Formal Methods in Computer Aided Design

Lugano, Switzerland • 20-23 October, 2010

Edited by Roderick Bloem and Natasha Sharygina

In cooperation with ACM's SIGPLAN (Special Interest Group on Programming Languages) and SIGSOFT (Special Interest Group on Software Engineering)

Technical Sponsorship from IEEE Council on Electronic Design Automation (CEDA)

2010 Formal Methods in Computer Aided Design

Preface

FMCAD 2010, held in Lugano Switzerland on October 20-23, was the tenth in a series of conferences on the theory and applications of formal methods in hardware and system verification. FMCAD provides a leading forum to researchers in academia and industry for presenting and discussing groundbreaking methods, technologies, theoretical results, and tools for reasoning formally about computing systems. FMCAD covers formal aspects of computer-aided system design including verification, specification, synthesis, and testing.

In the past, FMCAD was held in the United States on even years and its sister conference, CHARME, was held in Europe on odd years. The conferences merged in 2006 and FMCAD was held in the USA from 2006 to 2009. FMCAS was held in Europe for the first time this year.

For the first time, FMCAD10 featured an industrial track, dedicated to industry users of formal methods. We are very pleased at the interest shown in this track and the high quality of submissions we received, some from purely industrial teams and some from mixed industrial-academic teams, all describing results on real-life designs. To encourage professionals, we also accepted decks of slides. One of these, on property based formal methods for DFT logic verification, was accepted—work that otherwise might very well have passed us by. We are very grateful to Cindy Eiser (IBM Research - Haifa) and Wolgang Ecker (Infineon) for organizing the track and the accompanying exhibition.

We received 86 papers overall: 66 in the research track and 18 in the industrial track. We accepted 25 research and 7 industrial papers papers, of which 27 were long and 5 were short. The conference covered a wide range of formal topics, including model checking and theorem proving, verification from the arithmetic to the system level, synthesis from specifications and case studies.

We heard invited talks from Turing Award winner Joseph Sifakis (CNRS/Verimag, Schneider-INRIA Endowed Researcher Chair) titled *Embedded Systems Design: Scientific Challenges and Work Directions* and from Viresh Paruthi (IBM Austin) titled *Large-scale Formal Application: From Fiction to Fact.*

On the 20 October, we had tutorials organized by Helmut Veith (Vienna University of Technology) from Sumit Gulwani (Microsoft) on *Dimensions in Program Synthesis*, from Ken McMillan (Cadence) on *Invariant Generation*, from Warren A. Hunt (UT Austin) on *Verification of the VIA (Centaur) Nano Microprocessor using the ACL2 Theorem-Proving System*, and from Jin Yang (Intel) on *Post Silicon Verification.*

FMCAD 2010 also included a panel, organized by Tom Melham (Oxford) on

The Verification Challenge of Low-Level Embedded Software.

We sincerely thank the sponsors of FMCAD for their generous contributions: Centaur Technology, the HiPEAC Network of Excellence, IBM, Intel, the City of Lugano, Microsoft Research, NEC, and the University of Lugano.

FMCAD has in-cooperation status with ACM SIGPLAN and SIGSOFT and technical co-sponsorship with IEEE CEDA. FMCAD is committed to making the proceedings of FMCAD10 as available as possible. To that end, FMCAD will make the proceedings available online for free. In addition, the proceedings will be published in the ACM and IEEE digital libraries.

We would like to thank the organizing committee. Besides the people mentioned above, the committee includes Hana Chockler (publications), Jason Baumgartner (sponsoring) and Umberto Bondi, Daniela Dimitrova, Elisa Larghi, Francesco Regazzoni, and Mariagiovanna Sami, (local arrangements). The organizing committee provided invaluable help in organizing FMCAD 2010. We would also like to thank the steering committee (Jason Baumgartner, Aarti Gupta, Warren A. Hunt, Jr., Panagiotis Manolios, and Mary Sheeran) for their invaluable advice.

Most of all we would like to thank the FMCAD program committee for both tracks for their excellent work reviewing and discussing the submissions. Without them, FMCAD would not have achieved the quality that it has. Of course, the conference could not exist without authors choosing to submit their work to critical scrutiny at FMCAD.

Natasha Sharygina and Roderick Bloem (chairs)

Conference Organization

Program Chairs

Roderick Bloem, *Graz University of Technology, Austria*
Natasha Sharygina, *University of Lugano, Switzerland and CMU, USA*

Industrial Track Chairs

Wolfgang Ecker, *Infineon, Germany*
Cindy Eisner, *IBM Haifa Research Laboratory*

Tutorial Chair

Helmut Veith, *TU Wien*

Publication Chair

Hana Chockler, *IBM Haifa Research Laboratory*

Panel Chair

Tom Melham, *University of Oxford, UK*

Steering Committee

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

Local Arrangements

Umberto Bondi, *University of Lugano*
Daniela Dimitrova, *University of Lugano*
Elisa Larghi, *University of Lugano*
Francesco Regazzoni, *USI Lugano*, and *UCL Louvain-la-Neuve (Belgium)*
Mariagiovanna Sami, *University of Lugano*

Programme Committee

Technical Program:

