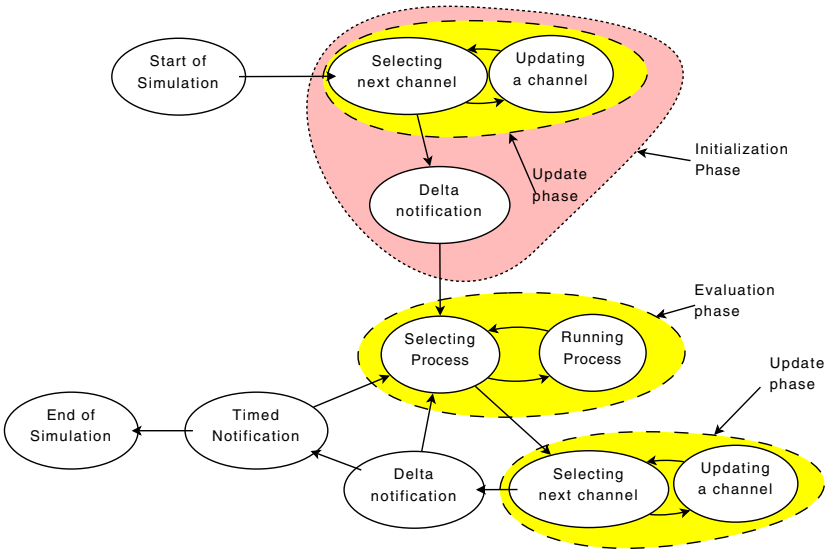


Fig. 3. Proposed Kernel States

between the kernel and the processes by splitting the Update end Evaluation phases into two subphases. Since SystemC uses the interleaving approach to concurrency, we believe that exposing transfer of control is important. Our abstraction of the kernel is more detailed than Moy et al.'s, therefore giving us more expressive power.

One might argue that keeping track of all phases of the simulation semantics is unnecessary because very few properties relate to those specific phases. We decided to take a generic approach and model all phases of the kernel that are described by the simulation semantics rather than try to pass judgment on which ones are the most important. As we show in the sequel, our approach enables users to use coarser abstractions if needed. Since we need to anticipate all possible uses of SystemC, exposing the semantic fully is the most justifiable approach.

2) SystemC Events: SystemC events are objects derived from the pre-defined class `sc_event`. A particular "waiting" process does not becomes "runnable" until the event on which the process is waiting is *notified*. For example, if a TLM channel is full, a thread that wishes to write to the channel may suspend itself by calling `wait(ok_to_put)`. As soon as there is free space on the channel, the channel notifies the `ok_to_put` event, and the waiting thread is moved to the pool of "runnable" processes among which the kernel selects the next process to run. Most core SystemC objects have an associated event that indicates that some change has occurred. For example, an `sc_signal` has an event that is notified when the signal changes; an `sc_fifo` has an event for writing to and an event for reading from the channel; an `sc_clock`'s positive and negative edges are represented by events. Thus, events are the fundamental synchronization mechanism in SystemC, and keeping track of when a particular event is notified allows us to pinpoint the instant in time when something important happens. In the particular example mentioned before, we might want to specify that every time `ok_to_put` is notified the number of items in the channel is strictly smaller than the capacity of the channel.

Events are notified by calling the `notify()` method of class `sc_event`. There are three types of event notification:

1) `notify()` with no arguments: immediate notification. Notification happens upon execution.
2) `notify(SC_ZERO_TIME)`: delta notification. Notification happens during the delta-notification phase.
3) `notify(`*`time`*`)` with a non-zero time argument: timed notification. Notification happens during a subsequent timed-notification phase.

Pending event notifications can be canceled using the `cancel()` method, pending timed notifications are canceled by delta notifications, and pending delta notifications are canceled by immediate notifications.

PSL's and Jeda's treatment of events is to allow them to be used as clock expressions, though the issue when events are actually notified is not discussed explicitly. We believe that the fundamental role played by events in the execution of SystemC models justifies fully exposing event notification in the kernel's state. In essence, we are elevating event notification to the Boolean layer. This means that properties can refer directly to event notification, for example, specifying that `ok_to_put` is notified at least once every clock cycle.

B. User Model State

The state of the user model is the full state of the C++ code of all processes in the model, which includes the values of the variables, the location counter, and the call stack. Our perspective is that of "white-box validation", which means that the state of the model should be fully exposed. Thus, the property languages should consider all class members to have **public** access, including those that are declared as **private** or **protected**.

As argued earlier, we believe that a property specification language for SystemC ought to consider SystemC as a system-level language, rather than a hardware-level language. This requires that the execution of a SystemC model be exposed also at the source-code level [5]. This should enable us to refer explicitly to statements being executed both via their label as well as by their syntax; for example, we should be able to refer to all invocations of a specific method. Formally, we add to the state a variable of type `string` that contains as a string the statement being executed at a given step. Furthermore, as method invocation is central to the execution of object-oriented code, the values of arguments passed to and returned from invoked methods should also be exposed, by exposing the values of the formal parameters of the method upon invocation. In essence, we are requiring traces of SystemsC model to include both state and transition descriptions, in contrast to standard models of temporal logics that are typically state based [28]. By exposing both semantics and syntax of the model, we enable properties that relate source code and execution; for example, we can specify that for every port connected to a specific channel, the `write()` interface method call is passed only positive numbers as arguments. In order to simplify the process of referring to individual statements in what could be many thousands of lines of code, we provide several predefined labels for important locations. This is discussed further in Section IV.

Finally, like Moy et al., we expose for each process its status, as *waiting*, *runnable*, or *running*, corresponding to the simulation semantics, per Figure 3.

C. Library Code State

Most SystemC models rely heavily on external libraries (the TLM library, for example). These libraries encapsulate crucial components of SystemC models, such as signals and channels. When formalizing the notion of SystemC state, we need to decide how to formalize the state of libraries. One approach would be to extend the white-box approach to library code, but users need to be familiar with libraries only at the API level, and not at the implementation level. Furthermore, while in many cases the code of the library is available, in others the library may be supplied as compiled code, thereby hiding the internal state.

Kroening and Sharygina [22] do not discuss how they handle library code. Moy et. al's approach [23], [25] is to provide specific state-machine models reflecting the functionality of TLM constructs. The benefit of this approach is that it preserves important information about the structure and behavior of the design. A major drawback is that this requires manual effort to develop formal models for libraries. These formal models may have to be revised when libraries are revised.

To attain generality, we believe that library code should be treated as a black box. For example, when it comes to `tlm_fifo`, the state of the queue should be exposed without exposing implementation details. Furthermore, the state of a library should be exposed only in terms of the API of that library. Consider, for example, the TLM 1.0 library. Properties should not have access to the state of the `tlm_fifo` other than via side-effect-free function calls, for example, via the `peek` method. Of course, when library source code is available, users can choose to treat it as a part of the user model and view it from a white-box perspective.

D. Trace

A SystemC trace is a sequence of states corresponding to the execution of the model. Such execution consists of an alternation of control between the kernel, on one hand, and the model and the libraries on the other hand. We have discussed so far how we formalize the state of the kernel, the model, and the libraries. It remains to discuss at what level of granularity we formalize the transition from one state to its successor.

When the kernel is executing, we follow transitions in the state machine described in Figure 3. When the kernel selects a process to run or a channel to update, control passes to that process, which then runs until it terminates or is suspended via a `wait` statement. With respect to transitions of processes, we follow the "large-step semantics" approach [32]. Under this approach we focus only on the overall effect of each statement, as opposed to considering the individual subexpressions. For example, `y = x++;` consists of two subexpressions (`y = x;` and `x = x + 1;`), but we ignore the valuations of the variables during the execution of the subexpressions. We believe that this matches the level at which programmers and verification engineers think about the source code. By following large-step semantics our framework may miss rare cases where a property is violated in a subexpression. For example, suppose that a program invariant requires that x must always be positive, and suppose that $x = 1$. During the execution of the expression `y = (x--) + (x++);` the value of x is temporarily set to 0 by the `x--` subexpression, but since the value of x is restored back to 1 by the `x++` subexpression, no violation of the property will be reported. Modern design practices discourage the use of complex subexpressions that change the valuation of variables, therefore the choice of large-step semantics over small-step semantics is justified.

Finally, we consider each invocation of a library method, for example, invoking a channel-interface method, to return in one step. This is consistent with our black-box view of libraries.

IV. Language Definition

After we have given a definition of an execution trace we can define a set of specification primitives for SystemC models. Notice that specification languages like PSL and SVA are already quite expressive temporarily, so no extension is necessary for their temporal layer in SystemC. Thus, our focus in this section is on extending the Boolean layer. The new set of primitives introduced here not only allows the specification of a richer set of properties, but also provides a uniform mechanism for controlling the temporal resolution of the specification language. Unlike Kasuya and Tesfaye's approach [21], which requires a separate language for clock-based and event-based models, our framework allows for greater

flexibility in the temporal granularity of the specification, and is general enough that it can readily be adapted for any temporal language with the notion of clock expressions.

Table I summarizes our proposed specification primitives.

SystemC_expr	::=	*model_expr* | *kernel_expr*
model_expr	::=	*loc_expr* | *arg_expr* | *proc_expr*
loc_expr	::=	[**before** | **after**] {code_label | *syntax_expr* } | *func_name:*{**entry** | **exit** | **call** | **return**}
syntax_expr	::=	$function_{\text{String} \rightarrow \text{Boolean}}$ (**curr_statement, ...**)
arg_expr	::=	*func_name* **:** non-negative_integer
proc_expr	::=	*proc_name*.**proc_state**
kernel_expr	::=	*phase_expr* | *event_expr*
phase_expr	::=	**kernel_phase**
event_expr	::=	*event_name*.**notified**

TABLE I
PROPOSED SPECIFICATION PRIMITIVES

A. User Model Primitives

A *model_expr* is a Boolean expression about the state of the user model. Under this category we include expressions about the location counter (*loc_expr*), the arguments and return values of functions (*arg_expr*), and expressions about the state of each process (*proc_expr*).

Our definition of a trace explicitly keeps track of the location counter during the execution of the user model. With each label `code_label` in the source code of the model we associate a Boolean variable that is true precisely in those states where the statement that is about to be executed is labeled by `code_label`. We introduce two optional modifiers, **before** and **after**, to refer, respectively, to the state immediately before and immediately after that statement is executed (the default behavior corresponds to specifying **before**). Using this primitive allows the specification of forbidden or mandatory paths in the execution of the compiled model, e.g., if execution reaches `lbl1` it should not reach `lbl2`. It also allows the specification of properties that must hold at particular locations in the code.

Inspired by BLAST [5], we also propose adding a primitive **curr_statement** of type `string` that exposes the syntax of the statement that is about to be executed. As mentioned earlier in Section III-B, this mechanism enables properties that relate syntax and semantics. PSL and SVA allow using functions defined in the underlying HDL language, which in the context of SystemC means that we can use a number of C++ functions that operate on strings and return Booleans (*syntax_expr*) and pass **curr_statement** as an argument. Of particular interest are regular expression matching and string comparison functions, because they allow the user to quickly identify a set of "important" locations in the source code without having to introduce labels manually. As an example, one can use this mechanism to identify all locations in the user model where a particular statement is executed.

Example 1 (Matching source code): Suppose that before a function *foo()* is called in any module, some consistency condition *bar* must hold. For this property we first define a function *match(string)* that returns true whenever the string contains "foo(", e.g. via pattern matching. The property then becomes `(before match(curr_statement)) -> bar`.

The specification of pre- and post-conditions requires evaluating assertions at specific locations in the source code that are difficult to identify automatically via the mechanisms described so far. Inspired by SLIC [4], we introduce two additional primitives, **entry** and **exit**, that refer to the location immediately before the first executable statement, and the location immediately after the last executable statement, in a function. In some cases the pre-condition may need to refer to the values of the formal parameters passed on to the function. If the function is a part of the user model, one can use the names of the variables on the parameter list. We also propose an alternative mechanism (previously used in both BLAST [5] and SLIC [4]) to refer to the value of each parameter according to its order. For a function *func(type1 param1, type2 param2, ...)*, we define implicit variables *func:1*, *func:2*, etc., whose values (and types) are equal to the values (and types) of the formal parameters of the function at the entry point (i.e., the values of the variables before the first statement in the function has been executed).

Example 2 (Precondition): One desirable precondition for a function *long_division(double dividend, double divisor)* is `ALWAYS (long_division:entry -> long_division:2 != 0)`.

This mechanism is inadequate for the specification of pre- and post-conditions of functions defined in a proprietary library because the source code is not exposed in the execution trace. For cases like this we introduce another set of primitives that we adopt from SLIC [4]. For each function call to *func()* we introduce the primitives *func*.**call** and *func*.**return** to refer, respectively, to the location in the source code that contains the function call and to the location immediately after the function call. (Note that here we assume that function calls are not nested.) The values of the arguments can be accessed via implicit variables *func:1*, *func:2*, etc., whose types match the types of the arguments to the function, and whose values are precisely the values of the actual parameters at *func*.**call**. Another implicit variable, *func:0*, is defined as the value returned by the function, and it is only defined at *func*.**return**. This mechanism allows the specification of properties of proprietary functions and objects even if the library does not expose their states directly (e.g., a proprietary channel). For example, we can ensure that a channel contains only positive values by specifying that the arguments to all relevant calls to `write()` are always positive. As a second example, we can express the property that the channel behaves like a queue by using PSL's modeling layer to temporarily remember two values written to the channel, and then verifying that the values are returned in the same order via the channel's `read()` method.

Finally, we propose adding a primitive **proc_state** that for each process name returns a value in the enumerated set { *waiting*, *runnable*, *running* } depending on the status of the process in the kernel. One can use this primitive to specify that a particular process is executed infinitely often, or is executed

at least once during each delta cycle.

B. Kernel Primitives

A *kernel_expr* is an expression about the state of the kernel. We introduce primitives for exposing the current phase (*phase_expr*) and when events are notified (*event_expr*).

When the kernel has the thread of control, the execution trace makes transitions that reflect the changing phases of the kernel. We propose adding a primitive **kernel_phase** that exposes the current phase. The primitive returns a value in an enumerated set { *startsim*, *init_select_channel*, *init_update_channel*, ..., *endsim* }, corresponding to our abstraction of the kernel in Figure 3. **kernel_phase** allows the user to define properties whose evaluation is triggered by different phases of the kernel.

Example 3 (Stable states): Variable p in process `proc` in module `mod` must be 0 in all stable states (i.e., states where no process is executing): `ALWAYS (! kernel_phase = eval_run_proc -> mod.proc.p = 0)`. The kernel phase `eval_run_proc` corresponds to the instances where a process is running (line 25 in Figure 1) in the evaluation loop (lines 23–26 on Figure 1).

Event notifications (*event_expr*) allow us to detect when the notifications actually take place. Note that the mechanisms described earlier expose function calls at the source-code level, and event notification requests and cancellations (i.e., calls to `notify()` and `cancel()`) are exposed via the user model primitives. However, these primitives do not expose the particular state when the actual notification is carried out (i.e., when the dependent processes are made *runnable* by the kernel). For each event we propose a primitive **notified** which is true whenever the kernel carries out the actual notification. For immediate notifications this happens concurrently with the function call to `notify()`; for delta-delayed notifications it happens during the earliest delta-notification phase; for time-delayed notifications it happens during the corresponding timed-notification phase. Note that both delta-delayed and time-delayed notification requests can be subsequently canceled, therefore a call to `notify()` with a non-negative argument does not guarantee that **notified** will be true in the future. The role of this primitive is particularly important when referring to events that are notified implicitly, e.g. when an `sc_signal` changes value, a built-in event named `value_changed_event` is notified implicitly by the kernel in the delta notification phase that immediately follows.

Example 4: The requirement that a signal changes in every delta cycle can be expressed as `ALWAYS (kernel_phase = delta_notify -> signal.value_changed_event.notified)`.

Example 5: Variable p in process `proc` in module `mod` must be 0 at the rising edge of clock `cl`: `ALWAYS (cl.posedge.notified -> mod.proc.p = 0)`.

C. Using Primitives as Clock Expressions

So far we have not discussed how the primitives described here can be used to control the temporal resolution of the specification language. Existing languages like PSL and SVA allow the use of "clock expressions" (CE), which are Boolean expressions that indicate when a state in the execution trace should be sampled. Traditionally (e.g., [14], [21], [31]) sampling is done at the boundary of clock cycles. Our framework can easily provide the same functionality by using the event notification primitive described earlier. Note that an `sc_clock` exposes two events, `posedge_event` and `negedge_event`, which are notified every time the value of the clock changes and the new value is, respectively, 1 and 0. Using `posedge_event.notified` and/or `negedge_event.notified` as CEs we can sample at the boundaries of half-clock or clock cycles. Clearly, we are not limited to the simulation clock. If finer grained resolution is required, one can sample at the boundary of delta cycles by using (**kernel_phase** = delta_notification) as a CE (sampling at the delta notification phases of the kernel), or at the end of execution of each process by sampling at (**kernel_phase** = next_proc_select), which corresponds to the phases where the kernel is selecting the next process to be executed. One can even sample at the boundary of the individual statements in the source code (which is the default sampling rate).

Example 6 (Clock expressions): For this example we borrow some of PSL's syntax. The property "A call to function *req()* is followed within 3 clock cycles (of clock `cl`) by a notification of event `ack`" can be expressed as

```
default clock = cl.posedge.notified;
ALWAYS (req:call -> next[3] ack.notified).
```

If the acknowledgment needs to be received within 3 delta cycles instead, all we need to do is change the clock expression:

```
default clock=(kernel_phase=delta_notify);
ALWAYS (req:call -> next[3] ack.notified).
```

The same mechanism allows specifying coarser sampling rates as well. In a transaction-level or system-level model one is typically interested in its behavior at event notification instances. Jeda's framework provides this functionality in a separate language (NSCv), but they require the user to setup callbacks (essentially, function calls) that "report an event occurrence at the point of transaction processing" [21]. In our framework this can be done by using as CEs the event-notification primitives introduced earlier. For example, in a TL model we can sample at the instances when the an `sc_fifo` is written to (by sampling at `data_written_event.notified`), or when a signal changes value (by sampling at `value_changed_event.notified`), etc. The advantage of our framework is that it is using the same language throughout the refinement process as the model is transformed from higher to lower levels of abstraction.

V. Discussion

To the best of our knowledge this is the first work that provides a principled discussion and definition of a SystemC execution trace. Our notion of a state encompasses information about the kernel (current phase and notification of events), as well as statement-level information about the user model, and

publicly exposed state of the libraries. The level of details preserved in the states allows us to define a rich set of new properties about the execution of the SystemC model. Moreover, the user can specify a range of sampling rates, from the most coarse (transaction- and system-level) to the most detailed (statement level) by combining clock expressions with the primitives introduced in this paper. Our framework is general enough that it can be adopted by most existing temporal specification languages by simply enriching the set of allowed atomic expressions.

Bringing techniques from software verification to the SystemC world is our second contribution. The fact that SystemC models should be viewed as software models has been ignored so far. The result is a minimal yet highly expressive extension of PSL/SVA.

The framework that we propose is equally applicable to dynamic verification and formal verification. Enabling a dynamic verification path would require a minimal "one-time" addition to SystemC's simulation kernel source code to expose a part of SystemC kernel's internal state and data structures. The user code will have to be instrumented to allow the monitors to observe the behavior of the relevant components, and the monitors will be compiled and executed together with the model. We are currently working on an implementation that automates the process.

Applying formal methods to SystemC is an active area of research with several different approaches (e.g. using communicating state machines [24], [25], Petri-nets [20], or leveraging Promela/SPIN [29]). All of these works propose some FSM-like abstraction of the SystemC kernel, and no two abstractions are the same. The model presented in this paper corresponds directly to the simulation semantics as described in the SystemC LRM [1] and is the most detailed model without making any assumptions about the particular kernel implementation. The FSM in Figure 3 can easily be adopted by existing and future formal verification approaches. Exposing the syntax further allows the analysis of the model from a purely software point of view. The techniques used in SLIC [4] and BLAST [5] can and should be applied to formal verification of SystemC.

REFERENCES

[1] *IEEE Std 1666 - 2005 IEEE Standard SystemC Language Reference Manual*, 2006.

[2] Standard for property specification language (PSL). *IEC 62531:2007 (E)*, pages 1–156, 2007.

[3] R. Armoni, L. Fix, A. Flaisher, R. Gerth, B. Ginsburg, T. Kanza, A. Landver, S. Mador-Haim, E. Singerman, A. Tiemeyer, M.Y. Vardi, and Y. Zbar. The ForSpec temporal logic: A new temporal property-specification logic. In *Proc. 8th ICTACAS*, volume 2280 of *LNCS*, pages 296–211, Grenoble, France, April 2002. Springer-Verlag.

[4] T. Ball and S. Rajamani. SLIC: A specification language for interface checking. Technical report, Microsoft Research, January 2002.

[5] D. Beyer, A. J. Chlipala, T. A. Henzinger, R. Jhala, and R. Majumdar. The BLAST query language for software verification. In *SAS*, pages 2–18, 2004.

[6] A. Bunker, G. Gopalakrishnan, and S. A. McKee. Formal Hardware Specification Languages for Protocol Compliance Verification. *ACM Transactions on Design Autom. of Elec. Sys.*, 9(1):1–32, January 2004.

[7] L. Burdy, Y. Cheon, D. Cok, M. D. Ernst, J. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An overview of JML tools and applications. *Software Tools for Technology Transfer*, 7(3):212–232, June 2005.

[8] L. Charest and E. M. Aboulhamid. A VHDL/SystemC comparison in handling design reuse. In *Proceedings of 2002 International Workshop on System-on-Chip for Real-Time Applications*, pages 79–85, Banff, Canada, July 2002.

[9] James C. Corbett, Matthew B. Dwyer, John Hatcliff, and Robby. Expressing checkable properties of dynamic systems: the Bandera Specification Language. *STTT*, 4(1):34–56, 2002.

[10] A. Dahan, D. Geist, L. Gluhovsky, D. Pidan, G. Shapir, Y. Wolfsthal, L. Benalycherif, R. Kamdem, and Y. Lahbib. Combining system level modeling with assertion based verification. In *ISQED*, pages 310–315, 2005.

[11] R. Drechsler and D. Große. Reachability analysis for formal verification of SystemC. In *DSD*, pages 337–340, 2002.

[12] W. Ecker, V. Esen, T. Steininger, M. Velten, and M. Hull. Specification language for transaction level assertions. *HLDVT*, pages 77–84, 2006.

[13] W Ecker, V Esen, T Steininger, M Velten, and J Smit. Implementation of a SystemC assertion library, 2005.

[14] C. Eisner and D. Fisman. *A Practical Introduction to PSL (Series on Integrated Circuits and Systems)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[15] D. Große and R. Drechsler. Formal verification of ltl formulas for SystemC designs. In *ISCAS (5)*, pages 245–248, 2003.

[16] T. Grotker, S. Liao, G. Martin, and S. Swan. *System Design with SystemC*. Kluwer Academic Publishers, Norwell, MA, USA, 2002.

[17] A. Habibi, A. Gawanmeh, and S. Tahar. Assertion based verification of PSL for SystemC designs. In *International Symposium on System-on-Chip*, pages 177–180, 2004.

[18] A. Habibi and S. Tahar. On the extension of SystemC by SystemVerilog assertions. *Electrical and Computer Engineering, 2004. Canadian Conference on*, 4:1869–1872 Vol.4, 2-5 May 2004.

[19] T. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Software verification with BLAST. In *Tenth International Workshop on Model Checking of Software (SPIN)*, volume LNCS 2648, 2003.

[20] D. Karlsson, P. Eles, and Z. Peng. Formal verification of SystemC designs using a Petri-net based representation. In *DATE '06: Proceedings of the conference on Design, automation and test in Europe*, pages 1228–1233, 3001 Leuven, Belgium, Belgium, 2006. European Design and Automation Association.

[21] A. Kasuya and T. Tesfaye. Verification methodologies in a tlm-to-rtl design flow. In *DAC*, pages 199–204, 2007.

[22] D. Kroening and N. Sharygina. Formal verification of SystemC by automatic hardware/software partitioning. In *MEMOCODE*, pages 101–110, 2005.

[23] M. Moy, F. Maraninchi, and L. Maillet-Contoz. LusSy: A toolbox for the analysis of systems-on-a-chip at the transactional level. In *International Conference on Application of Concurrency to System Design*, June 2005.

[24] M. Moy, F. Maraninchi, and L. Maillet-Contoz. LusSy: an open tool for the analysis of systems-on-a-chip at the transaction level. *Design Automation for Embedded Systems*, 2006.

[25] Matthieu Moy. *Techniques and Tools for the Verification of Systems-on-a-Chip at the Transaction Level*. PhD thesis, INPG, Grenoble, France, December 2005.

[26] James L. Peterson. *Petri Net Theory and the Modeling of Systems*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1981.

[27] L. Pierre and L. Ferro. A tractable and fast method for monitoring SystemC TLM specifications. *IEEE Transactions on Computers*, 2008.

[28] A. Pnueli. The temporal logic of programs. In *Proc. 18th IEEE Symp. on Foundation of Computer Science*, pages 46–57, 1977.

[29] C. Traulsen, J. Cornet, M. Moy, and F. Maraninchi. A SystemC/TLM semantics in Promela and its possible applications. In *14th Workshop on Model Checking Software SPIN*, July 2007.

[30] M. Y. Vardi. Formal techniques for SystemC verification. In *DAC '07: Proceedings of the 44th annual conference on Design automation*, pages 188–192, New York, NY, USA, 2007. ACM.

[31] S. Vijayaraghavan and M. Ramanathan. *A Practical Guide for SystemVerilog Assertions*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.

[32] G. Winskel. *The formal semantics of programming languages: an introduction*. MIT Press, Cambridge, MA, USA, 1993.

Augmenting a Regular Expression-Based Temporal Logic with Local Variables

Cindy Eisner
IBM Haifa Research Laboratory
Email: eisner@il.ibm.com

Dana Fisman
Hebrew University and
IBM Haifa Research Laboratory
Email: danafi@cs.huji.ac.il

***Abstract*—The semantics of temporal logic is usually defined with respect to a word representing a computation path over a set of atomic propositions. A temporal logic formula does not control the behavior of the atomic propositions, it merely observes their behavior. Local variables are a twist on this approach, in which the user can declare variables local to the formula and control their behavior from within the formula itself.**

Local variables were introduced in 2002, and a formal semantics was given to them in the context of SVA, the assertion language of SystemVerilog, in 2004. That semantics suffers from several drawbacks. In particular, it breaks distributivity of the operators corresponding to intersection and union. In this paper we present a formal semantics for local variables that solves that problem and others, and compare it to the previous solution.

I. Introduction

The semantics of temporal logic is usually defined with respect to a word representing a computation path over a set of atomic propositions. A temporal logic formula does not control the behavior of the atomic propositions, it merely observes their behavior. Local variables are a twist on this approach, in which the user can declare variables local to the formula and control their behavior from within the formula itself.

Local variables were introduced at the same time independently by Havlicek et al. [11] and by Oliveira and Hu [16]. The latter's work was based on work by Seawright and Brewer [18] that introduced the related concept of *action blocks*. Since then, local variables have made their way into SVA, the assertion language of SystemVerilog [2,14]. The semantics of [2,14] consider a temporal logic based on semi-extended regular expressions, which is at the core of the IEEE standard temporal logics SVA [2,14] and PSL [3,13]. In this paper, we term that core logic DRE (where the D stands for *dynamic modalities* and the RE for *regular expressions*) and use it as our base logic in a formal semantics for local variables that solves serious problems with those of [2,14].

The logic DRE is composed of standalone semi-extended regular expressions (SEREs) and the application of dynamic modalities and Boolean operators to them. The set of SEREs is built from atoms which are Boolean expressions and includes the standard operators concatenation, union, intersection and Kleene closure. If r is a SERE then $r!$ is a DRE formula that holds on words containing a non-empty prefix in the language of r. For example:

$$\{\mathit{true}^* \cdot a \cdot b^* \cdot c^* \cdot d\}! \tag{1}$$

holds on any word containing the letter a followed by some number of b's followed by some number of c's and then a d.

There are two dynamic modalities in DRE: $\mapsto$ (*suffix implication*) and its dual $\diamondsuit\!\!\rightarrow$ (*suffix conjunction*). If r is a SERE and φ is a DRE formula then $r \mapsto \varphi$ and $r \diamondsuit\!\!\rightarrow \varphi$ are DRE formulas.[1] The formula $r \mapsto \varphi$ holds on a word w if for every prefix v of w that is in the language of r, the suffix of w starting from the last letter of v satisfies φ; the formula $r \diamondsuit\!\!\rightarrow \varphi$ holds on a word w if there exists a prefix v of w that is in the language of r and the suffix of w starting from the last letter of v satisfies φ. For example, consider the following DRE formula:

$$\{\mathit{true}^* \cdot a \cdot\ b^* \cdot\ c^* \cdot\ d\} \mapsto \{e{=}6\}! \tag{2}$$

Informally, Formula 2 holds if whenever we see a word in the language of the SERE $\{a \cdot b^* \cdot c^* \cdot d\}$, we get that $e{=}6$ holds on the final letter of that word.

If we want to see $e{=}6$ only when the number of b's and c's is equal, DRE extended with local variables would allow us to do so like this:[2]

$$\begin{aligned}&\{\mathit{new}(i{\leftarrow}0, j{\leftarrow}0) \\ &\quad \mathit{true}^* \cdot a \cdot (b, i{+}{+})^* \cdot\ (c, j{+}{+})^* \cdot (d \wedge i{=}j)\} \mapsto \{e{=}6\}!\end{aligned} \tag{3}$$

Note that we use $\leftarrow$ for local variable assignment, and $=$ for equality. That is, $i{\leftarrow}0$ is an assignment while $e{=}6$ is a Boolean expression that uses the equality operator. We use $i{+}{+}$ and i `--` as abbreviations for the assignments $i{\leftarrow}i+1$ and $i{\leftarrow}i-1$, respectively. In Formula 3, local variables i and j are initialized to 0, and are incremented when we see a b or a c, respectively. At the occurrence of d, i is compared to j, and a word is in the language of the left-hand SERE only if they are equal.

The "locality" of a local variable stems from the fact that different "matches" of the same SERE can be considered to have different "copies" of the local variables. This is similar to programming languages in which each invocation of a function has its own copy of a local variable, even if those invocations

[1] In Dynamic Logic [8,10], $r \mapsto \varphi$ and $r \diamondsuit\!\!\rightarrow \varphi$ correspond to $[r]\varphi$ and $\langle r\rangle\varphi$, respectively.

[2] It might seem like we are increasing the expressive power to that of context free languages, but we limit ourselves to finite domains and therefore the expressive power does not increase. For simplicity, in this paper we omit the declaration of the domain and simply assume that it is finite.

978-1-4244-2735-2/08/$25.00 ©2008 IEEE

occur concomitantly (e.g., in recursion). Consider Formula 3 on a word such that a holds on the first letter and d on the fourth and sixth. If both b and c hold on all letters, then there are two ways to satisfy the formula by "matching" the left operand of the $\mapsto$. In one "match" we see one b and one c, thus at the occurrence of d we have that $i{=}j{=}1$. In another "match" we see two b's and two c's, thus at the occurrence of d we have that $i{=}j{=}2$.

It thus follows that on a word w such that a and b hold on the first letter and do not hold on any other letter, we expect that the following formulas should not hold:[3]

$$(\mathit{new}(v{\leftarrow}0)\ \{(a, v\texttt{++})^* \cdot (b, v\texttt{--})^* \cdot \neg b\} \mapsto \{v{=}\quad 1\}!) \quad (4)$$

$$(\mathit{new}(v{\leftarrow}0)\ \{(a, v\texttt{++})^* \cdot (b, v\texttt{--})^* \cdot \neg b\} \mapsto \{v{=}-1\}!) \quad (5)$$

This, because the left-hand SERE may match the first letter either by matching a or by matching b. When matching a, v is assigned 1 and when matching b it is assigned -1. According to the semantics of the suffix implication operator $\mapsto$, for every prefix in the language of the left-hand side there should exist a suffix matching the right-hand side. Since v may be assigned either 1 or -1 on the left-hand side, neither Formula 4 nor 5 should hold on w. The following formula, however, should hold on w:

$$(\mathit{new}(v{\leftarrow}0)\ \{(a, v\texttt{++})^* \cdot (b, v\texttt{--})^* \cdot \neg b\} \mapsto \{(v{=}1)\ \vee\ (v{=}-1)\}!) \quad (6)$$

Intuitively, local variables can be seen as a mechanism for both declaring quantified propositions [19] and constraining their behavior. While adding quantified propositions to LTL [17] increases its expressive power from ω-star-free regular expressions to ω-regular expressions [19], it does not increase the expressive power of DRE [2,15], which is ω-regular to begin with [1,15].

As stated earlier, the temporal logic DRE is at the core of the IEEE standard assertion languages SVA and PSL. The former standard includes local variables, while the latter is in the process of adding them. The motivation for local variables in these standards is pragmatic: while they do not add expressive power, they can ease formulation and readability. Formal semantics for local variables were first presented in [14, Annex E]. A closely related semantics was presented and its complexity analyzed in [2], where DRE is denoted SVA^{b+i}.

The semantics of [2,14] suffer from major drawbacks, the main one being that the union and intersection operators do not completely conform to the notions of set union and intersection, and thus break distributivity of $\cup$ over $\cap$. For example, as shown by [12, p. 49], the semantics of [14] give that $\{\mathit{true}\ \cup\ \{\mathit{false}\ \cap\ \{(\mathit{true}, v{\leftarrow}1)\}\}$ is not equivalent to $\{\mathit{true}\ \cup\ \mathit{false}\} \cap \{\mathit{true}\ \cup\ \{(\mathit{true}, v{\leftarrow}1)\}\}$, and the same is easily shown for the similar semantics of [2]. We examine this issue in depth in Sections IV and V.

Breaking standard algebraic properties such as distributivity is quite a serious problem, as most tools use heuristics that assume, for instance, that $A \cup \{B \cap C\}$ can be rewritten into $\{A \cup B\} \cap \{A \cup C\}$, and thus breaking distributivity breaks the tools. Of course, tools can be fixed, but even worse is that most users intuitively assume that the standard algebraic properties hold, and use them without thinking. Thus a logic that breaks them is at great risk of being misused.

In this paper we propose a formal semantics for augmenting DRE with local variables that solves the problems of [2,14]. In our semantics the standard operators on semi-extended regular expressions (concatenation, union, intersection and Kleene closure) are exactly as usual, and thus preserve conventional algebraic properties such as distributivity. In our solution, syntax explicitly determines the scope of a local variable and we assume that variables are not used (referenced or assigned) out of scope. Combining an explicit notion of scope with the standard interpretation of the SERE operators gives us a simple, elegant and intuitive semantics of the same complexity as those of [2,14], but without breaking distributivity or other standard algebraic properties.

As mentioned earlier, the addition of local variables to DRE does not increase expressive power, but may ease formulation of properties in certain cases. In this paper we do not motivate the addition of local variables to temporal logic, and it is not our intention to advocate for their use. Rather, we accept that others have found them useful [2, 11, 14, 16, 18] and in particular note that local variables are already a part of SVA and are in the process of being incorporated into PSL. Thus, given the widespread and growing use of these IEEE standard assertion languages, we feel it is important to get the definition of local variables right, and that is the goal we concentrate on in the sequel.

Our proposed solution is quite simple, which is on the one hand a virtue but on the other hand may give the impression that the problem itself was not a difficult one. However, we note that lack of distributivity has been a known issue in SVA since 2004, but never previously solved.

The remainder of this paper is structured as follows. In the following section we present the logic LVDRE — an extension of DRE with local variables. In Section III we show characteristics of our semantics, in Section IV we compare our semantics to previous work. In Section V we discuss in depth the intersection operator, which is the source of the problems of [2,14] and the motivation for our operator *free*. In Section VI we conclude. The proofs of all propositions are given in the full version of the paper.[4]

II. The logic DRE augmented with local variables

We now present the logic LVDRE, which is the logic DRE extended with local variables. We define LVDRE formulas with respect to a non-empty set of atomic propositions $\mathcal{P}$, a set of local variables $\mathcal{V}$ with domain $\mathcal{D}$, a set of (not necessarily Boolean) expressions $\mathcal{E}$ over $\mathcal{P} \cup \mathcal{V}$, and a set of Boolean expressions $\mathcal{B} \subseteq \mathcal{E}$. For simplicity we do not specify the syntax

[3]The examples of Formulas 4, 5 and 6 are due to Johan Mårtensson.

[4]The full version of the paper is accessible at http://www.cs.huji.ac.il/~danafi/publications/lv_full.pdf.

*Definition 1 (Semi-extended regular expressions (*SEREs*)):*

- If $b \in \mathcal{B}$ is a Boolean expression, X a sequence of local variables and E a sequence of expressions of the same length as X, then the following are SEREs:
  - b - $(b, \text{X}\leftarrow\text{E})$
- If r, r_1, and r_2 are SEREs, X a sequence of local variables and E a sequence of expressions of the same length as X, then the following are SEREs:
  - λ - $r_1 \cdot r_2$ - $r_1 \cup r_2$ - $r_1 \cap r_2$ - r^* - $\{\mathit{new}(\text{X})\ r\}$ - $\{\mathit{new}(\text{X}\leftarrow\text{E})\ r\}$ - $\{\mathit{free}(\text{X})\ r\}$

*Definition 2 (*LVDRE *formulas):* If φ and ψ are LVDRE formulas, r a SERE, X a sequence of local variables and E a sequence of expressions of the same length as X then the following are LVDRE formulas:

- $\neg\varphi$ - $\varphi \wedge \psi$ - $r!$ - $r \mapsto \varphi$ - $r \diamondsuit\!\!\rightarrow \varphi$ - $(\mathit{new}(\text{X})\ \varphi)$ - $(\mathit{new}(\text{X}\leftarrow\text{E})\ \varphi)$

Fig. 1: The syntax of LVDRE

of expressions, but rather assume that a set of expressions is given. Furthermore we assume that all local variables share the same domain $\mathcal{D}$, and that T and F are in $\mathcal{D}$. Then SEREs extended with local variables and LVDRE formulas are defined as shown in Figure 1.

The scope of a variable is defined as the SERE or formula in which it was declared using *new*, unless it was explicitly taken out of scope by *free*, and we assume that a variable not in scope is not used (neither referenced nor assigned). Thus, for example, in the formula $\varphi := \{r_1 \cdot \{\mathit{new}(x)\ r_2\} \cdot r_3\}$ the local variable x can be used in r_2 yet it cannot be used in r_1 or r_3. In the formula $\varphi' := \{\mathit{new}(x)\ r_1 \cdot \{\mathit{free}(x)\ r_2\} \cdot r_3\}$ the variable x can be used in r_1 and r_3 yet it cannot be used in r_2. Note that the formula φ' differs from the formula $\varphi'' := \{\{\mathit{new}(x)\ r_1\} \cdot r_2 \cdot \{\mathit{new}(x)\ r_3\}\}$. In both, x can be used in r_1 and r_3 but cannot be used in r_2. However, in the formula φ'' the variable x used in r_1 is different than the variable x used in r_3, while in φ' there is only a single variable x used in both r_1 and r_3. The formal semantics distinguishes between these two situations by assuming a preprocessing step that gives each variable a unique name. Thus, for example, instead of φ'' the formal semantics will see $\varphi''' := \{\{\mathit{new}(x_1)\ r'_1\} \cdot r_2 \cdot \{\mathit{new}(x_3)\ r'_3\}\}$, where r'_1 and r'_3 result from r_1 and r_3 by replacing every appearance of x with x_1 or x_3, respectively. Note that this is standard practice in any formal definition of a language with local variables.

We define the semantics of LVDRE formulas with respect to a word from the alphabet $\Sigma = 2^{\mathcal{P}}$ (the set of all possible valuations of the atomic propositions) and a letter from the alphabet $\Gamma = \mathcal{D}^{\mathcal{V}}$ (the set of all possible valuations of the local variables). The semantics of SEREs is defined with respect to words from the alphabet $\Lambda = \Sigma \times \Gamma \times \Gamma$. We call words over Λ *extended words*. A letter $\langle \sigma, \gamma, \gamma' \rangle$ of an extended word provides a valuation σ of the atomic propositions and two valuations γ and γ' of the local variables. The valuation γ corresponds to the value of the local variables before assignments have taken place, the *pre-value*, and the valuation γ' corresponds to the value of the local variables after they have taken place, the *post-value*. Working with extended words is part of the key to preserving the complete notions of union and intersection and thus preserving distributivity and other standard algebraic properties. This is since an extended word records the result of the assignment that took place in every letter (rather than existentially quantifying over the end result as is done in the semantics of [2,14]). Thus, we can check that the assignments in both operands of an intersection agree on every letter, and we can do so without altering the usual semantics of set intersection.

In the sequel we use σ to denote letters from the alphabet Σ, γ and $\hat{\gamma}$ to denote letters from Γ, and $\mathbf{a}$ to denote letters from Λ. We use u, v, w to denote words over Σ and $\mathbf{u}, \mathbf{v}, \mathbf{w}$ to denote words over Λ.

We use i, j and k to denote non-negative integers. We denote the i^{th} letter of v ($\mathbf{v}$) by v^{i-1} ($\mathbf{v}^{i-1}$) (since counting of letters starts at zero). We denote by $v^{i\cdot\cdot}$ ($\mathbf{v}^{i\cdot\cdot}$) the suffix of v ($\mathbf{v}$) starting at v^i ($\mathbf{v}^i$), and by $v^{i..j}$ ($\mathbf{v}^{i..j}$) the finite sequence of letters starting from v^i ($\mathbf{v}^i$) and ending at v^j ($\mathbf{v}^j$).

Let $\mathbf{v} = \langle \sigma_0, \gamma_0, \gamma'_0 \rangle \langle \sigma_1, \gamma_1, \gamma'_1 \rangle \cdots$ be a word over Λ. We use $\mathbf{v}|_\sigma$,$\mathbf{v}|_\gamma$,$\mathbf{v}|_{\gamma'}$ to denote the word obtained from $\mathbf{v}$ by leaving only the first, second or third component, respectively, of each letter. That is, $\mathbf{v}|_\sigma = \sigma_0\sigma_1\cdots$, $\mathbf{v}|_\gamma = \gamma_0\gamma_1\cdots$, and $\mathbf{v}|_{\gamma'} = \gamma'_0\gamma'_1\cdots$. We use $\mathbf{v}|_{\sigma\gamma}$ to denote the word obtained by leaving both the first and second components. That is, $\mathbf{v}|_{\sigma\gamma} = \langle \sigma_0, \gamma_0 \rangle \langle \sigma_1, \gamma_1 \rangle \cdots$. We say that $\mathbf{v}$ is *good* if for every $i \in \{0, 1, \ldots, |\mathbf{v}| - 2\}$ we have $(\mathbf{v}|_\gamma)^{i+1} = (\mathbf{v}|_{\gamma'})^i$, i.e., the pre-value of the local variables at letter $i+1$ is the post-value at position i.

Semantics of expressions We identify an expression $e \in \mathcal{E}$ over $\mathcal{P} \cup \mathcal{V}$ with a mapping $e : \Sigma \times \Gamma \mapsto \mathcal{D}$ where $\mathcal{D}$ is the domain of variables in $\mathcal{V}$. A Boolean expression $b \in \mathcal{B}$ is an expression whose domain is $\{\text{T}, \text{F}\}$, and we define *true* and *false* to be the Boolean expressions whose domains are $\{\text{T}\}$ and $\{\text{F}\}$, respectively.

We assume that for an atomic proposition p we have that $p(\sigma, \gamma) = \text{T}$ if $p \in \sigma$ and F otherwise, and that for a local variable v we have that $v(\sigma, \gamma)$ returns the value of v in γ. We sometimes abuse notation by writing simply $p(\sigma)$ and $v(\gamma)$.

We assume that operators are closed under $\mathcal{D}$ and behave in the usual manner, i.e. that for $\sigma \in \Sigma$, $\gamma \in \Gamma$, $e, e_1, e_2 \in \mathcal{E}$, a binary operator $\otimes$ and a unary operator $\odot$ we have $e_1(\sigma, \gamma) \otimes$

Definition 3 (Tight satisfaction): Let $\mathrm{Z} \subseteq \mathcal{V}$, and let $\gamma_1 \overset{\mathrm{Z}}{\sim} \gamma_2$ (read "γ_1 agrees with γ_2 relative to Z") denote that for every $z \in \mathrm{Z}$ we have that $z(\gamma_1) = z(\gamma_2)$. The notation $\mathbf{v} \models_{\mathrm{Z}} r$ means that $\mathbf{v}$ models tightly r with respect to controlled variables Z.

- $\mathbf{v} \models_{\mathrm{Z}} b \iff |\mathbf{v}| = 1$ and $b(\mathbf{v}^0|_{\sigma\gamma}) = \mathrm{T}$ and $\mathbf{v}^0|_{\gamma'} \overset{\mathrm{Z}}{\sim} \mathbf{v}^0|_{\gamma}$
- $\mathbf{v} \models_{\mathrm{Z}} b, \mathrm{X}{\leftarrow}\mathrm{E} \iff |\mathbf{v}| = 1$ and $b(\mathbf{v}^0|_{\sigma\gamma}) = \mathrm{T}$ and $\mathbf{v}^0|_{\gamma'} \overset{\mathrm{Z}}{\sim} [\mathrm{X}{\leftarrow}\mathrm{E}](\mathbf{v}^0|_{\sigma\gamma})$

- $\mathbf{v} \models_{\mathrm{Z}} \lambda \iff \mathbf{v} = \epsilon$
- $\mathbf{v} \models_{\mathrm{Z}} r_1 \cdot r_2 \iff \exists \mathbf{v}_1, \mathbf{v}_2$ such that $\mathbf{v} = \mathbf{v}_1\mathbf{v}_2$ and $\mathbf{v}_1 \models_{\mathrm{Z}} r_1$ and $\mathbf{v}_2 \models_{\mathrm{Z}} r_2$
- $\mathbf{v} \models_{\mathrm{Z}} r_1 \cup r_2 \iff \mathbf{v} \models_{\mathrm{Z}} r_1$ or $\mathbf{v} \models_{\mathrm{Z}} r_2$
- $\mathbf{v} \models_{\mathrm{Z}} r_1 \cap r_2 \iff \mathbf{v} \models_{\mathrm{Z}} r_1$ and $\mathbf{v} \models_{\mathrm{Z}} r_2$
- $\mathbf{v} \models_{\mathrm{Z}} r^* \iff$ either $\mathbf{v} = \epsilon$ or $\exists \mathbf{v}_1, \mathbf{v}_2$ such that ($\mathbf{v}_1 \neq \epsilon$ and $\mathbf{v} = \mathbf{v}_1\mathbf{v}_2$ and $\mathbf{v}_1 \models_{\mathrm{Z}} r$ and $\mathbf{v}_2 \models_{\mathrm{Z}} r^*$)

- $\mathbf{v} \models_{\mathrm{Z}} \{\mathit{new}(\mathrm{X})\ r\} \iff \mathbf{v} \models_{\mathrm{Z}\cup\mathrm{X}} r$
- $\mathbf{v} \models_{\mathrm{Z}} \{\mathit{new}(\mathrm{X}{\leftarrow}\mathrm{E})\ r\} \iff$ either ($\mathbf{v}{=}\epsilon$ and $\epsilon \models_{\mathrm{Z}} r$) or $\langle \mathbf{v}^0|_{\sigma}, [\mathrm{X}{\leftarrow}\mathrm{E}](\mathbf{v}^0|_{\sigma\gamma}), \mathbf{v}^0|_{\gamma'}\rangle \mathbf{v}^{1..} \models_{\mathrm{Z}\cup\mathrm{X}} r$
- $\mathbf{v} \models_{\mathrm{Z}} \{\mathit{free}(\mathrm{X})\ r\} \iff \mathbf{v} \models_{\mathrm{Z}\setminus\mathrm{X}} r$

Fig. 2: The semantics of SEREs

$e_2(\sigma,\gamma) = (e_1 \otimes e_2)(\sigma,\gamma)$ and $\odot(e(\sigma,\gamma)) = (\odot e)(\sigma,\gamma)$. In particular, we assume that Boolean disjunction, conjunction and negation behave in the usual manner.

Given a local variable x and an expression e we write $[x{\leftarrow}e](\sigma,\gamma)$ to denote the valuation $\hat{\gamma}$ such that $x(\hat{\gamma}) = e(\sigma,\gamma)$ and for every local variable $v \in \mathcal{V}\setminus\{x\}$ we have that $v(\hat{\gamma}) = v(\gamma)$. Given a sequence of local variable $\mathrm{X} = x_1,\ldots,x_n$ and a sequence of expressions $\mathrm{E} = e_1,\ldots,e_n$ of the same length we write $[x_1{\leftarrow}e_1,\ldots,x_n{\leftarrow}e_n](\sigma,\gamma)$ to denote the recursive application $[x_2{\leftarrow}e_2,\ldots,x_n{\leftarrow}e_n](\sigma,[x_1{\leftarrow}e_1](\sigma,\gamma))$. We write $\mathrm{X}{\leftarrow}\mathrm{E}$ to abbreviate $x_1{\leftarrow}e_1,\ldots,x_n{\leftarrow}e_n$.

Semantics of SEREs The semantics of SEREs is defined with respect to a finite good word over Λ and a set of local variables $\mathrm{Z} \subseteq \mathcal{V}$, and is given in Definition 3 in Figure 2. The role of Z, which we call the set of *controlled variables*, is to support scoping. Any variable in Z (this will correspond to the variables in scope) must keep its value if not assigned and take on the assigned value otherwise, whereas any variable not in Z (this will correspond to the variables not in scope) is free to take on any value.

Examine the light gray section of Definition 3. A SERE b holds on an extended word if the word consists of exactly one letter such that b holds on that letter and the γ' components of the letter record no assignments (that is, the post-value of every local variable in scope is the same as its pre-value). The SERE $(b, \mathrm{X}{\leftarrow}\mathrm{E})$ holds on an extended word if the word consists of exactly one letter such that b holds on that letter, and the γ' components of the letter record the assignments $\mathrm{X}{\leftarrow}\mathrm{E}$.

The dark gray section of Definition 3 shows the operators *new* and *free*, which are used to declare a variable (and thus add it to the current scope) and to free a variable (remove it from the current scope), respectively.

Examine now the medium gray section of Definition 3. The semantics of the empty SERE λ and of SERE concatenation, union, intersection and Kleene closure are exactly as usual.

Semantics of formulas The semantics of formulas, shown in Definition 4 in Figure 3, is defined with respect to a finite/infinite word over Σ, a valuation γ of the local variables and a set $\mathrm{Z} \subseteq \mathcal{V}$ of local variables. The role of Z is to support scoping, exactly as in tight satisfaction. The role of γ is to supply a current valuation of the local variables.

Examine the light gray section of Definition 4. When evaluating a formula containing a SERE over a word w we quantify over enhancements to w with respect to γ, where informally, an extended word $\mathbf{w}$ enhances w with respect to γ, denoted $\mathbf{w} \sqsupset \langle w,\gamma\rangle$, if $\mathbf{w}$ preserves w on $w|_\sigma$ and uses γ as the starting pre-value (this is defined formally in the preamble of the definition). If we are evaluating a formula of the form $r!$ or $r \diamondsuit\!\!\rightarrow \varphi$, then the quantification is existential, to match the existential quantification on prefixes of w in the semantics of DRE without local variables. If we are evaluating a formula of the form $r \mapsto \varphi$, then the quantification is universal, to match the universal quantification on prefixes of w in the semantics of DRE without local variables. For $\mapsto$ and $\diamondsuit\!\!\rightarrow$ we use the post-value of the last letter that "matched" r as the starting valuation of φ. For example, in the following formula:

$$(\mathit{new}(i)\ \{\mathit{true}^* \cdot (a, i{\leftarrow}0) \cdot (b, i{+}{+})^* \cdot c\} \mapsto \{(d, i{-}{-})^* \cdot (e \wedge i{=}0)\}!) \tag{7}$$

the post-value of i at the end of the "match" of the left operand records the number of b's that were seen, and is the starting point for the evaluation of the right operand. Thus, we get that Formula 7 holds on words such that if we see an a followed by some number (call it i) of b's followed by a c, then starting

Definition 4 (Satisfaction): We say that a word $\mathbf{w}$ over Λ *enhances* $\langle w, \gamma\rangle$, denoted $\mathbf{w} \sqsupset \langle w, \gamma\rangle$, iff $\mathbf{w}$ is good, $\mathbf{w}|_\sigma = w$, and $\mathbf{w}^0|_\gamma = \gamma$. The notation $\langle w, \gamma\rangle \models_{\mathrm{Z}} \varphi$ means that the word w satisfies φ with respect to controlled variables $\mathrm{Z} \subseteq \mathcal{V}$ and current valuation of variables γ.

- $\langle w, \gamma\rangle \models_{\mathrm{Z}} r!$ $\iff$ $\exists \mathbf{w} \sqsupset \langle w, \gamma\rangle$, $j < |w|$ such that $\mathbf{w}^{0..j} \not\models_{\mathrm{Z}} r$
- $\langle w, \gamma\rangle \models_{\mathrm{Z}} r \mapsto \varphi$ $\iff$ $\forall \mathbf{w} \sqsupset \langle w, \gamma\rangle$ and $j < |w|$: if $\mathbf{w}^{0..j} \not\models_{\mathrm{Z}} r$ then $\langle w^{j\cdot\cdot}, \mathbf{w}^j|_{\gamma'}\rangle \models_{\mathrm{Z}} \varphi$
- $\langle w, \gamma\rangle \models_{\mathrm{Z}} r \diamondsuit\!\!\mapsto \varphi$ $\iff$ $\exists \mathbf{w} \sqsupset \langle w, \gamma\rangle$, $j < |w|$ such that $\mathbf{w}^{0..j} \not\models_{\mathrm{Z}} r$ and $\langle w^{j\cdot\cdot}, \mathbf{w}^j|_{\gamma'}\rangle \models_{\mathrm{Z}} \varphi$

- $\langle w, \gamma\rangle \models_{\mathrm{Z}} \neg\varphi$ $\iff$ $\langle w, \gamma\rangle \not\models_{\mathrm{Z}} \varphi$
- $\langle w, \gamma\rangle \models_{\mathrm{Z}} \varphi \wedge \psi$ $\iff$ $\langle w, \gamma\rangle \models_{\mathrm{Z}} \varphi$ and $\langle w, \gamma\rangle \models_{\mathrm{Z}} \psi$

- $\langle w, \gamma\rangle \models_{\mathrm{Z}} (\mathit{new}(\mathrm{X})\ \varphi)$ $\iff$ $\langle w, \gamma\rangle \models_{\mathrm{Z}\cup\mathrm{X}} \varphi$
- $\langle w, \gamma\rangle \models_{\mathrm{Z}} (\mathit{new}(\mathrm{X}{\leftarrow}\mathrm{E})\ \varphi)$ $\iff$ $\langle w, [\mathrm{X}{\leftarrow}\mathrm{E}](w^0, \gamma)\rangle \models_{\mathrm{Z}\cup\mathrm{X}} \varphi$

Fig. 3: The semantics of LVDRE formulas

from the letter that "matched" c we should see i d's followed by an e.

The dark and medium gray sections of Definition 4 show the *new* operator, similar to that of tight satisfaction, and the operators $\neg$ and $\wedge$, which have the usual semantics.

Semantics with respect to a model Given a model M and a computation path π of M we use $L(\pi)$ to denote the word over Σ that results from mapping each state in M to a letter in Σ in the obvious way. The notation $M \models \varphi$ means that for every computation path π of M, and every $\gamma \in \Gamma$ we have $\langle L(\pi), \gamma\rangle \models_\emptyset \varphi$. By universally quantifying over all contexts of local variables we consider all possible initial values for them. The set Z is initially empty, as a variable is in scope only if it was declared (and not freed).

Extending the definition of local variables to all of PSL and SVA Our formal semantics is easily extendable from LVDRE to all of PSL and SVA, although the details of doing so are beyond the scope of this paper. The extension to PSL includes LTL-style temporal operators [17], the fusion operator [3,13], weak SEREs [5], clock [7,9] and abort [6] operators and is detailed in [4]. The extension to SVA, which includes a subset of the features in [4], is similar.

III. Characteristics

In this section we state several properties of the proposed semantics (all are proven in the full version of the paper).

Proposition 1: Let r be a SERE and φ be an LVDRE formula. Let $\mathcal{P}$ be the set of atomic propositions in r (resp. φ). Let $\mathcal{V}$ be the set of local variables in r (resp. φ) and let $\mathcal{D}$ be their domain. Let $\Sigma = 2^{\mathcal{P}}$ and $\Gamma = \mathcal{D}^{\mathcal{V}}$. Let $\mathrm{Z} \subseteq \mathcal{V}$ and $\gamma \in \Gamma$.

- There exists a non-deterministic finite automaton (NFW) $N_{\mathrm{Z},\gamma}(r)$ with $O(\mathcal{D}^{|r|\cdot 2|\mathcal{V}|})$ states that accepts exactly the set of words $\mathbf{v}$ such that $\mathbf{v} \not\models_{\mathrm{Z}} r$ and $\mathbf{v}^0|_\gamma = \gamma$.
- There exists an alternating Büchi automaton (ABW) B^γ_φ with $O(\mathcal{D}^{|\varphi|\cdot 2|\mathcal{V}|})$ states that accepts exactly the set of words w such that $\langle w, \gamma\rangle \models_{\mathrm{Z}} \varphi$.

When the domain of local variables is $\{\mathrm{T}, \mathrm{F}\}$ we get that the NFW and ABW are of sizes $O(2^{|r|\cdot 2|\mathcal{V}|})$ and $O(2^{|\varphi|\cdot 2|\mathcal{V}|})$, respectively (the source of the exponent in the size of the SERE/formula is the SERE intersection operator). Thus the automata that we build here are slightly bigger than those of [2] for which there exists an NFW (ABW) recognizing a given SERE (formula) of size $O(2^{|r|\cdot|\mathcal{V}|})$ (resp. $O(2^{|\varphi|\cdot|\mathcal{V}|})$. This is since our automata need to take into account the set Z of controlled local variables. At any rate, the complexity of the satisfiability and model checking problems is exactly as in [2], as stated in Proposition 2. The proof follows that of [2, Theorems 1 and 2].

Proposition 2: The satisfiability and model checking problems for properties in LVDRE are EXPSPACE-complete.

Proposition 3 below shows that our semantics preserves standard algebraic properties and in particular distributivity. Proposition 4 shows that idempotence for union and intersection is exactly as usual and that the identity element for union and concatenation is exactly as usual, while the identity element for intersection (usually *true**) changes slightly under our semantics, because we need to take all local variables out of scope explicitly using *free*.

Proposition 3: The following properties hold for the semantics of LVDRE.

- The operators $\cap$ and $\cup$ are commutative.
- The operators $\cap$, $\cup$, $\cdot$ are associative.
- The operator $\cap$ distributes over the operator $\cup$.
- The operator $\cup$ distributes over the operator $\cap$.

Proposition 4: Let r be a SERE and let $\mathcal{V}$ be the set of local variables. Then the following equivalences hold for the semantics of LVDRE.

- $r \equiv \{r\ \cap\ r\}$
- $r \equiv \{\{\mathit{free}(\mathcal{V})\ \mathit{true}\}^*\ \cap\ r\}$
- $r \equiv \{\lambda \cdot r\} \equiv \{r \cdot \lambda\}$
- $r \equiv \{r \cup r\}$
- $r \equiv \{\mathit{false}\ \cup\ r\}$

IV. Comparison with previous approaches

We now compare our semantics to those of [2,14]. Due to space limitations, we show only the semantics of the SERE

$$w, L_0, L_1 \models r_1 \cap r_2 \iff \exists L', L'' \text{ such that } w, L_0, L' \models r_1 \text{ and } w, L_0, L'' \models r_2 \text{ and } L_1(v) = L'(v) \text{ if } v \text{ is assigned in } r_1 \text{ and } L_1(v) = L''(v) \text{ otherwise}$$

$$w, L_0, L_1 \models r_1 \cap r_2 \iff \exists L', L'' \text{ s.t. } w, L_0, L' \models r_1 \text{ and } w, L_0, L'' \models r_2 \text{ and } L_1 = L'|_{D'} \cup L''|_{D''}, \text{ where } D' = \textbf{\textit{flow}}(dom(L_0), r_1) - (\textbf{\textit{block}}(r_1 \cap r_2) \cup \textbf{\textit{sample}}(r_2)) \quad D'' = \textbf{\textit{flow}}(dom(L_0), r_2) - (\textbf{\textit{block}}(r_1 \cap r_2) \cup \textbf{\textit{sample}}(r_1))$$

Fig. 4: The semantics of [2] (shown in light gray) and of [14] (shown in medium gray) for SERE intersection

$$\begin{array}{lcl} \bullet\ \textbf{\textit{sample}}(r_1 \cap r_2) & = & \textbf{\textit{sample}}(r_1) \cup \textbf{\textit{sample}}(r_2) \\ \bullet\ \textbf{\textit{block}}(r_1 \cap r_2) & = & \textbf{\textit{block}}(r_1) \cup \textbf{\textit{block}}(r_2) \cup (\textbf{\textit{sample}}(r_1) \cap \textbf{\textit{sample}}(r_2)) \\ \bullet\ \textbf{\textit{flow}}(X, r_1 \cap r_2) & = & (\textbf{\textit{flow}}(X, r_1) \cup \textbf{\textit{flow}}(X, r_2)) - \textbf{\textit{block}}(r_1 \cap r_2) \end{array}$$

Fig. 5: The functions sample, block and flow of [2,14] for SERE intersection

intersection operator (given in Figure 4). The comparison is done along two axes, substance and style.

Regarding substance, we first note that the non-standard semantics of the intersection operator in [2,14] gives startlingly unintuitive interpretations in some cases. For example, consider the following SEREs:

$$\{v{=}4\} \cap \{v{=}13\} \tag{8}$$

$$\{r_1 \cdot v{=}4\} \cap \{r_2 \cdot v{=}13\} \tag{9}$$

Intuitively, the language of SERE 8 should be empty, as should the language of SERE 9 for any SEREs r_1 and r_2. In the semantics of [2,14], the language of SERE 8 is indeed empty, but there exist r_1 and r_2 such that the language of SERE 9 is not empty. For example, letting $r_1 = (\textit{true}, v{\leftarrow}4)$ and $r_2 = \textit{true}$ results in a satisfiable SERE which when concatenated with $v{=}4$ gives the following satisfiable SERE:

$$\{\{(\textit{true}, v{\leftarrow}4) \cdot v{=}4\} \cap \{\textit{true} \cdot v{=}13\}\} \cdot v{=}4 \tag{10}$$

For example, SERE 10 is satisfiable by $\langle w, L_0, L_1 \rangle$ such that w is any word of length 2, in L_0 we have that $v{=}13$ and in L_1 we have that $v{=}4$.

Informally, in the semantics of [2,14] assignment to a local variable on one or both operands of an intersection operator "splits" it into two copies. If it is assigned on only one operand, then the value assigned by that operand is the one that "flows out" of the intersection. If both operands make an assignment, then "syntactic restrictions" are intended to prevent the use of the local variable until it has been reassigned. Under our semantics, all of SEREs 8, 9 and 10 are unsatisfiable.

If we omit the $\cap$ operator, our semantics and those of [2,14] are equivalent.[5] For formulas containing the $\cap$ operator, the semantics are different: in our semantics $\cap$ is pure intersection, while [2,14] uses existential quantification in order to get that the same local variable may be assigned different values in each operand of $\cap$. For this reason the semantics of [2,14] break distributivity, while ours does not. Nevertheless, the two semantics are of the same expressive power (because adding local variables with a finite domain does not change the expressive power, which is ω-regular to begin with).

Regarding style, the semantics of [14] are clearly quite complicated (recall that the dark gray section of Figure 4 gives only the semantics of intersection). In addition it is obvious that a violation of the *sample*, *block* and *flow* rules, whose intention is to provide the above mentioned syntactic restrictions, results in a specific truth value rather than an illegal formula. The semantics of [2] (shown in the light gray section of Figure 4) appear to be somewhat simpler. Note however the asymmetry between r_1 and r_2, which according to [2] is supposed to be resolved by the *sample*, *block* and *flow* rules. While intuitively the intention of this is clear, the details are hazy. The stated intent of the restrictions is to prevent a reference to a local variable at places where it does not have a well-defined value. However, the *sample*, *block* and *flow* rules were designed for semantic use (see the dark gray section of Figure 4) and the exact details of their use as syntactic restrictions are not obvious and are never defined.

In our semantics, on the other hand, there is a single, purely syntactic restriction — that a variable is not used out of scope — that is extremely easy to check. The simplicity of our semantics stems from the fact that treatment of local variables is restricted to the operators that explicitly refer to local variables, and the semantics of the SERE operators that correspond to standard operations on automata is exactly as in standard automata theory.

V. Discussion

In this section we explain why we chose to give explicit control of the scope to users (by introducing the *new* and *free* operators) instead of determining it automatically as done in [2,14].

The SERE intersection operator presents a difficulty. On the one hand it should completely preserve the notion of

[5]To be precise, without $\cap$ and modulo the fact that in our logic the user has to declare the variables, our semantics is equivalent to those of [2]. The semantics of [14] differ subtly from that of [2] for the $\cup$ operator, for a reason related to the sample, block and flow rules; the details are beyond the scope of this paper.

intersection, but on the other hand, doing so seems to restrict the usefulness of local variables. For example, suppose we want a SERE whose language includes only words where the number of a's between s and e equals the number of b's between s and e. Trying:

$$\{\mathit{new}(i{\leftarrow}0) \quad \{s \cdot \{\neg a^* \cdot (a, i{+}{+})\}^* \cdot e\} \cap \quad \{s \cdot \{\neg b^* \cdot (b, i{+}{+})\}^* \cdot e\} \ \} \tag{11}$$

gives us a language that includes only words where the a's and b's occur simultaneously. That is, we get the language of $\{s \cdot \{\neg(a \vee b)^* \cdot (a \wedge b)\}^* \cdot e\}$ rather than the language we are looking for. If we try to base our solution on

$$\{ \ \{\mathit{new}(i_1{\leftarrow}0) \ \{s \cdot \{\neg a^* \cdot (a, i_1{+}{+})\}^* \cdot e\}\} \cap \ \{\mathit{new}(i_2{\leftarrow}0) \ \{s \cdot \{\neg\, b^* \cdot (b, i_2{+}{+})\}^* \cdot e\}\} \ \} \tag{12}$$

we are left with a SERE where i_1 and i_2 cannot be compared because their scope is disjoint. We might try this:

$$\{\mathit{new}(i_1{\leftarrow}0, i_2{\leftarrow}0) \quad s \cdot \{ \ \{\neg a^* \cdot (a, i_1{+}{+})\}^* \cap \ \{\neg\, b^* \cdot (b, i_2{+}{+})\}^* \ \} \cdot e \wedge (i_1{=}i_2)\} \tag{13}$$

but the result is a SERE that does not hold tightly on any word containing at least one a or one b, because the left-hand side of the intersection increments i_1 while the right-hand side does not and vice-versa for i_2.

One way to get what we want is to keep the restricted scope of i_1 and i_2 as in SERE 12 while adding a third local variable whose scope is the whole SERE. Thus,

$$\{\mathit{new}(i) \quad \{\mathit{new}(i_1{\leftarrow}0)\, \{s \cdot \{\neg a^* \cdot (a, i_1{+}{+})\}^* \cdot (e, i{\leftarrow}i_1)\}\} \ \cap \ \{\mathit{new}(i_2{\leftarrow}0)\, \{s \cdot \{\neg\, b^* \cdot (b, i_2{+}{+})\}^* \cdot (e, i{\leftarrow}i_2)\}\}\} \tag{14}$$

gives us what we want. By assigning both i_1 and i_2 to i, we get that the SERE will "match" only if i_1 and i_2 are equal.

Another way is to use the *free* operator. It provides a way to change the context Z by removing a local variable from the set Z, thus "freeing" it to take on any value. Using the *free* operator, we can express our desired property as follows:

$$\{\mathit{new}(i_1{\leftarrow}0, i_2{\leftarrow}0) \quad s \cdot \{ \ \{\mathit{free}(i_2) \ \neg a^* \cdot (a, i_1{+}{+})\}^* \cap \ \{\mathit{free}(i_1) \ \neg\, b^* \cdot (b, i_2{+}{+})\}^*\} \ \cdot e \wedge (i_1{=}i_2)\} \tag{15}$$

The local variable i_2 is free to take on any value at all in the left operand of the intersection, and in particular the same value as in the right. And vice-versa, i_1 is free to take on any value at all in the right operand, and in particular the same value as in the left.

It might seem at first glance that control over the scope should be taken care of automatically. For example, define Z to be exactly the set of local variables used in the SERE, and ensure that at least one of the operands of $\cap$ "takes responsibility" for the variable, as shown in the alternative semantics of Figure 6 (which uses existential quantification and thus is somewhat similar in spirit to the semantics of [2,14]).

$$\mathbf{v} \models_{\mathrm{Z}} r_1 \ \cap \ r_2 \iff \exists \mathrm{Z}_1, \mathrm{Z}_2 \text{ s.t. } \mathrm{Z}_1 \cup \mathrm{Z}_2 = \mathrm{Z} \text{ and } \mathbf{v} \models_{\mathrm{Z}_1} r_1 \text{ and } \mathbf{v} \models_{\mathrm{Z}_2} r_2$$

Fig. 6: An alternative semantics for SERE intersection

For $r = \{b\} \cap \{(c, v{\leftarrow}3)\}$ the semantics of Figure 6 works as follows. Let the full set of local variables Z be $\{v\}$. Then we can let Z_1 be $\emptyset$ and Z_2 be $\{v\}$. Thus we get that under the context Z, r holds tightly on a word consisting of a single letter $\mathbf{a}$ where both b and c hold, and $v(\mathbf{a}|_{\gamma'}) = 3$.

The problem with this approach is that it breaks distributivity. Consider the following SEREs (these are the examples used to illustrate the lack of distributivity of [14] as given in [12, p. 49]):

$$\{\mathit{true}\} \cup \{\mathit{false} \ \cap \ \{(\mathit{true}, v{\leftarrow}1)\}\} \tag{16}$$

$$\{\mathit{true} \cup \mathit{false}\} \ \cap \ \{\mathit{true} \cup \{(\mathit{true}, v{\leftarrow}1)\}\} \tag{17}$$

Let $\mathbf{v}$ be a word consisting of a single letter $\mathbf{a}$ such that $v(\mathbf{a}|_{\gamma}) = 0$ and $v(\mathbf{a}|_{\gamma'}) = 1$. Then under the semantics of Figure 6, for the context $\mathrm{Z} = \{v\}$, SERE 17 holds tightly on $\mathbf{v}$, and SERE 16 does not. To see this, note that SERE 17 can choose Z_1 to be $\emptyset$ and Z_2 to be the set $\{v\}$, so both sides of the $\cap$ hold. Whereas SERE 16 must give the full set Z to both sides of the $\cup$, and thus $\mathbf{v}$ satisfies neither $\{\mathit{true}\}$ (because v is not assigned a value by it, yet we have that $v(\mathbf{a}|_{\gamma})$ and $v(\mathbf{a}|_{\gamma'})$ differ), nor $\{\mathit{false} \ \cap \ \{\mathit{true}, v{\leftarrow}1\}\}$ (because of course it does not satisfy *false*). We cannot solve this by letting the $\cup$ operator divide the responsibility between its operands in a similar way that $\cap$ does in Figure 6 above. This is since we are going to see either the left side of the $\cup$ operator or the right side, and in both cases we need one operand to take full responsibility.

Note that all semantics (including those of [2,14]) that try to automatically divide the responsibility for the set Z between r_1 and r_2 will have similar problems. This should be clear, because the semantics of Figure 6 is general in that it says simply that there exist suitable Z_1 and Z_2. If there do not exist Z_1 and Z_2 that work for SEREs 16 and 17, then being more specific cannot help.

Since the scoping cannot be determined automatically without breaking important algebraic properties, our logic includes the *new* and *free* operators that allow explicit control of the scope. Having the scoping determined by the syntax, and the SERE operators adhering to their standard semantics, we get a semantics that preserves distributivity as well as other standard algebraic properties. This is of course very important, since we want the user (and the algorithms and tools) to be able to use intuitive and standard rewrites such as $r_1 \cup \{r_2 \ \cap \ r_3\} \equiv \{r_1 \cup r_2\} \ \cap \ \{r_1 \cup r_3\}$ without having to stop and consider carefully the implications that would entail with an alternative definition that breaks distributivity such as that of Figure 6 or of [2,14].

We note that the basic utility of local variables is available without the *free* operator. The purpose of the *free* operator

is only to provide added flexibility for easy expression of properties such as that expressed by Formula 15. The semantics of [2,14] obtain that flexibility by using non-standard semantics for the intersection operator and so breaking distributivity, whereas we have chosen to preserve distributivity at the expense of giving explicit control of the scope to the user.

VI. Conclusions

We have presented a relatively simple semantics for local variables that solves serious problems in the widely used industry standard assertion language SVA. In SVA, the non-standard semantics of the intersection operator cause distributivity to be broken, so that users of the logic cannot use the standard intuitions such as that $A \cup \{B \cap C\} \equiv \{A \cup B\} \cap \{A \cup C\}$. In our semantics, distributivity and other standard algebraic properties are preserved, so that the user can freely use his or her existing intuitions when working with the logic.

Our solution is based on the understanding that the scope of local variables cannot be determined automatically by the semantics without breaking important algebraic properties (and as a consequence exhibiting unintuitive interpretations in certain cases). We thus require the user to explicitly control the scope and so equip our logic with operators $\mathit{new}(x)$ and $\mathit{free}(x)$ that add and remove x from scope, respectively. The standard operators of semi-extended regular expressions (concatenation, union, intersection, and Kleene closure) have their usual semantics. This separation between the parts of the semantics that explicitly deal with local variables and the usual operators on SEREs is the key to preserving algebraic properties and not breaking intuition.

Our aim was to solve the drawbacks of [2,14] while maintaining the same complexity. Indeed, though the automata constructed for recognizing our semantics are slightly bigger than those of [2], the satisfiability and model checking problems for formulas in LVDRE have the same complexity under our semantics as under the semantics of [2,14].

We remind the reader that our motivation was not to advocate the use of local variables in temporal logic, but rather to fix a serious problem in SVA, and to prevent a similar problem in PSL. Our solution is easily extendable to cover all of the features in these widely used IEEE assertion languages. As shown in [4], our semantics are easily extendable to all of PSL, including LTL-style temporal operators, the fusion operator, weak SEREs, clocks, and the abort operators; the extension to all of SVA, which includes a subset of the features in [4], is similar.

Acknowledgments

We would like to thank Shoham Ben-David, Doron Bustan, John Havlicek, Erich Marschner, Johan Mårtensson, Avigail Orni, Dmitry Pidan and Sitvanit Ruah for many interesting discussions on the subject of local variables. Thanks to the latter three for insightful comments on previous versions of this paper. Finally, special thanks for Avigail Orni for important comments on an early version of the semantics.

References

[1] D. Bustan, D. Fisman, and J. Havlicek. Automata construction for PSL. Technical Report MCS05-04, The Weizmann Institute of Science, May 2005.

[2] D. Bustan and J. Havlicek. Some complexity results for SystemVerilog assertions. In *Proc. CAV 2006*, LNCS 4144, pages 205–218. Springer, 2006.

[3] C. Eisner and D. Fisman. *A Practical Introduction to PSL*. Springer, 2006.

[4] C. Eisner and D. Fisman. Proposed New Appendix B for IEEE 1850 (PSL). Technical Report H-0257, IBM, 2008.

[5] C. Eisner, D. Fisman, and J. Havlicek. A topological characterization of weakness. In *Proc. PODC '05*, pages 1–8, New York, NY, USA, 2005. ACM Press.

[6] C. Eisner, D. Fisman, J. Havlicek, Y. Lustig, A. McIsaac, and D. Van Campenhout. Reasoning with temporal logic on truncated paths. In *Proc. CAV '03*, LNCS 2725, pages 27–40. Springer-Verlag, July 2003.

[7] C. Eisner, D. Fisman, J. Havlicek, A. McIsaac, and D. Van Campenhout. The definition of a temporal clock operator. In *Proc. ICALP '03*, LNCS 2719, pages 857–870. Springer-Verlag, June 2003.

[8] M. J. Fischer and R. E. Lander. Propositional dynamic logic of regular programs. In *J. Comput. Syst. Sci.*, pages 18(2), 194–211, 1979.

[9] D. Fisman. On the characterization of until as a fixed point under clocked semantics. In *Proc. Haifa Verification Conference*, volume 4899 of *LNCS*, pages 19–33. Springer, 2007.

[10] D. Harel, D. Kozen, and J. Tiuryn. *Dynamic Logic*. MIT Press, Cambridge, MA, USA, 2000.

[11] J. Havlicek, N. Levi, H. Miller, and K. Shultz. Extended CBV statement semantics, partial proposal presented to the Accellera Formal Verification Technical Committee, April 2002. At http://www.eda.org/vfv/hm/att-0772/01-ecbv_statement_semantics.ps.gz.

[12] J. Havlicek, K. Shultz, R. Armoni, S. Dudani, and E. Cerny. Notes on the semantics of local variables in Accellera SystemVerilog 3.1 concurrent assertions. Technical Report 2004.01, Accellera, May 2004.

[13] IEEE Standard for Property Specification Language (PSL). IEEE Std 1850™-2005.

[14] IEEE Standard for SystemVerilog Unified Hardware Design, Specification, and Verification Language. IEEE Std 1800™-2005.

[15] M. Lange. Linear time logics around PSL: Complexity, expressiveness, and a little bit of succinctness. In *Proc. CONCUR 2007*.

[16] M. T. Oliveira and A. J. Hu. High-level specification and automatic generation of IP interface monitors. In *DAC '02: Proceedings of the 39th conference on Design automation*, pages 129–134, New York, NY, USA, 2002. ACM.

[17] A. Pnueli. A temporal logic of concurrent programs. *Theoretical Computer Science*, 13:45–60, 1981.

[18] A. Seawright and F. Brewer. High-level symbolic construction technique for high performance sequential synthesis. In *DAC '93: Proceedings of the 30th international conference on Design automation*, pages 424–428, New York, NY, USA, 1993. ACM.

[19] A. P. Sistla, M. Y. Vardi, and P. L. Wolper. The Complementation Problem for Büchi Automata, with Applications to Temporal Logic. *Theoretical Computer Science*, 49:217–237, 1987.

2008-08-26 15:19

Beyond Vacuity: Towards the Strongest Passing Formula

Hana Chockler
IBM Haifa Research Lab, Haifa, Israel. hanac@il.ibm.com
Arie Gurfinkel
Software Engineering Institute, Pittsburgh, USA. arie@sei.cmu.edu
Ofer Strichman
Information Systems Engineering, IE, Technion, Haifa, Israel. ofers@ie.technion.ac.il

***Abstract*—Given an LTL formula φ in negation normal form, it can be *strengthened* by replacing some of its literals with FALSE. Given such a formula and a model M that satisfies it, *vacuity* and *mutual vacuity* attempt to find one or a maximal set of literals, respectively, with which φ can be strengthened while still being satisfied by M. We study the problem of finding the strongest LTL formula that satisfies M and is in the Boolean closure of strengthened versions of φ as defined above. This formula is stronger or equally strong to any formula that can be obtained by vacuity and mutual vacuity. We present our algorithms in the framework of lattice automata.**

I. Introduction

In finite-state *model checking* [CE81], [QS82], [LP85] we verify the correctness of a system modeled as a labeled state-transition graph, with respect to a desired behavior which is expressed in terms of a temporal logic formula or a finite automaton. Beyond being fully automatic, an additional attraction of model-checking tools is their ability to accompany a negative answer to the correctness query by a counterexample to the satisfaction of the specification in the system. Thus, together with a negative answer, the model checker returns some erroneous execution of the system. These counterexamples can be essential in detecting subtle errors in complex designs [CGMZ95].

On the other hand, when the answer to the correctness query is positive, most model-checking tools terminate with no further information to the user. Since a positive answer means that the system is correct with respect to the specification, this at first seems like a reasonable policy. This policy was first questioned in [BB94], where it was shown that *antecedent failure* leads to meaningless satisfaction. Beer et al. generalized this problem by formally defining the notion of *vacuity* and claimed that it is a serious problem: "Our experience has shown that typically 20% of specifications pass vacuously during the first formal verification runs of a new hardware design, and that vacuous passes always point to a real problem in either the design or its specification or the environment" [BBER01]. The main reasons for a vacuous pass are possible errors in the modeling of the system and a possible incompleteness in the specification. *Vacuity detection* [BBER01] (see a survey in [Kup06]) is probably the most common method for systematically searching for such errors. The reason for a vacuous pass of the model-checking procedure is, as in antecedent failure, that parts of the specification are irrelevant to its satisfaction. For example, the specification $\varphi = \mathbf{G}(req \rightarrow \mathbf{F}grant)$ ("every request is eventually followed by a grant") is vacuously satisfied in a system in which req is always FALSE (that is, requests are never sent). Thus, a specification might be satisfied for a reason unexpected by the user. Vacuity thus attempts to discover *why* M satisfies φ. The explanation may lead to insights and possible corrections to the property, the model or the environment.

The definition of Beer et al. is based on the notion of subformulas that do not affect the satisfaction of the specification in the system, as follows. Consider a system M and a specification φ. A subformula ψ of φ *does not affect* φ *in* M if and only if ψ can be replaced with any formula θ without changing the truth value of φ in M. A specification φ is *satisfied vacuously in* M, denoted by $M \models_v \varphi$, if $M \models \varphi$ and there exists ψ that does not affect φ in M.

Several variants of vacuity were published through the years: vacuity with respect to subformulas as in the original definition of [BBER01], vacuity with respect to subformula *occurrences* [KV03], and vacuity with respect to *literal occurrences* [GC04] (which is equivalent, but practically cheaper to compute than the former). In both cases, if the formula is given in negation normal form, it is enough to consider replacing each occurrence separately with FALSE [KV03]. Indeed, in the example above, after replacing $grant$ with FALSE φ is still satisfied in a system in which req is always false, and is hence vacuous.

Gurfinkel and Chechik extended vacuity to *mutual vacuity*, where the problem is to find the maximal set of literal occurrences that can be replaced simultaneously with FALSE without falsifying the property in the model [GC04]. Checking for mutual vacuity can lead to stronger properties than those obtained by standard vacuity checks, but is also harder because we have to consider subsets of literals [CS07].

In this paper we examine the possibility of finding a property stronger than φ and any strengthening of φ by either vacuity or mutual vacuity, that is still satisfied in M, for a given LTL property φ and model M such that $M \models \varphi$. In fact we claim that among formulas that satisfy M, the formula that we find is the strongest possible by replacing literals in φ with

978-1-4244-2735-2/08/$25.00 ©2008 IEEE

FALSE and any Boolean combination (excluding negations) of such formulas.

In the range between M and φ there are many possible formulas, i.e., formulas that are stronger than φ but weaker than M. The question of which formula is the most useful for the user can only be answered subjectively (if we ignore the complexity of finding this formula). It has to represent the user's abstract view of M so it can be contrasted with it, but on the other hand it should be strong enough to expose behaviors that violate the intended specification. The point in this range that we target here seems to us reasonable in that respect: it is still built based on properties given by the user, but it is potentially much stronger and closer to M.

The following examples demonstrate that indeed stronger formulas can be found.

Example 1 *Consider the three models and properties in Fig. 1. Each of these models satisfy its associated property non-vacuously, yet none of these properties capture well the behavior of the model. Better and stronger properties are, respectively,*

$$\begin{aligned} \varphi'_1 &= (\mathbf{G}a \vee \mathbf{G}b) \\ \varphi'_2 &= a \,\mathbf{U}\, c \vee b \,\mathbf{U}\, c \\ \varphi'_3 &= \mathbf{G}(a \vee b) \vee \mathbf{G}(a \vee c) \ . \end{aligned} \tag{1}$$

Example 1 exposes the following fact, first observed by Kupferman and Vardi in [KV03], about vacuity checks: even when all the paths in the model pass vacuously, together they might require all parts of the formula, hence leading to a non-vacuous pass of the property.

Better formulas can be obtained by distinguishing between the different ways in which paths in M satisfy φ. In particular, the different paths of a model induce equivalence classes, such that two paths are in the same class if and only if they have the same *vacuity value* with respect to the input formula φ. These classes can be computed and used for constructing strong formulas as discussed above.

We continue with several preliminaries and definitions in Sect. II. Section III is dedicated to the type of formulas that we wish to compute. Section IV presents algorithms, in the framework of *vacuity lattices*, for computing these formulas.