Jason Baumgartner, *IBM, USA*
Valeria Bertacco, *University of Michigan, USA*
Armin Biere, *Johannes Kepler University, Austria*
Per Bjesse, *Synopsys, Inc., USA*
Roderick Bloem, *Graz University of Technology, Austria*
Doron Bustan, *Intel, Israel*
Gianpiero Cabodi, *Politecnico di Torino, Italy*
Alessandro Cimatti, *IRST, Italy*
Koen Claessen, *Chalmers University of Technology, Sweden*
Ganesh Gopalakrishnan, *University of Utah, USA*
Aarti Gupta, *NEC Labs, USA*
Ziyad Hanna, *Jasper Design Automation, USA*
Alan Hu, *University of British Columbia, Canada*
Jie-Hong Roland Jiang, *National Taiwan University, Taiwan*
Barbara Jobstmann, *CNRS-Verimag, France*
Vineet Kahlon, *NEC, USA*
Gerwin Klein, *NICTA, Australia*
Daniel Kroening, *Oxford University Computing Laboratory, UK*
Thomas Kropf, *University of Tübingen, Germany*
Marta Kwiatkowska, *Oxford University Computing Laboratory, UK*
Oded Maler, *CNRS-Verimag, France*
Panagiotis Manolios, *Northeastern University, USA*
Ken McMillan, *Cadence Berkeley Labs, USA*
Tom Melham, *Oxford University Computing Laboratory, UK*
John O'Leary, *Intel, USA*
Lee Pike, *Galois, Inc., USA*
Marc Pouzet, *University of Paris-Sud, France*
Julien Schmaltz, *Open University of The Netherlands, Netherlands*
Natarajan Shankar, *Computer Science Laboratory, SRI, USA*
Natasha Sharygina, *University of Lugano, Switzerland and CMU, USA*
Satnam Singh, *Microsoft Research*
Anna Slobodova, *Centaur Technology, USA*
Sofiene Tahar, *Concordia University, Canada*
Helmut Veith, *Vienna University of Technology, Austria*
Karen Yorav, *IBM Haifa Research Laboratory, Israel*

Industrial track:

Pranav Ashar, *RealIntent, USA*
Lyes Benalycherif, *ST-Ericsson, France*
Per Bjesse, *Synopsys, Inc., USA*
Holger Busch, *Infineon, Germany*

Joachim Gerlach, *Albstadt-Sigmaringen University, Germany*
Ziyad Hanna, *Jasper Design Automation, USA*
Christian Jacobi, *IBM, Germany*
Peter Jensen, *SyoSil Aps, Denmark*
Andreas Kuehlmann, *Cadence Berkeley Labs, USA*
Ajeetha Kumari, *CVC Pvt Ltd, India*
Viresh Paruthi, *IBM, USA*
Axel Scherer, *Cadence Design Systems. Inc, USA*
Michael Siegel, *OneSpin Solutions, Germany*

External Reviewers

Naeem Abbasi, Rawan Abdel-Khalek, Sa'ed Abed, Oshri Adler, Behzad Akbarpour, Holzer Andreas, Gadiel Auerbach, Shoham Ben-David, Jesse Bingham, Magnus Bjork, Nicolas Blanc, Sascha Böhme, Ahmed Bouajjani, Marius Bozga, Marco Bozzano, Sebastian Burckhardt, Michael Case, Paul Caspi, Harsh Raju Chamarthi, Debapriya Chatterjee, Wei-Fan Chiang, Chris Chilton, Hana Chockler, David Cock, Claudionor Coelho Jr., Scott Cotton, William Denman, Andrew DeOrio, Flavio M. De Paula, Alastair Donaldson, Vijay D'silva, Ashvin Dsouza, Bruno Dutertre, Wolfgang Ecker, Kai Engelhardt, Levent Erkök, Ranan Fraer, Amjad Gawanmeh, Steven German, Amit Goel, Eugene Goldberg, Joseph Greathouse, David Greenaway, Andreas Griesmayer, Alberto Griggio, Ali Habibi, Leopold Haller, Hakan Hjort, Andreas Holzer, Wei-Lun Hung, Norris Ip, Alexander Ivrii, Himanshu Jain, Geert Janssen, Matti Järvisalo, Susmit Jha, Ian Johnson, Krishnan Kailas, Alexander Kaiser, Mark Kattenbelt, Zurab Khasidashvili, Johannes Kinder, Udo Krautz, Smita Krishnaswamy, Sumit Kumar Jha, Sudipta Kundu, Julien Legriel, Rebekah Leslie, Djones Lettnin, Guodong Li, Michael Lifshits, Lars Lundgren, Louis Mandel, Johan Martensson, John Matthews, Stephan Merz, Alan Mishchenko, David Monniaux, Hari Mony, In-Ho Moon, John Moondanos, Shiri Moran, Cesar Munoz, Iman Narasamdya, Rajeev Narayanan, Ziv Nevo, Dejan Nickovic, Sergio Nocco, Michael Norrish, Steven Obua, Ritesh Parikh, David Parker, Grant Olney Passmore, Andrea Pellegrini, Florence Plateau, Hongyang Qu, Stefano Quer, Kairong Quian, Sivan Rabinovitch, Silvio Ranise, Rajeev Ranjan, Pascal Raymond, Marco Roveri, Philipp Ruemmer, Georgia Safe, Bastian Schlich, Viktor Schuppan, Subodh Sharma, Nishant Sinha, Sebastian Skalberg, Jeremy Sproston, Stefan Staber, Muralidhar Talupur, Murali Talupur, Michael Tautschnig, Andrei Tchaltsev, Ashish Tiwari, Aaron Tomb, Stefano Tonetta, Rob van Glabbeek, Tatyana Veksler, Srinivasan Venkataramanan, Freek Verbeek, Anh Vo, Kai Weber, Georg Weissenbacher, Tobias Welp, Andy Yu, Florian Zuleger.

Table of Contents

Dimensions in Program Synthesis

(Tutorial)

Sumit Gulwani
Microsoft Research
Redmond, WA, 98052
Email: sumitg@microsoft.com

Abstract

Program Synthesis, which is the task of discovering programs that realize user intent, can be useful in several scenarios: discovery of new algorithms, helping regular programmers automatically discover tricky/mundane programming details, enabling people with no programming background to develop scripts for performing repetitive tasks (end-user programming), and even problem solving in the context of automating teaching.

In this tutorial, I will describe the three key dimensions that should be taken into account in designing any program synthesis system: expression of user intent, space of programs over which to search, and the search technique [1]. (i) The user intent can be expressed in the form of logical relations between inputs and outputs, input-output examples, demonstrations, natural language, and inefficient or related programs. (ii) The search space can be over imperative or functional programs (with possible restrictions on the control structure or the operator set), or over restricted models of computations such as regular/context-free grammars/transducers, or succinct logical representations. (iii) The search technique can be based on exhaustive search, version space algebras, machine learning techniques (such as belief propagation or genetic programming), or logical reasoning techniques based on SAT/SMT solvers.

I will illustrate these concepts by brief description of various program synthesis projects that target synthesis of a wide variety of programs such as standard undergraduate textbook algorithms (e.g., sorting, dynamic programming), program inverses (e.g., decoders, deserializers), bitvector manipulation routines, deobfuscated programs, graph algorithms, text-manipulating routines, geometry algorithms etc.

REFERENCES

[1] S. Gulwani. Dimensions in program synthesis. In *ACM Symposium on PPDP*, 2010.

Verifying VIA Nano Microprocessor Components

Warren A. Hunt, Jr.
Centaur Technology, Inc.
7600-C North Capital of Texas Hwy, Suite 300
Austin, Texas 78731-1180
Email: hunt@centtech.com

***Abstract*—We verify parts of the VIA Nano X86-compatible microprocessor design using the ACL2 theorem-proving system. We translate Nano RTL Verilog into the `EMOD` hardware description language. We specify properties of the Nano in the ACL2 logic and use a combination of theorem-proving and automated techniques to verify the correctness of Nano design elements.**

I. Introduction

We have specified and verified parts of VIA's X86-compatible Nano microprocessor using the ACL2 theorem-proving system. The VIA Nano microprocessor is a full X86-64 design, including VMX, AES, DES, and SHA instructions. The current Nano is implemented in a 40-nanometer process with around 100 million transistors. The Nano design contains a security co-processor; it runs over 40 different operating system (such as Windows, MacOS, Linux, FreeBSD); and it supports four different virtual-machine implementations. The RTL Nano specification is written in Verilog, which we translate into our formalized `EMOD` hardware description language (HDL). We use this `EMOD` representation both as a specification for transistor-level circuit elements and as an implementation for more abstract properties.

The design for the Nano is represented with 570,000 lines of Verilog. This is a hierarchical description that includes specifications for all Nano circuit elements, and it can be simulated using a Verilog simulator. To verify parts of the Nano design, we first translate modules of interest into the `EMOD` formal hardware description language, which is embedded it within the ACL2 logic. We use the ACL2 logic to specify the operation of Nano hardware elements. Finally, we use the ACL2 theorem-proving system to verify the correctness of Nano design elements.

Our verification efforts have been focused on the media and floating-point units. The Nano media unit can add/subtract four pairs of floating-point numbers every clock cycle with a two-cycle latency. Depending on the size of the operands, the Nano multiplier can multiply one, two, or four pairs of operands every clock cycle with a three- or four-cycle latency. The Nano divider is implemented with a special 4-bit divider unit augmented with a microcode program.

We have verified hardware the add, subtract, multiply, divide (microcode only), compare, convert, logical, shuffle, blend, insert, extract, and min-max instructions [8]. To verify Nano components, we symbolically simulate design fragments and compare the results to specifications written in ACL2. Separately, we also verify that our ACL2 specifications implement various floating-point operations.

In this paper, we describe some of the models used by VIA to implement the Nano. We have formalized subsets of several models, and we are working to formalize the entire Nano design. The Nano is continuously updated and extended; for example, this last year the Nano was extended with 256-bit SSE instructions. The Nano will soon be offered as a multi-core, which required an internal rearrangement of many design elements. As the design is altered, we re-run our evolving set of formal verification scripts to ensure that the latest design continues to satisfy the properties we have been able to formally specify and mechanically verify. Thus, we must be able to very quickly translate existing design representations into a form suitable for our tool suite.

II. The Centaur FV Toolflow

Nano circuits are initially represented in Verilog. We translate the Nano Verilog model into our `EMOD` hardware description language; and we analyze the result of such translations by comparing them to specification functions. The relationships between these various models are shown in Figure 1.

Starting in the upper-left-hand-corner of the diagram is the `Nano ''Golden'' Model`; this is a C-language program that is used as a specification for the `VIA Nano Verilog`. The operation of the `VIA Nano Verilog` is compared to the specification using co-simulation; both models are simulated, and after each (clock cycle) step, register and memory values are compared. This procedure is the primary pre-silicon verification approach for ensuring that the Nano satisfies its specification. Once functional silicon Nano processors are available, this same kind of co-simulation is done but the actual Nano microprocessor is used in place of the Nano Verilog; results are still compared to the `Nano ''Golden'' Model`. Of course, once working microprocessors are available, they are also installed in computing systems and subjected to a wide variety of tests; the results of these tests are compared to known-good results.

We use formal verification to augment the already extensive simulation being performed. Formal verification has found errors not detected during testing and in commercial usage that are generally very subtle. Of course, if such bugs were easy to find, they would have been uncovered by simulation. Our formal verification process begins by translating the Nano