II. Vacuity: Background and Definitions

Throughout the paper, we denote by φ an LTL formula in negation normal form (NNF), and by M a model. We only consider vacuity with respect to occurrences of literals. Without loss of generality, we assume that each literal appears only once. If this is not the case, different occurrences can be renamed (a literal that has several such copies is also assumed to be duplicated in the model that we wish to check, and further, it is assumed that their value in every state is equivalent). Hence, from now on when we speak of *literals* the meaning is *literal occurrences*. We denote by $\varphi[l \leftarrow \text{FALSE}]$ the result of replacing l in φ by FALSE, and by $lit(\varphi)$ the set of literals in φ. For a set $s \subseteq lit(\varphi)$, we define

$$\varphi^s \doteq \varphi[l \leftarrow \text{FALSE} | l \in s] \ . \tag{2}$$

Definition 1 (Vacuity) *A formula φ is satisfied vacuously in M, denoted $M \models_v \varphi$, if there exists a literal l in φ such that $M \models \varphi[l \leftarrow \text{FALSE}]$.*

Definition 2 (Mutual vacuity) *A formula φ is mutually vacuous in M with respect to a set of literals $s \subseteq lit(\varphi)$ if $M \models \varphi^s$.*

A. The vacuity lattice

A path in M is a model by itself, and therefore definitions 1 and 2 can be applied to individual paths of M. We write $vac(\pi, \varphi)$ for the set of all sets of literals in which φ is vacuous in a path π. We call it the *vacuity value* of φ in π. Formally,

$$vac(\pi, \varphi) \doteq \{s \mid \pi \models \varphi^s\} \ . \tag{3}$$

For any two sets of literals t and s, such that $t \subseteq s$, φ^t is weaker than φ^s. Thus, a vacuity value is always subset closed. We use this to represent such values by their maximal elements enclosed in $\{\!|\ |\!\}$. For example, the value $\{\{a,b\},\{a\},\{b\},\emptyset\}$ is represented as $\{\!|\{a,b\}|\!\}$.

Example 2 *Figure 2 presents several paths and their vacuity values with respect to the property $\varphi = \mathbf{G}(a \vee b \vee c \vee d)$. For example, $vac(\pi_2, \varphi) = \{\!|\{a,c\},\{a,d\},\{b,d\}|\!\}$ means that on π_2, φ is mutually vacuous separately in a, c, in a, d, and in b, d. $vac(\pi_3, \varphi) = \{\!|\emptyset|\!\}$ means that π_3 satisfies φ non-vacuously: no literal in φ can be replaced with* FALSE *without falsifying it in π_3.*

We note that the definition of an *interesting witness to non-vacuity* in [BBER01] corresponds to the case of a witness path π for φ for which $vac(\pi, \varphi) = \{\!|\emptyset|\!\}$ (in the example above π_3 is such a witness). That is, π satisfies φ non-vacuously.

We write $V(P)$ for the set of all vacuity values over a set of literals P. $V(P)$ is a lattice under subset ordering. For example, $V(\{a,b\})$ under such ordering is shown in Fig. 3. We call $V(P)$ the *positive vacuity lattice* since it corresponds to the "top half" of the full vacuity lattice used in [GC04]. We use $\sqsubseteq$, $\sqcap$ and $\sqcup$ to denote the ordering, meet, and join, respectively. For example, $\{\!|\{a\}|\!\} \sqsubseteq \{\!|\{a,b\}|\!\}$. We write MIN($S$) for the minimal elements of a set S with respect to $\sqsubseteq$.

The lattice $V(P)$ is De Morgan [Bir67]: (i) meet and join distribute over each other, i.e.,

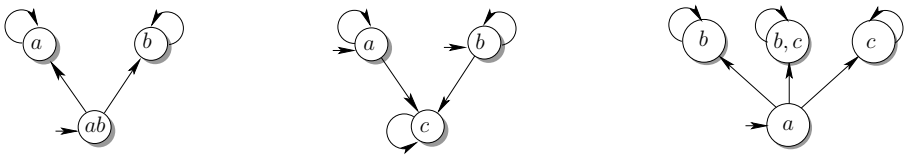
$$a \sqcap (b \sqcup c) = (a \sqcap b) \sqcup (a \sqcap c) \ ;$$

(ii) there is a De Morgan negation $\sim : V(P) \to V(P)$ such that for any lattice element a, $\sim\!\sim a = a$, and De Morgan laws hold. In our case, $\sim$ is defined as

$$\sim v \doteq \mathcal{P}(P) \setminus \{P \setminus x \mid x \in v\} \ , \tag{4}$$

where $\mathcal{P}(P)$ denotes a powerset over the elements in P. For example,

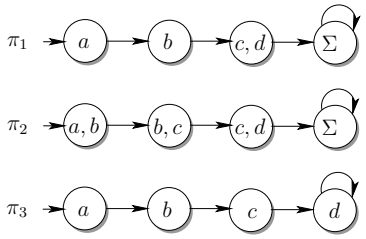
$$\sim\{\!|\{a\},\{b\}|\!\} = \mathcal{P}(\{a,b\}) \setminus \{\{b\},\{a\},\{a,b\}\} = \{\!|\emptyset|\!\} \ .$$



M_1 M_2 M_3

$\varphi_1 = \mathbf{G}(a \vee b)$ $\varphi_2 = a\,\mathbf{U}\,(b\,\mathbf{U}\,c))$ $\varphi_3 = \mathbf{G}(a \vee b \vee c)$

Fig. 1. Three example models and properties.



$vac(\pi_1, \varphi) = \{\!|\{c\}\{d\}|\!\}$

$vac(\pi_2, \varphi) = \{\!|\{a, c\}\{a, d\}\{b, d\}|\!\}$

$vac(\pi_3, \varphi) = \{\!|\emptyset|\!\}$

Fig. 2. Example values of vac for various traces and the property $\varphi = \mathbf{G}(a \vee b \vee c \vee d)$.

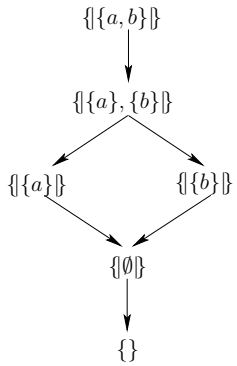


Fig. 3. The lattice corresponding to $V(\{a, b\})$.

The size of the vacuity lattice $V(P)$. The size of $V(P)$ is somewhere between a single and double exponential in the size of P. Davey and Priestley show in [DP02] a calculation of the size of such lattices, up to an input of eight elements (in this case, the size is around $5.6 \cdot 10^{23}$, whereas $2^{2^8} \approx 1.1 \cdot 10^{77}$), and note that this value is unknown for larger input sets.

B. Useful restrictions of the vacuity lattice

So far, we have only considered the vacuity lattice $V(P)$ with respect to a set of literals P. We now consider several useful restrictions thereof.

1) Given an LTL formula φ, we denote by $V(\varphi)$ the subset of $V(lit(\varphi))$ that does not render φ unsatisfiable. Formally,

$$V(\varphi) \doteq \{v \in V(lit(\varphi)) \mid \exists \pi.\ vac(\pi, \varphi) = v\}\ . \quad (5)$$

There are formulas φ for which $V(\varphi)$ is strictly smaller than $V(lit(\varphi))$. For example, let $\varphi = a\,\mathbf{U}\,(b\,\mathbf{U}\,c)$. Replacing c with FALSE makes φ unsatisfiable. Hence, $\{\!|\{c\}|\!\} \in V(\{a, b, c\}) \setminus V(\varphi)$. Specifically, for this formula, $V(\varphi)$ happens to be the lattice in Figure 3.

2) Given a model M, we denote by $V(M, \varphi)$ the subset of $V(\varphi)$ that corresponds to paths of M. Formally,

$$V(M, \varphi) \doteq \{v \in V(\varphi) \mid \exists \pi \in M.\ vac(\pi, \varphi) = v\}\ . \quad (6)$$

3) Given a model M, we denote by $V_{\min}(M, \varphi)$ the restriction of $V(M, \varphi)$ to its minimal elements. Formally,

$$V_{\min}(M, \varphi) \doteq \{v \in V(M, \varphi) \mid \neg\exists \pi \in M.\ vac(\pi, \varphi) \sqsubset v\}\ . \quad (7)$$

Thus, the restrictions we defined maintain the following relation:

$$V_{\min}(M, \varphi) \subseteq V(M, \varphi) \subseteq V(\varphi) \subseteq V(lit(\varphi))\ . \quad (8)$$

We note that $V(\varphi)$ is not necessarily a lattice (rather only a partially-ordered set), and, therefore, so are its restrictions $V(M, \varphi)$ and $V_{\min}(M, \varphi)$. The reason is that literal occurrences corresponding to the same literal are consistent in any state of the model, even if we renamed them in the formula. For example, although two complementing literals are given different names, they can never label the same state in the model. This fact results in missing values in $V(\varphi)$ (which makes it a non-lattice), as demonstrated in the following example.

Example 3 *Let $\varphi = \mathbf{G}p \vee \mathbf{G}\neg p$. Clearly, $v_1 = \{\!|\{p\}|\!\} \in V(\varphi)$ and $v_2 = \{\!|\{\neg p\}|\!\} \in V(\varphi)$, since $\pi_1 = \neg p^\omega$ is a witness for v_1 and $\pi_2 = p^\omega$ is a witness for v_2. However, neither $v_1 \vee v_2 = \{\!|\{p\}, \{\neg p\}|\!\}$, nor $v_1 \wedge v_2 = \{\!|\emptyset|\!\}$ are in $V(\varphi)$, since there is no path that is labeled with p and with $\neg p$ everywhere, and on the other hand, there is no path that satisfies φ non-vacuously. This implies that $V(\varphi)$ is not a lattice.* ⌟

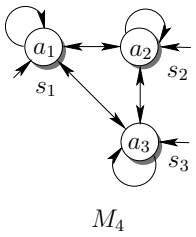


Fig. 4. A model for Example 4.

The size of $V(\varphi)$. Eq. (8) (and also Eq. (5)) implies that the size of $V(\varphi)$ is bounded by the size of $V(lit(\varphi))$ which we considered earlier. But in fact there is a tighter bound if we examine the structure of φ. As a simple example, every propositional subformula that is in the scope of a top-level conjunction falsifies φ in all models, and hence is not in $V(\varphi)$. Generalizing to temporal subformulas, we say that the left-hand side of $\mathbf{U}$ and both sides of $\mathbf{R}$ are *disjunctive*. What separates these locations is that replacing the subformula at these locations does not render the formula unsatisfiable. For example, FALSE $\mathbf{U}\, p$ is not false in all models, but $q\, \mathbf{U}$ FALSE is. Now, consider the parse tree of the formula. Denote by *disj_lit*(φ) the set of literals that on their path to the root there is a disjunction or they fall on a disjunctive side as described above. Clearly everything outside *disj_lit*(φ) is also not in $V(\varphi)$. Thus, $V(\varphi) \subseteq V(\textit{disj\_lit}(\varphi))$.

The size of $V(M,\varphi)$. Eq. (8) implies that the size of $V(M,\varphi)$ is bounded by that of $V(\varphi)$. It is also bounded by the number of paths in M, which is exponential in $|M|$ (up to unwinding), because in the worst case each path is associated with a different vacuity value. One may think that there is a much tighter upper bound on the size of $V(M,\varphi)$, but the following example shows that there are cases in which the size of $V(M,\varphi)$ is exponential with respect to the number of states of M.

Example 4 *Consider the model M_4 in Fig. 4, which has three states s_1, s_2, and s_3, with s_i labeled with a_i, for $i = 1..3$. All states of M are initial, and there are transitions between each pair of states, including self-loops. Consider the formula $\varphi = \mathbf{G}(a_1 \vee a_2 \vee a_3)$. The vacuity lattice induced by φ contains all downwards-closed sets of subsets of $L = \{a_1, a_2, a_3\}$. For each such set v there exists a path π_v in M that satisfies $vac(\pi_v, \varphi) = v$. For instance, for $v = \{\!|\{a, b\}|\!\}$, we have $\pi_v = s_3^\omega$.*

This construction can be generalized to a model with n states over the set of n literals, showing that the size of the vacuity lattice is exponential in the number of states of the model. ⌟

C. Vacuity values of sets

We now extend the definition of *vac* from individual paths to sets of paths. First, consider the set of paths that satisfy a given formula φ. We define *vac*(φ) as the meet of all the values in $V(\varphi)$:

$$\textit{vac}(\varphi) \doteq \bigsqcap_{\pi \models \varphi} vac(\pi, \varphi) = \bigsqcap_{v \in V(\varphi)} v \,. \tag{9}$$

If $\textit{vac}(\varphi) \neq \emptyset$ then φ is vacuous in any model that satisfies it. We will discuss this matter further in Sect. V.

Similarly, with respect to a given model M, we define *vac*(M, φ) as the meet over all elements of $V(M, \varphi)$:[1]

$$\textit{vac}(M, \varphi) \doteq \bigsqcap_{\pi \in M} vac(\pi, \varphi) = \bigsqcap_{v \in V(M,\varphi)} v \,. \tag{10}$$

The value $vac(M, \varphi)$ is the maximal set of sets of literals that can be replaced with FALSE simultaneously, and hence can be thought of as the goal of mutual vacuity. If $vac(M, \varphi) = \{\!|\{\emptyset\}|\!\}$ then φ is satisfied nonvacuously in M.

III. Towards the Strongest Formula

We mentioned in the introduction the fact that standard vacuity and mutual vacuity do not compute the strongest possible formulas. There are two reasons for this: (1) there can be several maximum replacements of a subset of literals with FALSE; and (2) examining the model as a whole may obscure information about the exact way in which the formula is satisfied by different paths of the model, as was demonstrated in Example 1.

With regard to the first problem, we define, for a vacuity value $v \in V(P)$:

$$C_\varphi(v) \doteq \bigwedge_{s \in v} \varphi^s \,. \tag{11}$$

As a small optimization we consider only the maximal elements in v, which results in an equivalent yet smaller formula. For instance, for π_1 in Example 2, we have $vac(\pi_1, \varphi) = \{\!|\{c\}, \{d\}|\!\}$, and hence

$$\begin{array}{l} C_\varphi(vac(\pi_1, \varphi)) = \\ \varphi[c \leftarrow \text{FALSE}] \wedge \varphi[d \leftarrow \text{FALSE}] = \\ \mathbf{G}(a \vee b \vee d) \wedge \mathbf{G}(a \vee b \vee c) \,. \end{array}$$

The formula $C_\varphi(vac(M, \varphi))$ is a conjunction over all possible results of mutual vacuity, and is hence stronger.

The following lemma is due to monotonicity of NNF formulas:

Lemma 1 *For $v_1, v_2 \in V(M, \varphi)$, $v_1 \sqsubseteq v_2$ if and only if $C_\varphi(v_2) \implies C_\varphi(v_1)$.*

With regard to the second problem, we define

$$\Phi(M, \varphi) \doteq \bigvee_{v \in V(M,\varphi)} C_\varphi(v) \,, \tag{12}$$

a disjunction over all the possible ways in which paths in M satisfy φ.

Thus, $\Phi(M, \varphi)$ improves the result of (mutual) vacuity detection in two dimensions: it strengthens the formula obtained with vacuity checks by using the conjunction of formulas

[1] The formula *vac*(M, φ) was already introduced in [GC04] in the context of CTL formulas.

$C_\varphi(v)$ for each $v \in V(M,\varphi)$, and at the same time it refines the granularity of vacuity checks, since it distinguishes between vacuity values of different paths.

Following are several observations regarding $\Phi(M,\varphi)$:

1) $\Phi(M,\varphi)$ is stronger or equal to φ, but is still weaker or equal to M. This observation is also true about formulas derived through standard vacuity checks.
2) Each path $\pi \in M$ is an interesting witness for exactly one disjunct in $\Phi(M,\varphi)$. In other words, for one disjunct φ' of $\Phi(M,\varphi)$, $vac(\pi,\varphi') = \{\!|\emptyset|\!\}$.
3) For every disjunct in $\Phi(M,\varphi)$, there is an interesting witness in M.[2]
4) $\Phi(M,\varphi)$ may contain redundant disjuncts (as will be shown in the example that follows), which means that it can be vacuous in M even if φ is not.

Example 5 *Consider again the models and properties in Fig. 1 and the suggested, stronger formulas $\varphi'_1,\ldots,\varphi'_3$ in Example 1. For the first two we have:*

$$\begin{array}{rcl} \Phi(M_1,\varphi_1) & = & \varphi'_1 \;=\; \mathbf{G}a \vee \mathbf{G}b \\ \Phi(M_2,\varphi_2) & = & \varphi'_2 \;=\; a\,\mathbf{U}\,c \vee b\,\mathbf{U}\,c \end{array}$$

For M_3 we do not reach φ'_3 yet. There are three paths in M, denoted, from left to right, by π_1,π_2 and π_3. The corresponding vacuity values are $vac(\pi_1,\varphi_3) = \{\!|\{c\}|\!\}$, $vac(\pi_2,\varphi) = \{\!|\{b\},\{c\}|\!\}$ and $vac(\pi_3,\varphi_3) = \{\!|\{b\}|\!\}$. Then,

$$\Phi(M_3,\varphi_3) = \mathbf{G}(a\vee b) \vee (\mathbf{G}(a\vee c) \wedge \mathbf{G}(a\vee b)) \vee \mathbf{G}(a\vee c)\ .$$

Indeed, π_1,π_2 and π_3 are interesting with respect to the 1-st, 2-nd, and 3-rd disjuncts in $\Phi(M_3,\varphi_3)$. The middle disjunct is redundant, which leads us to seek for a better alternative. ⌟

Generalizing the case of M_3 and φ_3 in the example above, we proceed by defining $\Phi_{\min}(M,\varphi)$, a non-vacuous equivalent of $\Phi(M,\varphi)$:

$$\Phi_{\min}(M,\varphi) \doteq \bigvee_{v\in V_{\min}(M,\varphi)} C_\varphi(v)\ . \tag{13}$$

For instance, in Example 5,

$$\Phi_{\min}(M_3,\varphi_3) = \mathbf{G}(a\vee b) \vee \mathbf{G}(a\vee c)\ , \tag{14}$$

since $\{\!|\{c\}|\!\}$ and $\{\!|\{b\}|\!\}$ are the minimal elements of $V(M_3,\varphi_3)$. This gives us φ'_3 of Example 1.

Following are several general observations about $\Phi_{\min}(M,\varphi)$:

1) We only remove from $\Phi(M,\varphi)$ those disjuncts that imply other disjuncts, and, therefore, $\Phi_{\min}(M,\varphi) \Leftrightarrow \Phi(M,\varphi)$. Hence, like $\Phi(M,\varphi)$, it is stronger or equal to φ and weaker or equal to M.
2) $\Phi_{\min}(M,\varphi)$ has no redundant disjuncts.
3) For every disjunct in $\Phi_{\min}(M,\varphi)$ there is an interesting witness in M. Hence $\Phi_{\min}(M,\varphi)$ is satisfied non-vacuously in M.
4) Not every path in M is interesting with respect to some disjunct in $\Phi_{\min}(M,\varphi)$.

[2] That is, C_φ is a surjective function from the paths of M to the disjuncts of $\Phi(M,\varphi)$.

Theorem 1 *For a model M and an LTL property φ in negation normal form, the formula $\Phi(M,\varphi)$ and its non-vacuous equivalent $\Phi_{\min}(M,\varphi)$ are the strongest LTL formulas that satisfy M and are in the positive (i.e., without negation) Boolean closure of strengthened versions of φ obtained by replacing subsets of φ's literals with* FALSE.

(Proofs can be found in a technical report [CGS08]).

The straightforward computation of $\Phi(M,\varphi)$ and $\Phi_{\min}(M,\varphi)$ considers every path of M separately, which is impractical since the number of different paths can be exponential in M. In the next section we describe two algorithms for computing $\Phi(M,\varphi)$ and $\Phi_{\min}(M,\varphi)$ that are based on automata and might be more efficient in practice.

IV. Computing $\Phi_{\min}$ with Lattice Automata

In this section we present algorithms for computing Φ and $\Phi_{\min}$, based on lattice automata.

A. Lattice automata

Lattice automata [CDG01], [KL07] are an extension of classical finite state automata that allows transitions to be labeled with values from a De Morgan algebra. As in the classical case, such automata can operate over finite or infinite words.

Definition 3 *A Lattice Nondeterministic Büchi automaton on infinite Words (LNBW) $\mathcal{A}$ is a tuple $\langle \mathcal{L},\Sigma,Q,Q_0,\delta,F\rangle$, where $\mathcal{L}$ is a De Morgan lattice, Σ is the finite alphabet, Q is a set of states, $Q_0 \subseteq Q$ is the set of initial states, $\delta : Q\times\Sigma\times Q \mapsto \mathcal{L}$ maps transitions to lattice elements, and $F \subseteq Q$ is a set of final states.*

A run of $\mathcal{A}$ on a word $w = \sigma_1\cdot\sigma_2\cdots$ is an infinite sequence $r = q_0,q_1,\ldots$ of states such that $q_0 \in Q_0$. The traversal value of r on w is

$$val(r,w) = \prod_{i\geq 0} \delta(q_i,\sigma_{i+1},q_{i+1})\ .$$

A run r is accepting iff it goes through an accepting state infinitely often:

$$acc(r) \doteq \prod_{i\geq 0}\bigsqcup_{j\geq i}(q_j \in F)\ .$$

The value of $\mathcal{A}$ on w is

$$val(\mathcal{A},w) \;\doteq \bigsqcup\{val(r,w) \mid acc(r)\}\ .$$

For brevity, we write $\mathcal{A}(w)$ for $val(\mathcal{A},w)$. The $\mathcal{L}$-language of $\mathcal{A}$ maps each word w to its value in $\mathcal{A}$. That is, $L(\mathcal{A})(w) = \mathcal{A}(w)$. The emptiness (*e_val*) and universality (*u_val*) values of a lattice automaton $\mathcal{A}$ are defined as follows:

$$\begin{array}{rl} e\_val(\mathcal{A}) & \doteq \bigsqcup\{\mathcal{A}(w) \mid w\in\Sigma^\omega\} \\ u\_val(\mathcal{A}) & \doteq \bigsqcap\{\mathcal{A}(w) \mid w\in\Sigma^\omega\}\ . \end{array}$$

Intuitively, $e\_val(\mathcal{A}) = \bot$ means that $\mathcal{A}$ is empty, and $u\_val(\mathcal{A}) = \top$ means that $\mathcal{A}$ accepts any word.

Connection with Vacuity. We say that $\mathcal{A}_{\varphi_v}$ is a *vacuity automaton of* φ if it maps each path π to the vacuity value of φ on π, i.e., $L(\mathcal{A}_{\varphi_v})(\pi) = \mathit{vac}(\pi, \varphi)$ for every path π. For every LTL formula φ there is such an automaton $\mathcal{A}_{\varphi_v}$ with the same number of states and transitions as $\mathcal{A}_\varphi$ — the classical Büchi automaton for φ.

Example 6 *Figure 5 shows a Büchi and a vacuity automaton for* $\varphi = \mathbf{G}(a \vee \mathbf{F}b)$ *with the alphabet* $\mathcal{P}(\{a, b\})$ *and the vacuity lattice* $V(\{a, b\})$*. Intuitively, at each step the automaton* $\mathcal{A}_{\varphi_v}$ *keeps track of which atomic propositions were used and which were ignored. If a value of a run is* $\{\!|\emptyset|\!\}$ *then all propositions where used at some point on it. For example, consider the word* $\pi = \{a\}, \{b\}^\omega$*. The automaton in Fig. 5(b) has two types of runs on it: a run that is stuck in the initial state has a value* $\{\!|\{b\}|\!\} \wedge \{\!|\{a\}|\!\} \wedge \{\!|\{a\}|\!\} \cdots = \{\!|\emptyset|\!\}$*, and a run that goes through the second state has a value* $\{\!|\{a\}|\!\}$*. Thus, the automaton assigns* π *the join of these values, namely* $\{\!|\{a\}|\!\}$*, meaning that* φ *is satisfied vacuously in* a *on* π*.* ∎

B. Vacuity decision problems

The decision problems and their corresponding search problems that will be discussed in this and in the next subsection, will be used in Sect. IV-D for describing algorithms for computing the elements of $V(M, \varphi)$ and $V_{\min}(M, \varphi)$.

Problem 1 *Given a model* M*, an LTL formula* φ*, and a vacuity value* v*, decide whether* $v \sqsubseteq \mathit{vac}(M, \varphi)$*.*

Solution: This problem is reducible to model-checking the formula $C_\varphi(v)$, which is defined in Eq. (11).

Example 7 *Consider a formula* φ *over alphabet* $\Sigma = \{a, b\}$ *and the corresponding lattice over* $V(\Sigma)$*, as depicted in Fig. 3. Assume that for some model* M*,*

$$V(M, \varphi) = \{\{\!|\{a, b\}|\!\},\ \{\!|\{b\}|\!\}\}\ .$$

(in other words, M *contains witnesses to exactly these vacuity values). Then,*

$$\mathit{vac}(M, \varphi) = \{\!|\{b\}|\!\}\ .$$

Let $v = \{\!|\{a\}|\!\}$*. Then, model-checking* $\varphi[a \leftarrow \text{FALSE}]$ *in* M *fails, meaning that* $v \not\sqsubseteq \mathit{vac}(M, \varphi)$*. Indeed, since* $\{\!|\{b\}|\!\} \in V(M, \varphi)$*, there exists a path* $\pi \in M$ *such that* $\mathit{vac}(\pi, \varphi) = \{\!|\{b\}|\!\}$*. The path* π *is a counterexample to* $\varphi[a \leftarrow \text{FALSE}]$ *in* M*.* ∎

Problem 2 *Given a model* M*, an LTL formula* φ*, and a vacuity value* v*, decide whether* $\forall \pi \in M.\ v = \mathit{vac}(\pi, \varphi)$*.*

Solution: To decide Prob. 2, we need to show that $v \sqsubseteq \mathit{vac}(M, \varphi)$ and that $\forall \pi \in M.\ \mathit{vac}(\pi, \varphi) \sqsubseteq v$. The first part is simply Prob. 1. For the second part, let $P_\varphi(v)$ be the set of "parents" of v in $V(\varphi)$:

$$P_\varphi(v) \doteq \{u \in V(\varphi) \mid v \sqsubset u \wedge \neg \exists u'.\ v \sqsubset u' \sqsubset u\}\ . \quad (15)$$

For example, recalling that Figure 3 represents $V(a\,\mathbf{U}\,(b\,\mathbf{U}\,c))$, the (single) parent of the vacuity value $v = \{\!|\{b\}|\!\}$ is $\{\!|\{a\}, \{b\}|\!\}$.

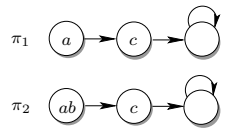


Fig. 6. Paths for Example 8.

Then, $\mathit{vac}(M, \varphi)$ is not greater than v iff it is not greater or equal than any element in $P_\varphi(v)$. This can be decided by model-checking the formula

$$D_\varphi(v) \doteq C_\varphi(v) \wedge \bigwedge_{u \in P_\varphi(v)} \neg C_\varphi(u)\ . \quad (16)$$

Note that any path π of a model M is also a model. Thus, path version of Prob. 1 and Prob. 2 in which the model M is replaced by a path π, can be solved in the same way.

Example 8 (Computing $D_\varphi(v)$) *Let* $\varphi = a\,\mathbf{U}\,(b\,\mathbf{U}\,c)$*. For the vacuity value* $v = \{\!|\{b\}|\!\}$*, we have*

$$C_\varphi(v) = a\,\mathbf{U}\,c. \quad (17)$$

The (single) parent of v *in* $V(\varphi)$ *is* $u = \{\!|\{a\}, \{b\}|\!\}$*, and hence*

$$D_\varphi(v) = (a\,\mathbf{U}\,c) \wedge \neg((a\,\mathbf{U}\,c) \wedge (b\,\mathbf{U}\,c)) = (a\,\mathbf{U}\,c) \wedge \neg(b\,\mathbf{U}\,c)\ . \quad (18)$$

In Figure 6, the vacuity value of π_1 *is* v *because* $\pi_1 \models D_\varphi(v)$*. The path* π_2*, on the other hand, does not satisfy* $D_\varphi(v)$*. Indeed, while* π_2 *satisfies* $a\,\mathbf{U}\,c$*, it also satisfies* $C_\varphi(u) = (a\,\mathbf{U}\,c) \wedge (b\,\mathbf{U}\,c)$*.* ∎

The two problems can be extended to multiple vacuity values, and in fact this is the way we are going to use them later on. Let V be a set of vacuity values. Define $C_\varphi(V) \doteq \bigvee_{v \in V} C_\varphi(v)$ and $D_\varphi(V) \doteq \bigvee_{v \in V} D_\varphi(v)$. Then, $M \models C_\varphi(V)$ iff every path of M has vacuity value greater or equal to some v in V, and $M \models D_\varphi(V)$ iff every path in M has a vacuity value equal to some v in V.

C. Vacuity search problems

We now show that the search problems corresponding to the decision problems in Sect. IV-B are reducible to computing the emptiness value of a lattice automaton.

Problem 3 *Given a path* π *and an LTL formula* φ*, compute* $\mathit{vac}(\pi, \varphi)$*.*

Solution: We assume that the path π is given as an automaton $\mathcal{A}_\pi$ that recognizes π, i.e., $L(\mathcal{A}_\pi) = \{\pi\}$. Let $\mathcal{A}_{\varphi_v}$ be the vacuity automaton of φ. Consider the product automaton $\mathcal{A}_{\varphi_v} \times A_\pi$. Its language maps every word w to $\perp$ if w is different from π, or to $\mathcal{A}_{\varphi_v}(w)$, i.e., the vacuity value of w on φ. Thus, $\mathit{e\_val}(\mathcal{A}_{\varphi_v} \times A_\pi) = \mathcal{A}_{\varphi_v}(\pi) = \mathit{vac}(\pi, \varphi)$. ∎

Problem 4 *Given a model* M *and an LTL formula* φ*, compute* $\mathit{vac}(M, \varphi)$ *(see Eq. (10)).*

Solution: We assume that the model M is given by an automaton A_M that recognizes all computations of M. We show that

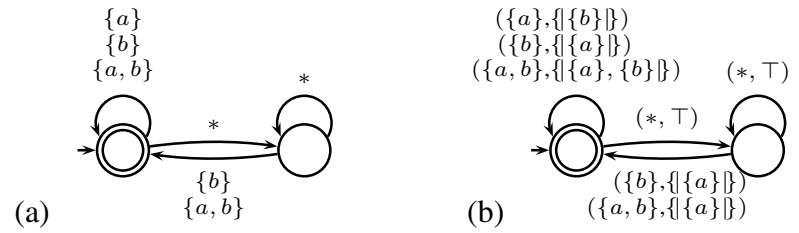


Fig. 5. Automata for $\mathbf{G}(a \vee \mathbf{F}b)$: (a) Büchi, and (b) vacuity. Transition label (x, y) indicates the letter and the value of the transition, respectively.

computing vacuity of φ in M is reducible to computing the emptiness value of the product of A_M and the complement of the vacuity automaton of φ. The reduction is similar to the classical automata-based model-checking for LTL.

Let $\mathcal{A}_{\varphi_v}$ be the vacuity automaton of φ. The complement of $\mathcal{A}_{\varphi_v}$ is an automaton $\mathcal{A}_{\overline{\varphi}_v}$ such that $L(\mathcal{A}_{\overline{\varphi}_v})(w) = \sim L(\mathcal{A}_{\varphi_v})(w)$, where $\sim$ is De Morgan negation as defined in Sect. II. By Theorem 12 in [KL07], such an automaton always exists. Consider the product automaton $A_{M\times\overline{\varphi}_v} = A_M \times \mathcal{A}_{\overline{\varphi}_v}$. Its language is:

$$L(\mathcal{A}_{M\times\overline{\varphi}_v})(w) = \begin{cases} \sim \mathit{vac}(w,\varphi) & \text{if } w \in M \\ \bot & \text{otherwise} \end{cases} \tag{19}$$

That is, a word w is mapped to $\bot$ if w is not in M, and to the complement of the vacuity value of w otherwise. The emptiness value of $A_{M\times\overline{\varphi}_v}$ is then the complement of the vacuity value of φ in M:

$$\begin{aligned} \mathit{e\_val}(A_{M\times\overline{\varphi}_v}) = \bigsqcup_{w\in M} \sim \mathit{vac}(w,\varphi) = \\ \sim \bigsqcap_{w\in M} \mathit{vac}(w,\varphi) = \sim \mathit{vac}(M,\varphi)\ . \end{aligned} \tag{20}$$

This leads to:

Theorem 2 *Let M be a model, and $\mathcal{A}_{\overline{\varphi}_v}$ the complement of the vacuity automaton of φ. Then, $\mathit{vac}(M,\varphi) = \sim \mathit{e\_val}(A_M \times \mathcal{A}_{\overline{\varphi}_v})$.*

We say that a set of lattice values W is a witness to $v = \mathit{e\_val}(\mathcal{A})$ if the following two conditions hold:

- $\forall w \in W.\ \exists \pi.\ \mathcal{A}(\pi) = w$ and
- $\bigsqcup W = v$.

We note that the emptiness value $\mathit{e\_val}(\mathcal{A})$ and its witness W can be computed through multi-valued (i.e., lattice-based) model-checking [CDEG03], [GC03].

D. Computing Φ and $\Phi_{\min}$

We now have the necessary components for presenting the algorithms for computing the sets $V(M,\varphi)$ and $V_{\min}(M,\varphi)$. The formulas $\Phi(M,\varphi)$ and $\Phi_{\min}(M,\varphi)$ are constructed from these sets following Eqs. (12) and (13), respectively.

The idea behind the algorithm for computing $V(M,\varphi)$ is to iteratively enumerate all the vacuity equivalence classes by choosing representative paths from the model. The algorithm is shown in Fig. 7(a). Checking whether a set of vacuity values V is a complete set of equivalence classes is done by model-checking $D_\varphi(V)$, as described in Sect. IV-B. Whenever V is not complete, the model-checker produces a counterexample, which is a path with a new vacuity value, not yet in V. Vacuity of the path is computed using the algorithm from Sect. IV-C.

The algorithm for computing $V_{\min}(M,\varphi)$ is shown in Fig. 7(b). There are two differences from the first algorithm: First, the algorithm only keeps track of incomparable minimal vacuity elements (line 5); Second, at every iteration it checks whether the upward closure of V is the complete set of equivalence classes using model-checking of $C_\varphi(V)$ (line 2), as described in Sect. IV-B.

We now consider a variant of the algorithm in Fig. 7(b), appearing in Fig. 7(c), which potentially progresses faster, because it updates the set V with multiple lattice elements in each iteration rather than only one. The idea is to compute the vacuity value of the formula on M, and use the witness to this value, which comprises lattice elements of $V(M,\varphi)$. In line 4, we use VAC to denote a procedure that computes the vacuity value v and a witness W such that $v = \sqcap W$ as described in Theorem 2. Note that by definition $C_\varphi(\{|\emptyset|\}) = \varphi$. Therefore the first iteration of the algorithm computes $\mathit{vac}(M,\varphi)$ which is already a formula that is stronger or equal to what can be computed by mutual vacuity.

Complexity. Constructing $V(M,\varphi)$ amounts to invoking model-checking for each lattice element in the output, i.e., in $V(M,\varphi)$. The number of sets of literals in each such element is worst-case exponential in $|\varphi|$, and hence the length of the formula to be checked is in $O(|\varphi| \cdot 2^{O(|\varphi|)})$. The worst-case complexity of constructing $V(M,\varphi)$ is thus $O(|V(M,\varphi)| \cdot |M| \cdot 2^{O(|\varphi| \cdot 2^{O(|\varphi|)})})$. The complexity of computing $V_{\min}(M,\varphi)$ is similar in the worst case, but is expected to be much better in practice, as in each iteration it removes the entire upset of the current set V.

V. Observations on the Structure of $V(\varphi)$

Recall that $V(\varphi)$ is the set of vacuity values that do not render φ unsatisfiable. We used $V(\varphi)$ in several ways so far: to restrict the size of $V(M,\varphi)$, to define $P_\varphi(v)$ in (15), and to define $\mathit{vac}(\varphi)$ in (9). We now examine several properties of this set and show connection between the structure of $V(\varphi)$ and the ways it can be satisfied.

- If $\{|\{\}|\}$ is the (single) minimal element of $V(\varphi)$, then $V(\varphi)$ admits a single interesting witness to its non-vacuity (of course, such a witness needs not to exist in every model). Otherwise, $V(\varphi)$ can either have multiple minimal elements, or have a single minimal element,

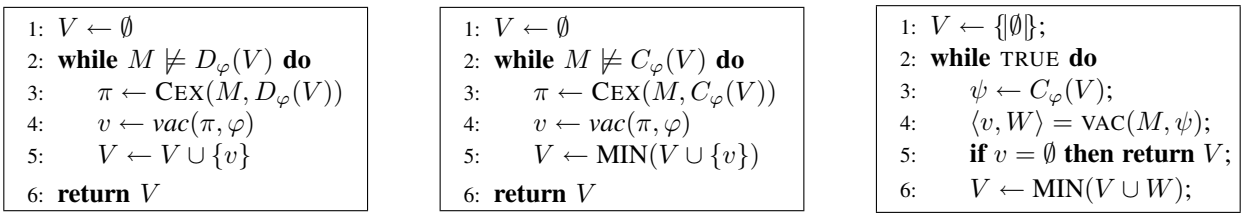


Fig. 7. Algorithms for computing (a) $V(M, \varphi)$, (b) $V_{\min}(M, \varphi)$ and (c) a potentially faster algorithm for computing $V_{\min}(M, \varphi)$.

which is not $\{\!|\{\}|\!\}$. When there are multiple minimal elements, it is impossible to construct an interesting witness to non-vacuity, but φ can be satisfied non-vacuously in M. When $V(\varphi)$ has a single minimal element which is different from $\{\!|\{\}|\!\}$, the formula φ is satisfied vacuously in all models. A (somewhat trivial) example of such a formula is $a \vee \text{TRUE}$. Indeed, $V(a \vee \text{TRUE}) = \{\!|\{a\}|\!\}$, and $a \vee \text{TRUE}$ is satisfied vacuously in all models.

- All elements that are between the maximal and the minimal elements of $V(\varphi)$ are in $V(\varphi)$ due to monotonicity of the NNF representation.
- The maximal element of $V(lit(\varphi))$, which corresponds to assigning all literals of φ the value FALSE, is not in $V(\varphi)$ for all formulas except trivial tautologies (a tautology such as $\text{TRUE} \vee \varphi'$, in which the constant TRUE remains regardless of replacements of literals). Indeed, if φ does not have a constant value, assigning FALSE to all of its literals results in an unsatisfiable formula.
- The set $V(\varphi)$ can have more than one maximal element. Indeed, consider, the specification $\varphi_1 = \mathbf{G}p \vee \mathbf{G}\neg p$ (recall that we consider each occurrence of a subformula separately). The witness π_1 for $v_1 = \{\!|\{p\}|\!\}$ is the path $(\neg p)^\omega$, and the witness π_2 for $v_2 = \{\!|\{\neg p\}|\!\}$ is the path p^ω. There is no witness, however, for the common parent of v_1 and v_2, which is $v_3 = \{\!|\{p\}, \{\neg p\}|\!\}$, as it would require a path to be labeled with p and with $\neg p$ everywhere, which is impossible[3].

VI. Conclusions

Vacuity and mutual vacuity detection is aimed at finding errors in the model, the environment, and in the specification. Frequently the specification is not strong enough with respect to the model, and these checks suggest stronger formulas to replace them. This can be useful when the model is still subject to change: future model-checking runs will be made against a tighter specification. We studied the problem of deriving formulas that are stronger than those obtained by standard vacuity and mutual vacuity checks. Further, these formulas are the strongest possible that are still satisfied by M – see Theorem 1 for a precise formulation of this claim. The complexity of the procedures that we suggest for computing these formulas is dominated by the size of $V(M, \varphi)$, which, recall, is bounded by $V(\varphi)$. As future work, it is interesting to characterize properties for which $V(\varphi)$ is small. Perhaps in such cases the worst-case double-exponential upper-bound can be reduced, which will make the algorithms useful even for large formulas.

[3]This example demonstrates the difference between formulas in which each subformula occurs only once and formulas in which subformulas can occur more than once but we treat each occurrence separately. We thank Orna Kupferman for bringing this difference to our attention.

References

[BB94] D. Beaty and R. Bryant. "Formally Verifying a Microprocessor Using a Simulation Methodology". In *Proc. of DAC'94*, pages 596–602, 1994.

[BBER01] I. Beer, S. Ben-David, C. Eisner, and Y. Rodeh. "Efficient Detection of Vacuity in Temporal Model Checking". *FMSD*, 18(2):141–162, 2001.

[Bir67] G. Birkhoff. *Lattice Theory*. AMS, 3rd edition, 1967.

[CDEG03] M. Chechik, B. Devereux, S. M. Easterbrook, and A. Gurfinkel. "Multi-Valued Symbolic Model-Checking". *TSE*, 12(4):371–408, 2003.

[CDG01] M. Chechik, B. Devereux, and A. Gurfinkel. "Model-Checking Infinite State-Space Systems with Fine-Grained Abstractions Using SPIN". In *Proc. of SPIN'01*, volume 2057 of *LNCS*, pages 16–36, Toronto, Canada, 2001. Springer.

[CE81] E.M. Clarke and E.A. Emerson. "Design and Synthesis of Synchronization Skeletons Using Branching Time Temporal Logic". In *Proc. of Workshop on Logic of Programs*, volume 131 of *LNCS*, pages 52–71. Springer, 1981.

[CGMZ95] E.M. Clarke, O. Grumberg, K.L. McMillan, and X. Zhao. "Efficient Generation of Counterexamples and Witnesses in Symbolic Model Checking". In *Proc. of DAC'95*, pages 427–432, 1995.

[CGS08] H. Chockler, A. Gurfinkel, and O. Strichman. "Beyond Vacuity: Towards the Strongest Passing Formula (full version)". Technical Report IE/IS-2008-03, Technion, IE, 2008.

[CS07] Hana Chockler and Ofer Strichman. "Easier and More Informative Vacuity Checks". In *Proc. of MEMOCODE'07*, pages 189–198. IEEE, 2007.

[DP02] B.A. Davey and H. A. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, 2002.

[GC03] A. Gurfinkel and M. Chechik. "Generating Counterexamples for Multi-valued Model-Checking". In *Proc. of FM'03*, pages 503–521, 2003.