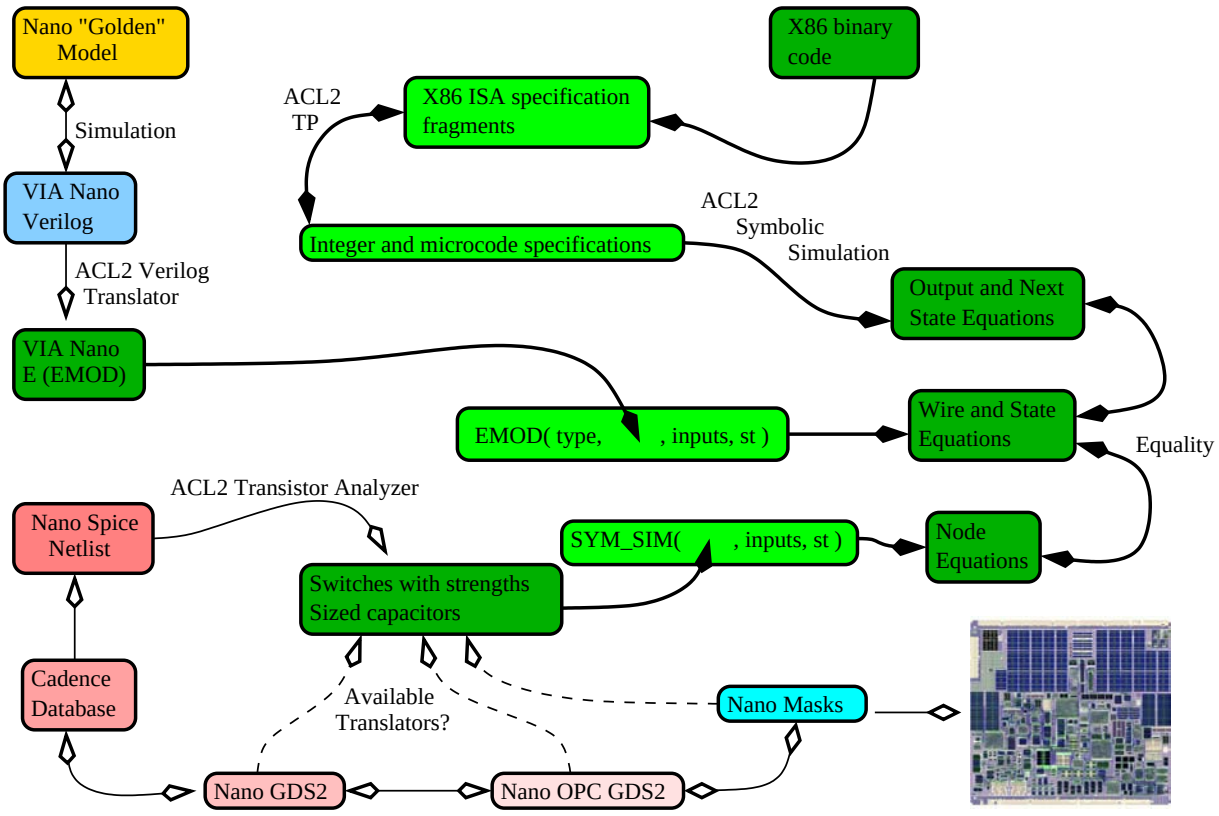


Fig. 1. Verification relationships between the models

Verilog into the `EMOD` HDL, and then symbolically simulating it to get ``Wire and State Equations`. For more complex specifications, we generally write ACL2 code that is designed to mimic the behavior of the source Verilog, and then compare the results produced. In some cases, we compare our `Integer and microcode specifications` to even more abstract specifications, such as when we verified the Nano media-unit [8] instructions.

In addition to the kind of Verilog verification so far discussed, we also verify transistor-level circuit implementations. A large part of the Nano implementation is custom-designed, transistor-level circuits. In fact, almost all of the Nano design is full custom, except for a number of auto-place-and-route blocks that primarily implement control logic. As show in Figure 1, we verify transistor-level models by translating their Spice-level circuit representations into a `Switches with strengths -- Sized capacitors` form. Using `SYM_SIM`, we symbolically simulate the resulting circuit models and compare the resulting `Node Equations` values with the expected `Wire and State Equations` results.

At a high level, our current verification efforts could be described as co-simulation with symbolic test vectors. Boolean data is represented with Boolean variables instead of with specific Boolean values. In some cases, such as exist in the execution cluster, the specification of correctness is relatively straight forward, although voluminous and detailed. In other cases, such as with the bus interface, the specification is much more complex and subtle because of the very large number interactions with other units. Before we attempt to explain our use of `EMOD`, we provide a simple embedding example.

III. A Simple Embedded Language

We now illustrate the embedding of a very simple language within the ACL2 logic. This language, based on IF trees, is defined by two functions: a recognizer (the permitted syntax) for IF expressions and an evaluator (the semantics) for IF expressions.

Here are some syntactically, well-formed examples in language. Note that these expressions are “quoted”; that is, they are ACL2 (and Lisp) data constants.

```
'(IF c a b)        '(IF 1 2 3)

'(IF r (IF c T NIL) q)
```

We can check whether these forms are indeed acceptable using our syntactic recognizer function `if-termp`, which takes a single argument and recognizes whether this argument is a valid IF-expression. If `term` is an `atom` then it must be recognized by the `eqlablep` predicate, which recognizes atoms that are numbers, symbols, or characters. Otherwise, this predicate requires an object of the form `(IF a b c)`, where the argument recursively recognized by `if-termp`.

```
(defun if-termp (term)
   (if (atom term)
       (eqlablep term)
     (let ((fn (car term))
           (args (cdr term)))
       (and (consp args)
            (consp (cdr args))
            (consp (cddr args))
            (null  (cdddr args))
            (eql fn 'if)
            (if-termp (car args))
            (if-termp (cadr args))
            (if-termp (caddr args))))))
```

The function `if-evl` evaluates the `term` argument, recognized by `if-termp`, using assignments of values to variables as given in `alist`.

```
(defun if-evl (term alist)
   (if (atom term)
       (cdr (assoc term alist))
     (if (if-evl (cadr term) alist)
         (if-evl (caddr term) alist)
       (if-evl (cadddr term) alist))))
```

For instance, by binding the variables `1`, `2`, and `3` to themselves, we get:

```
(IF-EVL '(IF 1 2 3)
        '((1 . 1) (2 . 2) (3 . 3)))
  ==>
2
```

Given these two functions, we have defined the syntax and semantics of our IF-expression language. This is a very simple language embedding; we use the same technique to embed our hardware description language with ACL2.

We can now prove theorems about descriptions involving formulas our IF-expression language. For instance, we can prove:

```
(let ((if-expr  '(IF A B C))
      (bindings (list (cons 'A a)
                      (cons 'B b)
                      (cons 'C c))))
(implies
 (and (if-termp if-expr)
      (eqlable-alistp bindings))
 (equal (if-evl if-expr bindings)
        (if a b c))))
```

This shows for any `a`, `b`, and `c`, that the evaluation of the expression `'(IF A B C)` with the bindings shown is the same as `(if a b c)`.

IV. Our Verification Approach

We verify Verilog circuit descriptions by translating them into a HDL-form that ACL2 can process. We then use our ACL2-based definition of our HDL to symbolically simulate these translations, and we compare the simulation results to ACL2 specifications.

A. Our Verilog-to-EMOD Translator

We have written a translator that converts a Verilog design description into the `EMOD` language. This translation is meant to be principally a syntactic transformation; however, because of the complexity of Verilog it involves a number of semantic transformations.

To implement the translation of Verilog into `EMOD`, we adopt a program-transformation-like [17] style: to begin with, the entire parse tree for the Verilog sources is constructed; we then apply a number of rewriting passes to the tree which result in simpler Verilog versions of each module. The final conversion into `EMOD` is really almost incidental, with the resulting `EMOD` modules differing from our most-simplified Verilog modules only in minor syntactic ways. Since each rewriting pass produces well-formed Verilog modules, we can simulate the original and simplified Verilog modules against each other, either at the end of the simplification process or at any intermediate point.

- We instantiate modules to eliminate parameters introducing new modules for each instantiation size.
- Wires and registers in Verilog can have varying widths, and we resolve all such expressions to constants.
- We reduce the variety of operators we need to deal with by simply rewriting some operators away. In particular, we perform rewrites such as:

  `a && b` $\rightarrow$ `(|a) & (|b)`,
  `a != b` $\rightarrow$ `|(a ^ b)`, and
  `a < b` $\rightarrow$ `~(a >= b)`.

  This process eliminates all logical operators, equality comparisons, negated reduction operators, and standardizes all inequality comparisons.
- We annotate every expression with its type (sign) and width. The rules for determining widths are subtle, and if they are not properly implemented then, signals might be inappropriately kept or dropped.
- We introduce explicit wires to hold the intermediate values in expressions.
- Verilog allows for implicit truncations in assignment statements; for instance, one can assign the result of a five-bit addition `a + b` to a three-bit bus (collection of wires), `w`. We make these truncations explicit by introducing a new wire for the intermediate result. We replace expressions like `a + b` with basic module instances.

We have left out many minor transformations like naming any unnamed instances, eliminating supply wires, and minor optimizations. Together, our simplifications leave us with a new list of modules where only simple gate and module instances are used. From this we can produce either `EMOD` or simplified Verilog. The simplified Verilog can be co-simulated with the original Verilog as a translation sanity check.

B. The EMOD HDL

Our EMOD-language analysis approach permits the hierarchical verification of cooperating finite-state machines. We have been investigating such languages for over 20 years. Our initial attempt was the HEVAL language [2]; this combinational-only language was embedded in the NQTHM logic [4]. This led us to the development of the DUAL-EVAL HDL which was used as the target for the FM9001 microprocessor verification [3]. As we were the designers of the FM9001, we actually created and verified a DUAL-EVAL description of the FM9001 before translating it into LSI Logic's NDL language for implementation [12].

The DE HDL [6] was our first HDL embedded into the ACL2 [11] logic. Later, we extended DE by adding parameters and busses; we called this the DE2 [7] language. To validate a data-network circuit, the logic was represented in DE2 and then this design fragment was verified using ACL2 [14]. Our latest effort is the EMOD HDL, which is used as a target for Nano circuits. Other groups [5] have pursued a similar approach using HOL [16] to provide the formal semantics. Intel has done extensive formal verification of the Intel®Core i7™ processor architecture [10]. AMD is also using formal verification to aid the verification of their processors [15].

The semantics of EMOD are specified by a deeply-embedded interpreter written in the ACL2 logic; this interpreter permits multiple signal evaluation styles: BDDs, AIGs, definedness, dependency, and delay. We believe EMOD is the first formally-specified language to support multiple interpretations of HDL descriptions within a single system, and EMOD is the first formally-defined HDL to be used in a commercial design flow.

Although EMOD language circuit descriptions have the form of a HDL, its structure allows it to be accessed and updated much like a database. Annotations may be attached to every module definition and occurrence; such annotations include information such as signaling conventions, functional requirements, warnings, and clock disciplines. Thus, we eventually imagine that a post-silicon design engineer may interrogate an EMOD-language design with database-like queries to determine properties that were specified and proven by pre-silicon designers. And, a post-silicon engineer may exhaustively establish properties using the speed of fabricated circuits, and then add these properties to the evolving EMOD-based design (database).

In support of commercial design verification, we have defined edge-triggered and level-sensitive, state-holding primitives, and using these primitives, a user may define and verify multi-clock (derived from one master clock) circuits. Verification of gated-clock circuits is supported, indeed, required for the Nano design style. Verifying bi-directional, tri-state busses and pass-transistor circuits requires four-valued equations to be used, and since our transistor-level circuit analyzer targets our four-valued logic, mixed transistor-gate-level designs may also be verified.

C. Our Circuit Models

We formally verify fragments of the Nano by translating them from Verilog to our formal EMOD language, and then performing symbolic analysis. Instead of trying to write a formal semantics for Verilog, we choose to formally define a simpler language and then analyze the results of our translator, which is labeled ACL2 Verilog Translator in Figure 1. The EMOD language contains mechanisms that allow us to represent all of the interface and module names that appear in Verilog design representations, and we verify EMOD circuit representations using ACL2.

We verify EMOD circuit representations to more abstract specifications that we write in ACL2. As depicted in Figure 1, we write Integer and microcode specifications in ACL2, and then symbolically simulate these specifications [1]; this produces, either as AIGs or BDDs, results that we compare to the symbolic simulation of EMOD circuit representations. Sometimes, independently of the Nano design, we may write an even more abstract X86 ISA specification fragments, such as for the floating-point operations, and compare our Integer specifications to these higher-level specifications. For instance, we have such X86 ISA specification fragments for the basic floating-point operations; these specifications are independent from the Nano; they conform to the IEEE floating-point specifications [9].

In a large number of cases, there are custom implementations for various Nano circuits; these circuits are implemented at the transistor level. To verify such transistor-level circuit descriptions, we use our ACL2 Transistor Analyzer which converts a Spice-level circuit representation [13] into a model that has Switches with strengths and Sized capacitors. This kind of model can be symbolically simulated using the SYM_SIM circuit simulator, and we compare the results of such simulations to higher-level, symbolic simulations.