[GC04] A. Gurfinkel and M. Chechik. "How Vacuous is Vacuous?". In *Proc. of TACAS'04*, volume 2988 of *LNCS*, pages 451–466, March 2004.

[KL07] O. Kupferman and Y. Lustig. "Lattice Automata". In *Proc. of VMCAI'07*, volume 4349 of *LNCS*, pages 199 – 213, 2007.

[Kup06] O. Kupferman. "Sanity Checks in Formal Verification". In *Proc. of CONCUR'06*, volume 4137 of *LNCS*, pages 37–51, 2006.

[KV03] O. Kupferman and M.Y. Vardi. "Vacuity Detection in Temporal Model Checking". *STTT*, 4(2):224–233, February 2003.

[LP85] O. Lichtenstein and A. Pnueli. "Checking that Finite State Concurrent Programs Satisfy their Linear Specification". In *Proc. POPL'85*, pages 97–107, January 1985.

[QS82] J.P. Queille and J. Sifakis. "Specification and Verification of Concurrent Systems in CESAR". In *Proc. of 5th Int. Symp. on Programming*, volume 137 of *LNCS*, pages 337–351, 1982.

A Theory of Mutations with Applications to Vacuity, Coverage, and Fault Tolerance

Orna Kupferman
Hebrew University
orna@cs.huji.ac.il

Wenchao Li
UC Berkeley
wenchao@berkeley.edu

Sanjit A. Seshia
UC Berkeley
sseshia@eecs.berkeley.edu

***Abstract*— The quality of formal specifications and the circuits they are written for can be evaluated through checks such as vacuity and coverage. Both checks involve mutations to the specification or the circuit implementation. In this context, we study and prove properties of mutations to finite-state systems. Since faults can be viewed as mutations, our theory of mutations can also be used in a formal approach to fault injection. We demonstrate theoretically and with experimental results how relations and orders amongst mutations can be used to improve specifications and reason about coverage of fault tolerant circuits.**

I. Introduction

Model checking [9] has proved successful in verifying the correctness of finite-state systems with respect to their specifications. In recent years, however, there has been growing awareness of the importance of challenging positive answers of model-checking tools. The main reasons include the possibility of errors or imprecision in the modeling of the system or the specification. In order to detect such errors, one needs to pose and answer questions about the quality of the modeling and the exhaustiveness of the specification.

Vacuity and *coverage* are two checks proposed to answer these questions. In vacuity, the goal is to detect cases where the system satisfies the specification in some unintended trivial way. For example, the temporal logic specification $\varphi = \mathbf{G}(req \rightarrow \mathbf{F}grant)$ ("every request is eventually followed by a grant") is satisfied in a system in which requests are never sent, but that clearly points to an error in the model or the specification. In coverage, the goal is to increase the exhaustiveness of the specification by detecting components of system behavior that do not play a role in the verification process (i.e., are "not covered by the specification"). For example, a system in which a request is followed by two grants satisfies the specification φ above, but only one of the grants plays a role in the satisfaction.

Vacuity and coverage have a lot in common; in particular, both checks involve iterating the verification procedure on a *mutated input*. In vacuity, mutations are introduced in the specification, whereas in coverage, the system is mutated. If verification succeeds even with a mutated implementation (specification), it indicates to the user that some component of the specification (implementation) is redundant. The user can then inspect the implementation and specification, possibly with a trace illustrating the redundancy, and decide how to tighten that component of the design. However, there is currently little guidance to users, in the form of automated tool support, on how the specification can be tightened to improve coverage or eliminate vacuity.

A particular challenge is posed in the verification of fault-tolerant designs. The reason is that faults can be viewed as mutations to the design. Thus, an ability to mutate the implementation and still satisfy the specification is somehow inherent in a fault-tolerant design, and need not indicate low coverage [19]. How then can we check coverage of fault-tolerant designs?

In this paper, we present a new theory of mutations, with applications to certifying the correctness of fault-tolerant circuits, checking coverage and vacuity of specifications, and potentially also to the synthesis of environment assumptions. We build upon our previous work [15], which formally proved that vacuity and coverage are *dual notions* using a basic set of mutations. Roughly speaking, a mutation μ is the dual of a mutation ν if proving lack of coverage using μ implies showing vacuity using ν, or vice-versa. Duality gives us a systematic approach to modifying the specification when low coverage is detected.

This paper makes the following novel contributions:

- *New useful mutations:* We extend the set of mutations defined in [15] and show how this extended set allows for better improvement of specifications for our target applications.
- *Properties of mutations:* We define properties of mutations such as *monotonicity* and *invertability* and explore relationships amongst them. A mutation μ is monotonic if its application on both the implementation and the specification preserves satisfaction. Mutations are inverse of each other if composing them results in the identity mutation. We prove, for example, that monotonic mutations that are inverse of each other are dual. These properties and results enable us to find more dual mutations, and to obtain simple proofs for duality of mutations.
- *Aggressiveness order between mutations:* We introduce the notion of an *aggressiveness order* between mutations. Intuitively, mutation μ is more aggressive than mutation ν, if applying μ to the implementation or specification makes satisfaction harder than applying ν. Aggressiveness orders amongst several mutations are presented. We demonstrate how the aggressiveness order can be used to improve the coverage of specifications.
- *Coverage for fault tolerant circuits:* It has been observed that some circuits are inherently fault-tolerant, in the sense that the specification is satisfied even when a fault is injected in some circuit component [19]. The impact of a fault is often only a small performance penalty which does

978-1-4244-2735-2/08/$25.00 ©2008 IEEE

not make the circuit incorrect.

Using the notion of an aggressiveness order in this setting enables us to define coverage for fault-tolerant circuits. In essence, there is low coverage if we can apply to the implementation a mutation that is more aggressive than the one it is designated to tolerate, and still satisfy the specification. The least aggressive mutation that causes the implementation to fail the specification can then be returned to the designer as a formal indicator of low coverage of the specification.

We demonstrate our approach to checking coverage and improving specifications on RTL implementations of circuits, including VIS benchmarks [20], and the Verilog design of a chip multiprocessor router [18].

Related Work. There is much previous work on vacuity and coverage, which we do not have space to survey here; we instead point the reader to some relevant papers ([1], [2], [4], [5], [7], [8], [14], [16]).

Our work is the first to formalize the connection between vacuity, coverage, and fault tolerance. It does so with a new theory of mutations, many of which are inspired by physical faults. Some past work on coverage can be expressed concisely in terms of the theory we develop here. For example, Große et al. [12] describe a method for estimating coverage for bounded model checking which can be viewed as applying the flip mutation (defined in Section III-B) at a specific cycle, and then asking the model checker for a witness. Similarly, Fummi et al. [11] use the single stuck-at fault (mutation) to estimate coverage of specifications, which is an instance of coverage checking using the stuck-at mutations we define here. The ordering amongst mutations that we derive can potentially be used to formally compare these different proposals for coverage checking. Our theory of mutations can also be used with implementation-independent techniques for estimating coverage (e.g., [10], which uses single stuck-at faults).

Outline of Paper. We begin, in Section II, with an overview of our approach, applied to a simple fault-tolerant design. Section III gives the formal model and definitions for developing our theory of mutations. In Section IV, we describe some key properties of mutations. We introduce the notion of an aggressiveness order in Section V. Section VI discusses some applications of our theory and presents experimental results.

II. Overview

We give an overview of how our mutation-based approach can be used to certify fault tolerance of the arbiter module of a chip multiprocessor (CMP) router design [18].

In the general case, the CMP router has N input channels and N output channels, where N is a parameter. We focus our discussion on the arbiter module of the router (see Figure 1), which implements a round-robin arbitration for each output channel. The vector $\vec{r}$ represents an input N-bit request vector from the input channels (r_i is high whenever channel i issues a request), and the vector $\vec{g}$ represents the corresponding output N-bit grant vector (g_i is high whenever channel i is granted access). The signals p_{ij}, for $0 \leq i < j \leq N-1$, maintain the priorities of the channels, with p_{ij} being high indicating that channel i has higher priority over input channel j The policy of the arbiter is to grant access to a channel i that requests an access only if every other channel that issues a request has lower priority than i. For example, in Figure 1, both r_0 and r_1 are high, but since p_{01} is high and no other requests are high, g_0 is high. The arbiter then inverts all p_{0j}'s in the next cycle.

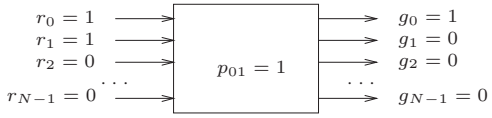


Fig. 1. The arbiter in the CMP router.

Let us consider the case $N = 2$, and the specification $\mathcal{S}$ asserting that whenever Channel 0 issues a request, then access should be granted within 12 cycles. Thus, in temporal logic, $\mathcal{S}$ is given by

$$\mathcal{S} = \mathbf{G}\,(r_0 \rightarrow \mathbf{X}^{\leq 12} g_0)$$

The implementation $\mathcal{I}$ of the CMP router was model checked against $\mathcal{S}$ and it was found that $\mathcal{I}$ satisfies $\mathcal{S}$.

We then analyzed the design for resilience to *single event upsets* (also called soft errors). A single event upset (formally defined in Section III-B) is a fault where, in any run of the system, a single state-holding Boolean element can take the opposite value of what it is supposed to take, at a single but arbitrary cycle of operation. In particular, we checked whether $\mathcal{I}$ satisfies $\mathcal{S}$ even when a soft error is injected into the arbiter priority bit p_{01} of $\mathcal{I}$. (This experiment was performed using the methodology of verification-guided error resilience [19].)

We found that the arbiter bit p_{01} is resilient to a single soft error with respect to $\mathcal{S}$. Does this mean that $\mathcal{S}$ is not a good specification?

To analyze this, we applied a more aggressive mutation, introducing up to two soft errors in bit p_{01}. It turned out that $\mathcal{S}$ is still satisfied! The designer must decide if this additional level of fault tolerance is really necessary. Let us assume that it is not – that the circuit need not be resilient to two soft errors in p_{01}. The question is: how can we improve $\mathcal{S}$?

The soft error mutation does not yet have an associated dual mutation, so in order to improve $\mathcal{S}$, we analyzed the impact of other mutations on $\mathcal{I}$ for which duality results are available. One such mutation, indicated by the form of the LTL property, is to delay the output g_0 of the arbiter module by k cycles, for increasing values of k.

It turned out that the specification $\mathcal{S}$ is satisfied even after applying these delay mutations to $\mathcal{I}$ for $k = 1, 2, 3, 4$, but not $k = 5$. The dual mutation for the delay of 4 cycles is to make the output g_0 in the specification change prematurely by 4 cycles. Applying this prematureness mutation to $\mathcal{S}$ gives us the new mutant specification $\mathcal{S}' = \mathbf{G}\,(r_0 \rightarrow \mathbf{X}^{\leq 8} g_0)$,

We then checked $\mathcal{I}$ mutated with one and two soft errors against $\mathcal{S}'$: the former satisfied $\mathcal{S}'$ but the latter did not. Thus, the new specification $\mathcal{S}'$ is a good specification for analyzing fault tolerance: it certifies the resilience of p_{01} against a single soft error, but catches the occurrence of more than one soft error.

III. Formal Model and Definitions

We present a unified formal notation to model implementations and specifications as *circuits* (Section III-A). We then

present several mutations including those proposed earlier as well as newer mutations (Section III-B).

A. Circuits

A sequential circuit (*circuit*, for short)[1] is a tuple $\mathcal{C} = \langle I, O, C, \theta, \delta, \rho \rangle$, where I is a set of input signals, O is a set of output signals, and C is a set of control signals that induce the state space 2^C. Accordingly, the three other components of the tuple are defined as:

- $\theta : 2^I \rightarrow 2^{2^C} \setminus \emptyset$ is a nondeterministic initialization function that maps every input assignment (that is, assignment to the input signals) to a nonempty set of initial states
- $\delta : 2^C \times 2^I \rightarrow 2^{2^C} \setminus \emptyset$ is a nondeterministic transition function that maps every state and input assignment to a nonempty set of possible successor states
- $\rho : O \rightarrow \mathcal{B}(C)$ is an output function that maps every output signal in O to a Boolean formula over the set of control signals; a signal $x \in O$ holds (is 1) at exactly the states $t \in 2^C$ that satisfy $\rho(x)$.

It is required that $I \cap C = I \cap O = \emptyset$. Possibly $O \cap C \neq \emptyset$, in which case for all $x \in O \cap C$, we have $\rho(x) = c$, and x is termed an observable control signal. Note that the interaction between the circuit and its environment is initiated by the environment. Once the environment generates an input assignment $i \in 2^I$, the circuit starts reacting with it from one of the states in $\theta(i)$. Note also that $\theta(i)$ and $\delta(s, i)$ are not empty for all $i \in 2^I$ and $s \in 2^C$. Thus, $\mathcal{C}$ is *receptive*, in the sense it never gets stuck.

We will interpret a state s (or an input i or output o) as a set comprising exactly those signals that evaluate to 1 in that state. Thus, $s \cup \{x\}$ denotes the state in which signal x is 1 and which agrees with s on all other signal values. Similarly, $s \setminus \{x\}$ denotes the state with signal $x = 0$ and agreeing with s on all other signal values.

Given an input sequence $\xi = i_0, i_1, \ldots \in (2^I)^\omega$, a *computation* of $\mathcal{C}$ on ξ is a word $w = w_0, w_1, \ldots \in (2^{I \cup O})^\omega$ such that there is a *path* $s = s_0, s_1, \ldots \in (2^C)^\omega$ in $\mathcal{C}$ that can be traversed while reading ξ, and w describes the input and output along this traversal. Formally, $s_0 \in \theta(i_0)$ and for all $j \geq 0$, we have $s_{j+1} \in \delta(s_j, i_j)$ and $w_j = i_j \cup \rho(s_j)$. The *language* of $\mathcal{C}$, denoted $L(\mathcal{C})$, is the union of all its computations.

Consider an implementation $\mathcal{I} = \langle I, O, C, \theta, \delta, \rho \rangle$ and a specification $\mathcal{S} = \langle I', O', C', \theta', \delta', \rho' \rangle$. We assume that $I' \subseteq I$, $O' \subseteq O$, and $C' \subseteq C$. In applications such as hierarchical refinement, the implementation may still not be precise, so we allow nondeterminism in both $\mathcal{I}$ and $\mathcal{S}$. Satisfaction of $\mathcal{S}$ in $\mathcal{I}$ can be formalized in both the linear and the branching frameworks for specification. In the linear framework, we say that $\mathcal{I}$ is *contained* in $\mathcal{S}$, denoted $\mathcal{I} \subseteq \mathcal{S}$, if $L(\mathcal{I}) \subseteq L(\mathcal{S})$. In the branching framework, we define satisfaction by means of *simulation*. A binary relation $H \subseteq 2^C \times 2^{C'}$ is a *simulation* from $\mathcal{I}$ to $\mathcal{S}$ if for all $\langle s, s' \rangle \in H$, the following conditions hold: (1) $\rho(s) \cap O' = \rho'(s')$, and (2) For each $i \in 2^I$, and $t \in \delta(s, i)$ there is $t' \in \delta'(s', i \cap I'))$ such that $H(t, t')$. We say that H is *initial with respect to $\mathcal{I}$ and $\mathcal{S}$* if for every input assignment $i \in 2^I$ and state $s \in \theta(i)$, there is a state $s' \in \theta'(i \cap I')$ such that $H(s, s')$. We say that $\mathcal{S}$ *simulates* $\mathcal{I}$, denoted $\mathcal{I} \leq \mathcal{S}$, if there is an initial simulation from $\mathcal{I}$ to $\mathcal{S}$. Intuitively, it means that $\mathcal{S}$ has more observable behaviors than $\mathcal{I}$. Formally, every universal property over the observable signals $I' \cup O'$ that is satisfied in $\mathcal{S}$ is also satisfied in $\mathcal{I}$ [3], [13]. The branching approach is stronger, in the sense that $\mathcal{I} \leq \mathcal{S}$ implies that $\mathcal{I} \subseteq \mathcal{S}$, but not vice versa. For a recent survey comparing the linear and branching approaches see [17]. We say that $\mathcal{I}$ satisfies $\mathcal{S}$, denoted $\mathcal{I} \models \mathcal{S}$ if $\mathcal{I} \subseteq \mathcal{S}$ (in the linear approach) or $\mathcal{I} \leq \mathcal{S}$ (in the branching approach).

B. Mutations on Circuits

In previous work [15], various circuit mutations have been defined. The mutations are partitioned into three classes: those that remove behaviors, modify behaviors, and add behaviors. It is possible to control the temporal pattern of mutations, specifying the cycles at which mutations occur. It is also possible to apply mutations on top of each other (like function composition).[2]

Below, we briefly describe the previously proposed mutations along with some new mutations we have devised.

Mutations that remove behaviors. Consider a specification $\mathcal{S}$ and an implementation $\mathcal{I}$ such that $\mathcal{I} \models \mathcal{S}$. If there is a mutant $\mathcal{S}'$ of $\mathcal{S}$ such that $\mathcal{S}'$ has fewer behaviors than $\mathcal{S}$ and still $\mathcal{I} \models \mathcal{S}'$, we say that $\mathcal{I}$ *vacuously satisfies* $\mathcal{S}$. Vacuous satisfaction suggests that behaviors expected by the specifier have not been exhibited in the implementation. We list some typical mutations below.

- *Arbitrary removal of behaviors*. This mutation restricts the functions θ and δ according to a given set of transitions to be removed. Mutations in this class are useful for modeling temporal-logic vacuity, for finding maximal vacuity, vacuity along computations, and arbitrary delays that can be avoided.
- *Removing behaviors that depend on a signal*. This mutation is parameterized by a signal $x \in I \cup C \setminus O$ and is obtained from the original circuit by removing transitions that depend on x. Thus, a transition stays only if it exists with both $t \setminus \{x\}$ and $t \cup \{x\}$, for either input assignments $t \in 2^I$ or states $t \in 2^C$. We term the mutation DEP_x.
- *Restricting a signal to a value*. A mutation in this class is parameterized by a signal $x \in C \cup O$ and it restricts the value of x to 0 (or 1) by disabling transitions to states in which the value of x is 1 (resp. 0). We term the mutations REST_x_TO_0 and REST_x_TO_1.

Mutations that modify behaviors. Consider a specification $\mathcal{S}$ and an implementation $\mathcal{I}$ such that $\mathcal{I} \models \mathcal{S}$. If there is a mutant $\mathcal{S}'$ of $\mathcal{S}$ such that $\mathcal{S}'$ has different behaviors than $\mathcal{S}$ and still $\mathcal{I} \models \mathcal{S}'$, or there is a mutant $\mathcal{I}'$ of $\mathcal{I}$ such that $\mathcal{I}'$ has different behaviors than $\mathcal{I}$ and still $\mathcal{I}' \models \mathcal{S}$, we say that $\mathcal{I}$ *diversely satisfies* $\mathcal{S}$. Diverse satisfaction suggests that some components in the implementation or the specification do not play a role in the satisfaction.

[1] Note that, although we focus mainly on applications to hardware verification in this paper, the formalism is a *finite-state transducer*, and the results also apply to finite-state models of software.

[2] Recall that we assume that both the implementation and its specification are given by means of circuits. As demonstrated in [15], many mutations on circuits have analogous mutations on temporal-logic formulas, which can be applied when the specification is given by means of a formula rather than a circuit.

- *Forcing a signal to be flipped*. This mutation is parameterized by a signal x, and it consistently flips the value of x; i.e., x always takes the opposite value of what it is supposed to take. We term the mutation FLIP_x.
- *Forcing a signal to get stuck*. A mutation in this class is parameterized by a signal $x \in I \cup O \cup C$ and it forces x to get stuck at 0 (or 1) by acting as if $x = 0$ (resp. $x = 1$) regardless of its actual value. For example, if $x \in C$, then the valuation of the initialization, transition, and output functions may change, as θ, δ and ρ are now applied to the evaluation containing the possibly-modified value of x. We term the mutations STICK_x_AT_0 and STICK_x_AT_1.
- *Forcing a delayed or a premature output*. A mutation in this class causes the output of the circuit to be delayed or prematured (by a fixed number of cycles, specified by the user). We term the mutations DELAY_k and PREMATURE_k, where k is the number of cycles.
- *Inserting perturbation*. A mutation in this class inserts small local perturbation to the circuit. The mutations include permuting the output sequence, stuttering the output, or applying other noise on it [15].

Mutations that add behaviors. Consider a specification $\mathcal{S}$ and an implementation $\mathcal{I}$ such that $\mathcal{I} \models \mathcal{S}$. If there is a mutant $\mathcal{I}'$ of $\mathcal{S}$ such that $\mathcal{I}'$ has more behaviors than $\mathcal{S}$ and still $\mathcal{I}' \models \mathcal{S}$, we say that $\mathcal{I}$ *loosely satisfies* $\mathcal{S}$. Loose satisfaction suggests that some components of the implementation are not covered by the specification.

- *Adding an arbitrary set of behaviors*. A mutation in this class extends the functions θ and δ according to a given set of transitions to be added. Mutations in this class are useful for detecting computations and delays that are not covered by the specifications.
- *Freeing a signal*. A mutation in this class is parameterized by a signal x and it adds to the original circuit behaviors that agree with existing behaviors of it on everything but x. We term this mutation FREE_x.

Additional mutations. The formalism used in [15] is slightly different, and the output function of a circuit there is $\rho : 2^C \rightarrow 2^O$. We have changed the formalism in order to extend the set of mutations. In our formalism, $\rho : O \rightarrow \mathcal{B}(C)$ associates with each output signal a formula over the control signals. This enables us to define a *conditional stuck-at* mutation, which, as we later show in Example 4.3, has a natural dual mutation. We also define mutations for soft errors, those that remove or add transitions according to guards in $\mathcal{B}(C)$, and variants to the DELAY_k and PREMATURE_k mutations.

- *Single event upset (SEU).* (modifies behaviors) Recall from the start of this section that we can qualify a mutation with a pattern that specifies the cycle(s) at which that mutation occurs. This pattern can reflect non-deterministic behavior. One prominent example of such a pattern-mutation is the single event upset (SEU), also called a *soft error*. A k-SEU_x is obtained by combining FLIP_x with a pattern that non-deterministically chooses to perform FLIP_x at k non-deterministically chosen cycles of operation. A 1-SEU_x is also simply called an SEU.
- *Conditional stuck-at.* (modifies behaviors) Consider an output signal $x \in O$. For a formula $\beta \in \mathcal{B}(C)$, let β_STICK_x_AT_1 be the mutation that replaces ρ_x by $\rho_x \vee \beta$, and let β_STICK_x_AT_0 be the mutation that replaces ρ_x by $\rho_x \wedge \neg\beta$. Intuitively, β_STICK_x_AT_1 sticks x to 1 in states that satisfy the condition β, whereas β_STICK_x_AT_0 sticks x to 0 in such states.
- *Conditional addition/removal of transitions.* (adds/removes behaviors) For formulas $\beta, \gamma \in \mathcal{B}(C)$, let (β, γ)_ADD be the mutation that adds transitions from all states that satisfy β to all states that satisfy γ. In the mutated circuit $\mathcal{C}_\mu$, we have $t \in \delta_\mu(s, i)$ iff $t \in \delta(s, i)$ or both $\beta(s)$ and $\gamma(t)$. Likewise, the mutation (β, γ)_REMOVE removes all transitions from states that satisfy β to states that satisfy γ.
- *Bounded delay/prematureness in output.* (modifies behaviors) DELAY_LEQ_k and PREMATURE_LEQ_k are variants of the DELAY_k and PREMATURE_k mutations. Here, the parameter k does not specify the exact delay or prematureness, but an upper bound on it. Thus, the output is delayed or prematured by *at most* k cycles.

IV. Properties of Mutations

Let M_a, M_m, and M_r be the sets of mutations that respectively add, modify, and remove behaviors. Then, $M_{imp} = M_a \cup M_m$ is the set of mutations to apply to implementations, and $M_{spec} = M_r \cup M_m$ is the set of mutations to apply to specifications. Let $M = M_a \cup M_m \cup M_r$. For a circuit $\mathcal{C}$ and a mutation μ, let $\mathcal{C}_\mu$ denote the mutant circuit obtained from $\mathcal{C}$ by applying μ.

We study and classify mutations to circuits with respect to the following properties.

1. **Duality:** Mutations $\mu \in M_{imp}$ and $\nu \in M_{spec}$ are *dual*, denoted $dual(\mu, \nu)$, if for all implementations $\mathcal{I}$ and specifications $\mathcal{S}$, we have that $\mathcal{I}_\mu \models \mathcal{S}$ iff $\mathcal{I} \models \mathcal{S}_\nu$.
   We also consider single-sided versions of the definition: If $\mathcal{I}_\mu \models \mathcal{S}$ implies $\mathcal{I} \models \mathcal{S}_\nu$, then ν is *specification dual* to μ. Also, if $\mathcal{I} \models \mathcal{S}_\nu$ implies $\mathcal{I}_\mu \models \mathcal{S}$, then μ is *implementation dual* to ν.
2. **Invertability:** Mutation $\nu \in M$ is the *inverse* of mutation $\mu \in M$ if for all circuits $\mathcal{C}$, we have that $(\mathcal{C}_\mu)_\nu = \mathcal{C}$.
   We also consider single-sided versions of the definition: Consider mutations $\nu \in M_{spec}$ and $\mu \in M_{imp}$. If for all circuits $\mathcal{C}$, we have that $\mathcal{C} \models (\mathcal{C}_\mu)_\nu$, then ν *decreases* μ. Also, if for all circuits $\mathcal{C}$, we have that $(\mathcal{C}_\nu)_\mu \models \mathcal{C}$, then μ *increases* ν.
3. **Monotonicity:** Mutation $\mu \in M$ is *monotone* if for all circuits $\mathcal{I}$ and $\mathcal{S}$, if $\mathcal{I} \models \mathcal{S}$ then $\mathcal{I}_\mu \models \mathcal{S}_\mu$.

The following theorem states useful relations amongst the above properties.

Theorem 4.1: Consider two mutations $\mu \in M_{imp}$ and $\nu \in M_{spec}$.

1. **[Specification duality]** If ν is monotone, and ν decreases μ, then ν is specification dual to μ.
2. **[Implementation duality]** If μ is monotone, and μ increases ν, then μ is implementation dual to ν.
3. **[Duality]** If μ and ν are monotone and inverse of each other, then they are dual.

Proof: We start with the first claim. We have to prove that for all implementations $\mathcal{I}$ and specifications $\mathcal{S}$, we have

that $\mathcal{I}_\mu \models \mathcal{S}$ implies $\mathcal{I} \models \mathcal{S}_\nu$. Assume that $\mathcal{I}_\mu \models \mathcal{S}$. By monotonicity, $(\mathcal{I}_\mu)_\nu \models \mathcal{S}_\nu$. Since ν decreases μ, we have $\mathcal{I} \models (\mathcal{I}_\mu)_\nu$. Hence, by transitivity of $\models$, also $\mathcal{I} \models \mathcal{S}_\nu$.

The proof of the second claim is similar. We have to prove that for all implementations $\mathcal{I}$ and specifications $\mathcal{S}$, we have that $\mathcal{I} \models \mathcal{S}_\nu$ implies $\mathcal{I}_\mu \models \mathcal{S}$. Assume that $\mathcal{I} \models \mathcal{S}_\nu$. By monotonicity, $\mathcal{I}_\mu \models (\mathcal{S}_\nu)_\mu$. Since μ increases ν, we have that $(\mathcal{S}_\nu)_\mu \models \mathcal{S}$. Hence, by transitivity of $\models$, also $\mathcal{I}_\mu \models \mathcal{S}$.

By definition, if μ is the inverse of ν, then μ both increases and decreases ν. Thus, the third claim follows from the first two claims. ■

The conditions in the above theorem are sufficient for duality, but they are not necessary. To see this, let $\mu \in M_{imp}$ be the mutation that adds behaviors that dualize the value of a signal x. Also, let $\nu \in M_{spec}$ be the mutation DEP_x that removes behaviors that depend on signal x. In [15], the authors show that ν is specification dual to μ. On the other hand, while μ is monotone, it does not increase ν. Indeed $(\mathcal{C}_\nu)_\mu = \mathcal{C}_\nu$, since if we remove all behaviors dependent on x, there is nothing to add back.

We now illustrate the use of the above theorem in proving duality results about mutations introduced in Section III-B.

For lack of space, we omit the full proofs of both propositions. The proofs are based on showing that the studied mutation μ is monotone and dual to itself (Proposition 4.2) or that the studied mutations μ and ν are monotone and increase or decrease each other (Proposition 4.3).

Proposition 4.2: **[Flipping the value of a signal is self-dual]** Let μ be FLIP_x, for $x \in I$. The mutation μ is dual to itself.

Proof sketch: Let μ be FLIP_x, for $x \in I$. The mutation μ is clearly the inverse of itself. Also, it is easy to see that μ is monotone, because applying a flip to both the implementation and specification changes their runs in exactly the same way, so that trace containment or simulation is maintained.

Thus μ is dual to itself. □

In earlier work [15], it was shown that FLIP_x is also self-dual for $x \in O$ and $x \in C$ under certain reasonable conditions. The above proposition is just an illustration of the use of the newly defined properties of mutations.

Recall that it is possible to control the temporal pattern in which mutations are applied. For example, we can flip the value of x at exactly all even positions or at specific cycles. When the pattern is deterministic (as in "flip in all even positions" but not as in "flip exactly once at non-deterministically chosen cycle"), then monotonicity and invertability are maintained, and so is the duality.

Proposition 4.3: **[Conditional stuck-at is dual]** Consider a signal $x \in O$ and a formula $\beta \in \mathcal{B}(C)$. Let μ be β_STICK_x_AT_1 and ν be β_STICK_x_AT_0. If $\rho_x \rightarrow \neg\beta$ and β is over observable control signals, then ν is specification dual to μ. Similarly, if $\beta \rightarrow \rho_x$ and β is over observable control signals, then μ is implementation dual to ν.

Proof sketch: It is easy to see that ν is the inverse of μ when $\rho_x \rightarrow \neg\beta$ and μ is the inverse of ν when $\beta \rightarrow \rho_x$: this is obtained by syntactically evaluating $((\rho_x)_\mu)_\nu$ and $((\rho_x)_\nu)_\mu$ and verifying that they simplify to ρ_x.

The monotonicity follows from the crucial observation that the implementation and specification agree on the values of the observable control signals, and hence on the values of β at each step. Hence, they will continue to agree on the values of observable signals even after mutation by ν (for specification duality) or μ (for implementation duality). □

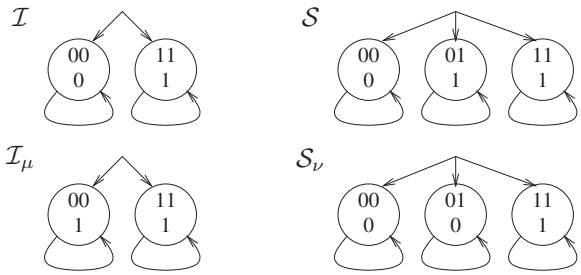


Fig. 2. Duality for conditional stuck-at mutation.

Note that, in the first case, if a designer has in mind a condition γ that is not disjoint from ρ_x, he can take $\beta = \gamma \wedge \neg\rho_x$ as the condition that would satisfy $\rho_x \rightarrow \neg\beta$. Likewise, in the second case, if a designer has in mind a condition γ that does not imply ρ_x, he can take $\beta = \gamma \wedge \rho_x$ as the condition that satisfies $\beta \rightarrow \rho_x$.

Example 1: Consider the implementation $\mathcal{I}$ and specification $\mathcal{S}$ in Figure 2. In both, $I = \emptyset$, $C = \{y, z\}$, and $O = \{x, y\}$. In the figure, the upper line describe the values of y and z, and the bottom line describes the value of x. In the specification, the labeling function $\rho_x^{\mathcal{S}}$ of x is $y \vee z$, whereas in the implementation, $\rho_x^{\mathcal{I}}$ is $y \wedge z$. It is easy to see that $\mathcal{I}$ satisfies $\mathcal{S}$.

Assume we apply to $\mathcal{I}$ a mutation μ that sticks x to 1 whenever $y = 0$. Thus, $\beta = \neg y$ and in the mutated implementation $\mathcal{I}_\mu$, we have that $(\rho_x^{\mathcal{I}})_\mu$ is $(y \wedge z) \vee \neg y$. Note that $\mathcal{I}_\mu \models \mathcal{S}$ and, as required, $\rho_x^{\mathcal{I}} \rightarrow \neg\beta$ and β is over observable control signals. Thus, the mutation ν that replaces $\rho_x^{\mathcal{S}}$ by $\rho_x^{\mathcal{S}} \wedge \neg\beta$ is specification-dual to μ. In the mutated specification $\mathcal{S}_\nu$, we have that $(\rho_x^{\mathcal{S}})_\nu$ is $(y \vee z) \wedge \neg\neg y = y$, and, as guaranteed from the duality, $\mathcal{I} \models \mathcal{S}_\nu$.

Remark 1: Let μ and ν be dual mutations. Suppose that $\mathcal{I}_\mu \models \mathcal{S}$. Then, we know that $\mathcal{I} \models \mathcal{S}_\nu$. What do we know about the satisfaction of $\mathcal{S}_\nu$ in $\mathcal{I}_\mu$? It turns out that for some dual mutations, $\mathcal{I}_\mu \models \mathcal{S}_\nu$ is equivalent to $\mathcal{I} \models \mathcal{S}$ (for example, when $\mu = \nu =$FLIP_x), for some it "doubles the effect" of the mutation (for example, when μ is DELAY_k and ν is PREMATURE_k), and for some it is equivalent to checking $\mathcal{I}_\mu \models \mathcal{S}$ or $\mathcal{I} \models \mathcal{S}_\nu$ (for example, when μ is FREE_x and ν is DEP_x), and for some, it cannot be that $\mathcal{I}_\mu \models \mathcal{S}_\nu$ (for example, in conditional stuck at). Thus, it is impossible to come up with a general recommendation about such a check. □

V. Aggressiveness Order Between Mutations

We now present a set of aggressiveness orders between many of the mutations we have presented in this paper. These orders help to organize the space of mutations, and are especially useful in reasoning about fault-tolerant circuits, as we will demonstrate in Section VI.

The aggressiveness order between a pair of mutations is a partial order parameterized by a *polarity*. The polarity indicates whether the mutations are applied to the implementation

(in which case, the mutations are in M_{imp}, and the more behaviors the mutation adds, the more aggressive it is) or to the specification (in which case, the mutations are in M_{spec} and the more behaviors the mutation removes, the more aggressive it is). We first define the aggressiveness order and study its properties, and then describe some natural orders.

Definition 1: **[Aggressiveness order for implementations]** Let μ and ν be mutations in M_{imp}. We say that μ is *more implementation-aggressive than* ν, denoted $\mu \geq_{imp} \nu$, if for every implementation $\mathcal{I}$ and specification $\mathcal{S}$, we have that $\mathcal{I}_\mu \models \mathcal{S}$ implies $\mathcal{I}_\nu \models \mathcal{S}$.

Definition 2: **[Aggressiveness order for specifications]** Let μ and ν be mutations in M_{spec}. We say that ν is *more specification-aggressive than* μ, denoted $\nu \geq_{spec} \mu$, if for every implementation $\mathcal{I}$ and specification $\mathcal{S}$, we have that $\mathcal{I} \models \mathcal{S}_\nu$ implies $\mathcal{I} \models \mathcal{S}_\mu$.

Intuitively, $\mu \geq_{imp} \nu$ means that the mutation μ makes it harder for the implementation to satisfy the specification. Dually, $\nu \geq_{spec} \mu$ means that the mutation ν makes it harder for the specification to be satisfied by the implementation. Formally, we have the following.

Theorem 5.1: Consider mutations μ and ν. The following are equivalent: (1) For every circuit $\mathcal{C}$, we have $C_\nu \models \mathcal{C}_\mu$ (2) $\mu \geq_{imp} \nu$ (3) $\nu \geq_{spec} \mu$.

Proof: We first prove that (1) implies (2) and (3). Consider an implementation $\mathcal{I}$ and specification $\mathcal{S}$. Assume that $\mathcal{I}_\mu \models \mathcal{S}$. By the assumption (1), we have that $\mathcal{I}_\nu \models \mathcal{I}_\mu$. Now, transitivity of $\models$ implies that $\mathcal{I}_\nu \models \mathcal{S}$, thus $\mu \geq_{imp} \nu$. Assume now that $\mathcal{I} \models \mathcal{S}_\nu$. By (1), we have $\mathcal{S}_\nu \models \mathcal{S}_\mu$. The transitivity of $\models$ implies that $\mathcal{I} \models \mathcal{S}_\mu$, thus $\nu \geq_{spec} \mu$.

We now prove that (2) implies (1). For a circuit $\mathcal{C}$, we refer to $\mathcal{C}$ as an implementation and to $\mathcal{C}_\mu$ as its specification. By the assumption (2), if $\mathcal{C}_\mu \models \mathcal{C}_\mu$, then $\mathcal{C}_\nu \models \mathcal{C}_\mu$. Hence, by the reflexivity of $\models$, we have $\mathcal{C}_\nu \models \mathcal{C}_\mu$.

It is left to prove that (3) implies (1). For a circuit $\mathcal{C}$, we refer to $\mathcal{C}_\nu$ as an implementation and to $\mathcal{C}$ as its specification. By the assumption (3), if $\mathcal{C}_\nu \models \mathcal{C}_\nu$, then $\mathcal{C}_\nu \models \mathcal{C}_\mu$. Hence, by the reflexivity of $\models$, we have $\mathcal{C}_\nu \models \mathcal{C}_\mu$. ■

Aggressiveness orders arise in a few different ways. We explore each of these ways in a separate subsection below.

A. Syntactic Aggressiveness

Syntactic aggressiveness involves mutations that change the syntactic structure of a circuit, such as deleting transitions, or changing conditionals.

We say that a mutation μ is *clean* if it only adds and/or removes transitions of the circuit. That is, for every circuit $\mathcal{C}$, there is a pair $\langle a_\delta, r_\delta \rangle$ of functions $a_\delta, r_\delta : 2^C \times 2^I \to 2^{2^C}$ such that the circuit obtained form $\mathcal{C}$ by applying μ is $\mathcal{C}' = \langle I, O, C, \theta, \delta', \rho \rangle$, where for all $i \in 2^I$ and $s \in 2^C$, we have that $\delta'(s,i) = \delta(s,i) \cup a_\delta(s,i) \setminus r_\delta(s,i)$. Thus, $\mathcal{C}'$ is obtained from $\mathcal{C}$ by increasing the nondeterminism in δ according to a_δ and them decreasing it according to r_δ. Note that the application of μ may result in a circuit that is not receptive.[3]

Intuitively, the more transitions a mutation adds or removes, the more aggressive it is. Formally, we have to consider also the polarity of the aggressiveness, and we have the following. Consider two clean mutations $\mu = \langle a_\delta, r_\delta \rangle$ and $\nu = \langle a'_\delta, r'_\delta \rangle$. If $a'_\delta \subseteq a_\delta$ and $r_\delta \subseteq r'_\delta$, then we say that μ is *syntactically more implementation-aggressive than* ν. Dually, if $a_\delta \subseteq a'_\delta$ and $r'_\delta \subseteq r_\delta$, then we say that ν is *syntactically more specification-aggressive than* μ.

Proposition 5.2: **[Syntactic aggressiveness implies aggressiveness]** If μ is syntactically more implementation-aggressive than ν, or ν is syntactically more specification-aggressive than μ, then $\mu \geq_{imp} \nu$ and $\nu \geq_{spec} \mu$.

Proof: Assume that μ is syntactically more implementation-aggressive than ν, or that ν is syntactically more specification-aggressive than μ. Thus, $a'_\delta \subseteq a_\delta$ and $r_\delta \subseteq r'_\delta$, or $a_\delta \subseteq a'_\delta$ and $r'_\delta \subseteq r_\delta$. It is easy to see that in both cases, for every circuit $\mathcal{C}$, we have $\mathcal{C}_\nu \models \mathcal{C}_\mu$. Indeed, in the branching approach, we can restrict the identity relation to an initial simulation, and in the linear approach all the computations of $\mathcal{C}_\nu$ have matching computations of $\mathcal{C}_\mu$. Hence, by Theorem 5.1, $\mu \geq_{imp} \nu$ and $\nu \geq_{spec} \mu$. ■

Since adding and removing transitions is a clean mutation, and the more we add/remove the more syntactically aggressive we are, Proposition 5.2 implies the following.

Proposition 5.3: If $\beta_1 \to \beta_2$ and $\gamma_1 \to \gamma_2$, then (β_2, γ_2)_ADD $\geq_{imp}$ (β_1, γ_1)_ADD and (β_2, γ_2)_REMOVE $\geq_{spec}$ (β_1, γ_1)_REMOVE.

B. Orders for Mutations on Signals

We exhibit aggressiveness orders for different kinds of mutations involving signals of the circuit.

The first set of orders are given in the following proposition.

Proposition 5.4:

- DEP_x $\geq_{spec}$ REST_x_TO_0 $\geq_{spec}$ STICK_x_AT_0,
- DEP_x $\geq_{spec}$ REST_x_TO_1 $\geq_{spec}$ STICK_x_AT_1.
- DEP_x $\geq_{spec}$ FLIP_x.
- For all $k \geq 1$, DELAY_LEQ_$k+1$ $\geq_{imp}$ DELAY_LEQ_k.

Proof sketch: Recall that in REST_x_TO_0, we disable transitions to states in which the value of x is not 0, whereas in STICK_x_AT_0, we leave such states reachable but behave as if the value of x in them is 0. Also, in DEP_x, we disable all transitions that depend on the value of x. Hence, it is not hard to prove that for every circuit $\mathcal{C}$, we have $\mathcal{C}_{\text{DEP\_}x} \models \mathcal{C}_{\text{REST\_}x\text{\_TO\_0}} \models \mathcal{C}_{\text{STICK\_}x\text{\_AT\_0}}$ in both the linear and branching approaches (and similarly with REST_x_TO_1 and STICK_x_AT_1). Hence, by Theorem 5.1, the aggressiveness orders follow.