V. ECC Circuit Analysis

We present a memory error-detection-and-correction circuit (ECC) and its analysis. This circuit detects and corrects single-bit memory errors; it also detects double-bit memory errors. A descendant of this circuit is used in the VIA Nano, and we verified its operation using the procedures outlined above. As shown in Figure 2, the circuit is composed of two identical syndrome generators and an ECC element that drives 64 exclusive-OR gates. The "Memory" block is a model we developed to model the operation of the real memory; this block is modeled with 72 exclusive-OR gates, which allows, using the 72 error inputs, to model any number of inversion failures. We have three verification goals:

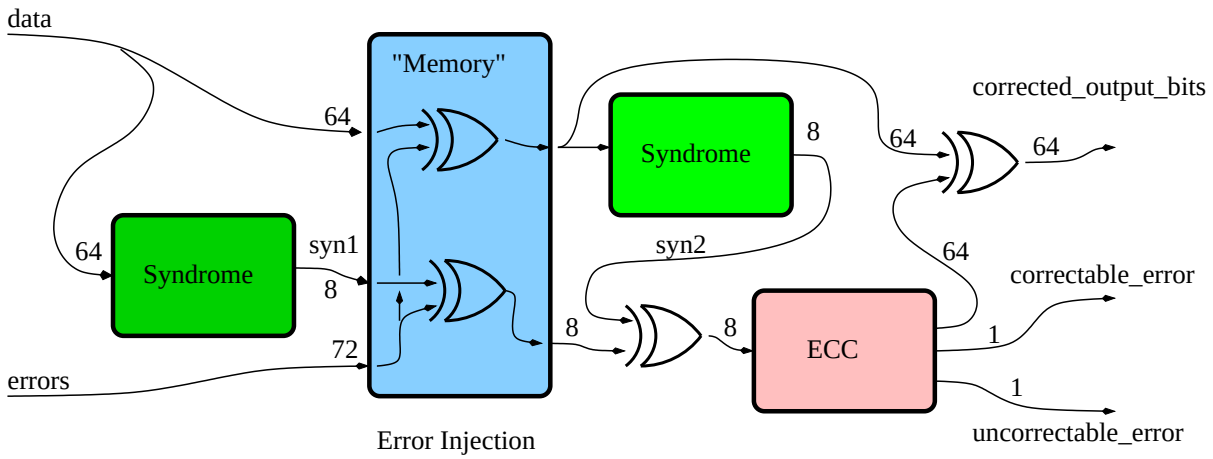


Fig. 2. Error-Correction-Circuitry Diagram

- when there are no memory errors, the output is correct, and no error is indicated;
- when there is one memory error, the output is correct, the `correctable_error` output bit is set, and the `uncorrectable_error` is not set; and
- when there are two memory errors, we only check that the `uncorrectable_error` is set.

We approach this verification considering all of the possible combinations of memory bits and errors that are possible:

- no memory errors: 2^{64} = 18446744073709551616;
- one error: $(2^{64} * 72)$ = 1328165573307087716352; and
- $(2^{64} * (72 * 71)/2)$ = 47149877852401613930496 for when there are two errors.

In the two-error case, the error positions are symmetric. We encode the possible errors explicitly by a one- or two-hot encoding on the `error` input vector; we will later see that our specification functions model these errors.

Below is the Verilog source we will attempt to validate. We constructed this model so we could check that modules `ecc_gen` and `ecc_decode` perform the intended operation. Of course, the operation of this circuit model also depends on the exclusive-OR operations (gates) that are part of the circuit. The exclusive-OR gates in the memory are just part of our model; these gates allow us to model "bit-flips" in the memory.

```
module ecc_model
  (data,                    // Input Data
   errors,                  // Error Injection
   corrected_output_bits,   // Output Data
   correctable_error,       // Corrected?
   uncorrectable_error);    // Can't be corrected

   input  [63:0]  data;     // Data inputs
   wire   [63:0]  data;
   input  [71:0]  errors;   // Error injection bits
   wire   [71:0]  errors;

   output [63:0]  corrected_output_bits; // Output
   wire   [63:0]  corrected_output_bits;
   output         correctable_error;     // Good?
   wire           correctable_error;
   output         uncorrectable_error;   // Bad
   wire           uncorrectable_error;

   wire [7:0]  syn1;     // from first ecc_gen
   wire [7:0]  syn2;     // from second ecc_gen
   wire [63:0] data_err; // Possibly flawed data
   wire [7:0]  syn_err;  // Memory syndrome bits
   wire [63:0] bit_to_correct; // correct outputs

   // Generate syndrome bits for "memory"
   ecc_gen gen1 (syn1, data);

   // Fault injection in memory model.
   assign data_err = data ^ errors[63:0];
   assign syn_err  = syn1 ^ errors[71:64];

   // Thus, using the "errors" input we can create
   // faults that could be considered memory errors.

   // Syndrome bits for "memory" output
   ecc_gen gen2 (syn2, data_err);

   wire [7:0] syn_backwards_xor;
   // Compute syndrome
   assign syn_backwards_xor = syn_err ^ syn2;

   ecc_decode make_outs (bit_to_correct,
                         correctable_error,
                         uncorrectable_error,
                         syn_backwards_xor);

   // Finally, correct the output.
   assign corrected_output_bits
                    = bit_to_correct ^ data_err;
endmodule
```

We now present the ACL2 commands used to define and verify our example ECC circuit. Some details are omitted, but we attempt to supply sufficient detail so a reader can understand the process. After placing ourselves in the directory containing the Verilog above, we start our ACL2-based analysis system and execute the commands below. The `defmodules` command reads the Verilog source and converts it into the `EMOD` language. The `find-input` commands collect and group the input wire names. Similarly, the `find-output` commands collect the output wire names. We use these command because it allows rearrangement of the

module interface without it concerning our effort.

```
; Convert the Verilog ECC model and its inferior
; components into the EMOD language.

(defmodules *ecc* :start-files (list "ecc_model.v")
                  :search-path '("."))

; By name, collect the input data and error inputs.

(defconst *ecc_model/data*
          (find-input "data"   64  |*ecc_model*|))
(defconst *ecc_model/errors*
          (find-input "errors" 72  |*ecc_model*|))

; By name, collect the output and the correctable
; and uncorrectable error indications.

(defconst *ecc_model/corrected-output-bits*
  (find-output
   "corrected_output_bits" 64 |*ecc_model*|))
(defconst *ecc_model/correctable_error*
  (find-output
   "correctable_error" 1 |*ecc_model*|))
(defconst *ecc_model/uncorrectable_error*
  (find-output
   "uncorrectable_error" 1 |*ecc_model*|))
```

The two functions below allow us to form the inputs by name. With the function `create-input`, we construct an association list pairing names with their values, and then we generate an appropriate pattern. This frees us from being concerned about the position of the arguments in the `|*ecc_model*|` model. The next two functions below perform a similar function for the output; that is, we construct three outputs based on the output wire names.

```
(defun create-input (data errors)
  (b* ((alist
        (make-fast-alist
         (ap (pairlis$ *ecc_model/data*   data)
             (pairlis$ *ecc_model/errors* errors))))
       (pat (gsal (gpl :i |*ecc_model*|)
                  alist 'fail))
       (- (fast-alist-free alist)))
      pat))

(defun alist-extract (keys alist)
  (declare (xargs :guard t))
  (if (atom keys)
      nil
    (cons (cdr (hons-get (car keys) alist))
          (alist-extract (cdr keys) alist))))

(defun get-output (output)
  (b* ((alist (pal (gpl :o |*ecc_model*|)
                   output nil))
       (corrected_output_bits
        (alist-extract
         *ecc_model/corrected-output-bits*
         alist))
       (correctable_error
        (car (alist-extract
              *ecc_model/correctable_error*
              alist)))
       (uncorrectable_error
        (car (alist-extract
              *ecc_model/uncorrectable_error*
              alist))))
      (mv corrected_output_bits
          correctable_error
          uncorrectable_error)))
```

We next specify our error-correction circuit. We define the `q-not-nth` function that (symbolically) inverts a bit of `x` at position `n`. If `n` is larger than the length of the list `x`, no change is made. The next three functions specify the operation of our `|*ecc_model*|` when there are no memory errors, when one error is introduced, and when two errors are inserted.

```
(defun q-not-nth (n x)
  ;; Invert bit N of X.
  (if (atom x)
      nil
    (if (zp n)
        (cons (q-not (car x)) (cdr x))
      (cons (car x)
            (q-not-nth (1- n) (cdr x))))))

(defun no-problems ()
  ;; Check output correctness if no errors injected.
  (b* ((data   (qv-list 0 1 64))
       (errors (make-list 72 :initial-element nil))
       (inputs (create-input data errors))
       ((mv & o) (emod 'two |*ecc_model*|
                       inputs nil))
       ((mv corrected_output_bits
            correctable_error
            uncorrectable_error)
        (get-output o)))
      (and (equal corrected_output_bits data)
           (not   correctable_error)
           (not   uncorrectable_error))))

(defun one-bit-error-predicate (bad-bit)
  ;; Check output correctness if one error injected.
  (b* ((data     (qv-list 0 1 64))
       (err-bits (make-list 72
                            :initial-element nil))
       (errors   (q-not-nth bad-bit err-bits))
       (inputs   (create-input data errors))
       ((mv & o) (emod 'two |*ecc_model*|
                       inputs nil))
       ((mv corrected_output_bits
            correctable_error
            uncorrectable_error)
        (get-output o)))
      (and (equal corrected_output_bits data)
           (equal correctable_error (< bad-bit 64))
           (not   uncorrectable_error))))

(defun two-bit-error-predicate (x y)
  ;; For two-bit errors, we only check that the
  ;; uncorrectable error is signaled.
  (if (eql x y)
      ;; If only one error bit is injected.
      (one-bit-error-predicate x)
    (b* ((data     (qv-list 0 1 64))
         (err-bits (make-list
                    72 :initial-element nil))
         (errors   (q-not-nth
                    x (q-not-nth
                       y err-bits)))
         (inputs   (create-input data errors))
         ((mv & o) (emod 'two |*ecc_model*|
                         inputs nil))
         ((mv & & uncorrectable_error)
          (get-output o)))
        uncorrectable_error)))
```

Finally, we introduce functions that generate input suitable to check all one- and two-bit errors. Thus, these functions provide the top-level requirements for the ECC circuit.

```
(defun all-one-bit-errors (x)
  (and (or (one-bit-error-predicate x)
           (cw "one-bit-error ~x0~%" x))
       (if (zp x)
           t
         (all-one-bit-errors (1- x)))))

(defun all-two-bit-errors-help (x y)
  (and (or (two-bit-error-predicate x y)
           (cw "two-bit-error ~x0 ~x1~%" x y))
       (if (zp x)
           t
         (all-two-bit-errors-help (1- x) y))))

(defun all-two-bit-errors (y)
  (if (zp y)
      t
    (and (all-two-bit-errors-help (1- y) y)
         (all-two-bit-errors (1- y)))))

(defun all-zero-one-two-bit-errors (z)
  (and (or (no-problems)
           (cw "no-problems ~%"))
       (all-one-bit-errors z)
       (all-two-bit-errors z)))

(time$ (all-zero-one-two-bit-errors 71))
```

This is not the most efficient way to investigate all such errors, but it is straightforward. We could have introduced additional symbolic variables to indicate input-error positions, and then performed one symbolic computation. However, in spite of the fact that over 5000 symbolic executions of the EEC circuit are performed, it takes less than 30 seconds to consider all of the combinations. When we considered this problem, the ECC circuit designers wanted a quick answer, and this was a simple way to check their intent. But, we realized a few days later that our specification, and therefore, the circuit had an error – our `one-bit-error-predicate` only checks that a flawed data bit is detected, but it does not check if one of the redundant check bits (positions 64 to 71) is itself flawed.

```
(equal correctable_error (< bad-bit 64))
```

This was a problem of there being an error in both the circuit and the specification; subsequently, this error was fixed.

In specification for the ECC circuit we just described, we did not symbolically co-simulate a corresponding ACL2 specification; we directly specified what we expected as answers. Thus, as pictured in Figure 1, instead of comparing `Output and Next State Equations` to `Wire and State Equations`, we just inspected the latter. For our proofs about the Nano media unit [8], we wrote ACL2 specifications that we believed correctly specified it operation. We later verified that our media-unit specifications were valid by proving that they implemented our IEEE floating-point specification.

VI. Conclusion

We have verified parts of the VIA Nano Verilog design using the ACL2 theorem prover. Much of the verification "work" is done with symbolic simulation, and we use the ACL2 theorem prover both to verify high-level properties and to orchestrate the various verification techniques we use. All of our proofs are carried out with the ACL2 theorem prover, and the BDD and AIG algorithms we use have also been verified using the ACL2 theorem prover.