It is also clear that, for all $k \geq 1$, $\mathcal{C}_{\text{DELAY\_LEQ\_k+1}}$ exhibits a superset of behaviors of $\mathcal{C}_{\text{DELAY\_LEQ\_k}}$. Therefore, DELAY_LEQ_$k+1$ $\geq_{imp}$ DELAY_LEQ_k. □

Remark 2: Note that it is not the case that DELAY_$k+1$ $\geq_{imp}$ DELAY_k. Indeed, when the mutation specifies an exact bound, a larger delay may lead to satisfaction of properties that are not satisfied in a smaller delay. A related phenomenon in temporal-logic vacuity is the fact that there is no order between vacuity detection that is done by mutating a single occurrence of a sub-formula and multiple occurrences of it. □

The implementation-aggressiveness order is dual to the specification-aggressiveness order:

Theorem 5.5: If $dual(\mu, \tilde{\mu})$ and $dual(\nu, \tilde{\nu})$, then $\mu \geq_{imp} \nu$ iff $\tilde{\mu} \geq_{spec} \tilde{\nu}$.

[3] In the definition above, we ignored initial transitions described by θ. Extending the definition with $a_\theta, r_\theta : 2^I \to 2^{2^C}$ is straightforward.

Proof: Assume that $\mu \geq_{imp} \nu$. Consider an implementation $\mathcal{I}$ and specification $\mathcal{S}$. Assume that $\mathcal{I} \models \mathcal{S}_{\tilde{\mu}}$. By the duality of μ and $\tilde{\mu}$, we have that $\mathcal{I}_\mu \models \mathcal{S}$. Since $\mu \geq_{imp} \nu$, it follows that $\mathcal{I}_\nu \models \mathcal{S}$. The duality of ν and $\tilde{\nu}$ implies that $\mathcal{I} \models \mathcal{S}_{\tilde{\nu}}$, and we are done. The other direction is dual. ■

It is shown in [15] that FREE_x and DEP_x are dual, and FLIP_x is self-dual. Hence, by Proposition 5.4 and Theorem 5.5, FREE_$x \geq_{imp}$ FLIP_x. Likewise, since DELAY_LEQ_k and PREMATURE_LEQ_k are dual, Theorem 5.5 implies that PREMATURE_LEQ_$k+1 \geq_{spec}$ PREMATURE_LEQ_k.

Note that FREE_$x \geq_{imp}$ STICK_x_AT_1 and FREE_$x \geq_{imp}$ STICK_x_AT_0, because the FREE_x mutation adds behaviors that stick x to 1 as well as those that stick x to 0.

Further, note that the FREE_x mutation is equivalent to the mutation that non-deterministically, at each step, either flips x or doesn't. In other words, FREE_x is identical to allowing an infinite number of SEUs to occur in x. This yields that FREE_$x \geq_{imp}$ k-SEU_x for all $k \geq 1$.

C. Duality and Aggressiveness

For mutations that do not have duals that are independent of the circuit $\mathcal{I}$, it is useful to define an *implementation-specific duality*. This notion has an associated *circuit-dependent aggressiveness order*.

Definition 3: **[Dual aggressiveness]** Consider a mutation $\mu \in M_{imp}$. For a circuit $\mathcal{I}$ and a mutation $\nu_\mathcal{I} \in M_{spec}$, we say that $\nu_\mathcal{I}$ *dualizes* μ *for* $\mathcal{I}$ if, for all specifications $\mathcal{S}$, we have that $\mathcal{I}_\mu \models \mathcal{S}$ implies that $\mathcal{I} \models \mathcal{S}_{\nu_\mathcal{I}}$. Then, we say that a mutation $\nu \in M_{spec}$ is *more dual-implementation-aggressive* than μ if for all implementations $\mathcal{I}$ and mutations $\nu_\mathcal{I}$ that dualize μ for $\mathcal{I}$, we have $\nu \geq_{spec} \nu_\mathcal{I}$.

Example 2: Consider the mutation μ = (β,γ)_ADD. Assume that β and γ are over observable control signals. For a given implementation $\mathcal{I}$, let $\nu_\mathcal{I}$ be the mutation that removes exactly all the transitions added by (β,γ)_ADD. Note that since $\mathcal{I}$ may contain transitions from states that satisfy β to states that satisfy γ, the mutation $\nu_\mathcal{I}$ does not coincide with the mutation (β,γ)_REMOVE. Indeed, the latter removes all transitions between states that satisfy β to states that satisfy γ, and not only these added by μ. It is easy to see, however, that (β,γ)_REMOVE is syntactically more specification aggressive than $\nu_\mathcal{I}$, and hence, by Proposition 5.2, (β,γ)_REMOVE is more specification aggressive than $\nu_\mathcal{I}$.

Since β and γ are over observable control signal, the mutation $\nu_\mathcal{I}$ is monotonic. Also, since $(\mathcal{I}_\mu)_{\nu_\mathcal{I}} = \mathcal{I}$, it follows that for every specification $\mathcal{S}$, if $\mathcal{I}_\mu \models \mathcal{S}$ then $\mathcal{I} \models \mathcal{S}_{\nu_\mathcal{I}}$. Thus, $\nu_\mathcal{I}$ dualizes μ for $\mathcal{I}$. Hence, as (β,γ)_REMOVE is more specification aggressive than $\nu_\mathcal{I}$ for all implementations $\mathcal{I}$ and mutations $\nu_\mathcal{I}$, we conclude that (β,γ)_REMOVE is more dual-implementation-aggressive than μ.

For easy reference, we summarize some useful aggressiveness orders in Table I. Each row of the table states an aggressiveness order along with any conditions under which it holds.

D. Coverage for Fault-Tolerant Systems

We now describe how coverage of specifications can be computed for fault-tolerant circuits.

Faults mutate the implementation. Thus, an implementation $\mathcal{I}$ is said to tolerate faults modeled by $\mu_1, \mu_2, \ldots, \mu_k$ in M_{imp} if $\mathcal{I}$ continues to satisfy its specification even in the presence of these faults.

Aggressiveness order	Conditions
DEP_$x \geq_{spec}$ REST_x_TO_0 $\geq_{spec}$ STICK_x_AT_0	none
DEP_$x \geq_{spec}$ REST_x_TO_1 $\geq_{spec}$ STICK_x_AT_1	none
DEP_$x \geq_{spec}$ FLIP_x	none
PREMATURE_LEQ_$k+1 \geq_{spec}$ PREMATURE_LEQ_k	$k \geq 1$
DELAY_LEQ_$k+1 \geq_{imp}$ DELAY_LEQ_k	$k \geq 1$
FREE_$x \geq_{imp}$ STICK_x_AT_1	none
FREE_$x \geq_{imp}$ STICK_x_AT_0	none
FREE_$x \geq_{imp}$ FLIP_x	none
FREE_$x \geq_{imp}$ k-SEU_x	$k \geq 1$
(β_2,γ_2)_ADD $\geq_{imp}$ (β_1,γ_1)_ADD	$\beta_1 \rightarrow \beta_2$
(β_2,γ_2)_REMOVE $\geq_{spec}$ (β_1,γ_1)_REMOVE	and $\gamma_1 \rightarrow \gamma_2$

TABLE I
SUMMARY OF AGGRESSIVENESS ORDERS

Note that by Theorem 5.1, the above definition also means that $\mathcal{I}$ continues to satisfy its specification when less aggressive faults are applied to it.

Formally, we model a *fault-tolerant* system by an implementation $\mathcal{I}$ along with a set of mutations $\mu_1, \mu_2, \ldots, \mu_k$, such that for all specifications $\mathcal{S}$, if $\mathcal{I} \models \mathcal{S}$, then $\mathcal{I}_{\mu_j} \models \mathcal{S}$ for for all $1 \leq j \leq k$.

Measuring the coverage of a fault-tolerance system is challenging, as satisfaction of a specification under mutations (loose satisfaction) need not imply low coverage. Thus, in the context of fault-tolerant systems, we have to redefine loose satisfaction to take the tolerance into account and apply to the system mutations that are more implementation-aggressive than those it tolerates. Formally, we say that a fault tolerant system $\langle \mathcal{I}, \{\mu_1, \mu_2, \ldots, \mu_k\}\rangle$, *loosely satisfies* a specification $\mathcal{S}$, if there is a mutation μ such that $\mu_j \not\geq_{imp} \mu$ for all $1 \leq j \leq k$, and $\mathcal{I}_\mu \models \mathcal{S}$.

In particular, if $\mathcal{I}_\mu$ satisfies $\mathcal{S}$ and $\mathcal{I}$ is designed to be tolerant only to μ, and further if $\mathcal{I}_\nu$ satisfies $\mathcal{S}$ for $\nu \geq_{imp} \mu$, then we say that $\mathcal{S}$ has low coverage.

We will see in the next section how the theory of this section can be applied in practice.

VI. EXPERIMENTAL RESULTS

We now present experimental results demonstrating the application of our theory of mutations in the context of fault-tolerant circuits. We demonstrate how analysis of fault tolerance can also provide vacuity and coverage information. Due to lack of space, we describe specifications using LTL rather than draw the finite-state transducers – the correspondence with LTL is described in [15].[4]

Experiments were performed using the Cadence SMV model checker with option "-absref3" (BDD-based counterexample guided abstraction refinement) on an Intel Core2Duo 2 GHz machine. Each model checking run involved checking whether a possibly-mutated implementation satisfied a possibly-mutated specification. For each experiment on a benchmark, we have an associated set M comprising mutations of interest. For example, to check coverage of a circuit

[4]We note that in order to handle full LTL, one has to consider circuits with fairness, and, accordingly, extend the containment and simulation relations to ones that account for fairness. In our experiments, the only LTL formulas we mutated were safety LTL formulas, for which fairness is not needed.

Benchmark Benchmark	Number of latches	Number of LTL properties	Coverage (%)
Ibuf	6	8	83.3
Lock	9	2	55.6
Am2910	99	4	99.0
SyncArb	48	48	33.3

TABLE II
FLIP_x MUTATION ON VIS BENCHMARKS

with respect to the SEU mutation, the set M comprises an 1-SEU_x mutation for every latch x in the design. Thus, for each benchmark, a coverage experiment would involve the following two steps for each element μ of the set M of mutations: (a) automatically create a new SMV model of the circuit mutated with μ, and then (b) use SMV to check whether the mutated model satisfies its specification.

The scalability of our approach is naturally dependent on the scalability of model checking for the circuits of interest. In our experiments, for any benchmark and set of mutations M, the total run-time was on the order of a few minutes; in fact, for each benchmark listed in Tables II-IV, the total run-time of all experiments performed on that benchmark was less than 500 seconds. We believe that incremental approaches to model checking that re-use work performed in different runs can be quite effective at further reducing run-time, since mutations tend to be local and mutated models tend to be similar.

A. Vacuity from Coverage

The first experiment illustrates how useful vacuity information can be obtained for free once coverage has been checked. A number of benchmarks are selected from the VIS benchmark suite [20]. We use the mutation FLIP_x to investigate the resilience of latches in the circuits with the given specifications. Hence, a latch x is *covered* by a specification $\mathcal{S}$ with respect to FLIP_x if, after mutation by FLIP_x, the circuit fails $\mathcal{S}$.

Table II summarizes our results. The coverage numbers in the table give the percentage of latches which, when mutated, caused the circuit to fail its specification. We note that none of the four circuits have 100% coverage[5]. This suggests that at least some of the latches are resilient to FLIP_x in these circuits. A verification engineer might wonder: why are they resilient? At this stage, the engineer is usually left with two choices. Either she can try to write more specifications and hope they will cover those latches, or she can go back to the circuit designer and investigate whether they were intended to be fault-tolerant. These two approaches are both time-consuming. We propose that we can use *duality* to help answer this question.

Recall from Section IV that the FLIP_x mutation is self-dual. Consider the following LTL specification from the benchmark "Am2910".

$$spec: \ assert\ \mathbf{G}(sp[2..0] = 110 \rightarrow \mathbf{X}(sp[2..0] = 111));$$

In our experiment, applying the FLIP_x mutation to $x = sp[0]$ in the circuit still satisfies the above specification.

[5]The LTL specifications used here are the ones that verified to be true for the original circuit in SMV. Hence the number of specifications may be smaller than the one found in the benchmark suite.

Benchmark	1-SEU_x coverage	20-SEU_x coverage	FREE_x coverage
Ibuf	83.3%	83.3%	100%
Lock	0%	0%	100%
Am2910	99.0%	99.0%	99.0%
SyncArb	33.3%	33.3%	66.7%

TABLE III
COVERAGE OF k-SEU_x VS. FREE_x

Benchmark	STICK_x_AT_0 Coverage	FREE_x Coverage
Ibuf	83.3%	100%
Lock	100%	100%
Am2910	97.0%	99.0%
SyncArb	66.7%	66.7%

TABLE IV
COVERAGE OF STICK_x_AT_0 VS. FREE_x

Thus, applying the dual mutation (FLIP_x again) to the above LTL specification gives us the property below, because the transition from the state 110 goes to 110 after the flip in the 0th bit of sp instead of 111.

$$spec': \ assert\ \mathbf{G}(sp[2..0] = 110 \rightarrow \mathbf{X}(sp[2..0] = 110));$$

The only way that both specifications are satisfied is that they are *vacuously* true, i.e. $sp[2..0]$ never reaches 6. We verified that indeed $\mathbf{G}(\neg sp[2..0] = 110)$ was satisfied.

B. Coverage for Fault-Tolerant Circuits

The second experiment illustrates how the *aggressiveness orders* can be used to analyze coverage in fault-tolerant circuits. We use the same VIS benchmarks from the previous experiment, but apply a 1-SEU_x mutation to each latch x in each benchmark. We also tried applying k-SEU_x for increasing k, as well as the FREE_x mutation. Table III shows our coverage results for three mutations on the VIS benchmarks (we only show $k = 1$ and $k = 20$). As argued in the preceding section, FREE_x $\geq_{imp}$ 20-SEU_x $\geq_{imp}$ 1-SEU_x

We considered specifications of the form $\mathbf{G}(\alpha \rightarrow F\ \beta)$, and found cases where the circuits are tolerant to a large number of of SEUs but not to an infinite number of them (the FREE_x). Hence, we can certify these specifications as being covered by the FREE_x mutation, and the circuit is tolerant to k-SEU_x for any k, but not to the FREE_x mutation. Thus, the FREE_x mutation defines a "lower bound" for the specification.

Observe that for benchmark "Am2910", coverage is not 100% even if we make a latch totally nondeterministic (the FREE_x mutation). This result is due to the vacuity bug found in the previous experiment. A similar result holds for the "SyncArb" benchmark: here the reason for less than 100% coverage of FREE_x is that one-third of the latches involve latched inputs (request lines) which can be changed arbitrarily by the environment in any case.

Similarly, we analyzed the coverage of fault-tolerance to STICK_x_AT_0 using FREE_x since we know that FREE_x $\geq_{imp}$ STICK_x_AT_0. Table IV shows the results for the same benchmarks. For benchmarks "Ibuf" and "Am2910", there are latches resilient to STICK_x_AT_0 but not FREE_x.

In general, the aggressiveness order helps pinpoint the *fault-tolerance boundary* of a system by finding faults that the system tolerates as well as more aggressive faults that the

system does not tolerate. Designers and verification engineers can use information about this boundary to either optimize the circuit or evaluate the quality of their specifications.

C. Improving Specifications

We next show how our methodology can lead to improved specifications. Consider once again, the CMP router mentioned in Section II. For simplicity, again assume that $N = 2$. We started with a specification $\mathcal{S}$ asserting that if both channels issue a request and Channel 1 has priority over Channel 0, then it is not the case that both channels are granted access in the next cycle. Thus, $\mathcal{S} = \mathbf{G}\left(((\vec{r} = 11) \wedge \neg p_{01}) \rightarrow \mathbf{X}\left(\neg(\vec{g} = 11)\right)\right)$.

The form of this LTL property ($\mathbf{G}(\alpha \rightarrow \mathbf{X}\alpha')$) indicates that modifying transitions might be a reasonable form of mutation. We start by trying (β, γ)-REMOVE where $\beta = ((\vec{r} = 11) \wedge \neg p_{01})$ and $\gamma = (\vec{g} = 00)$. Denote this mutation by ν_1. The resulting specification $\mathcal{S}_1 = \mathbf{G}\left(((\vec{r} = 11) \wedge \neg p_{01}) \rightarrow \mathbf{X}\left((\vec{g} = 01) \vee (\vec{g} = 10)\right)\right)$ continues to be satisfied. Continuing, we can apply the same kind of mutation, with the same β but $\gamma = (\vec{g} = 01)$ and find that the resulting property $\mathcal{S}_2$ is also satisfied. Let the mutation we applied in this step be denoted by ν_2; we note that $\mathcal{S}_2$ is obtained by mutating $\mathcal{S}$ with the composition $\nu_1 \circ \nu_2$. (However, if we attempt to use $\gamma = (\vec{g} = 10)$, this is not satisfied.) We now obtain $\mathcal{S}_2 = \mathbf{G}\left(((\vec{r} = 11) \wedge \neg p_{01}) \rightarrow \mathbf{X}\left(\vec{g} = 10\right)\right)$ which looks like what we want. However, we can tighten it still further: by again using the same kind of mutation, but with $\beta = (r_1 = 1)$ (i.e., the second channel need not make a request). Denote this mutation by ν_3. The resulting mutated specification is still satisfied, giving us our final result: $\mathcal{S}_3 = \mathbf{G}\left(((r_1 = 1) \wedge \neg p_{01}) \rightarrow \mathbf{X}\left(\vec{g} = 10\right)\right)$.

It is easy to see that the mutations we applied satisfy the aggressiveness order $\nu_1 \circ \nu_2 \circ \nu_3 \geq_{spec} \nu_1 \circ \nu_2 \geq_{spec} \nu_1$.

VII. Discussion

We have given a novel theory of mutations in this paper, and demonstrated its usefulness in reasoning about vacuity, coverage, and fault-tolerance in a unified way. We conclude with a brief discussion of other potential applications of our theory.

The results of this paper are also relevant for *tightening specifications to catch bugs*. Consider an implementation $\mathcal{I}$ that satisfies its specification $\mathcal{S}$. The ultimate goal of checking coverage of $\mathcal{I}$ by $\mathcal{S}$ is to find errors in $\mathcal{I}$. We suggest two ways to use the aggressiveness order in order to automatically generate a specification that reveals errors. The first is to apply to $\mathcal{S}$ a mutation ν' that is more specification aggressive than ν. If, for example, μ introduces some delay, ν' introduces a bigger prematureness. This may be done automatically, or with user guidance, especially in mutations that involve guards. For example, if ν is (β, γ)-REMOVE, then ν' can be (β', γ')-REMOVE, for $\beta' \rightarrow \beta$ and/or $\gamma' \rightarrow \gamma$. The user is aware of the fact that $\mathcal{I}$ loosely satisfies $\mathcal{S}$ with the mutation (β, γ)-ADD, and can use this awareness in order to come up with the stronger β' and γ'. The second approach is to use dual aggressiveness and apply a mutation that is more dually-aggressive than the one with which loose satisfaction has been detected. Note that in this case, as demonstrated in Example 2, the mutation with which loose satisfaction has been detected need not have a a dual mutation.

We also note that tightening of specifications *can be applied also in the context of assume-guarantee reasoning*. There, a specification for a component is of the form $\langle \varphi, \psi \rangle$, and the component has to satisfy a guarantee ψ under the assumption that its environment satisfies φ. Clearly, the tighter φ is, the more likely it is to imply ψ. Our mutations offer an automatic tightening: if $\langle \varphi, \psi \rangle$ does not hold, we can use mutations on the environment in order to automatically generate a tighter assumption φ_μ for which $\langle \varphi_\mu, \psi \rangle$ does hold. Note that this application is to prove properties, rather than to detect errors.

Acknowledgments. The authors gratefully acknowledge the support of the Gigascale Systems Research Focus Center, one of five research centers funded under the Focus Center Research Program, a Semiconductor Research Corporation program. This work was done while the first author was at UC Berkeley.

References

[1] R. Armoni, L. Fix, A. Flaisher, O. Grumberg, N. Piterman, A. Tiemeyer, and M. Vardi. Enhanced vacuity detection for linear temporal logic. In *Proc 15th CAV*, 2003.

[2] I. Beer, S. Ben-David, C. Eisner, and Y. Rodeh. Efficient detection of vacuity in ACTL formulas. *FMSD*, 18(2):141–162, 2001.

[3] M. Browne, E. Clarke, and O. Grumberg. Characterizing finite Kripke structures in propositional temporal logic. *TCS*, 59:115–131, 1988.

[4] D. Bustan, A. Flaisher, O. Grumberg, O. Kupferman, and M. Vardi. Regular vacuity. In *CHARME*, LNCS 3725, pages 191–206, 2005.

[5] M. Chechik and A. Gurfinkel. Extending extended vacuity. In *Proc. 5th FMCAD*, LNCS 3312, pages 306–321, 2004.

[6] H. Chockler, J. Halpern, and O. Kupferman. What causes a system to satisfy a specification? CoRR cs.LO/0312036, 2003.

[7] H. Chockler, O. Kupferman, R. Kurshan, and M. Vardi. A practical approach to coverage in model checking. In *13th CAV*, LNCS 2102, pages 66–78, 2001.

[8] H. Chockler, O. Kupferman, and M. Vardi. Coverage metrics for temporal logic model checking. In *Proc. 7th TACAS*, LNCS 2031, pages 528 – 542, 2001.

[9] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.

[10] S. Das, A. Banerjee, P. Basu, P. Dasgupta, P. P. Chakrabarti, C. R. Mohan, and L. Fix. Formal methods for analyzing the completeness of an assertion suite against a high-level fault model. VLSI Design 2005, pages 201–206.

[11] F. Fummi, G. Pravadelli, A. Fedeli, U. Rossi, and F. Toto. On the use of a high-level fault model to check properties incompleteness. In MEMOCODE 2003, pages 145–152.

[12] D. Große, U. Kühne, and R. Drechsler. Estimating functional coverage in bounded model checking. In *Design Automation and Test in Europe* (DATE), 2007, pages 1176–1181.

[13] O. Grumberg and D. Long. Model checking and modular verification. *ACM TOPLAS*, 16(3):843–871, 1994.

[14] Y. Hoskote, T. Kam, P.-H. Ho, and X. Zhao. Coverage estimation for symbolic model checking. In *Proc. 36th DAC*, pages 300–305, 1999.

[15] O. Kupferman, W. Li, and S. A. Seshia. On the duality between vacuity and coverage. Technical report UCB/EECS-2008-26, EECS Department, UC Berkeley, March 2008.

[16] O. Kupferman and M. Vardi. Vacuity detection in temporal model checking. *Software Tools for Technology Transfer*, 4(2):224–233, 2003.

[17] S. Nain and M. Vardi. Branching vs. linear time: Semantical perspective. In *5th ATVA*, LNCS 4762, 2007.

[18] L.-S. Peh. *Flow Control and Micro-Architectural Mechanisms for Extending the Performance of Interconnection Networks*. PhD thesis, Stanford University, August 2001.

[19] S. A. Seshia, W. Li, and S. Mitra. Verification-guided soft error resilience. In *Design Automation and Test in Europe* (DATE), pages 1442–1447. ACM Press, 2007.

[20] VIS verification benchmarks. Available at `ftp://vlsi.colorado.edu/pub/vis/vis-verilog-models-1.0.tar.gz`.

Trading-off SAT search and Variable Quantifications for effective Unbounded Model Checking

G. Cabodi, P. Camurati, L. Garcia, M. Murciano, S. Nocco, S. Quer
Dipartimento di Automatica ed Informatica
Politecnico di Torino - Torino, Italy
Email: {gianpiero.cabodi, paolo.camurati, luz.garcia, marco.murciano, sergio.nocco, stefano.quer}@polito.it

***Abstract*—Interpolant-based model checking has been shown effective on large verification instances, as it efficiently combines automated abstraction and fixed-point checks. On the other hand, methods based on variable quantification have proved their ability to remove free inputs, thus projecting the search space over state variables.**

In this paper we propose an integrated approach combining the abstraction power of interpolation with techniques relying on AIG and/or BDD representations of states, supporting variable quantification and fixed-point checks. The underlying idea of this combination is to adopt AIG- or BDD-based quantifications to limit and restrict the search space (and the complexity) of the interpolant-based approach. The exploited strategies, individually well-known, are integrated with a new flavor, specifically designed to improve their effectiveness on large verification instances.

Experimental results, oriented to hard-to-solve verification problems, show the robustness of our approach.

I. Introduction

State set representations, based on characteristic functions and on efficient symbolic quantification, have been a key factor for the success of Binary Decision Diagrams (BDDs) in symbolic model checking. Nonetheless, scalability problems practically limit the use of BDDs to designs with up to a few hundred latches.

On the other hand, the usefulness of quantification operators within SAT-based model checking is still unproven. Several optimizations have been proposed over the years, but a widespread applicability is still out of reach. Williams et al. [1] adopted Boolean Expression Diagrams (BEDs) for the removal of quantifiers. Abdulla et al. [2] used Reduced Boolean Circuits (RBCs), a variant of BEDs, to represent formulas on which they performed existential quantifier elimination through substitution, scope reduction, etc. McMillan [3], later followed by Kang and Park [4], proposed quantifier elimination through the enumeration of SAT solutions (*all-solutions* SAT). Ganai et al. [5] extended the previous approaches by using *circuit co-factoring*, where a circuit was used to capture a large set of states in every SAT enumeration step.

Abstraction techniques have been shown very effective to remove those parts of a system that are not relevant for the verification task. McMillan [6] used Craig interpolants for Unbounded Model Checking (UMC), and improvements were proposed in [7], [8], [9], [10]. Craig interpolants exploit the ability of modern SAT solvers to generate proofs of unsatisfiability. Starting from refutation proofs of (unsatisfied) BMC-like runs, the method computes over-approximations of the reachable states. Unfortunately, the main strength of interpolants, i.e., the automated exploitation of SAT refutation proofs, is also their weakness, as interpolants can become very large when SAT checks are complex.

A. Motivations and contributions

Our experience [11] shows that SAT-based Craig interpolants, combined with preliminary computations of inductive invariants, can prove a broad range of verification instances. A careful analysis of the unproved problems led us to the following observations:

- Craig interpolants tend to produce highly redundant circuit representations of state sets. Combinational logic optimizers are often not scalable and powerful enough to compact large interpolants.
- Although abstraction usually reduces the sequential depth of state transition graphs, over-approximation can trigger state space explorations within unreachable state areas, with direct consequences in terms of state visits and representations.
- Circuit- and SAT-based state set representations and variable quantifications often work with sub-sets of problems/variables, whereas memory blow-up occurs when attempting complete quantifications and state set representations.

Given the above listed problems, we complement interpolant-based verification with a set of cooperating techniques, based on (partial and/or partitioned) state set representations, appropriately combined and tailored to accomplish the verification task. More specifically, we adopt a divide-and-conquer, incremental approach, based on the general idea of computing and representing state sets as partitioned functions, partially in a quantified form (whenever convenient/possible) and partially without quantification. In other words, we exploit interpolants for abstraction, while using quantification and state set representations for (partial) reachability and search space restriction. The main contributions of the paper are:

- A novel SAT-based technique for partial quantification and/or under-estimation of state space, by means of *lazy quantification*.
- A partitioned formulation of interpolant-based image computations, working with disjunctively partitioned representations of backward circuit unrollings.

978-1-4244-2735-2/08/$25.00 ©2008 IEEE

- An integrated approach combining different techniques within a powerful unified SAT-based verification engine.

Our methodology is more than just a mix of several competing techniques, such that, for any given problem, one of them wins over the others. Our combination of different methods often produces better results than any of those strategies running separately. Our experimental results, on the suite collected for the 2007 Model Checking Competition [11], show improvements in terms of speed, robustness and scalability.

B. Outline

The paper is organized as follows. Section II introduces background notions on Bounded, Unbounded, and Interpolant Model Checking. Sections III and IV present our main contributions on efficiently computing, representing, and exploiting state sets. Section V shows our integrated approach to Unbounded Model Checking. Section VI discusses the experiments we performed. Section VII concludes the paper with some summarizing remarks.

II. Background

A. Model and notation

We address systems modeled by labeled state transition structures, and represented implicitly by Boolean formulas. The state space and the primary inputs are defined by indexed sets of Boolean variables $V = \{v_1, \ldots, v_n\}$ and $W = \{w_1, \ldots, w_m\}$, respectively. States correspond to the valuations of variables in V, whereas transition labels correspond to the valuations of variables in W. We indicate next states with the primed variable set $V' = \{v'_1, \ldots, v'_n\}$. Whenever we explicitly need time frame variables, we use $V^i = \{v^i_1, \ldots, v^i_n\}$ and $W^i = \{w^i_1, \ldots, w^i_m\}$ for variable instances at the $i-$th time frame. We also adopt the short notation $V^{i..j}$ ($W^{i..j}$) for $V^i, V^{i+1}, \ldots, V^j$ ($W^i, W^{i+1}, \ldots, W^j$)[1].

A set of states is expressed by a state predicate $S(V)$ (or $S(V')$ for the next state space). $I(V)$ is the initial state predicate. We use $P(V)$ to denote an invariant property, and $F(V) = \neg P(V)$ for its complement (as it is often used as target for bug search). With abuse of notation, in the rest of this paper we make no distinction between the characteristic function of a set and the set itself.

$T(V, W, V')$ is the transition relation of the system. We assume T given by a *circuit graph*, with state variables mapped to latches. Present and next state variables correspond to latch outputs and inputs, respectively. The input of the $i-$th latch is fed by a combinational circuit, described by the $\delta_i(V, W)$ Boolean function. Hence, the transition relation can be expressed as:

$$T(V, W, V') \;=\; \textstyle\bigwedge_i t_i(V, W, v'_i) \;=\; \bigwedge_i (v'_i \Leftrightarrow \delta_i(V, W))$$

A state path of length k is a sequence of states $\sigma_0, \ldots, \sigma_k$ such that $T(\sigma_i, \nu_i, \sigma_{i+1})$ is true, given some input pattern ν_i, for all $0 \leq i < k$.

[1] $V^{i..j}$ and $W^{i..j}$ are appropriately defined if $i \leq j$, otherwise we conventionally define them as empty variable sets.

A state set S' is reachable from state set S in k steps if there exists a path of length k, in the labeled state transition structure, connecting a state in S to another state in S':

$$S(V^0) \wedge \big(\textstyle\bigwedge_{i=0}^{k-1} T(V^i, W^i, V^{i+1})\big) \wedge S'(V^k) \quad \neq \quad 0$$

The image operator $\textsc{Img}(T, From)$ computes the set of states $Image$ reachable in one step from the states in $From$:

$$\begin{array}{rcl} Image(V') & = & \textsc{Img}(T(V, W, V'),\ From(V)) \\ & = & \exists_{V,W}(From(V) \wedge T(V, W, V')) \end{array}$$

Pre-image is dual, with the only difference that existential quantification of functionally computed state variables can be obtained by composition:

$$\begin{array}{rcl} Image(V) & = & \textsc{PreImg}(T(V, W, V'),\ From(V')) \\ & = & \exists_{W,V'}(From(V') \wedge T(V, W, V')) \\ & = & \exists_W From(\delta(V, W)) \end{array}$$

An over-approximate image is any state set including the exact image:

$$Image^+ \;=\; \textsc{Img}^+(T,\ From) \;\supseteq\; \textsc{Img}(T,\ From)$$

B. Bounded Model Checking

SAT-based Bounded Model Checking (BMC) [12] considers only $k-$bounded reachability, as expressed by the propositional formula:

$$\textsc{Bmc}_0^k \;=\; I(V^0) \wedge \big(\textstyle\bigwedge_{i=0}^{k-1} T(V^i, W^i, V^{i+1})\big) \wedge F(V^k)$$

$\textsc{Bmc}_0^k$ is satisfiable *iff* there is a counter-example (a path from I to F) of length k. In the case of circuits, existential quantification can be applied to intermediate sets of state variables:

$$\begin{array}{rl} \textsc{Bmc}_0^k & = I(V^0) \wedge \exists_{V^{1..k}} \big(\textstyle\bigwedge_{i=0}^{k-1} T(V^i, W^i, V^{i+1}) \wedge F(V^k)\big) \\ & = I(V^0) \wedge Cone_k(V^0, W^{0..k-1}) \end{array}$$

where $Cone_k$ represents a combinational single output circuit unrolling, formally defined by exploiting quantification by functional composition:

$$\begin{array}{rcl} Cone_k & = & Cone_k(V^0, W^{0..k-1}) \\ & = & \exists_{V^{1..k}} \big(\textstyle\bigwedge_{i=0}^{k-1} (V^{i+1} \Leftrightarrow \delta(V^i, W^i)) \wedge F(V^k)\big) \\ & = & F(\delta(\ldots\delta(\delta(V^0, W^0))\ldots), W^{k-1}) \end{array} \tag{1}$$

The main advantage of using $Cone_k$ (a combinational circuit) instead of transition relation instances ($\bigwedge_i T_i$), is that several circuit-based simplifications, from constant propagation to combinational optimization, are possible on $Cone_k$ before going to CNF- or circuit-based SAT.

C. State sets and fixed-points in Unbounded Model Checking

Although reachability is usually formulated in terms of the image and/or pre-image operators, we will here express backward reachability using $Cone_k$, as previously defined. SAT-based model checking approaches generally keep explicit representations of circuit unrollings instead of (exact) state set representations, due to the inherent complexity of quantification operators. AIGs, or cognate (non canonical) circuit-based representations [13], are often used for that purpose.

The set of backward reachable states from F in (exactly) k steps can be obtained by primary input quantification over a circuit unrolling:

$$BckR_k(V) \quad = \quad \exists_{W^{0..k-1}} Cone_k(V, W^{0..k-1})$$

The overall set of backward reachable states is the union of all reachable states up to depth k:

$$\begin{aligned} BckR_{0..k}(V) &= \bigcup_{i=0}^{k} BckR_i(V) \\ &= \bigcup_{i=0}^{k} \exists_{W^{0..i-1}} Cone_i(V, W^{0..i-1}) \\ &= \exists_{W^{0..k-1}} \bigcup_{i=0}^{k} Cone_i(V, W^{0..i-1}) \end{aligned}$$

where distributivity of existential quantification over union has been applied. We introduce the short notation $Cone_{0..k}$ for $\bigcup_{i=0}^{k} Cone_i$. Following this notation, backward reachable states are defined as:

$$BckR_{0..k}(V) \quad = \quad \exists_{W^{0..k-1}} Cone_{0..k}(V, W^{0..k-1})$$

A backward reachability fixed-point could be checked by a SAT run on the following Boolean formula:

$$BckR_{k+1}(V) \wedge \neg BckR_{0..k}(V)$$

or on a simpler one, with quantified state sets on the second term only:

$$Cone_{k+1}(V, W^{0..k}) \wedge \neg BckR_{0..k}(V) \tag{2}$$

Unfortunately, both the above formulations are difficult to manipulate in many practical cases, due to the complexity of circuit-based quantification.

D. Craig interpolants in Model Checking

Given two inconsistent formulas A and B ($A \wedge B = 0$), an interpolant C is a formula such that:

1) It is implied by A.
2) It is inconsistent with B, i.e., $C \wedge B$ is unsatisfiable.
3) It is expressed over the common variables of A and B.

A Craig interpolant $C = \text{ITP}\ (A,\ B)$ is an AND/OR circuit directly derived from refutation proof of $A \wedge B$. Albeit the computation of ITP from the proof has a linear cost, the refutation proof can be exponentially larger than A and B.

An over-approximate image of a state set S is k–adequate, with respect to F, if it does not intersect any state on paths of length k to F. It can be computed as follows[2]:

$$\begin{aligned} &\text{IMG}^{+}_{Adq}(T, S, Cone_{0..k}) \ = \\ &\quad \text{ITP}(S(V^0) \wedge T(V^0, W^0, V^1),\ Cone_{0..k}(V^1, W^{1..k})) \end{aligned}$$

The $Cone_{0..k}$ circuit unrolling encodes all $k-$step bounded paths to F. A k–adequate over-approximate image IMG^{+}_{Adq} is `undefined` *iff* the exact image is not k–adequate:

$$S(V^0) \wedge T(V^0, W^0, V^1) \wedge Cone_{0..k}(V^1, W^{1..k}) \quad \neq \quad 0$$

An image is called adequate if it is k–adequate for any k, i.e., no state within the image is backward reachable from F. Since the model is finite, a k–adequate image is adequate if $k \geq d$, where d is the diameter of the state transition graph.

Figure 1 shows a fully SAT-based UMC algorithm [6], exploiting Craig interpolants.

```
INTERPOLANTMC (I, T, F)
  k = 0
  do
    Cone_{0..k} = CIRCUITUNROLL(F, δ, k)
    res = FINITERUN (I, T, Cone_{0..k})
    k = k + 1
  while (res = undecided)

FINITERUN (I, T, Cone)
  if (SAT(I ∧ T ∧ Cone)
    return (reachable)
  R = I
  while (true)
    Image+ = IMG+_Adq (T, R, Cone)
    if (Image+ = undefined)
      return (undecided)
    if (Image+ ⇒ R)
      return (unreachable)
    R = R ∨ Image+
```

Fig. 1. Interpolant-based Verification.

While INTERPOLANTMC is the entry point of the algorithm, routine FINITERUN takes care of the interpolant-based over-approximate traversal. The latter function may end up with three possible results:

- `reachable`, if it proves F reachable from I in k steps, hence the property has been disproved.
- `unreachable`, if the approximate traversal using the IMG^{+}_{Adq} image computation reaches a fixed-point. In this case the property has been proved.
- `undecided`, if F is intersected by the over-approximate state sets. So k is increased for a new FINITERUN call.

The algorithm is sound and complete [6]. In synthesis, let us assume I and F mutually unreachable. If $k < d$, a k–adequate set can produce a non k–adequate image. In this case, `undecided` is returned and k is increased. Otherwise ($k \geq d$), IMG^{+}_{Adq} is adequate, and the algorithm will terminate with an approximate reachability fixed-point.

III. COMPUTING AND REPRESENTING STATE SETS

We adopt (quantified) state sets to represent a part of the state space, and circuit unrollings for the rest of the space. As a consequence, our solution is based on a partitioned representation of backward cones and reachable state sets, and it lies in between two extremes:

- *Full quantification* and exact state set representation, within a backward traversal scheme.
- *No quantification* at all, i.e., "states" represented as a backward circuit unrolling, within a pure interpolant-based approach.

We describe in this section our state representation strategy based on partial quantification, whereas disjunctive partitioning is the subject of the next section.

[2]Variable indices have been shifted up in $Cone_{0..k}$ in order to use W^0, V^0 and V^1 in the forward T instance.

A. Partial quantification

As we analyzed in Section II-C (see Equation 2), state sets support fixed-point checks. For this reason, as long as we can compute state sets with an acceptable cost, we prefer them to backward unrollings. In general, we will use both (partial) state sets and (partial) unrollings.

Let us use $BckR^{-}$ and $Cone^{-}$ to indicate partial state sets and partial unrollings:

$$\begin{array}{rcl} BckR_k^-(V) & \subseteq & BckR_k(V) \\ BckR_{0..k}^-(V) & \subseteq & BckR_{0..k}(V) \\ Cone_k^-(V, W^{0..k-1}) & \subseteq & Cone_k(V, W^{0..k-1}) \\ Cone_{0..k}^-(V, W^{0..k-1}) & \subseteq & Cone_{0..k}(V, W^{0..k-1}) \end{array}$$

A $(BckR_k^-, Cone_k^-)$ couple is *complete*, if:

$$\begin{array}{rcl} BckR_k(V) & = & \exists_{W^{0..k-1}} Cone_k(V, W^{0..k-1}) \\ & = & BckR_k^-(V) \cup \exists_{W^{0..k-1}} Cone_k^-(V, W^{0..k-1}) \end{array}$$

In other words, a *complete* $(BckR_k^-, Cone_k^-)$ couple can fully replace a $Cone_k$ in backward reachability and/or in the FINITERUN function, still preserving the completeness of the approach. A similar consideration holds for $(BckR_{0..k}^-, Cone_{0..k}^-)$.

We propose two ways to compute partial state sets, as described in the following two paragraphs.

1) Bounded reachability: We compute exact (quantified) state sets up to a given bound $l < k$. Then $BckR_{0..k}^- = BckR_{0..l} \subseteq BckR_{0..k}$[3]. This captures the common case where the complexity of exact traversals is accepted up to a given depth. Cones at higher depths are not quantified.

In order to decide whether or not to accept a quantification, we introduce *lazy quantification* operators, based on AIG and/or BDD implementations, controlled by space and time limits.

Let us define lazy existential quantification as follows. For each variable to quantify, we optionally accept or reject the quantification, based on a target maximum size increase. The global effect of the procedure is to filter out critical variable quantifications, i.e., those responsible of size explosion. Lazy quantification can produce intermediate results in the range going from no quantification at all to full quantification.

Figures 2 and 3 show two versions of the lazy quantification procedure. The former one is completely based on AIG manipulations, whereas the latter one relies on AIG to BDD (and vice-versa) transformations.

Function LAZYE of Figure 2 shows how we selectively quantify the W variables from $Cone$[4]. Quantification is achieved by looping through each w_i variable (with $w_i \in W$), and heuristically deciding whether the corresponding quantification is acceptable or not. The α parameter expresses the maximum allowed size increase. A given quantification is maintained only if the circuit size (evaluated in terms of AIG nodes) after quantification is acceptable (line 5).

[3] $BckR_{0..l} = BckR_{0..k}$ just at the fixed-point.

[4] For the sake of simplicity, in the rest of this section we do not explicitly indicate k-bounds of unrolling, so $Cone$ stands for either $Cone_k$ or $Cone_{0..k}$, and W stands for $W^{0..k}$.

```
1  LAZYE (Cone, W, α)
2    G = Cone
3    forall w_i ∈ W
4      tmp = ∃_{w_i} G
5      if (|tmp| < α · |G|)
6        // Accept quantification
7        G = tmp
8    return (G)
```

Fig. 2. Lazy, i.e., partial, quantification based on AIGs.

Figure 3 proposes an alternative implementation of partial quantification, based on BDDs. It is activated as far as the number of domain variables is in a reasonable range for BDDs. We convert $Cone$ from AIG to BDD, then we perform BDD-based quantification. BDDs are generated by the AIG2BDD function (line 2), which builds BDDs ($ConeBdd$) with cut-points ($cutF$) and auxiliary variables at each cut ($cutV$).

```
1  LAZYEBDD (Cone, W, β)
2    (ConeBdd, CutV, CutF) = AIG2BDD (Cone, th_cut)
3    G = ANDEBDD (ConeBdd ∧ ⋀_i (CutV_i ⇔ CutF_i),
                  CutV ∪ W)
4    if (|G| < β · |Cone|)
5      // Accept quantification
6      return (BDD2AIG(G))
7    else
8      // Reject quantification
9      return (Cone)
```

Fig. 3. Lazy, i.e., partial, quantification based on BDDs.

Cut-points are dynamically selected to keep the size of the BDDs below the th_{cut} threshold and prevent memory blow-up [14]. The th_{cut} value is selected by the user. Given a BDD representation with cut-points, an and-exist procedure (function ANDEBDD, line 3) tries to build a monolithic BDD by using an early quantification schedule [15]. In order to bound the size of the BDD representation, we compare the size of BDD G with the AIG size of $Cone$ scaled by the factor β (line 4). The β factor takes into account the BDD to AIG isomorphism, where each BDD node is converted into a multiplexer. If the size of the result is below the bound, it is accepted and converted back into an AIG (function BDD2AIG, line 6). Otherwise the quantification is rejected and the original function $Cone$ is returned as a result (line 9).

2) Circuit sub-setting: Within the field of BDD-based verification, Ravi and Somenzi [16] proposed high density in order to maximize the number of states represented by a BDD of bounded size. Ganai et al. [5] used circuit cofactors as circuit representations of sub-sets of states. The common idea of the two techniques is to assign constant values to a set of variables, thus restricting the represented state set within a given sub-space. Sub-setting performed on state or input variables can attain the additional result of removing quantification variables.

Following this trend, let us suppose to work on the Boolean function $Cone(V, W)$, such that a state set $BckR(V)$ can be

obtained by quantifying out the W variables:

$$BckR(V) \quad = \quad \exists_W Cone(V, W)$$

Then a sub-set $BckR^-(V)$ can be computed by assigning constant values to W variables. Let σ be a variable assignment to W, satisfying $Cone$ (generated by a SAT solver run). Then:

$$BckR^-_\sigma(V) \quad = \quad Cone(V, \sigma) \quad \subseteq \quad BckR(V)$$

The overall level of under-estimation of $BckR(V)$ by $BckR^-_\sigma(V)$ strongly depends on the choice of σ. Most of the heuristics described in [5] are oriented to drive a circuit-based SAT solver towards more representative input solutions. We propose an alternative hybrid approach to sub-setting, where, in order to reduce the dependency from the choice of σ, some primary inputs are existentially quantified and the others are set to a constant value. A simplified implementation of such a procedure is shown in Figure 4.

```
1   LAZYESUBSET (Cone, W, γ)
2     G = Cone
3     σ = SAT (Cone)
4     forall w_i ∈ W
5       tmp = ∃_{w_i} G
6       if (|tmp| < γ · |G|)
7         // Accept quantification
8         G = tmp
9       else
10        // Reject quantification and subset
11        G = G|_{w_i=σ[w_i]}
12    return (G)
```

Fig. 4. Lazy, i.e., partial, quantification with sub-setting.

Function LAZYESUBSET first computes the variable assignment σ satisfying $Cone$ (line 3). Then, for each variable $w_i \in W$, it evaluates $\exists_{w_i} G = G|_{w_i=0} \vee G|_{w_i=1}$ (line 5). After that, it compares the size of the result with the size of G scaled by the factor γ (line 6). If the result is too large, the quantification is rejected, and only one of the cofactors is retained (line 11). The selection of the cofactor is based on the value of σ, computed as previously described. The produced sub-set includes σ by construction (some variables are existentially quantified, the other ones are assigned according to σ). Since the variable assignment σ satisfies $Cone$, sub-setting will never end up finding an empty sub-set. An alternative choice would be computing both cofactors and dynamically selecting one of them, based on support or size heuristics. Anyhow, in this case the procedure may incur in dead-ends, i.e., it may find an empty sub-set, unless some kind of backtracking is introduced.

Our procedure can obtain potentially denser sub-sets than other sub-setting operators, since it assigns constant values only to variables that are critical for existential quantification.

Note that the previous pseudo-codes hide some implementation details, such as:

- Variable scoring based on history of accepted/rejected quantifications: We directly reject quantification of variables with high rejection rate.
- Two–step quantification: A first loop just quantifies accepted variables, with no sub-setting, a second loop through the previously rejected variables enables both quantification and sub-setting.
- Time thresholds for the entire quantification procedure and for single variable quantifications, with inner SAT sweepings to simplify the result.

We will show in the sequel how the results given by the procedures analyzed in this section are exploited in our overall verification procedure.

IV. EXPLOITING PARTIAL STATE SETS IN UMC

As we introduced in Section III, $Cone_{0..k}(V, W^{0..k-1})$ represents backward reachable states, and the related input assignments, whereas $BckR_{0..k}(V)$ is a set of reachable states with no information on paths to the F target. Obviously $Cone_{0..k} \subseteq BckR_{0..k}$. The core idea of our method is to represent backward reachable state sets in terms of a disjunctively partitioned form:

$$\widehat{Cone}_{0..k} \quad = \quad BckR^-_{0..k} \cup Cone^-_{0..k}$$

such that

$$Cone_{0..k} \quad \subseteq \quad \widehat{Cone}_{0..k} \quad \subseteq \quad BckR_{0..k}$$

With respect to $Cone_{0..k}$ and $BckR_{0..k}$, $\widehat{Cone}_{0..k}$ is an intermediate representation, where input paths explicitly appear just for a sub-set of the states.

We generate sub-sets of states by partial quantification, as described in Section III. Whenever $BckR^-_{0..k} = BckR_{0..k}$, our model checking procedure behaves like a standard exact backward reachability function. Otherwise, we use partial state sets ($BckR^-_{0..k}(V)$) as *don't care* sets for computation of partial cones:

$$Cone^-_{0..k} \quad = \quad \text{SIMPLIFY}(Cone_{0..k}, \neg BckR^-_{0..k}) \qquad (3)$$

Based on redundancy removal techniques, function SIMPLIFY produces more compact cones. The use of $\neg BckR^-_{0..k}(V)$ as *external care* causes cone sub-setting. $\neg BckR^-_{0..k}(V)$ will be further exploited to restrict the search space while computing interpolants.

Furthermore, we use disjoint decompositions for k-adequate over-approximate image computation within FINITERUN. Let us remark the fact that reachable state sets and circuit unrollings are unions of sub-sets (see Sections II-B and II-C):

$$\begin{aligned} BckR_{0..k} &= \bigcup_i BckR_i \\ Cone_{0..k} &= \bigcup_i Cone_i \end{aligned}$$

A similar observation holds for partial state sets and for partial circuit unrollings. So our most general representation is:

$$\begin{aligned} \widehat{Cone}_{0..k} &= BckR^-_{0..k} \cup Cone^-_{0..k} \\ &= \bigcup_i BckR^-_i \cup \bigcup_j Cone^-_j \end{aligned}$$

We apply a divide-and-conquer approach to adequate image computation by generating a conjunctively partitioned image starting from a disjunctively partitioned cone. For the sake

of simplicity, let us formulate partitioned (k-adequate) image just for the case of a generic backward cone disjunctively decomposed in two sub-sets (the extension to $k-$bounded cones with more partitions is straightforward):

$$\widehat{Cone}(V,W) \quad = \quad Cone^0(V,W) \cup Cone^1(V,W)$$

Both $Cone^0$ and $Cone^1$ are either partial state sets or partial circuit unrollings.

Theorem 1:

$$\text{IMG}^+_{Adq}(T, From, Cone) = \text{IMG}^+_{Adq}(T, From, \widehat{Cone}) = \text{IMG}^+_{Adq}(T, From, Cone^0) \wedge \text{IMG}^+_{Adq}(T, From, Cone^1)$$

Sketch of proof. As $Cone^0$ and $Cone^1$ are subsets of $Cone$, if the image is adequate with respect to $Cone$, then it is adequate for both the subsets as well. On the other hand, the conjunction guarantees that the overall result is adequate to both $Cone^0$ and $Cone^1$. The same applies to their union ($Cone$). □

We exploit the above formulation for complex images, sorting disjunctive components by size and combinational depth, and computing "easier" sub-images first.

The $BckR^-$ partial sets can be used in a similar way, with additional benefits:

- As they already represent state sets (no W variables appear in their support), $\neg BckR^-$ can be used as partial image, due to the fact that:

$$\text{IMG}^+_{Adq}(T, From, BckR^-) \subseteq \neg BckR^-$$

- They can be used as don't care sets for circuit simplification of other cones (see Equation 3).
- They can be exploited as additional constraints in every SAT run for computing an adequate image with respect to a (partial) cone:

$$Image^+ = \text{IMG}^+_{Adq}(T, From, Cone \wedge \neg BckR^-)$$

Our decomposed image differs from usual partitioned images in model checking, that typically follow either the disjunctive model (disjunctively partitioned image working with disjunctive state sets and/or transition relation) or the conjunctive one (conjunctively partitioned transition relation with monolithic state sets). Our scheme is specifically oriented to interpolant-based model checking, as we rely on monolithic sets and transition relations. We use disjunctively decomposed backward unrollings, and we produce conjunctively decomposed images.

V. An integrated approach

In this section we describe our model checking procedure which combines the previously mentioned techniques.

Figure 5 shows function IntegratedMC. The procedure loops through a main iteration (line 4), following a standard backward traversal scheme. At each iteration, it computes a deeper backward cone (line 6), starting from the target F (line 3). Then, it tests the "failure" condition by checking the intersection of $Cone_k$ with the initial state I (line 7). If such an intersection is not empty, the `reachable` result is returned (line 8).

In line 9, CheckFP verifies whether the fixed-point is hit. As already pointed out, fixed-point checks are the critical operation, which we perform in the following way:

- If the backward cone is fully quantified (it is a backward state set), CheckFP consists in a pure SAT check.
- If the unrolling cone includes input variables, CheckFP performs a heuristically limited number of SAT runs (inspired by [17] and [5]), trying to find a state reached by $Cone_k$ and not included in the previously reached state set ($BckR^-$ or $Cone_{0..k-1}$). If the bound on the SAT calls is exceeded, CheckFP ends up with an `undecided` result.

If a fixed-point is hit, the `unreachable` result is returned (line 11).

After that, we attempt BDD-based (LazyEBdd, line 12) and circuit-based (LazyE, line 13) quantification, and evaluate the $BckR^-$ state set using the LazyESubset procedure (line 14). $BckR^-$ is used as a don't care set to optimize $Cone_{0..k}$, computing its sub-set $Cone^-_{0..k}$ (line 16). Furthermore, $BckR^-$ helps both SAT-based fixed-point checks and interpolant traversals.

Finally, we activate the interpolant-based forward procedure FiniteRun2 (line 17). This is a variant of FiniteRun, which uses the optimized (divide-and-conquer) image computation based on partial sub-sets and exploits $BckR^-$ for search space restriction, as discussed in Section IV.

```
1   IntegratedMC (I, T, F, α, β, γ)
2     k = 1
3     BckR⁻ = Cone_0 = F
4     do
5       res = undecided
6       Cone_k(V, W^{1..k}) = Cone_{k-1}(δ(V, W^1), W^{2..k})
7       if (SAT(I ∧ (Cone_k ∨ BckR⁻)))
8         return (reachable)
9       fp = CheckFP (Cone_k, BckR⁻, Cone_{0..k-1})
10      if (fp = true)
11        return (unreachable)
12      Cone_k = LazyEBdd (Cone_k, W^{1..k}, β)
13      Cone_k = LazyE (Cone_k, W^{1..k}, α)
14      BckR⁻ = BckR⁻ ∨
                  LazyESubset (Cone_k, W^{1..k}, γ)
15      if (fp = undecided)
16        Cone⁻_{0..k} = Simplify(Cone_{0..k}, ¬BckR⁻)
17        res = FiniteRun2 (I, T, Cone⁻_{0..k}, BckR⁻)
18      k = k + 1
19    while (res = undecided)
20    return (res)
```

Fig. 5. Verification Procedure: An integrated approach dovetailing interpolation, lazy quantification and lazy quantification with sub-setting.

Theorem 2: Function IntegratedMC is sound.

Sketch of proof. CheckFP is sound as it is based on an optimized version of Equation 2. FiniteRun2 is sound as it is based on complete decompositions as described in Section IV: A complete decomposition $\widehat{Cone}$ guarantees that no forward fixed-point can be reached, including states backward reachable from F. □

VI. Experimental Results

We implemented a prototype version of our methodology, on top of the PdTrav tool, a state-of-the-art verification framework which won two of the sub-categories at the Model Checking Competition 2007 [11].

We present results on circuits derived from the Model Checking Competition [11] suite, a few standard benchmarks from the VIS distribution [18], and some industrial benchmarks coming from STMicroelectronics. Some of the circuits, denoted with the $vis.pm$ prefix, are modified versions of the original ones, where we check the equivalence of the original version against a sequentially optimized instance[5].

We compare results of our tool with and without the proposed methodology. Our experiments ran on a Dual Core Pentium IV 3 GHz Workstation with 3 GBytes of main memory, hosting a Debian Linux distribution. We concentrate on true properties, since they are the most interesting ones (false instances are usually easily captured by BMC runs).

The scattered plot in Figure 6 represents verification times on 259 benchmarks. In all those experiments, we used a time limit of 900 seconds. The graph compares a standard interpolant-based UMC technique (time on the X-axis) against the same strategy improved as suggested in this paper (time on the Y-axis). Markers below the main diagonal represent an advantage for the methods presented in this paper. The plot clearly shows a set of "easy" benchmarks, i.e., the ones which are solved in few seconds or minutes in either technique. It also exhibits 20 problems that are not solved within the time limit by the original strategy, whereas they are solved by the proposed optimization. The overall results clearly prove the robustness of the proposed approach.

Table I reports more detailed data on a few selected hard-to-solve verification instances. For these benchmarks we ran all the engines we submitted to the 2007 Model Checking Competition, i.e., BDD-based verification, circuit-based quantification, inductive invariants, and standard interpolation. Among these, we selected the fastest technique on each single experiment, and we compared it with the hybrid method presented in this paper. In all these experiments, we allowed a time limit of 2 hours.

The meaning of the columns is the following. Model is the instance name, # PI, # FF and # Nodes represent the number of primary inputs, memory elements and AIG nodes of the circuit, respectively. For both the original technique and the one proposed in this paper, we report the CPU time (in seconds), and the Bound, i.e., the verification depth necessary to hit the fixed-point. For the original methods, each instances is tagged with the technique used to complete the experiment (column Method). The tags for the original methods are:

- ITP: standard interpolation
- INV: inductive invariants
- BDD: BDD-based reachability
- CBQ: circuit-based quantification

[5]We applied re-timing and combinational optimization algorithms using the ABC tool [19].

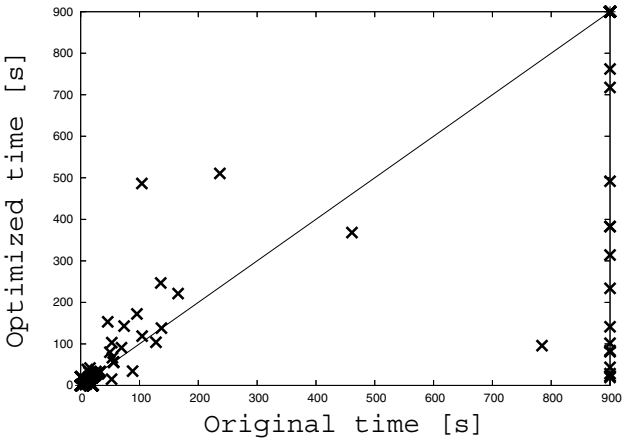


Fig. 6. Verification results: Comparison in terms of CPU time.

On the contrary, the new method is tagged with either ITP or BR depending on whether the fixed-point was proved by interpolation or by SAT checks on state sets.

The data in the table clearly show the robustness of our method. Besides being able to run benchmarks that could not be completed with any of the original methods, in several cases the integrated approach outperformed the best of our previously submitted techniques. Furthermore, we observe that both the techniques for detecting the fixed-point (ITP and BR) are well-balanced with respect to the number of successfully verified instances. This fact seems to give credit to the proposed methodology.

VII. Conclusions

This paper shows how to improve interpolant-based model checking by means of an integrated approach combining partial quantification, sub-setting, disjunctive partitioning, and interpolation. The core idea of this combination is to adopt quantifications and circuit-based representations of sub-spaces whenever convenient (and not too expensive). The obtained advantages derive from a limitation and a restriction of the search space of interpolant-based methods.

Experimental results, specifically oriented to hard verification problems, show the robustness of our approach, as implemented on a state-of-the-art verification framework.

References

[1] P. F. Williams, A. Biere, E. M. Clarke, and A. Gupta, "Combining Decision Diagrams and SAT Procedures for Efficient Symbolic Model Checking," in *Proc. Computer Aided Verification*, ser. LNCS, E. A. Emerson and A. P. Sistla, Eds., vol. 2102. Chicago, Illinois: Springer-Verlag, Jul. 2000, pp. 124–138.

[2] P. A. Abdulla, P. Bjesse, and N. Een, "Symbolic Reachability Analysis based on SAT-Solvers," in *Tools and Algorithms for the Construction and Analysis of Systems*, S. Graf and M. I. Schwartzbach, Eds., vol. 1785. Berlin, Germany: Springer-Verlag, Apr. 2000, pp. 411–425.

[3] K. L. McMillan, "Applying SAT Methods in Unbounded Symbolic Model Checking," in *Proc. Computer Aided Verification*, ser. LNCS, E. Brinksma and K. G. Larsen, Eds., vol. 2404. Copenhagen, Denmark: Springer, 2002, pp. 250–264.

[4] H. J. Kang and L. C. Park, "SAT-based unbounded symbolic model checking," in *Proc. 40th Design Automation Conf.* Anaheim, CA: IEEE Computer Society, 2003, pp. 840–843.

TABLE I
VERIFICATION RESULTS: DETAILED ANALYSIS OF SOME HARD-TO-SOLVE DESIGNS. THE SYMBOL – MEANS OVERFLOW.

Model	#PI	#FF	#Nodes	Original Method			New Method		
				Time [s]	Method	Bound	Time [s]	Method	Bound
intel_006	345	350	3265	195.80	ITP	9	197.72	ITP	9
intel_024	352	357	5710	6344.47	ITP	15	454.47	ITP	15
intel_029	559	564	8816	–			620.09	ITP	18
vis.blackjack-inv	5	103	3979	3359.29	BDD	10	110.02	ITP	11
vis.vsaR	17	66	2321	1131.25	BDD	12	371.66	BR	6
nusmv.tcas^3.B	146	169	2914	87.38	ITP	6	37.02	ITP	7
vis.coherence^3.E	6	29	1214	2439.24	INV	10	236.70	ITP	11
vis.pm.am2901	26	136	2416	1764.57	CBQ	3	83.40	BR	2
vis.pm.FPMult	17	215	1347	1895.51	ITP	3	85.49	BR	2
vis.pm.palu	14	220	2347	–			390.14	ITP	5
vis.ns3$_1$	21	103	3598	606.45	ITP	7	83.75	ITP	7
vis.ns3$_2$	21	103	3598	1004.25	ITP	7	149.92	ITP	7
vis.feistel	68	296	6821	392.09	INV	15	749.65	BR	13
eijk.bs3271	26	305	2546	1391.00	ITP	17	327.33	BR	13
eijk.bs6669	83	506	4879	–			132.04	BR	5
eijk.bs3384	43	689	3069	–			532.07	BR	7
IndustrialA$_1$	5	99	2657	1761.86	ITP	11	71.92	BR	7
IndustrialA$_2$	37	250	4521	1192.51	CBQ	7	517.21	BR	4
IndustrialA$_3$	51	333	1275	1933.09	CBQ	8	470.07	BR	8
IndustrialB$_1$	12	190	3324	–			17.08	ITP	17
IndustrialB$_2$	12	193	6782	–			154.21	ITP	11
IndustrialB$_3$	15	309	1592	1341.60	ITP	9	49.76	ITP	9
IndustrialB$_4$	18	416	5409	–			265.49	ITP	5
IndustrialB$_5$	18	425	4391	–			457.17	ITP	9
IndustrialC$_1$	21	116	1098	91.27	BDD	12	98.10	ITP	12
IndustrialC$_2$	67	351	2021	950.08	ITP	15	98.55	ITP	15
IndustrialC$_3$	96	359	3692	–			719.24	ITP	15
IndustrialC$_4$	105	377	5279	–			415.62	BR	19
IndustrialC$_5$	138	608	1003	720.15	CBQ	6	315.63	BR	6
IndustrialD$_1$	119	76	1075	478.90	ITP	37	372.25	ITP	37
IndustrialD$_2$	138	97	2172	7157.35	ITP	35	378.91	ITP	35
IndustrialD$_3$	25	88	498	7124.54	ITP	45	25.55	BR	67
IndustrialD$_4$	21	116	3879	795.24	ITP	9	103.41	BR	9
IndustrialD$_5$	96	355	6360	–			507.27	ITP	10
IndustrialD$_6$	95	353	6348	5408.67	ITP	10	771.49	ITP	10

[5] M. K. Ganai, A. Gupta, and P. Ashar, "Efficient SAT-based Unbounded Symbolic Model Checking using Circuit Cofactoring," in *Proc. Int'l Conf. on Computer-Aided Design*. San Jose, California: IEEE Computer Society, Nov. 2004.

[6] K. L. McMillan, "Interpolation and SAT-Based Model Checking," in *Proc. Computer Aided Verification*, ser. LNCS, W. A. H. Jr. and F. Somenzi, Eds., vol. 2725. Boulder, CO, USA: Springer, 2003, pp. 1–13.

[7] K. L. McMillan and R. Jhala, "Interpolation and SAT-Based Model Checking," in *Proc. Computer Aided Verification*, ser. LNCS, T. Ball and R. B. Jones, Eds., vol. 3725. Edimburgh, Scotlan, UK: Springer, 2005, pp. 39–51.

[8] J. Marques-Silva, "Improvements to the Implementation of Interpolant–Based Model Checking," in *Proc. Computer Aided Verification*, ser. LNCS, D. Borrione and W. Paul, Eds., vol. 3725. Edimburgh, Scotlan, UK: Springer, 2005, pp. 367–370.

[9] G. Cabodi, M. Murciano, S. Nocco, and S. Quer, "Stepping Forward with Interpolants in Unbounded Model Checking," in *Proc. Int'l Conf. on Computer-Aided Design*. San Jose, California: ACM Press, Nov. 2006.

[10] B. Li and F. Somenzi, "Efficient Abstraction Refinement in Interpolation-Based Unbounded Model Checking," in *Tools and Algorithms for the Construction and Analysis of Systems*, vol. 3920, 2006, pp. 227–241.

[11] A. Biere and T. Jussila, "The Model Checking Competition Web Page, http://fmv.jku.at/hwmcc," 2007.

[12] A. Biere, A. Cimatti, E. M. Clarke, M. Fujita, and Y. Zhu, "Symbolic Model Checking using SAT Procedures instead of BDDs," in *Proc. 36th Design Automation Conf.* New Orleans, Louisiana: IEEE Computer Society, Jun. 1999, pp. 317–320.

[13] P. Bjesse and A. Boralv, "DAG-Aware Circuit Compression for Formal Verification," in *Proc. Int'l Conf. on Computer-Aided Design*. San Jose, California: IEEE Computer Society, Nov. 2004.

[14] G. Cabodi, P. Camurati, and S. Quer, "Auxiliary Variables for BDD–based Representation and Manipulation of Boolean Functions," *ACM Transactions on Design Automation of Electronic Systems*, vol. 3, no. 3, pp. 309–340, Jul. 1998.

[15] R. Hojati, S. C. Krishnan, and R. K. Brayton, "Early Quantification and Partitioned Transition Relation," in *Proc. Int'l Conf. on Computer Design*, Austin, Texas, Oct. 1996, pp. 12–19.

[16] K. Ravi and F. Somenzi, "High–Density Reachability Analysis," in *Proc. Int'l Conf. on Computer-Aided Design*, San Jose, California, Nov. 1995, pp. 154–158.

[17] M. Mneimneh and K. Sakallah, "SAT-based Sequential Depth Computation," in *Proc. Int'l Conf. on Asia South Pacific Design Automation*. New York, NY, USA: ACM, 2003, pp. 87–92.

[18] R. K. B. et al., "VIS," in *Proc. Formal Methods in Computer-Aided Design*, ser. LNCS, M. Srivas and A. Camilleri, Eds., vol. 1166. Palo Alto, California: Springer, Nov. 1996, pp. 248–256.

[19] A. Mishchenko, "ABC: A System for Sequential Synthesis and Verification, http://www.eecs.berkeley.edu/∼alanmi/abc/," 2005.

Automatic Generation of Local Repairs for Boolean Programs

Roopsha Samanta, Jyotirmoy V. Deshmukh and E. Allen Emerson
Department of Electrical and Computer Engineering and Department of Computer Sciences,
The University of Texas at Austin, Austin TX 78712, USA
{roopsha,deshmukh,emerson}@cs.utexas.edu

***Abstract*—Automatic techniques for software verification focus on obtaining witnesses of program failure. Such counterexamples often fail to localize the precise cause of an error and usually do not suggest a repair strategy. We present an efficient algorithm to automatically generate a repair for an incorrect sequential Boolean program where program correctness is specified using a pre-condition and a post-condition. Our approach draws on standard techniques from predicate calculus to obtain annotations for the program statements. These annotations are then used to generate a synthesis query for each program statement, which if successful, yields a repair. Furthermore, we show that if a repair exists for a given program under specified conditions, our technique is always able to find it.**

I. Introduction

In spite of their significance in ensuring program correctness, techniques for automatic error localization and repair are relatively unexplored. Error traces generated by model checking and static analysis tools may be long, encumbered with unnecessary data, and may give little insight into the actual location of the error within the program. Generally, verification fails to suggest a concrete strategy for error localization or repair, and developers end up relying on their own intuition and experience to debug and fix faulty programs. On the other end of the spectrum, synthesis of programs from high level specifications has been well-studied [1]. While synthesis solves an important theoretical problem, synthesizing a large piece of code is a computationally intensive task, and mandates writing a very rich and detailed specification. Thus, it is potentially more cost-effective to debug and repair existing software than attempt to synthesize it from scratch. While one cannot hope to completely eliminate the human programmer from the debugging process, bugs that are amenable to automatic repair should be dealt with accordingly.

In this paper, we present a method for efficient and automatic repair of programs that tries to bridge the gap between manual repair and automatic synthesis. Our strategy is applicable to a large subset of sequential Boolean programs [2], and has complexity comparable to that of model checking. Popular verification tools such as SLAM [3], and BLAST [4] use Boolean programs as abstractions for programs written in higher level programming languages, as they are inherently simpler to analyze than concrete programs.

Using the imperative programming paradigm, we specify *total correctness* of a Boolean program $\mathcal{P}$ as a Hoare(-style) triple $\langle\varphi\rangle\mathcal{P}\langle\psi\rangle$, where φ is a pre-condition that specifies initial values, and ψ is a post-condition that expresses the relation between the desired final values and the initial values. Our repair algorithm has two main steps. In the first step, we annotate $\mathcal{P}$ by propagating φ and ψ through the program statements. In the second step, we choose either an automatically generated or a user-specified order to target statements for repair. For every chosen statement, we attempt to synthesize a local repair using the propagated φ and ψ. Local synthesis serves to minimize modifications made to the original program, a desirable property for any automatic repair engine. Once such a synthesis query succeeds, a repair is extracted and our algorithm halts. If all queries fail, the algorithm reports that the program is irreparable within the constraints imposed by the repair model. We prove that our algorithm is sound, and specify the conditions under which it is complete. The worst-case complexity of our algorithm is exponential in the number of program variables. In practice, the algorithm can be much more effficient due to symbolic implementation.

A. Related Work

Extant work in automatic repair of software programs is directed towards repair of both sequential and concurrent programs using different program models like Kripke structures, UNITY [5] programs, game graphs and Boolean programs. Repair is geared towards satisfying various correctness specifications like propositional assertions, invariants, UNITY properties and temporal logics. The most pertinent approach repairs Boolean programs that violate propositional assertions [6]. This work is an extension of the authors' earlier work on repairing programs with LTL specifications [7], [8], and reduces automatic repair to finding a winning strategy for a pushdown game graph (obtained from the original program). The size of such a game graph can be quite formidable, and the complexity of finding a winning strategy is doubly exponential in the number of program variables.

There has been some work in proof-directed error localization and repair [9], [10], which borrows a proof planning framework from Artificial Intelligence (AI) for proof automation. This work is still in a nascent phase, and its use is likely to be limited to functional programs. In another paper [11], which also draws on principles from AI, the authors propose a repair algorithm based on abductive reasoning and theory revision. This work focuses on reactive programs with specifications in CTL, and is not very efficient as it requires invocation of a

978-1-4244-2735-2/08/$25.00 ©2008 IEEE

model checker to ascertain the validity of each repair.

Repair of Kripke structures with respect to CTL and other temporal logic formulas has received attention in both the AI [12] and formal verification [13] communities. Work done in automatic addition of UNITY properties to programs [14] is also relevant. It is worthwhile to mention *sketching* [15], [16], which is a software synthesis technique that utilizes a partial program or a *sketch* written by a programmer, and completes it to satisfy a specification.

There has been a lot of work on automatically correcting sequential and combinational circuits, and we refer the reader to a paper by Staber *et al.* [8] for a summary of such techniques. We also note a recent paper [17] in which the authors translate the problem of localizing and fixing faults in sequential circuits to solving Quantified Boolean Formulas (QBFs). While we use QBFs in our approach as well, our QBFs have only two alternating quantifiers unlike theirs, and are more tractable.

In addition to the above work, a multitude of algorithms [18], [19], [20] have been proposed for error localization based on analyzing error traces. Many of these techniques try to localize faults by finding correct program traces most similar to an incorrect counterexample trace. Some of these techniques could be exploited to obtain heuristics for the order in which we target program statements for repair.

The rest of the paper is organized as follows. In Section II we briefly look at the syntax and semantics of Boolean programs. We explain the propagation mechanism in Section III, and present our repair strategy in Section IV. We present an extension of our approach to programs with function calls in Section V, and conclude in Section VI.

II. Preliminaries

Boolean programs have a control-flow structure that is similar to their source code and closely resemble C programs. The key difference is that all variables in a Boolean program are Boolean variables. Moreover, Boolean programs support nondeterminism, an inherent by-product of the counterexample-guided abstraction refinement (CEGAR) paradigm used in tools that generate Boolean programs. In this section, we present a programming language with a syntax that greatly resembles that of Boolean programs as defined in [2], and is more amenable to our technique. For instance, we forego parallel assignments in favour of single assignments. We remove `assert` statements from the programming language and replace such statements with specifications based on pre- and post-conditions. Note that any sequential Boolean program with the syntax of [2] can be converted into an equivalent program with our syntax and correctness specification. Thus, we consider it unnecessary to distinguish between the two, and refer to our programs, with their marginally modified syntax, as Boolean programs as well.

A. Syntax and Semantics

A Boolean program $\mathcal{P}$ is defined as a tuple $(\mathcal{F}, \texttt{main}, \mathcal{V})$, where $\mathcal{F}$ is a set of functions, `main` is the initial function, and $\mathcal{V} = \{v_1, v_2, \ldots, v_t\}$ is a set of Boolean global variables. The `main` function is a tuple $(S, \mathcal{V})$, where $S = (s_1; s_2; \ldots; s_n)$ is a sequence of n statements. A function $f \in \mathcal{F}$ is a tuple $(S_f, \mathcal{V}_{f,l})$, where $S_f = (s_{f,1}; s_{f,2}; \ldots; s_{f,q})$ is a sequence of q statements and $\mathcal{V}_{f,l}$ is a set of Boolean local variables. The set of variables *visible* to function f is denoted by $\mathcal{V}_f$.

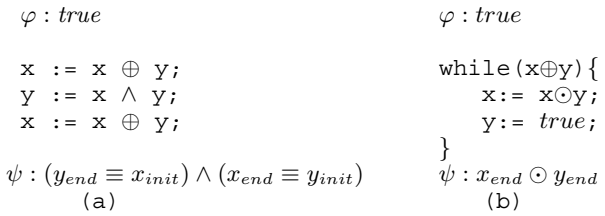


Fig. 1. Examples of Boolean programs

An expression in our programming language is a Boolean expression, and is allowed to contain a *fair* nondeterministic choice expression, $nd(0,1)$, which nondeterministically evaluates to *true* or *false*. For example, the expression $v_2 \wedge nd(0,1)$ evaluates to *false* when $v_2 = false$, and nondeterministically evaluates to *false* or *true* when $v_2 = true$. The choice is fair in the sense that it does not permanently evaluate to a single value. Program statements are either function calls or `return`, `skip`, assignment, conditional or loop statements. As in [2], every function call is required to be followed by a `skip` statement. Every function has a `return` statement. The syntax of the other statements is defined below.

Assignment statement: v_j `:=` E, where E is an expression over $2^{\mathcal{V}}$, or a function call. Each assignment statement is an assignment to a single program variable. For an assignment statement in function f, E is an expression over $2^{\mathcal{V}_f}$ or a function call.

Conditional statement: `if` (G) S_{if} `else` S_{else}, where G is an expression, and S_{if}, S_{else} are sequences of statements.

Loop statement: `while` (G) S_{body}, where G is an expression, and S_{body} is a sequence of statements.

The guard G in a conditional or loop statement is either a deterministic expression, or, the expression $nd(0,1)$. G is not allowed to be an arbitrary nondeterministic expression, which conforms with most reasonable abstraction frameworks. We show two example Boolean programs in Figure 1. Program (a) is meant to swap values of two variables without the use of a temporary variable, while Program (b) is meant to iterate till the values of x and y are the same. Both programs are incorrect.

We further associate a state transition graph with a Boolean program, with a state corresponding to a specific statement in the program and a valuation of the visible variables prior to the execution of the statement. The action of a statement in conjunction with the control-flow at that point of the program defines a transition from one state to another. In this paper, we will interchangeably use both the syntactic model and the state transition graph of a Boolean program without distinguishing them unless necessary. We now briefly describe our specification and repair model.

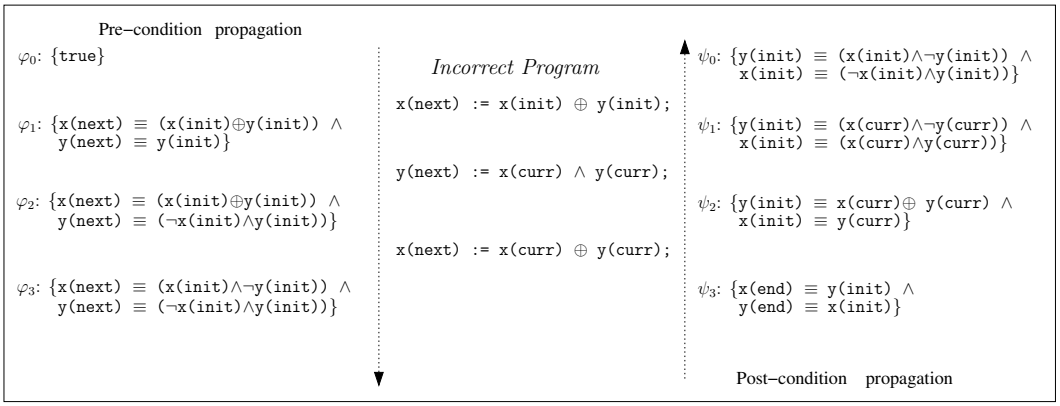


Fig. 2. Pre-condition and Post-condition Propagation

B. Specification and Repair Model

We assume that program correctness is specified as a pair of Boolean expressions known as the pre-condition φ, and the post-condition ψ. φ represents the initial states of a Boolean program and is a Boolean expression over the initial values of the global program variables $\mathcal{V}$. ψ represents the desired final states for initial states specified by φ, and is a Boolean expression relating the initial and final values of $\mathcal{V}$. φ and ψ do not contain any $nd(0,1)$ expressions. Formally, we specify total correctness of the program $\mathcal{P}$ using the Hoare triple: $\langle\varphi\rangle\mathcal{P}\langle\psi\rangle$. Further, we interpret the nondeterminism in an abstract Boolean program as Dijkstra's *demonic* nondeterminism [21]. Thus, $\mathcal{P}$ is correct if and only if the execution of $\mathcal{P}$, begun in any state satisfying φ, terminates in a state satisfying ψ, for *all* choices that $\mathcal{P}$ might make.

Assumptions made about the types of errors that could occur in a program are said to constitute an *error model*. Many repair techniques start with a set of preconceived targets for repair based on a programmer's notion of what the program should have been. Consequently, a lot of effort is expended on error localization. However, there may be a way to realize the desired specifications by "fixing" an entirely different statement than what the developer had conceived. We use this fact in our approach, and attempt to repair any statement that makes the Hoare-triple $\langle\varphi\rangle\mathcal{P}\langle\psi\rangle$ *true*. Thus, since our technique obviates the need for error localization, we focus on a *repair model*, as opposed to an error model.

The repair model captures the kinds of repairs that can be automatically generated by our technique. Our algorithm tries to repair individual program statements by either deleting them or modifying them without changing their type. A modified assignment statement may differ from the original one in both the left and right hand side of the assignment operator. For a conditional statement, a modification constitutes a change to either the test condition G or a statement within S_{if} or S_{else}. Similarly, in a loop statement, a modification changes either the loop guard G or a statement within S_{body}.

In our approach, the repair process has two main steps.

1) We annotate the program text by propagating φ and ψ through each program statement.
2) We use these annotations to inspect statements for repairability, and generate a repair if possible.

In the following sections, we treat each of these steps in detail. For ease of exposition, we first describe our algorithm for a program without any function calls, and extend our approach to programs with functions calls in Section V.

III. Step I: Program Annotation

Propagation of the initial pre-condition φ and the final post-condition ψ through program statements is based on the techniques used for Hoare logic [22], [23], with some modifications to the technique for propagating ψ. We denote the initial pre-condition φ by φ_0, and pre-conditions propagated forward through statements $s_1, \ldots, s_n$ by $\varphi_1, \ldots, \varphi_n$. Similarly, the final post-condition ψ is denoted by ψ_n, and post-conditions propagated back through statements $s_n, \ldots, s_1$ are denoted by $\psi_{n-1}, \ldots, \psi_0$.

To aid efficient computation and storage of propagated pre- and post-conditions, we add indices to identify certain valuations of the program variables. The initial set of variable values is denoted by $\mathcal{V}(init)$, and the final set of variable values obtained after execution of the n^{th} statement is denoted by $\mathcal{V}(end)$. Further, the sets of values of the variables *before* and *after* execution of any statement in the program are distinguished by representing them as $\mathcal{V}(curr)$ and $\mathcal{V}(next)$, respectively. We refer the reader to Figure 2 for the indexed and annotated version of Program (a) from Figure 1.

Henceforth, an expression E over $\mathcal{V}(curr)$ is sometimes denoted by $E(curr)$ for clarity. Moreover, if $E(curr)$ contains an $nd(0,1)$ expression, we use the notations $E(curr)|_0$ and $E(curr)|_1$ to denote the expressions obtained from $E(curr)$ when $nd(0,1)$ evaluates to $false$ and $true$, respectively. The need for maintaining these indices will become clear as we summarize the computations for propagating pre- and post-conditions in the following subsections. Note that our propagation techniques, combined with the absence of $nd(0,1)$ expressions in φ_0 and ψ_n, ensure the absence of $nd(0,1)$ expressions in all propagated pre- and post-conditions.

A. Post-condition Back Propagation

Propagation of a post-condition ψ_i through statement s_i corresponds to computing the weakest pre-condition $wp(s_i, \psi_i)$ that would make the Hoare triple $\langle \psi_{i-1} \rangle s_i \langle \psi_i \rangle$ $true$. In other words, $wp(s_i, \psi_i)$ represents the set of all *input* states such that execution of s_i is guaranteed to terminate in a state satisfying ψ_i, for all (nondeterministic) choices made by s_i. In the event that $wp(s_i, \psi_i)$ evaluates to $false$ for any statement s_i, our algorithm aborts propagation of post-conditions, and proceeds to the next phase, *i.e.*, pre-condition propagation (Section III-B) and repair generation (Section IV). Given a statement s_i and a post-condition ψ_i (an expression over $\mathcal{V}(next)$ and $\mathcal{V}(init)$), we obtain the weakest pre-condition $\psi_{i-1} = wp(s_i, \psi_i)$ using the following inductive rules:

1) Assignment statement: $v_j(next)$:= $E(curr)$*:* The weakest pre-condition is computed from ψ_i by replacing the variable $v_j(next)$ in ψ_i with its assigned expression $E(curr)$, and swapping all other variables $v_m(next)$ in ψ_i with $v_m(curr)$. Thus, the resulting expression ψ_{i-1} is an expression over $\mathcal{V}(curr)$ and $\mathcal{V}(init)$ and is given by[1]:

$$\psi_{i-1} = \psi_i[v_j(next) \rightarrow E(curr), \\ \text{for each } m \neq j, v_m(next) \rightarrow v_m(curr)].$$

If $E(curr)$ contains an $nd(0,1)$ expression, the weakest pre-condition is computed as the *conjunction* of the weakest pre-conditions over the statements given by $v_j(next)$:= $E(curr)|_0$ and $v_j(next)$:= $E(curr)|_1$.

2) Sequential composition: To propagate the post-condition ψ_i back through a sequence of statements $(s_{i-1}; s_i)$, it is first propagated through s_i to obtain $wp(s_i, \psi_i)$; $wp(s_i, \psi_i)$ is then used as the post-condition to be propagated through s_{i-1} to obtain the required weakest pre-condition over $(s_{i-1}; s_i)$. In order to maintain the correct indices, all variables $v_m(curr)$ in the expression for $wp(s_i, \psi_i)$ are swapped with $v_m(next)$ before propagating through s_{i-1}. This rule is expressed as:
$wp((s_{i-1}; s_i), \psi_i) = wp(s_{i-1}, wp(s_i, \psi_i))$.

3) Conditional statement: `if` $(G(curr))$ S_{if} `else` S_{else}*:* The weakest pre-condition of a conditional statement is computed to be the same as the weakest pre-condition over S_{if} if the guard G is $true$, and as the weakest pre-condition over S_{else} if G is $false$. The weakest pre-condition over S_{if} and S_{else} is computed inductively using Rule 2. Thus we have:

$$\psi_{i-1} \;=\; (G(curr) \Rightarrow wp(S_{if}, \psi_i)) \;\wedge \\ (\neg G(curr) \Rightarrow wp(S_{else}, \psi_i)).$$

If $G(curr) = nd(0,1)$, the weakest pre-condition is given by the *conjunction* of the weakest pre-conditions over S_{if} and S_{else}: $wp(S_{if}, \psi_i) \wedge wp(S_{else}, \psi_i)$.

[1]We use the standard notation $\delta[y \rightarrow x]$ to represent the expression obtained by replacing all occurrences of y in δ by x.

4) Loop statement: `while` $(G(curr))$ S_{body}*:* The post-condition propagated through loop statements is computed as a fixpoint over the weakest pre-condition of each loop iteration. We define the weakest pre-condition of the k^{th} loop iteration, $wp(S^k_{body}, \psi_i)$, as the set of input states that terminate in a state satisfying ψ_i after executing the loop *at most* k times. These sets of states are given by the following inductive expressions:

$$wp(S^0_{body}, \psi_i) = Y_0 \wedge \neg G,$$
$$wp(S^k_{body}, \psi_i) = \bigvee_{l=0}^{k-1} (wp(S^l_{body}, \psi_i)) \vee (Y_k \wedge G),$$

where, $Y_0 = \psi_i$, $Y_k = wp(S_{body}, Y_{k-1} \wedge \neg G)$.

The last term in the expression for $wp(S^k_{body}, \psi_i)$ represents the set of input states that enter the loop, and exit it after exactly k loop iterations, terminating in a state satisfying ψ_i. The weakest pre-condition of the $(k-1)^{th}$ loop iteration is computed in terms of variables in $\mathcal{V}(curr)$, which are then swapped with their respective variables in $\mathcal{V}(next)$, and used for computing the weakest pre-condition of the k^{th} iteration. Since Boolean programs have a finite number of states, fixpoint computation over these monotonically increasing sets of weakest pre-conditions terminates after a bounded number of iterations, say L, by the Tarski-Knaster Theorem. The weakest pre-condition for the loop statement s_i, which corresponds to states that exit the loop eventually and satisfy ψ_i, is given by:

$$\psi_{i-1} \;=\; \bigvee_{l=0}^{L} (wp(S^l_{body}, \psi_i)), \text{ or equivalently,}$$
$$\psi_{i-1} \;=\; (\psi_i \wedge \neg G) \vee \bigvee_{l=1}^{L} wp(S_{body}, Y_{l-1} \wedge \neg G).$$

If $G(curr) = nd(0,1)$, then we define a different set of iterants Z_k, which correspond to input states that terminate in a state satisfying ψ_i in the k^{th} loop iteration when G evalues to $false$. We remind the reader that $nd(0,1)$ expresses fair choice between $true$ and $false$. Hence, the loop guard is guaranteed to evaluate to $false$ eventually. The weakest pre-condition, which corresponds to all input states that terminate in a state satisfying ψ_i for all values of the guard, can be computed as the fixpoint: $\psi_{i-1} \;=\; \bigwedge_{l=0}^{L'} Z_l$, where, $Z_0 = \psi_i$, $Z_k = wp(S_{body}, Z_{k-1})$.

B. Pre-condition Forward Propagation

Traditionally, the definition of the strongest post-condition (sp) is the dual of the weakest *liberal* pre-condition (wlp). These are used in the *partial correctness* framework and do not concern themselves with program termination [23]. Hence, the traditional sp should ideally be called the strongest *liberal* post-condition. We redefine the strongest post-condition to be the dual of the weakest pre-condition for total correctness. Thus, propagation of a pre-condition φ_{i-1} through a statement s_i corresponds to computing the strongest post-condition $sp(s_i, \varphi_{i-1})$ that would make the Hoare triple $\langle \varphi_{i-1} \rangle s_i \langle \varphi_i \rangle$ $true$. In other words, $sp(s_i, \varphi_{i-1})$ represents the smallest set of *output* states such that execution of s_i, begun in all states

satisfying φ_{i-1}, is guaranteed to terminate in one of them for all (nondeterministic) choices that s_i might make. It is important to note that the strongest post-condition is undefined if there exists *some* input state for which *some* execution of s_i may not terminate. If the strongest post-condition $sp(s_i, \varphi_{i-1})$ evaluates to *false* or is undefined for any statement s_i, our algorithm aborts propagation of pre-conditions, and proceeds to the next phase, *i.e.*, repair generation (Section IV). We refer the interested reader to Appendix A for a detailed treatment of strongest post-condition computation for each type of program statement.

IV. Step II: Synthesis of Repairs

In this section, we present an algorithm to generate repairs given an annotated program $\mathcal{P}$ that does not satisfy its specification, *i.e.*, the Hoare triple, $\langle\varphi\rangle\mathcal{P}\langle\psi\rangle$ is *false*. We also prove that our algorithm is sound and complete with respect to our repair model. Before proceeding, we establish some groundwork in order to motivate the algorithm.

A. Groundwork

We first characterize our notion of correctness in the following lemma, which is stated without proof. The first part of the lemma is standard [23], and the second part of the lemma follows from our definition of strongest post-conditions (Appendix A).

Lemma 1: Characterization of total correctness:
For any statement s in a given program $\mathcal{P}$,

$$\langle\varphi\rangle s\langle\psi\rangle \equiv \varphi \Rightarrow wp(s,\psi),$$

$$\langle\varphi\rangle s\langle\psi\rangle \equiv \begin{cases} sp(s,\varphi) \Rightarrow \psi, & \text{when } sp(s,\varphi) \text{ is defined,} \\ false, & \text{otherwise.} \end{cases}$$

Thus, when $sp(s, \varphi)$ is undefined, the Hoare triple $\langle\varphi\rangle s\langle\psi\rangle$ is *false* for all possible ψ.

As shown in Section III, we annotate each program statement s_i with a propagated pre-condition φ_{i-1} and a propagated post-condition ψ_i. This provides us with n *local* Hoare triples corresponding to the n program statements. In the following lemma, we present an interesting relation between the local Hoare triples and the Hoare triple for the entire program. This lemma is the basis for our repair algorithm. We claim that $\langle\varphi\rangle\mathcal{P}\langle\psi\rangle$ is *false* if and only if all the local Hoare triples are *false*. Further, all the local Hoare triples are *false* if and only if any one local Hoare triple is *false*.

Lemma 2: For a Boolean program $\mathcal{P}$, composed of a sequence of n statements, $s_1; s_2; \ldots; s_n$, the following expressions are equivalent when all strongest post-conditions $\psi_1, \psi_2, \ldots, \psi_n$ are defined:

$\langle\varphi\rangle\mathcal{P}\langle\psi\rangle$,
$\langle\varphi_{i-1}\rangle s_i\langle\psi_i\rangle$, for any given i in $\{1, 2, \ldots, n\}$,
$\langle\varphi_{i-1}\rangle s_i\langle\psi_i\rangle$, for all i in $\{1, 2, \ldots, n\}$.

Proof: We first prove an equivalence between the second and third expressions. Let $(s_{i-1}; s_i; s_{i+1})$ be a sequence of any three consecutive statements of program $\mathcal{P}$. Consider the Hoare triple, $\langle\varphi_{i-1}\rangle s_i\langle\psi_i\rangle$. We have:

$$\begin{aligned} \langle\varphi_{i-1}\rangle s_i\langle\psi_i\rangle &\equiv \varphi_{i-1} \Rightarrow wp(s_i,\psi_i) && \text{(Lemma 1)} \\ &\equiv sp(s_{i-1},\varphi_{i-2}) \Rightarrow \psi_{i-1} && \text{(Definition)} \\ &\equiv \langle\varphi_{i-2}\rangle s_{i-1}\langle\psi_{i-1}\rangle && \text{(Lemma 1).} \end{aligned}$$

We can show $\langle\varphi_{i-1}\rangle s_i\langle\psi_i\rangle \equiv \langle\varphi_i\rangle s_{i+1}\langle\psi_{i+1}\rangle$ using a similar argument. Extending this result to program statements preceding s_{i-1} and succeeding s_{i+1}, we arrive at an equivalence between the second and third expressions.

We prove an equivalence between the first and second expressions by noting that: $\psi_0 = wp(s_1,\psi_1) = wp((s_1; s_2),\psi_2) = \ldots = wp((s_1; s_2; \ldots; s_n),\psi) = wp(\mathcal{P},\psi)$. Then, we have:

$$\begin{aligned} \langle\varphi\rangle\mathcal{P}\langle\psi\rangle &\equiv \varphi \Rightarrow wp(\mathcal{P},\psi) && \text{(Lemma 1)} \\ &\equiv \varphi_0 \Rightarrow wp(s_1,\psi_1) \\ &\equiv \langle\varphi_0\rangle s_1\langle\psi_1\rangle && \text{(Lemma 1).} \end{aligned}$$

A similar equivalence can be obtained between $\langle\varphi\rangle\mathcal{P}\langle\psi\rangle$, and $\langle\varphi_{i-1}\rangle s_i\langle\psi_i\rangle$, for any i.

■

B. Repair Algorithm

A consequence of Lemma 2 is that if the i^{th} Hoare triple is made *true* by modifying s_i, then the program so obtained satisfies the specification. Hence, our repair strategy proceeds by examining program statements in some specific order to identify potential candidates for repair. If propagation of post-conditions is aborted due to an empty weakest pre-condition for statement s_i, we check for repairable statements among the statements $(s_i; s_{i+1}; \ldots; s_n)$. Similarly, if propagation of pre-conditions is aborted due to an undefined or empty strongest post-condition for statement s_i, we check for repairable statements among the statements $(s_0; s_1; \ldots; s_i)$.

Before checking if a statement can be repaired by modifying it, we do a simple check to see if the statement can be repaired by deleting it. We can do this by computing the QBF:

$\forall_{\mathcal{V}(init)}\ \varphi_{i-1} \Rightarrow \psi_i(curr)$.

Here, $\psi_i(curr)$ is the post-condition ψ_i with all variables in $\mathcal{V}(next)$ swapped with their corresponding variables in $\mathcal{V}(curr)$. Thus, this QBF checks if the local Hoare triple given by $\langle\varphi_{i-1}\rangle s_i\langle\psi_i\rangle$ can be made *true* if s_i is deleted. If this QBF is *true*, we return a new program $\widehat{\mathcal{P}}$ with s_i deleted as the repaired program. If this QBF is *false*, we proceed with the algorithm outlined in the rest of this section, and assume that deletion of s_i is not an option for repair.

For every statement s_i to be inspected, we pose a query to check if s_i can be repaired. If the query returns a `yes`, we proceed to synthesize a repair and declare success. If not, we proceed to the next statement in the given order. If none of the statements can be repaired, we report failure in repairing the program under current constraints. We now explain how to formulate `Query` and `Repair` for each type of program statement.

1) Assignment Statement: Suppose s_i is an assignment statement that we wish to repair. Let $\hat{s}_{i,j}$ denote a potential repair for s_i that assigns an expression $expr$ to the variable v_j, *i.e.*, $\hat{s}_{i,j}$ is v_j `:=` $expr$. The `Query` function determines if there exists such an expression. To be able to do this, we use a variable z to denote $expr$, and pose the following question: does there exist some variable v_j such that for all initial values of the program variables, there exists an assignment z to v_j, which makes the local Hoare triple, $\langle\varphi_{i-1}\rangle v_j := z\langle\psi_i\rangle$, $true$. Formally, `Query` returns `yes` if, for any j, the following QBF is $true$, and `no` otherwise:

$\forall_{\mathcal{V}(init)}\exists_z\ \varphi_{i-1} \Rightarrow \widehat{\psi}_{i-1,j}$.

Each $\widehat{\psi}_{i-1,j}$ is the weakest pre-condition corresponding to the proposed repair $\hat{s}_{i,j}$: `v`$_j$ `:=` `z`, and is an expression over z, $\mathcal{V}(init)$ and $\mathcal{V}(curr)$.

Suppose the m^{th} QBF evaluates to $true$. The key challenge is in obtaining an expression for z in terms of variables in $\mathcal{V}(curr)$. We may use the QBF's certificate of validity to derive such a z. In this paper, we provide a more direct solution to obtain z. Let us define $T = \varphi_{i-1} \Rightarrow \hat{\psi}_{i-1,m}$. We denote the positive cofactor of T w.r.t. z as $T|_z$[2]. By properties of Boolean functions, we can show that $\exists_z\ T$ is equivalent to $T|_{[z\rightarrow T|_z]}$, where $T|_{[z\rightarrow T|_z]}$ represents the expression obtained by replacing all occurences of z in T by $T|_z$. Thus, $z = T|_z$ is a witness to the validity of the QBF and could yield $expr$.

However, $T|_z$ is an expression containing variables in $\mathcal{V}(init)$ and $\mathcal{V}(curr)$. Hence the repair algorithm tries to express variables in $\mathcal{V}(init)$ in terms of variables in $\mathcal{V}(curr)$. If this is feasible, $expr$ is obtained solely in terms of variables from $\mathcal{V}(curr)$. If this cannot be done, the repair algorithm suggests adding at most t new constants, $\{v_1(0), v_2(0), \ldots, v_t(0)\}$, to store the initial values of the program variables. Note that this is the only time the repair algorithm suggests addition of constants or insertion of statements. In the following lemma, we prove that the repair generated as above is sound.

Lemma 3: Replacing the statement s_i by the statement $\hat{s}_i$: v_m `:=` $expr$, where v_m is identified from the `Query` function, and $expr$ is obtained as above, makes the Hoare triple $\langle\varphi_{i-1}\rangle\hat{s}_i\langle\psi_i\rangle$ $true$.

Proof: Recall that T is $\varphi_{i-1} \Rightarrow \widehat{\psi}_{i-1,m}$. Furthermore, by Lemma 1, we know that T represents correctness of the local Hoare triple $\langle\varphi_{i-1}\rangle\hat{s}_i\langle\psi_i\rangle$. Since $z = T|_z$ is a witness to the validity of T, $z = T|_z$ is a witness to the validity of the local Hoare triple. In other words, our approach for obtaining $expr$ by selecting an appropriate z ensures that the local Hoare triple is made $true$. ■

2) Conditional Statement: Let s_i be a conditional statement that we wish to repair. `Query` for s_i checks for the possibility of repairing either the guard, or a statement in S_{if}, or a statement in S_{else}.

Let $\widehat{G}(curr)$ represent the repair for the guard. We check for its existence by formulating a similar QBF as above and checking for its validity. If the QBF is valid, we can derive $\widehat{G}(curr)$ from the cofactor of T in a similar manner. If the QBF is not valid, then we make two separate sets of queries to check if any statement in S_{if} or in S_{else} can be repaired to yield $\widehat{S}_{if}$ or $\widehat{S}_{else}$, respectively.

Lemma 4: Replacing the conditional statement s_i by the statement $\hat{s}_i$: `if` $(\widehat{G}(curr))$ S_{if} `else` S_{else} or the statement $\hat{s}_i$: `if` $(G(curr))$ $\widehat{S}_{if}$ `else` S_{else} or the statement $\hat{s}_i$: `if` $(G(curr))$ S_{if} `else` $\widehat{S}_{else}$, where $\widehat{G}(curr)$, $\widehat{S}_{if}$ and $\widehat{S}_{else}$ are obtained as above, makes the Hoare triple $\langle\varphi_{i-1}\rangle\hat{s}_i\langle\psi_i\rangle$ $true$.

Proof: The proof is similar to the one for Lemma 3. ■

3) Loop Statement: Let s_i be a loop statement that we wish to repair. As in the repair for a conditional statement, `Query` for a loop statement checks for the possibility of repairing either the guard or a statement in S_{body}. `Query` and `Repair` for loop statements have an additional responsibility to ensure termination of the repaired loop, and differ from the versions we have seen so far. We explain these in detail to highlight these differences.

To check if the loop guard can be repaired, `Query` checks if there exist states which may enter the loop, execute it iteratively and never satisfy the post-condition ψ_i on completion of a loop execution. Such states are non-terminating, irrespective of the loop guard, and hence, their presence implies the absence of a repair for the loop guard. This set of states can be computed using a terminating fixpoint over the monotonically decreasing sets, $\bigwedge_k Y_k$, where,
$Y_0 = \varphi_{i-1} \wedge \neg\psi_i$, $Y_k = sp(S_{body}, Y_{k-1}) \wedge \neg\psi_i$.

If this fixpoint is non-empty, `Query` returns a `no` and the algorithm declares that the loop guard cannot be repaired. If this fixpoint is empty, `Query` returns `yes`, and the `Repair` function computes another fixpoint to yield the desired repair for the loop guard, $\widehat{G}(curr)$. The construction of this fixpoint ensures that any state which exits the loop satisfies ψ_i. The iterants of this fixpoint are the same as above, and the actual fixpoint representing $\widehat{G}(curr)$ is computed using the monotonically increasing sets, $\bigvee_k Y_k$. The rationale for our repair choice for the loop guard is provided by the following lemma.

Lemma 5: Given that the statement s_i: `while` $(G(curr))$ S_{body} cannot be repaired by deleting it, it can be repaired by modifying its loop guard if and only if the fixpoint $\bigwedge_k Y_k$, where the Y_k's are computed as above, evaluates to $false$, in which case, the fixpoint $\bigvee_k Y_k$, yields a valid repair for s_i.

Proof Sketch: See Appendix B.

If the loop guard cannot be repaired, we formulate a set of queries to check if any statement in S_{body} can be repaired. For simplicity of explanation, we show how to formulate a query for checking if an assignment statement in S_{body} can be repaired. The method extends inductively to inner conditional and loop statements. Let the p^{th} statement in S_{body} be an assignment statement, $s_{i,p}$. As before, let $\hat{s}_{i,p,j}$ denote a potential repair for $s_{i,p}$ that assigns an expression $expr$ to the variable v_j, *i.e.*, $\hat{s}_{i,p,j}$: v_j `:=` $expr$. Using a variable z to

[2]This is standard terminology for the expression obtained by substituting all occurences of z in the Boolean expression T by $true$.

denote $expr$, `Query` computes the weakest pre-condition of the loop corresponding to this repair, $\widehat{\psi}_{i-1,p,j}$, and returns a `yes` if the following QBF is $true$, and `no` otherwise:

$\forall_{\mathcal{V}(init)} \exists_z \varphi_{i-1} \Rightarrow \widehat{\psi}_{i-1,p,j}$.

To compute the weakest pre-condition, we compute a fixpoint as outlined in Section III-A, noting that each iterant of the fixpoint, and hence, the final fixpoint would be an expression over z, $\mathcal{V}(init)$, and $\mathcal{V}(curr)$. If the QBF is valid, we can derive z from the positive cofactor of T or, $\varphi_{i-1} \Rightarrow \widehat{\psi}_{i-1,p,j}$ as before. Since the weakest pre-condition computation guarantees termination, the derived repair z ensures that the loop terminates. If the QBF is not valid, we attempt to repair another statement within the loop body.

Lemma 6: Replacing the loop statement s_i by the statement $\hat{s}_i$: `while` $(\widehat{G}(curr))$ S_{body} or the statement $\hat{s}_i$:`while` $(G(curr))$ $\widehat{S}_{body}$, where $\widehat{G}(curr)$ and $\widehat{S}_{body}$ are computed as above, makes the Hoare triple $\langle \varphi_{i-1} \rangle \hat{s}_i \langle \psi_i \rangle$ $true$.

Proof: The proof for soundness of $\hat{s}_i$: `while` $(\widehat{G}(curr))$ S_{body} follows from Lemma 5. The proof for soundness of $\hat{s}_i$: `while` $(G(curr))$ $\widehat{S}_{body}$ is similar to that of Lemma 3. ■

We now prove that our algorithm is sound and complete with respect to our repair model.

Theorem 1: Given a Hoare triple $\langle \varphi \rangle \mathcal{P} \langle \psi \rangle$ that is $false$ due to the erroneous Boolean program $\mathcal{P}$, if there exists another Boolean program $\widehat{\mathcal{P}}$ satisfying the conditions enumerated below, our algorithm finds one such $\widehat{\mathcal{P}}$. If there exists a unique $\widehat{\mathcal{P}}$, then the algorithm finds it. If our algorithm finds a $\widehat{\mathcal{P}}$, then $\langle \varphi \rangle \widehat{\mathcal{P}} \langle \psi \rangle$ evaluates to $true$.

1) $\widehat{\mathcal{P}}$ differs from $\mathcal{P}$ in exactly *one* statement, *i.e.*, there exists exactly one i such that the statement s_i is absent in $\widehat{\mathcal{P}}$, or is replaced by another statement $\hat{s}_i$ in $\widehat{\mathcal{P}}$,
2) If s_i is an assignment statement, $\hat{s}_i$ is an assignment statement which may differ both in the left and right hand sides of the assignment operator,
3) If s_i is a conditional statement, $\hat{s}_i$ is a conditional statement that differs from s_i in either the guard G or in a statement in S_{if} or S_{else} and
4) If s_i is a loop statement, $\hat{s}_i$ is a loop statement that differs from s_i in either the guard G or in S_{body}.

Proof: The soundness result follows from Lemmas 1, 2, 3, 4 and 6. The completeness result is a direct consequence of our query formulation, which checks for the existence of exactly those repairs that satisfy the above repair model. ■

Note that, by completeness, we refer to the completeness of our algorithm in the domain of Boolean programs. When such programs are used within the CEGAR framework, the algorithm can no longer be complete due to the nature of abstraction.

Example 1: In the annotated program from Figure 2, the QBF for the 2^{nd} statement evaluates to $true$, and generates the expected repair: `y(2) := x(1)` $\oplus$ `y(1);` for the program.

Example 2: Program (b) in Figure 1 does not satisfy its specification. In fact, it does not terminate for some input values. Since the program consists of one loop statement, it does not require any annotation. Let us replace the expression on the right-hand side of the 2^{nd} assignment statement within the loop with z, and invoke the `Query` operation. The weakest pre-condition of the loop can be computed as the expression: $\psi_{0,2,y} = (x \wedge y) \vee (\neg x \wedge \neg y) \vee (z \wedge x \wedge \neg y) \vee (z \wedge \neg x \wedge y)$, and the QBF given by $\forall_{x,y} \exists_z (true \rightarrow \psi_{0,2,y})$ evaluates to $true$. We then compute the cofactor of $\psi_{0,2,y}$ to obtain $z = true$, which is a valid repair.

C. Complexity Analysis and Implementation

All fixpoint computations in our algorithm can be done in time exponential in the number of program variables in the worst-case. Moreover, the `Query` function for assignment and conditional statements involves checking validity of a QBF with exactly two alternating quantifiers and lies in the second polynomial hierarchy, *i.e.*, it is $\Sigma_2 P-$complete in the number of program variables. All other operations like swapping of variables, subsititution, Boolean manipulation and cofactor computation can be done in either constant time or time exponential in the number of program variables in the worst-case.

Thus, the worst-case complexity of our algorithm is exponential in the number of program variables. In practice, most of these computations can be done very efficiently using BDDs. The initial results obtained from a preliminary implementation of our algorithm that uses a Java-based BDD library [24] have been promising. Use of BDDs allows symbolic storage and efficient manipulation of pre-conditions and post-conditions as well as efficient computation of fixpoints. The `forall` and `exist` methods for BDDs facilitate computing validity of a desired QBF and hence enable the `Query` check. Similarly, the `Repair` operation can be performed easily by computing the cofactor of a BDD.

V. Annotation and Repair in presence of Functions

A. Annotation

Our algorithm can be extended to programs containing non-recursive function calls, and function calls with a restricted form of recursion (*e.g.* tail recursion). The basic idea is to use *function summaries* to represent the action of function calls. Suppose the formal parameters of a function f are denoted by $arg_0, \ldots, arg_z$. A *forward summary* of the function f, denoted by $S_{\{f,+\}}$ can be obtained by strongest post-condition propagation through f, by assuming an initial pre-condition of the form $\bigwedge_y (arg_y \equiv x_y)$. Here the x_y's are symbolic values representing the actual parameters of f at its call-site. Similarly, a *backward summary* of f, denoted by $S_{\{f,-\}}$ can be obtained by weakest pre-condition propagation through f, assuming a final post-condition that represents the return value (ret).

$S_{\{f,+\}}$ is used at each call-site of f during pre-condition propagation through the function containing the call-site. To propagate φ_i across a call to f, the x_y's in $S_{\{f,+\}}$ are instantiated with the values of the actual parameters at the call-site, and the result is combined with additional conjunct φ_i

(appropriately renaming the $curr$ and $next$ copies at the call-site). $S_{\{f,-\}}$ is used at each call-site of f during post-condition propagation through the function containing the call-site. To propagate ψ_i across a call-site of the form $v_m := f(\ldots)$, we replace ret in $S_{\{f,-\}}$ by v_m, and conjoin the result with conjunct ψ_i (appropriately renaming the $curr$ and $next$ copies at the call-site).

Since we preclude recursive calls, the call-graph for our programs is acyclic, and hence can be sorted in reverse topological order. We compute function summaries in this order to ensure that the summaries are always available at the call-site of each function.

B. Repair

To repair a statement within the function, we proceed as in the case of assignment, loop and conditional statements, by making an expression within that statement unknown (by replacing it with z). However, a function could be called multiple times within a program. A repair to a statement within the function could impact the overall pre- and post-conditions. Hence, once a function is considered for repair, the program needs to be re-annotated, before we can solve for z.

VI. Conclusions and Future Work

In this paper, we present an efficient and automatic technique for the repair of a large subset of Boolean programs with respect to a total correctness specification denoted by the Hoare triple $\langle\varphi\rangle\mathcal{P}\langle\psi\rangle$. Our technique first annotates each program statement with propagated pre- and post-conditions. It then queries each statement to check if it can be deleted or modified locally to validate its corresponding local Hoare triple. Upon a successful query, the algorithm generates a suitable repair. We prove that our algorithm is sound and complete with respect to the repair model used. Our approach obviates the need for error localization. Moreover, our algorithm avoids performing an exhaustive search for possible repairs, and has tractable complexity.

We remark that it may be possible to extend our approach to repair multiple statements by using multiple variables $z_1, z_2, \ldots, z_h$ to denote desired repairs for suspect expressions, and formulating a QBF over all these variables; the repair can be extracted from the certificate of validity of the QBF. Further, we note that Boolean programs can model both sequential and combinational circuits, and hence, our techniques can be used for repairing such circuits.

As an extension to this work, it would be interesting to explore richer formalisms like pushdown systems to deal with Boolean programs with arbitrary recursive functions. Also, it would be useful to extend our current approach to programs with bounded integers. With appropriate usage of SMT solvers, we may be able to extend the current techniques to such programs. Reasoning over such programs may help us avoid the abstraction-refinement loop encountered during Boolean program repair, and also pave the way for adapting our technique for microcode repair.

References

[1] P. C. Attie, A. Arora, and E. A. Emerson, "Synthesis of Fault-tolerant Concurrent Programs," *ACM Trans. Program. Lang. Syst.*, vol. 26, no. 1, pp. 125–185, 2004.

[2] T. Ball and S. K. Rajamani, "Boolean Programs: A Model and Process for Software Analysis."

[3] ——, "Automatically Validating Temporal Safety Properties of Interfaces," in *Proceedings of the 8th International Workshop on Model Checking of Software (SPIN)*. Springer-Verlag, 2001, pp. 103–122.

[4] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre, "Software Verification with BLAST," in *Proceedings of the 10th International Workshop on Model Checking of Software (SPIN)*, 2003, pp. 235–239.

[5] K. M. Chandy and J. Misra, *Parallel Program Design: A Foundation*. Addison-Wesley, 1988.

[6] A. Griesmayer, R. Bloem, and B. Cook, "Repair of Boolean Programs with an Application to C," in *18th Conference on Computer Aided Verification (CAV)*, T. Ball and R. B. Jones, Eds., 2006, pp. 358–371.

[7] B. Jobstmann, A. Griesmayer, and R. Bloem, "Program Repair as a Game," in *17th Conference on Computer Aided Verification (CAV)*, K. Etessami and S. K. Rajamani, Eds. Springer-Verlag, 2005, pp. 226–238.

[8] S. Staber, B. Jobstmann, and R. Bloem, "Finding and Fixing Faults," in *13th Conference on Correct Hardware Design and Verification Methods (CHARME)*, D. Borrione and W. Paul, Eds. Springer-Verlag, 2005, pp. 35–49.

[9] L. A. Dennis, "Program Slicing and Middle-Out Reasoning for Error Location and Repair," in *Disproving: Non-Theorems, Non-Validity and Non-Provability*, 2006.

[10] L. A. Dennis, R. Monroy, and P. Nogueira, "Proof-directed Debugging and Repair," in *Seventh Symposium on Trends in Functional Programming*, H. Nilsson and M. van Eekelen, Eds., 2006, pp. 131–140.

[11] F. Buccafurri, T. Eiter, G. Gottlob, and N. Leone, "Enhancing Model Checking in Verification by AI Techniques," *Artif. Intell.*, vol. 112, no. 1-2, pp. 57–104, 1999.

[12] Y. Zhang and Y. Ding, "CTL Model Update for System Modifications," *J. of Artif. Intell. Res.*, vol. 31, pp. 113–155, 2008.

[13] P. C. Attie and J. Saklawi, "Model and Program Repair via SAT Solving," Tech. Rep., 2007.

[14] A. Ebnenasir, S. S. Kulkarni, and B. Bonakdarpour, "Revising UNITY Programs: Possibilities and Limitations," in *International Conference on Principles of Distributed Systems (OPODIS)*, 2005, pp. 275–290.

[15] A. Solar-Lezama, R. Rabbah, R. Bodik, and K. Ebcioglu, "Programming by Sketching for Bit-streaming Programs," in *ACM SIGPLAN conference on Programming language design and implementation (PLDI)*. ACM, 2005, pp. 281–294.

[16] A. Solar-Lezama, L. Tancau, R. Bodik, S. Seshia, and V. Saraswat, "Combinatorial Sketching for Finite Programs," in *12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 2006, pp. 404–415.

[17] S. Staber and R. Bloem, "Fault Localization and Correction with QBF," in *Tenth International Conference on Theory and Applications of Satisfiability Testing (SAT)*, 2007.

[18] A. Groce, S. Chaki, D. Kroening, and O. Strichman, "Error Explanation with Distance Metrics," *Int. J. Softw. Tools Technol. Transf.*, vol. 8, no. 3, pp. 229–247, 2006.

[19] T. Ball, M. Naik, and S. K. Rajamani, "From Symptom to Cause: Localizing Errors in Counterexample Traces," in *Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL)*. ACM, 2003, pp. 97–105.

[20] H. Jin, K. Ravi, and F. Somenzi, "Fate and Free Will in Error Traces," *Int. J. Softw. Tools Technol. Transf.*, vol. 6, no. 2, pp. 102–116, 2004.

[21] E. W. Dijkstra, *A Discipline of Programming*. Prentice Hall, 1976.

[22] M. Huth and M. Ryan, *Logic in Computer Science: Modelling and Reasoning about Systems*. Cambridge Univeristy Press, 2004.

[23] E. W. Dijkstra and C. S. Scholten, *Predicate Calculus and Program Semantics*. Springer-Verlag New York, Inc., 1990.

[24] J. Whaley, "JavaBDD: An Efficient BDD Library for Java," http://javabdd.sourceforge.net/.

APPENDIX A
STRONGEST POST-CONDITION COMPUTATION

Pre-condition propagation involves recording the value of each program variable in terms of the initial values of the variables, conjoined with the conditions imposed by the program's control-flow. As will be evident below, the propagated pre-condition at any point in the program is a conjunction of α, which is some condition imposed by the program control-flow, and, $\bigwedge_m\ (v_m(next) \equiv \mathcal{F}_m(init))$, where $\mathcal{F}_m(init)$ is a an expression over $\mathcal{V}(init)$ and represents v_m in terms of the initial values of the program variables. For an expression E over $\mathcal{V}(curr)$, we use a special notation $E_{v(curr)\rightarrow\mathcal{F}(init)}$ to denote the expression obtained by substituting each variable $v_m(curr)$ in E by its corresponding expression $\mathcal{F}_m(init)$. Thus, $E_{v(curr)\rightarrow\mathcal{F}(init)}$ is an expression over $\mathcal{V}(init)$.

For the purpose of explaining pre-condition propagation, we assume that we are given a a statement s_i and a pre-condition $\varphi_{i-1} = \alpha\ \wedge \bigwedge_m (v_m(curr) \equiv \mathcal{F}_m(init))$, which is an expression over $\mathcal{V}(init)$ and $\mathcal{V}(curr)$. The strongest post-condition $\varphi_i = sp(s_i, \varphi_{i-1})$ can also be expressed in the same form: $\alpha\ \wedge \bigwedge_m (v_m(next) \equiv \mathcal{F}_m(init))$ and is obtained by the following rules for inductive computation:

1) Assignment statement: $v_j(next)$ `:=` $E(curr)$*:* The strongest post-condition of an assignment statement, $sp(x{:=}E, \delta)$, is given by $\exists_y(\delta[x \rightarrow y] \wedge x \equiv E[x \rightarrow y])$, [23]. In this setup, y represents the "previous" value of x, prior to the assignment, with y being unknown. This necessitates existential quantification over y to obtain the post-condition. In our program syntax, the use of indices enables us to preserve history information, and thus skip existential quantification. Further, since the single assignment statement, $v_j(next)$ `:=` $E(curr)$ leaves all but the j^{th} variable unchanged, we can express $sp(v_j(i)$ `:=` $E,\ \varphi_{i-1})$ as:

$$\varphi_i = \ \varphi_{i-1} \wedge \bigwedge\ (v_j(next) \equiv E) \wedge \bigwedge_{m\neq j} (v_m(next) \equiv v_m(curr)).$$

Equivalently, we replace φ_{i-1} by including α and substituting all variables $v_m(curr)$ with their corresponding $\mathcal{F}_m(init)$ expressions to obtain an expression for φ_i in terms of $\mathcal{V}(init)$ and $\mathcal{V}(next)$ as follows:

$$\varphi_i = \ \alpha \wedge \bigwedge (v_j(next) \equiv E_{v(curr)\rightarrow\mathcal{F}(init)}) \wedge \bigwedge_{m\neq j} (v_m(next) \equiv \mathcal{F}_m(init)).$$

If $E(curr)$ contains an $nd(0,1)$ expression, the strongest post-condition is computed as the *disjunction* of the strongest post-conditions over the statements, $v_j(next) := E(curr)|_0$ and $v_j(next) := E(curr)|_1$.

2) Sequential composition: The rule for propagation of the pre-condition φ_{i-1} through a sequence of statements $(s_i; s_{i+1})$ is similar to the rule for propagation of a post-condition. As before, in order to maintain the correct indices, all variables, $v_m(next)$ in the expression for $sp(s_i, \varphi_{i-1})$ are swapped with $v_m(curr)$ before propagating through s_{i+1}. The rule can be expressed as:
$sp((s_i; s_{i+1}), \varphi_{i-1}) = sp(s_{i+1}; sp(s_i, \varphi_{i-1}))$.

3) Conditional statement: `if` $(G(curr))\ S_{if}$ `else` S_{else}*:* The strongest post-condition of a conditional statement is computed as the disjunction of the strongest post-conditions over S_{if} and S_{else}. Thus we have:

$$\varphi_i = sp(\varphi_{i-1} \wedge G_{v(curr)\rightarrow\mathcal{F}(init)}, S_{if}) \vee sp(\varphi_{i-1} \wedge \neg G_{v(curr)\rightarrow\mathcal{F}(init)}, S_{else}).$$

Replacing φ_{i-1} and invoking our induction hypothesis about the form of a computed strongest post-condition, we get the following expression:

$$\varphi_i = (\alpha \wedge G_{v(curr)\rightarrow\mathcal{F}(init)} \wedge \bigwedge_m (v_m(next) \equiv \mathcal{F}_{m,if}(init))) \vee (\alpha \wedge \neg G_{v(curr)\rightarrow\mathcal{F}(init)} \wedge \bigwedge_m (v_m(next) \equiv \mathcal{F}_{m,else}(init))).$$

Note that we use different notations, $\mathcal{F}_{m,if}(init)$ and $\mathcal{F}_{m,else}(init)$, to denote the expressions over $\mathcal{V}(init)$ for an arbitrary variable v_m, arising from propagation through two different sequences of statements, S_{if} and S_{else}. Further simplifying, we have:

$$\varphi_i = \ \alpha \wedge \bigwedge_m (\ v_m(next) \equiv (\ (G_{v(curr)\rightarrow\mathcal{F}(init)} \wedge \mathcal{F}_{m,if}(init)) \vee (\neg G_{v(curr)\rightarrow\mathcal{F}(init)} \wedge \mathcal{F}_{m,else}(init))\ )\ ).$$

If $G(curr) = nd(0,1)$, the strongest post-condition is computed as the *disjunction* of the strongest post-conditions over S_{if} and S_{else}, *i.e.*,

$$\varphi_i = \ \alpha \wedge \bigwedge_m (\ v_m(next) \equiv (\mathcal{F}_{m,if}(init) \vee \mathcal{F}_{m,else}(init))\ ).$$

4) Loop statement: `while` $(G)\ S_{body}$*:* As in post-condition propagation, pre-condition propagation of loop statements involves fixpoint computation over the sp operator. The propagated pre-condition of the k^{th} loop iteration is denoted as $sp(S^k_{body}, \varphi_{i-1})$. It is the set of output states, for which execution of the loop body terminates in *at most* k iterations, when begun in some input state satisfying φ_{i-1}. It is computed as follows:

$$sp(S^0_{body}, \varphi_{i-1}) = Y_0 \wedge \neg G,$$
$$sp(S^k_{body}, \varphi_{i-1}) = \bigvee_{l=0}^{k} (Y_l \wedge \neg G),$$

where, $Y_0 = \varphi_{i-1}, Y_k = sp(S_{body}, Y_{k-1} \wedge G)$.

The strongest post-condition of the $(k-1)^{th}$ loop iteration is computed in terms of variables in $\mathcal{V}(next)$ (and $\mathcal{V}(init)$), which are then swapped with their respective variables in $\mathcal{V}(curr)$, and used for computing the strongest post-condition of the k^{th} iteration. As before, let us suppose the fixpoint computation terminates after L iterations. The strongest post-condition for the loop statement, which corresponds to the smallest set of output states such that the loop, when begun in all states satisfying φ_{i-1} is guaranteed to terminate in one of them is thus given by:

$$\varphi_i = \ \neg G \wedge (\varphi_{i-1} \vee \bigvee_{l=1}^{L} sp(S_{body}, Y_{l-1} \wedge G)).$$

If each variable v_m has the value $\mathcal{F}_{m,k}(init)$ at the end of the l^{th} iteration, then the above expression can be rewritten as: $\varphi_i = \alpha' \wedge \bigwedge_m \Big(v_m(next) \equiv (\bigvee_{l=1}^{L} F_{m,l}(init))\Big)$.

If $G(curr) = nd(0,1)$, the strongest post-condition is expressed as the fixpoint: $\varphi_i = \bigvee_{l=0}^{L'} Z_l$, where $Z_0 = \varphi_{i-1}, Z_k = sp(S_{body}, Z_{k-1})$. Since the sets $\bigvee_l Z_l$ are monotonically increasing, the above fixpoint is guaranteed to terminate in, say, L' iterations.

The above computation for the strongest post-condition of a loop is accompanied by a check for termination for *all* input states satisfying φ_{i-1}. In the event that there exists even one input state which does not terminate for all choices made by the loop guard, the above computation for the strongest post-condition is invalid as the strongest post-condition is undefined.

The termination check can be done in multiple ways. One approach is to propagate the prospective strongest post-condition computed above, back through the loop, and check if the weakest pre-condition so obtained contains φ_{i-1}. If this check is $false$, it implies the existence of states in φ_{i-1} that do not terminate. This approach is based on a characterization of total correctness that is presented in Lemma 1 in Section IV.

Appendix B
Proof Sketch of Lemma 5

Since the loop statement cannot be repaired by deleting it, some states satisfying the pre-condition for the loop φ_{i-1} must enter the loop. Hence, any repair $\widehat{G}(curr)$ for the loop guard must contain some states satisfying φ_{i-1}. Consider the fixpoint over the sets $\bigwedge_k Y_k$, where

$$Y_0 = \varphi_{i-1} \wedge \neg\psi_i,$$
$$Y_k = sp(S_{body}, Y_{k-1}) \wedge \neg\psi_i.$$

This fixpoint represents the set of all states that satisfy the pre-condition φ_{i-1}, but not the post-condition ψ_i, enter the loop, execute it iteratively and never satisfy the post-condition ψ_i on completion of a loop execution. Suppose this fixpoint does not evaluate to $false$, *i.e.*, it contains some states. The loop guard that can repair the loop while including some states from φ_{i-1}, cannot exclude states from this set as such states do not satisfy ψ_i on exiting the loop. On the other hand, a loop guard cannot include states from this set as such states will never transition into states that can exit the loop suitably, satisfying ψ_i. Thus, the states from this set cannot belong to either $\widehat{G}(curr)$ or $\neg\widehat{G}(curr)$ leading to a contradiction.

Hence, if we cannot choose a loop-guard that is disjoint from φ_{i-1}, a choice for a loop guard that can help repair the loop statement exists if and only if the above fixpoint is empty.

Assuming that the above fixpoint is empty, consider the fixpoint over the sets $\bigwedge_k \widehat{G}_k$, where

$$\widehat{G}_0 = \varphi_{i-1} \wedge \neg\psi_i,$$
$$\widehat{G}_k = sp(S_{body}, \widehat{G}_{k-1}) \wedge \neg\psi_i.$$

By construction, this fixpoint accumulates all states that satisfy the pre-condition for any loop iteration, but not the post-condition ψ_i before entering the loop. If we select the loop guard to be this fixpoint, then any state that exits the loop is forced to satisfy ψ_i by construction. Thus each iterant of this fixpoint accumulates states that are chosen by the guard to execute the loop during that iteration. Moreoever, since the first fixpoint in this proof is empty, there are no states which do not terminate.

Note that the above fixpoint is non-empty as the first iterant $\varphi_{i-1} \wedge \neg\psi_i$ is required to be non-empty. If the first iterant were empty, no states from φ_{i-1} would enter the loop, thereby making the loop unnecessary and contradicting the assumption in the lemma.

Consistency Checking of All Different Constraints over Bit-Vectors within a SAT Solver

Armin Biere and Robert Brummayer
Institute for Formal Models and Verification
Johannes Kepler University, Linz, Austria
Email: {armin.biere,robert.brummayer}@jku.at

***Abstract*—This paper shows how all different constraints (ADCs) over bit-vectors can be handled within a SAT solver. It also contains encouraging experimental results in applying this technique to encode simple path constraints in bounded model checking. Finally, we present a new compact encoding of equalities and inequalities over bit-vectors in CNF.**

I. Introduction

Many applications require to reason about inequalities over bit-vectors. More specifically, one is often interested in constraining bit-vectors to be pairwise different. In SAT based bounded model checking [1] such all different constraints (ADCs) are used to model simple paths (loop-free paths), which are used to compute occurrence diameters (length of longest loop-free paths) [1], or reverse occurrence diameters for k-induction [2].

A straight-forward encoding of ADCs over bit-vectors to SAT is obviously quadratic in the number of bit-vectors. There are linear QBF encodings [3]–[5], but currently available QBF solvers have a hard time to take advantage of these more compact encodings. Non symbolic algorithms from constraint programming, such as [6], [7], are not applicable due to the large domain sizes of bit-vectors, e.g. a 32 bit-vector variable has a domain size of 2^{32} possible values.

In this paper we show how ADCs over bit-vectors can be embedded into a SAT solver. The technique is similar to the lazy approach in satisfiability modulo theories (SMT). See [8] for a recent survey on (lazy) SMT. In contrast to previous work [9], ADCs are checked inside the SAT solver. In our application ADCs are used to encode simple path constraints for k-induction [2].

We propose a new and effective way to extend a standard SAT solver to the theory of symbolic ADCs over bit-vectors. Our approach avoids costly restarts. Lemmas, generated for ADCs, are marked as learned clauses, which can be garbage collected. Furthermore, we present an efficient incremental consistency checking algorithm for symbolic all different constraints over bit-vectors. This technique allows us to solve many instances for which the classical approach [9] of incrementally adding bit-vector inequalities on demand fails. We always need less memory and also less time if we combine our technique with the refinement based approach [9].

II. External Consistency Checking of ADCs

Let $s_1, s_2, \ldots, s_m$ be bit-vectors of width n and $s_i(l)$ denote the l-th bit of bit-vector s_i. Then a simple but quadratic bit-level encoding of the ADC over the s_i is as follows:

$$\mathrm{adc}(s_1, \ldots, s_m) \equiv \bigwedge_{i<j} s_i \neq s_j \equiv \bigwedge_{i<j} \bigvee_{l=1}^{n} s_i(l) \oplus s_j(l)$$

where $\oplus$ denotes XOR. However, it turns out that in actual applications not all inequalities $s_i \neq s_j$ are required. Often, a small subset of inequalities is sufficient to conclude unsatisfiability. A standard technique to take advantage of this observation is to encode these inequalities lazily, and not eagerly [9], [10]. An inequality encoding is only added if the SAT solver returns a satisfying assignment that violates this particular inequality. In this case the SAT solver is restarted and the process continues until either a consistent assignment is found, or the SAT solver concludes the formula to be unsatisfiable. This technique is similar to an abstraction refinement loop in CEGAR [11].

However, in the worst case still quadratic many inequalities in m need to be encoded. Additionally, checking consistency outside of the SAT solver is expensive in this classical refinement based approach [9].

Consistency of a satisfying assignment returned by the SAT solver with respect to an ADC can be checked by either sorting the bit-vectors by their assigned values, or by hashing bit-vectors with their assigned values as key. In any case this check is linear in $m \cdot n$. Furthermore, the SAT solver has to be restarted. Finally, inconsistency can only be determined after the SAT solver generated a *full* assignment.

III. Internal Consistency Checking of ADCs

Our main contribution is a new consistency checking algorithm. It handles ADCs inside the SAT solver, which is an instance of Satisfiability Modulo Theories SMT.

To handle an ADC the SAT solver just needs its list of bit-vectors, which in turn are represented as lists of literals. These bit-vectors are added through the API in the same way as clauses. The size of the internal representation of these bit-vectors is linear in m, the number of bit-vectors in the ADC, and not quadratic in m as in an eager encoding. Internally, these bit-vectors are copied into all different objects (ADOs),

978-1-4244-2735-2/08/$25.00 ©2008 IEEE

which contain a list of literals for the individual bits of the bit-vector, and a reference to the ADC they belong to.

One unassigned literal is watched in each ADO. This technique is similar to the two watching literal scheme of CHAFF [12], in which two unassigned literals per clause are watched. Whenever a watched literal becomes assigned, a new unassigned literal has to be found. If the search fails and all literals of an ADO are assigned, then the watched literal remains unchanged.

In the latter case all literals of the bit-vector are assigned. This concrete assignment to the bit-vector, represented by an ADO, is used as key to a hash table, associated with each ADC. In this table the ADOs of the ADC are stored. If another ADO with the same key is already in the hash table, an ADC inconsistency has been found. Otherwise, the fully instantiated ADO is entered into the hash table.

If a conflict occurs, a temporary clause of length $2 \cdot n$ is constructed. It contains the negation of all literals in the two bit-vectors, which have been assigned to equal values. This clause is a conflicting clause in the current assignment and is used as starting point for conflict analysis [13]. After a new learned clause is generated from this conflict, the temporary clause is discarded. Using a temporary clause avoids changing the procedure for analyzing the implication graph.

We should emphasize again, that neither the temporary clause is added to the CNF, nor a symbolic representation of the inequality of which this clause is an instance. Only clauses learned through conflict analysis starting from the temporary clause are added. We conjecture that similar clauses are learned as if inequalities are encoded symbolically.

During backtracking, ADOs are removed in reverse chronological order from the hash table. This can be implemented efficiently by saving the hash table position of the ADO together with the variable, whose assignment triggered the ADO to be added to the hash table. Whenever this variable becomes unassigned, the entry in the hash table is reset. Since entries to the ADO hash table are added and removed in a stack like fashion, also a hash table with open addressing can be used.

To simplify our implementation, we actually require that every variable occurs in at most one ADO. This restriction can always be enforced by adding copies of variables and enforcing equality through binary clauses. Nevertheless, the algorithm can be easily extended to variables resp. literals occurring in multiple ADOs and ADCs.

Updating watches for ADOs is cheaper than updating clauses of the corresponding eager encoding. Hashing the keys can be more costly, since it is linear in the bit-width n. The cost for entering and comparing keys can be ignored.

Watching two literals per ADO and changing the hash function slightly would also allow to derive forced assignments, which in the context of SMT is known as theory propagation [14]. However, this extension either requires to generate learned clauses for each such propagation, or major changes to the analysis function, which derives learned clauses. We leave the investigation of this extension as future work.

IV. Encoding

Previous work, where ADCs are encoded either eagerly or lazily *outside* of the SAT solver, requires to encode a bit-vector inequality $s \neq t$, where s and t are bit-vectors of width n. We present a compact CNF encoding which up to our knowledge has not been described in the literature yet.[1] In order to encode $s \neq t$ to CNF, we introduce n fresh variables d_k:

$$\bigwedge_{k=1}^{n-1} \left((s_k \vee t_k \vee \bar{d}_k) \ \wedge \ (\bar{s}_k \vee \bar{t}_k \vee \bar{d}_k) \right)$$

The idea of this encoding is as follows. The variables d_k represent that s and t differ at position k. If $s_k = t_k$, then d_k is forced to false. However, if $s_k \neq t_k$, then d_k is unconstrained and can be set to true. Finally, we add the following *linking clause*, which enforces s and t to differ in at least one position:

$$\bigvee_{k=0}^{n-1} d_k$$

This idea can be extended to encode combinations of bit-vector inequalities and equalities, which have to be added in incremental refinement loops. Consider the following example:

$$i = j \ \Rightarrow \ v = w$$

This formula is an instance of Ackermann constraints [15] and can be used to enforce function congruence on demand, where i and j are unary function arguments, and v and w the results. First, we introduce a fresh variable e:

$$(i = j \ \Rightarrow \ e) \ \wedge \ (e \ \Rightarrow \ v = w)$$

This formula is weaker and can be rewritten into

$$(i \neq j \ \vee \ e) \ \wedge \ (\bar{e} \ \vee \ v = w)$$

Let n_1 be the number of bits of i and j, and let n_2 be the number of bits of v and w. As before, we introduce n_1 fresh variables d_k and encode $i \neq j$ as follows:

$$\bigwedge_{k=0}^{n_1-1} \left((i_k \vee j_k \vee \bar{d}_k) \ \wedge \ (\bar{i}_k \vee \bar{j}_k \vee \bar{d}_k) \right)$$

To encode $v = w$ we add the following clauses:

$$\bigwedge_{k=0}^{n_2-1} \left((\bar{v}_k \vee w_k \vee \bar{e}) \ \wedge \ (v_k \vee \bar{w}_k \vee \bar{e}) \right)$$

Finally, we relate the two parts through a *linking clause*:

$$e \vee \bigvee_{k=0}^{n_1-1} d_k$$

The idea of this encoding is as follows. If $i \neq j$, then they differ in at least one bit. Therefore, one d_k can be set to true to satisfy the linking clause. The variable e is now unconstrained and can be set to false. Therefore, v and w do not have to

[1] In our experiments this encoding is only used for eager and lazy encodings outside of the SAT solver, but not for our improved method for handling ADCs inside of the SAT solver.

TABLE I
OVERALL RESULTS

	partial	*complete*	*solved*	*unsolved*	*inconcl*	*sat*	*unsat*	*time* 10^3sec	*space* GB	*steps*
mixed	n	y	259	85	38	182	39	96	23.2	9736
refine	n	y	250	94	32	179	39	101	23.0	9698
sadc	n	y	244	100	36	171	37	103	17.2	9131
eager	n	y	242	102	27	177	38	102	31.9	9438
mixed	y	y	258	86	40	179	39	98	23.3	9792
sadc	y	y	243	101	33	172	38	104	17.1	9066
none	y	n	267	77	56	179	32	87	16.6	10877
base	y	n	283	61	96	187	0	70	28.9	15187

be equal. However, if $i = j$, all the d_k are forced to $false$. In order to satisfy the linking clause, e has to be $true$, which forces v and w to be equal.

V. EXPERIMENTS

We have built a simple SAT based bounded model checker MCAIGER.[2] It reads AIGER format [16] and uses PICOSAT [17] as back end. We have implemented the consistency checking algorithm described above in PICOSAT. PICOSAT provides an incremental API similar to MINISAT [9]. ADCs can be extended incrementally, by adding new bit-vectors.

MCAIGER follows [2], [9] to validate or falsify simple safety properties. It incrementally checks a *base case* and an *induction step* for increasing bounds. If the base case becomes satisfiable, the bad state is reachable. If the SAT instance of the induction step turns out to be unsatisfiable, then the bad state is unreachable. Otherwise, the new time frame is added, unless a limit on the number of steps is reached.

States are encoded as bit-vectors. Adding a new time frame incrementally extends the ADC of the states by adding the new state, unless no simple path constraints are used. Different strategies for enforcing simple paths through ADCs are discussed below.

In our experiments we used all the 344 benchmarks[3] of the Hardware Model Checking Competition in 2008 (HWMCC'08), which can be considered to be a representative set of model checking benchmarks. The setup is almost the same as in HWMCC'08: 900 seconds time limit, 1.5 GB memory limit. However, we only checked base and inductive case up to a bound of 100 steps. Therefore a benchmark is considered to be *solved*, if either

1) after at most 100 steps the base case is *satisfiable* and thus the bad state reachable,
2) after at most 100 steps the inductive case is *unsatisfiable*, i.e. the bad state can not be reached, **or**
3) the bound of 100 steps is reached without conclusive answer on the reachability of the bad state (*inconclusive*).

Thus, a benchmark is marked *unsolved* if during checking the base or inductive case the time or space limit is reached. [4]

[2]Available at http://fmv.jku.at/mcaiger

[3]Available at http://fmv.jku.at/hwmcc08

[4]The memory limit was only reached in two runs, where only the base case was checked. In all other cases *unsolved* instances are due to time out.

Table I summarizes the results. In the first column model checking algorithms are listed. The second and third columns show whether the algorithm uses simple path constraints in the base case and whether the algorithm is complete. The next five columns contain number of *solved*, *unsolved*, *inconclusive*, *satisfiable* (bad state reachable) and *unsatisfiable* (bad state proven not to be reachable) instances. The sum of the run times in seconds and the sum of the maximum memory in GB follow in the next two columns. Time and space outs contribute 900 seconds. The last column denotes the number of *steps* the algorithm was able to reach over all benchmarks.

The rows are partitioned into three parts. The lower part contains **base** case only checking, which is plain BMC without checking the inductive case, and **none**, which both checks the base and the inductive case. Both methods do not use simple path constraints and are thus incomplete. They do not solve the same problem as the other methods, but give a limit on what can be gained resp. lost by adding simple path constraints.

The upper part lists our new method **sadc**, which uses symbolic all different constraints handled in the SAT solver. It also contains the classical quadratic **eager** encoding of simple path constraints. The **refine** method [9] adds individual state inequalities as lemmas on demand [10] incrementally. Finally, **mixed** uses symbolic all different constraints as long the number of all different conflicts is small. If during one SAT call the number of conflicts due to all different constraints reaches a certain limit (1000 conflicts in these experiments) or the overall number of such conflicts in all calls to the SAT solver is above another limit (10000), then the model checker switches to the **refine** method.

Finally, the API of our SAT solver has been extended to temporarily disable detection of conflicts due to symbolic simple path constraints. This is useful for checking the base case only. In the inductive case consistency of symbolic simple path constraints is always enabled for complete methods. For **mixed** and **sadc** approach we list two experiments where we *partially* disable simple path constraints this way.

Symbolically handling all different constraints as in **sadc** needs much less memory. In its plain form it handles less examples than the **refine** approach of [9], but more than classical **eager** encoding. However, dynamically switching from **sadc** to **refine** as in the **mixed** approach can solve the largest number of examples.

To understand this result it is instructive to compare **sadc** with **refine** in more detail. The scatter plot of Fig. 1 reveals that there a lot of examples where **sadc** is much faster than **refine**, but also vice versa. These scatter plots use a double logarithmic scale. A cross marks the run time for **refine** on the vertical axis and for **sadc** on the horizontal axis. Thus, the region above the main diagonal shows runs where **sadc** is faster, below where **refine** is faster. The time limit of 900 seconds corresponds to the two light and dotted vertical resp. horizontal lines. The other diagonal dashed lines correspond to a factor of 2, 10, and 100 difference in run time.

Therefore, a combined approached can be beneficial. Even our simple strategy in **mixed**, which starts with **sadc** and

Fig. 1. **sadc** vs **refine**

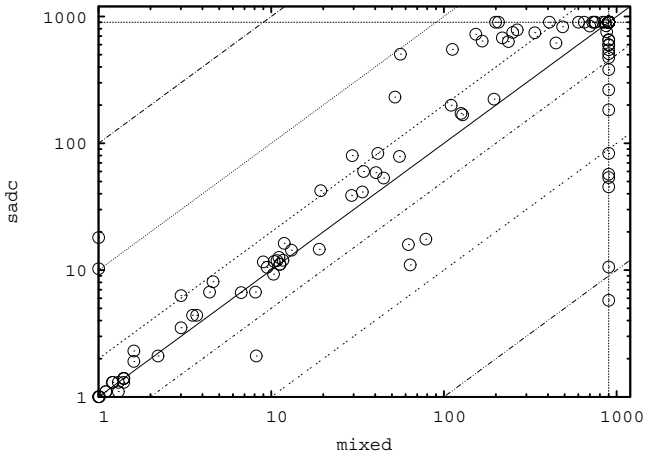


Fig. 2. **mixed** vs **refine**

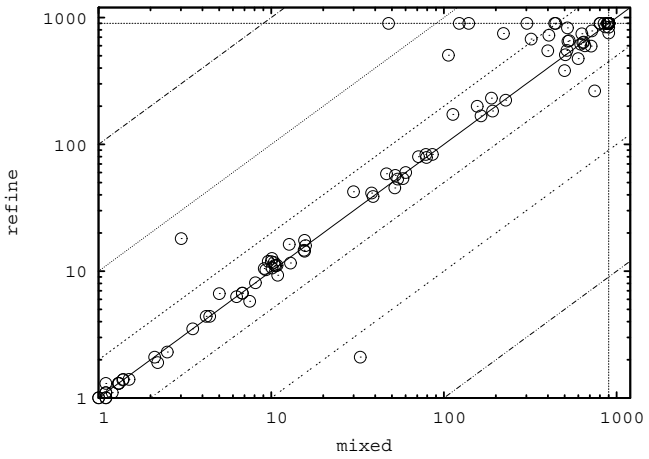
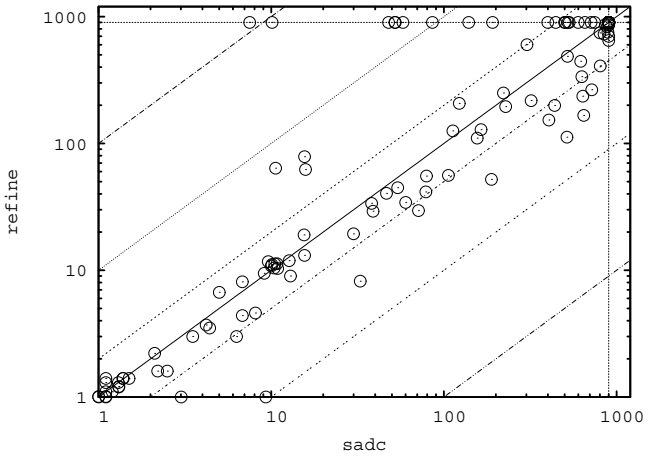


Fig. 3. **mixed** vs **sadc**



switches to **refine** as soon a conflict limit is reached seems to work as also the scatter plot of Fig. 2 and Fig. 3 show. The region above the main diagonal shows runs where **mixed** is faster than **refine** resp. **sadc**.

Tab. I also shows, that disabling simple path constraints temporarily in **sadc** or **refine** does not pay off. A more detailed analysis, not presented in this paper due to space constraints, reveals that the run-times are almost identical, except for a hand-full of benchmarks, where *partially* disabling simple path constraints hurts performance.

Finally, the reason that for some of the instances in the experiments, our new method performs worse than externally checking consistency, is most likely due the fact, that our current implementation does not propagate through ADCs.

VI. Conclusion

We have shown how all different constraints (ADCs) for bit-vectors can be handled inside a SAT solver symbolically. The technique does not require many changes to the SAT solver on the implementation side. Encouraging experimental results have been obtained.

So far we only check consistency of ADCs. In future work we want to use symbolic representations of ADCs for boolean constraint propagation (BCP) as well, which in the context of SMT is known as theory propagation [14]. We also think that it is worthwhile to apply similar techniques to other applications of equality logic over bit-vectors, such as encoding Ackerman constraints [15] for uninterpreted functions, or representing instances of McCarthy axioms for arrays [18].

References

[1] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu, "Symbolic model checking without BDDs," in *Proc. TACAS*, 1999.

[2] M. Sheeran, S. Singh, and G. Stålmarck, "Checking safety properties using induction and a SAT-solver," in *Proc. FMCAD*, 2000.

[3] N. Dershowitz, Z. Hanna, and J. Katz, "Bounded model checking with QBF," in *Proc. SAT*, 2005.

[4] T. Jussila and A. Biere, "Compressing BMC encodings with QBF," in *Proc. BMC*, 2006.

[5] H. Mangassarian, A. Veneris, S. Safarpour, M. Benedetti, and D. Smith, "A performance-driven QBF-based iterative logic array representation with applications to verification, debug and test," in *Proc. ICCAD*, 2007.

[6] J. Régin, "A filtering algorithm for constraints of difference in CSPs," in *Proc. AAAI*, 1994.

[7] J. Marques-Silva and I. Lynce, "Towards robust CNF encodings of cardinality constraints," in *Proc. CP*, 2007.

[8] R. Sebastiani, "Lazy satisfiability modulo theories," *JSAT*, vol. 3, 2007.

[9] N. Eén and N. Sörensson, "Temporal induction by incremental SAT solving," in *Proc. BMC*, 2003.

[10] L. de Moura and H. Rueß, "Lemmas on demand for satisfiability solvers," in *Proc. SAT*, 2002.

[11] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Counterexample-guided abstraction refinement for symbolic model checking," *J. ACM*, vol. 50, no. 5, 2003.

[12] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: Engineering an efficient SAT solver," in *Proc. DAC*, 2001.

[13] J. Marques-Silva and K. Sakallah, "GRASP: A Search Algorithm for Propositional Satisfiability," *IEEE T. on Computers*, vol. 48, no. 5, 1999.

[14] R. Nieuwnhuis and A. Oliveras, "DPLL(T) with exhaustive theory propagation and its applications to difference logic," in *Proc. CAV*, 2005.

[15] W. Ackermann, "Solvable cases of the decision problem," 1954.

[16] A. Biere, "AIGER And-Inverter Graph (AIG) format," fmv.jku.at/aiger.

[17] ——, "PicoSAT essentials," *JSAT*, 2008, submitted.

[18] J. McCarthy, "Towards a mathematical science of computation," in *Proc. IFIP Congress*, 1962.

Mechanized Information Flow Analysis through Inductive Assertions

Warren A. Hunt, Jr., Robert Bellarmine Krug, Sandip Ray, and William D. Young
Department of Computer Sciences
University of Texas at Austin
{hunt,rkrug,sandip,byoung} @cs.utexas.edu

***Abstract*— We present a method for verifying information flow properties of software programs using inductive assertions and theorem proving. Given a program annotated with information flow assertions at cutpoints, the method uses a theorem prover and operational semantics to generate and discharge verification conditions. This obviates the need to develop a verification condition generator (VCG) or a customized logic for information flow properties. The method is compositional: a subroutine needs to be analyzed once, rather than at each call site. The method is being mechanized in the ACL2 theorem prover, and we discuss initial results demonstrating its applicability.**

I. Introduction

Security of many critical computing systems depends on *information flow policies* that prohibit access to sensitive information without proper authorization. With the increasing application of software systems to secure applications, it is vital to ensure that a software implementation properly enforces information flow policies. The goal of this paper is to develop techniques for mechanized information flow analysis.

In its simplest form, modeling an information flow policy involves labeling certain program variables as classified (or high security), with the requirement that the value of an unclassified variable is not influenced by the initial values of any classified variable. Such a policy can be formalized by *noninterference* [1]. A deterministic program satisfies the policy if, from a pair of initial states differing only in classified variables, any pair of computations leads to final states with identical values for unclassified variables. Noninterference naturally generalizes to a lattice of security levels.

This paper proposes a method for verifying information flow properties of software programs through general-purpose theorem proving. Programs are formalized through an *operational semantics* of the underlying language defined by an interpreter that specifies the effect of executing instructions on the system state. Our approach uses *inductive assertions*. Given a program annotated with assertions at cutpoints, we derive *verification conditions* that ensure requisite information flow control, to be discharged with a theorem prover.

A key feature of our approach is that it obviates the need for implementing a custom verification condition generator (VCG) for information flow properties of the underlying language constructs. Instead, we show how to configure an off-the-shelf theorem prover to *mimic* a VCG through symbolic simulation of the operational model. The method is inspired by, and an extension of, our previous work [2] which showed how to prove functional correctness via symbolic simulation. The method is compositional; properties of subroutines can be verified individually rather than at each call site. We demonstrate the method by analyzing a small but illustrative program with the ACL2 theorem prover.

II. Basic Framework

We use operational semantics to model a program by its effects on the machine states. A state is a tuple of values of all machine variables—the program counter (pc), registers, memory, etc. The semantics is then given by a transition function $\mathit{next} : S \rightarrow S$ where S is the set of states: for a state s, $\mathit{next}(s)$ returns the state after executing one instruction from s. Executions are modeled by the function $\mathit{run} : S \times \mathbb{N} \rightarrow S$ which returns the state after n transitions from s.

$$\mathit{run}(s,n) \triangleq \begin{cases} s & \text{if } n = 0 \\ \mathit{run}(\mathit{next}(s), n-1) & \text{otherwise} \end{cases}$$

To illustrate how to formally specify information flow properties of programs using operational semantics, we consider the simple version of noninterference mentioned in Section I. Assume a partition of the variables into sets H (high) and L (low), corresponding to classified and unclassified data. Furthermore, assume that we have two predicates poise and exit on set S. For any state s, $\mathit{poise}(s)$ stipulates that s is poised to initiate execution of the program of interest: it specifies that the program is in the current call frame and the pc points to its first instruction. The predicate exit characterizes the termination states. To formalize the noninterference statement, we make use of the function esteps below, which returns, for any state s, the number of transitions to the first exit state reachable from s (if such an exit state exists).

$$\mathit{estpt}(s,i) \triangleq \begin{cases} i & \text{if } \mathit{exit}(s) \\ \mathit{estpt}(\mathit{next}(s), i+1) & \text{otherwise} \end{cases}$$
$$\mathit{esteps}(s) \triangleq \mathit{estpt}(s,0)$$

The definition of estpt is *partial*: its return value is unspecified if no exit state is reachable from s. Defining a recursive function generally requires a termination proof. However, since the definition is tail-recursive, it is admissible in theorem provers whose logics support Hilbert's choice operator [3].

A formalization of noninterference is shown in Fig. 1, and can be paraphrased as follows. "Let s and s' be any two states

978-1-4244-2735-2/08/$25.00 ©2008 IEEE

$\mathit{pre}(s, s') \triangleq \mathit{poise}(s) \wedge \mathit{poise}(s') \wedge (\bigwedge_{l \in L} l(s) = l(s'))$
$\mathit{post}(s, s') \triangleq (\bigwedge_{l \in L} l(s) = l(s'))$
$\mathit{nexte}(s) \triangleq \mathit{run}(s, \mathit{esteps}(s))$

Noninterference Condition:

$\mathit{pre}(s, s') \wedge \mathit{exit}(\mathit{run}(s, n))$
$\Rightarrow \mathit{exit}(\mathit{nexte}(s')) \wedge \mathit{post}(\mathit{nexte}(s), \mathit{nexte}(s'))$

Fig. 1. Formal Definition of Noninterference. Here $l(s)$ is assumed to be the value of variable l in state s.

$$\mathit{cstpt}(s, i) \triangleq \begin{cases} i & \text{if } \mathit{cut}(s) \\ \mathit{nextc}(\mathit{next}(s, i+1)) & \text{otherwise} \end{cases}$$

$$\mathit{cstps}(s) \triangleq \begin{cases} \mathit{cstpt}(s, 0) & \text{if } \mathit{cstpst}(s, 0) \in \mathbb{N} \\ \omega & \text{otherwise} \end{cases}$$

$\mathit{cut}(\mathrm{D}) \Leftrightarrow (\forall s : \mathit{cut}(s))$

$$\mathit{nextc}(s) \triangleq \begin{cases} \mathit{run}(s, \mathit{cstps}(s)) & \text{if } \mathit{cstps}(s) \in \mathbb{N} \\ \mathrm{D} & \text{otherwise} \end{cases}$$

Verification Conditions

1. $C(s, s') \Rightarrow \mathit{cut}(s) \wedge \mathit{cut}(s') \wedge (\mathit{exit}(s) \Leftrightarrow \mathit{exit}(s'))$
2. $\mathit{pre}(s, s') \Rightarrow C(s, s') \wedge \mathit{assert}(s, s')$
3. $\mathit{exit}(s) \Rightarrow \mathit{cut}(s)$
4. $C(s, s') \wedge \mathit{assert}(s, s') \wedge \neg \mathit{exit}(s) \wedge \mathit{exit}(\mathit{run}(s, n))$
   $\Rightarrow \mathit{assert}(\mathit{nextc}(\mathit{next}(s)), \mathit{nextc}(\mathit{next}(s')))$
5. $C(s, s') \wedge \mathit{assert}(s, s') \wedge \neg \mathit{exit}(s) \wedge \mathit{exit}(\mathit{run}(s, n))$
   $\Rightarrow C(\mathit{nextc}(\mathit{next}(s)), \mathit{nextc}(\mathit{next}(s')))$
6. $\mathit{assert}(s, s') \wedge \mathit{exit}(s) \wedge C(s, s') \Rightarrow \mathit{post}(s, s')$

Fig. 2. Verification conditions for information flow. Each verification condition is implicitly quantified over all free variables. Here ω is the first infinite ordinal. The function $\mathit{cstps}(s)$ returns the number of transitions to the closest cutpoint reachable from s if one exists, and ω otherwise. The reasons for returning ω when no cutpoint is reachable from s are technical, and not germane to this paper.

poised to execute the program, such that the variables in L have the same valuation in both s and s'. Suppose that there is an *exit* state reachable from s. Then the following conditions hold. (1) There is an *exit* state reachable from s'. (2) Let s_0 and s'_0 be the first *exit* states reachable from s and s' respectively; then s_0 and s'_0 have the same valuation for all variables in L."

Note that the statement of a practical information flow property might differ from the above. For instance, one might have a lattice of security levels. Or one might be interested only in a subset $L' \subseteq L$ at *exit*; then the conjunction in the definition of *post* would range over L'. Nevertheless, such concerns affect only the concrete definitions of *pre* and *post*; once they have been defined for the desired information flow requirements, the noninterference statement can be used as is.

Our approach is based on *inductive assertions* which involve annotating a program with assertions at cutpoints that include loop tests, program entry, and exit. For our purpose, the set of cutpoints is characterized by a predicate *cut* on S, commonly depending only on the pc. One then proves that whenever the program control reaches a cutpoint, the corresponding assertion holds. For functional correctness, this is achieved by a VCG as follows. A VCG crawls over an annotated program, generating verification conditions to be discharged by theorem proving. The guarantee from the VCG process is informally stated as follows. "Let p be any non-exit cutpoint satisfying the assertions. Let q be the next subsequent cutpoint. Then the assertions hold at q." It follows that the corresponding assertion holds whenever the program control reaches a cutpoint.

How do we extend the above for information flow properties? Since information flow is characterized by *pairs of states*, the assertions involved are formalized by a predicate *assert* over $S \times S$. The associated VCG guarantee is as follows. "Let p and p' be any two corresponding non-exit cutpoints of the program such that $\mathit{assert}(p, p')$ holds. (See below for an explanation of the role of "corresponding cutpoints.") Let q and q' be the next cutpoints from p and p' respectively. Then $\mathit{assert}(q, q')$ holds." Then, if additionally, (1) *assert* holds for the initial pair of states, and (2) for the pair of *exit* states *assert* implies *post*, it follows that the pair of *exit* states reachable from any pair of initial states satisfies *post*.

The formal verification conditions for information flow are shown in Fig. 2. Condition 4 formalizes the VCG guarantee. We now discuss the predicate C, which formalizes the notion of "corresponding cutpoints." Recall that the VCG guarantee specifies that when *assert* holds between a pair of cutpoints then it also holds for the subsequent pair. However, the next subsequent cutpoints might be out of sync. For instance, computation might exit from p and not from p'; thus the information flow theorem cannot be derived from the VCG guarantee. The predicate C eliminates this possibility by requiring the following: (1) if $\mathit{pre}(s, s')$ holds then s and s' must be corresponding; (2) for any two corresponding cutpoints, the subsequent cutpoint pair must be corresponding; and (3) if s satisfies *exit* and s' is a corresponding cutpoint, then s' must also satisfy *exit*. We assume that there is a binary predicate C on $S \times S$ characterizing the corresponding cutpoints; in practice, the definition of $C(s, s')$ will usually reduce to the condition that the pc values for s and s' are equal.

Conditions 4 and 5 involve multiple steps of program execution. Contrary to common practice, we discharge them *without a VCG* as follows. We prove the following two theorems, which are easy consequences of the definition of *nextc*:

SSR1: $\neg \mathit{cut}(s) \Rightarrow \mathit{nextc}(s) = \mathit{nextc}(\mathit{next}(s))$
SSR2: $\mathit{cut}(s) \Rightarrow \mathit{nextc}(s) = s$

We treat SSR1 and SSR2 as oriented conditional equations or *rewrite rules*. For any symbolic state s, the rules rewrite the term $\mathit{nextc}(s)$ to either s or $\mathit{nextc}(\mathit{next}(s))$; in the latter case the definition of *next* is symbolically expanded, simplified, and the rules applied again. The proof attempt causes the theorem prover to symbolically simulate the program from a cutpoint until the next cutpoint is reached; the process mimics a forward VCG. For symbolic simulation to terminate, each program loop must contain a cutpoint, as with traditional VCGs.

We briefly remark on how to automate the method above in ACL2. The derivation of noninterference from Conditions 1–6 is independent of the definitions of *pre*, *post*, etc. This allows the development of a *proof template* for generating the

```
tricky1 (int high, low, n) {
   int temp = low;
   for i = 0 to n do {
     if even(i) {
       out = out + temp;
       temp = high;
     } else {
       temp = low;
     }
   }
   out = out + 7;
   return out;
 }
```

Fig. 3. Pseudo-code for the Tricky Program. Variable `out` is incremented by `temp` only when `i` is even, and `temp` is equal to `low` in that case. Thus, the final value of `out` is independent of `high`.

verification conditions as follows. First, we introduce functions *pre*, *post*, *cut*, *assert*, *C*, and *next* constrained to satisfy Conditions 1–6, and prove the noninterference theorem for these constrained functions.[1] We can then implement an ACL2 macro that automates the information flow proofs as follows:

- Mechanically generate the concrete version of *nextc*.
- Establish conditions 1–6 for the concrete versions of *pre*, *post*, etc., using symbolic simulation.
- Derive the information flow theorem by functionally instantiating the generic version.

We have developed a corresponding proof template for (partial and total) functional correctness [2]. We are working on extending the template for information flow properties.

III. A "Tricky" Example

Our approach, although extremely simple, nevertheless provides a scalable framework for information flow analysis. One key strength is the ability to use expressive predicates for proving information flow theorems: if an information flow property depends on functional invariants of the system state, then assertions can easily account for such invariants. This is in stark contrast to traditional security type systems for information flow verification, which depend on the syntactic analysis of the program to deduce information flow [4].

As an illustration, consider the program shown in Fig. 3. The information flow specification for the program is that the final value of `out` depends only on the initial values of `low` and `n`. The program is adapted from one in a recent paper by Amtoft and Banerjee [5][2] which, although simple, was motivated by an actual program used in operational verification of hardware amplifiers provided by Rockwell Collins. The information flow property depends on a key observation: whenever the value of `out` is incremented by `temp`, the value of `i` is even and the value of `temp` is equal to `low`; thus the final value of `out` is dependent only on `low` (and the loop count `n`). The property cannot be inferred by type reasoning which would infer dependence of `temp` on `high` and `out` on `temp`.

We formalized the program through an operational semantics of a simple machine model, and proved the information flow specification using inductive assertions. The precondition stipulates that s and s' are poised to execute the program and the values of `low`, `n`, and `out` are the same in both states; the postcondition specifies that the value of `out` is the same after exiting the program. The only "creative" assertion is in the loop invariant. In addition to the boiler-plate assertion that the values of `low`, `n`, `out`, and `i` are the same in s and s', we need the condition that if `i` is even s and s' have the same value of `temp`. With this assertion, the verification conditions shown in Fig. 2 are easily verified through symbolic simulation.

It is instructive to compare our approach with that of Amtoft and Banerjee [5]. Their approach is built around the axiomatic semantics for a special logical construct $\bowtie$ stipulating *agreement assertions*: for a variable x, two states p and q satisfy $x \bowtie$ if and only if $x(p) = x(q)$. They develop axiomatic semantics for specifying loop flow and object flow using $\bowtie$, and a VCG for the semantics. In contrast, we generate and discharge the verification conditions directly through symbolic simulation of the operational semantics. Nevertheless, our approach requires no more creative insight than theirs, namely manually constructing the loop invariant above. On the other hand, our approach can harness the power of a general-purpose theorem prover for symbolic simulation and requires no axiomatic semantics for information flow.

IV. Compositionality

The above treatment did not consider compositionality. Consider verifying a program P that invokes a subroutine Q. Symbolic simulation from a cutpoint of P might encounter an invocation of Q, resulting in simulation of Q. We prefer to separately verify Q, and use the result for verifying P.

To achieve composition, we must handle the *frame conditions* to justify that P can continue execution after Q returns. Note that an information flow property of Q is not sufficient for this; for instance, we must show that the execution of Q does not corrupt the call stack. For functional correctness, the frame problem is addressed by phrasing the postcondition as an equality to characterize how each state component is modified by Q [2]. However, a full characterization of Q is often irrelevant to the information flow of P. The challenge is to effectively augment the postcondition of Q with frame conditions to facilitate symbolic simulation of P.

How do we address the challenge? Let V be the set of state components governing the control flow on return from Q; typically V includes the call stack. Then we define $\mathit{modify}_Q(s)$ to update each component of s as follows. For each component $v \in V$, $\mathit{modify}_Q(s)$ updates v by precisely characterizing its modification on exit from Q. The update to the call stack is characterized by popping the current call frame. For $c \notin V$, the update is simply $c(\mathit{nexte}_Q(s))$, which

[1] This proof has been completed with ACL2 at the time of this writing.

[2] The difference between Amtoft and Banerjee's program and that shown in Fig. 3 is that the former involved 7 iterations of `i` in the loop while we use `n` iterations. Note that if the number of loop iterations is a constant then the loop can be unrolled by symbolic simulation, obviating loop invariants.

```
int out;
main (int high, low, n, flag) {
  out = 0;
  if flag < 0 {
    tricky1(high, low, n);
  } else {
    tricky3(high, low, n);
  }
  out = out - 1;
}
```

Fig. 4. A Program to demonstrate compositionality

may or may not need to be characterized depending on the caller. We then prove the following two conditions.

1) $\mathit{poise}_Q(s) \wedge \mathit{exit}_Q(\mathit{run}(s,n)) \Rightarrow \mathit{nexte}_Q(s) = \mathit{modify}_Q(s)$
2) $\mathit{pre}_Q(s,s') \wedge \mathit{exit}_Q(\mathit{run}(s,n)) \Rightarrow \mathit{post}_Q(\mathit{modify}_Q(s), \mathit{modify}_Q(s'))$

Here, $\mathit{poise}_Q(s)$ states that s is poised to invoke Q. Its definition is derived from $\mathit{pre}_Q(s,s')$ by collecting the conjuncts that only mention s. Condition 2 follows from 1 and the information flow property of Q. We prove 1 through inductive assertions, viewing modify_Q as a functional characterization of Q [2]. The necessary assertions can be culled from the information flow proof of Q. Recall that the predicate $\mathit{assert}_Q(s,s')$ is strong enough to characterize the flow of control from each cutpoint of s (and s') to the next. Thus, the conjuncts in $\mathit{assert}_Q(s,s')$ that only involve s can be used for symbolic simulation from each cutpoint in s to prove 1. The theorems facilitate compositional reasoning. If states s and s' encountered during symbolic simulation of P are poised to execute Q, then 1 permits simulation to "skip past" Q, and 2 enables us to assume *post* on the generated pair $(\mathit{modify}_Q(s), \mathit{modify}_Q(s'))$ during subsequent simulation. Note that this can be automated using macros as hinted at in Section II for the basic framework.

We used the approach above to compositionally verify the program shown in Fig. 4. The program is artificial but illustrative. It invokes one of two separate versions of the `tricky` procedure depending on `flag`: `tricky1` is as shown in Fig. 3; `tricky3` iterates `3n` times instead of `n`. Our information flow specification is that the final value of `out` on *exit* from `main` is independent of `high`. Note that the information flow analysis of `tricky3` is exactly analogous to `tricky1`, but its return value is different. Thus, a complete functional characterization of the two routines would involve separate analysis. However, the actual return value of the subroutines is immaterial to the information flow of `main`. With our approach, `main` can be verified using only the noninterference of each subroutine and the frame conditions.

V. Related Work and Conclusion

Information flow analysis was formulated by Denning and Denning [6]. Sabelfeld and Meyers [4] contains a comprehensive survey of the area. Traditional approaches to information flow analysis involves *security type systems* [7], [8]: program variables and expressions are annotated with security levels, and flow of information is controlled by typing rules. There has also been significant recent work on axiomatic semantics for information flow. Clark *et al.* [9] develop a semantics for Idealized Algol. Joshi and Leino [10] develop a weakest precondition calculus for information flow. Darvas *et al.* [11] use dynamic logic to express information flow for Javacard.

Our work provides the first framework for information flow analysis through inductive assertions directly on operational semantics. No separate VCG or axiomatic semantics for information flow is necessary. Instead, the generation and discharge of verification conditions are handled by the theorem prover through symbolic simulation. Furthermore, we can compose information flow properties of subroutines without requiring full characterization of their functional specification. The framework is in an early stage of development. As mentioned in Section II, we are developing proof templates to facilitate the automation of information flow verification. Some planned future enhancements include (1) automated static analysis of data structure shapes, (2) extension to multithreaded programs, and (3) analysis of dynamic and declassification policies.

Acknowledgements: This material is based upon work supported by DARPA and the National Science Foundation under Grant No. CNS-0429591. Matt Kaufmann provided numerous comments and suggestions in course of this work.

References

[1] J. Goguen and J. Meseguer, "Security policies and security models," in *Proc. 1982 IEEE Symposium on Security and Privacy*, April 1982, pp. 11–20.

[2] J. Matthews, J. S. Moore, S. Ray, and D. Vroon, "Verification Condition Generation Via Theorem Proving," in *Proceedings of the 13th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR 2006)*, ser. LNCS, M. Hermann and A. Voronkov, Eds., vol. 4246, Nov. 2006, pp. 362–376.

[3] P. Manolios and J. S. Moore, "Partial Functions in ACL2," *Journal of Automated Reasoning*, vol. 31, no. 2, pp. 107–127, 2003.

[4] A. Sabelfeld and A. C. Myers, "Language-Based Information-Flow Security," *IEEE Journal on Selected Areas of Communication*, vol. 21, no. 3, Jan. 2003.

[5] T. Amtoft and A. Banerjee, "Verification Condition Generation for Conditional Information Flow," in *Proceedings of the 2007 ACM workshop on Formal methods in security engineering (FMSE 2007)*, P. Ning, V. D. G. V. Atluri, and H. Mantel, Eds. ACM Press, Nov. 2007, pp. 2–11.

[6] D. Denning and P. Denning, "Certification of Programs for Secure Information Flow," *Communications of the ACM*, vol. 20, no. 7, pp. 504–513, 1977.

[7] P. Orbaek and J. Palsberg, "Trust in the λ-calculus," *Journal of Functional Programming*, vol. 7, no. 6, pp. 557–591, 1997.

[8] D. Volpano and G. Smith, "A Type-Based Approach to Program Security," in *Proceedings of Theory and Practice of Software Development (TAPSOFT 1997)*, ser. LNCS. Springer-Verlag, 1997, pp. 607–621.

[9] D. Clark, C. Hankin, and S. Hunt, "Information Flow Analysis for Algol-like Languages," *Computer Languages*, vol. 28, no. 1, pp. 3–28, 2002.

[10] R. Joshi and K. R. M. Leino, "A Semantic Approach to Information Flow," *Science of Computer Programming*, vol. 37, pp. 113–138, 2000.

[11] A. Darvas, R. Hähnle, and D. Sands, "A Theorem Proving Approach to Analysis of Secure Information Flow," in *Proceedings of the 2nd International Conference Security in Pervasive Computing (SPC 2005)*, ser. LNCS. Springer, 2005, pp. 193–209.