Beyond the straightforward mechanisms described here, we often use additional verification techniques. The circuit descriptions we verify include state-holding elements, and we must either initialize such state-holding elements with suitable initial values or perform additional symbolic simulation that forces such storage elements into suitable (symbolic) states. We usually simulate a circuit for multiple steps, as it requires several clock cycles for such such circuits to complete their operations. With sequential circuits, it is necessary to specify the clocking discipline; that is, when and in what phases the clocks arrive is critical to circuit operation. For instance, for the verification work on the Nano media unit, we must correctly orchestrate 26 clock inputs. We use input parametrization, with appropriate choice of input space partitioning, to allow verifications where otherwise we might fail to create desired output equations – generally, we construct AIGs and then, through an iterative BDD construction procedure, we compare the results produced to their specifications. We have developed a general procedure for symbolically simulating any specification written in ACL2. Using this procedure, we symbolically evaluate ACL2-based specifications and compared to their results to an `EMOD` simulation. Separately, we prove desired correctness properties about such ACL2 specifications.

Most of our overall effort has been directed to verifying execution-cluster properties, much in the same way that Intel has done with the Nehalem family [10]. AMD has also been using ACL2 to verify elements from their Athlon processors [15]. We have begun to explore the use of our formal verification tools for other parts of the Nano design; for instance, we have recently been investigating the instruction decoder because a problem manifested itself that was not discovered by other tools; this was due to a lack of capacity, as the state machines being compared were too large for available commercial tools.

Our application of one formal system, specifically ACL2, may be broader than any single formal verification tool in use by other projects. We use ACL2 to read and translate the Verilog and to model the behavior of Nano circuits at the transistor level; this part of our verification flow has become more important as we experience the limitations of commercially available tools. We specify high-level operations, such as floating-point operations, in a manner that is independent of the specific operation of the Nano; these specifications are general and would likely be valid for many microprocessors. We are expanding the use of formal verification on future Nano microprocessors.

Acknowledgment

The author would like to thank Jared Davis, Anna Slobodova, and Sol Swords, for the work that they have done to make this effort possible, and for their contributions to this paper.

REFERENCES

[1] Robert S. Boyer and Warren A. Hunt, Jr.: "Symbolic Simulation in ACL2", with Robert S. Boyer, in the Proceedings of the Eighth International Workshop on the ACL2 Theorem Prover and its Applications, May, 2009.

[2] Bishop C. Brock and Warren A. Hunt, Jr.: "The Formalization of a Simple HDL", *Proceedings of the IFIP TC10/WG10.2/WG10.5 Workshop on Applied Formal Methods for Correct VLSI Design*, Elsevier Science Publishers, 1989.

[3] Bishop C. Brock and Warren A. Hunt, Jr.: "The `DUAL-EVAL` Hardware Description Language and Its Use in the Formal Specification and Verification of the FM9001 Microprocessor", in *Formal Methods in Systems Design*, Volume 11, pp. 71–105, Kluwer Academic Publishers, 1997.

[4] Robert S. Boyer and J Strother Moore: "A Computational Logic Handbook", Academic Press", 1988.

[5] Mike Gordon, "Relating event and trace semantics of Hardware Description Languages", in The Computer Journal, Volume 45, No. 1, 2002.

[6] Warren A. Hunt, Jr.: "The DE Language", in Computer-Aided Reasoning ACL2 Case Studies, edited by Matt Kaufmann, Panagiotis Manolios, and J Strother Moore, Kluwer Academic Publishers, 2000.

[7] Warren A. Hunt, Jr. and Erik Reeber: "Formalization of the DE2 Language", in Correct Hardware Design and Verification Methods (CHARME 2005), *Lecture Notes in Computer Science*, No. 3725, pp 20–34, Springer-Verlag, 2005.

[8] Warren A. Hunt, Jr. and Sol Otis Swords: "Centaur Technology Media Unit Verification", *Proceedings of the 21st International Conference on Computer Aided Verification (CAV 2009)*, In Computer-Aid Verification (CAV) 2009, LNCS No. 5643, pp 353–367, Springer-Verlag, June, 2009.

[9] IEEE Computer Society: IEEE Standard for Floating-Point Arithmetic. IEEE Std 754^{TM}-2008 edn.

[10] Roope Kaivola, Rajnish Ghughal, Naren Narasimhan, Amber Telfer Jesse Whittemore, Sudhindra Pandav, Anna Slobodová, Christopher Taylor, Vladimir Frolov, Erik Reeber and Armaghan Naik, *Replacing Testing with Formal Verification in Intel CoreTM i7 Processor Execution Engine Validation*. In Computer-Aid Verification (CAV) 2009, LNCS No. 5643, pp 414–429, Springer-Verlag, June, 2009.

[11] Matt Kaufmann, Panagiotis Manolios and J Strother Moore. Computer-Aided Reasoning: An Approach. Kluwer Academic Press, Boston, Massachusetts, 2000.

[12] LSI LOGIC. 1.5-Micron Array-Based Products Databook. LSI Logic Corporation, Milpitas, CA. 1990.

[13] T. Quarles, A. R. Newton, D. O. Pederson, A. Sangiovanii-Vincentelli. SPICE 3B1 User's Guide. Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, California. April, 1987.

[14] Erik Reeber and Warren A. Hunt, Jr., *A SAT-Based Decision Procedure for the Subclass of Unrollable List Formulas in ACL2 (SULFA)*. In the Third International Joint Conference on Automated Reasoning, Springer Verlag, Volume 4130, pp. 453–467.

[15] David Russinoff, *A Case Study in Formal Verification of Register-Transfer Logic with ACL2: the Floating-point Adder of the AMD Athlon (TM) Processor*. In: Formal Methods in Computer-Aided Design. (2000) 22–55

[16] Konrad Slind and Michael Norrish, *A Brief Overview of HOL4*. In TPHOL, pp. 28-32, 2008.

[17] Eelco Visser, *A Survey of Strategies in Rule-Based Program Transformation Systems*. In the Journal of Symbolic Computation, Volume 40, Issue 1, pp. 831–873, July, 2005.

Embedded Systems Design – Scientific Challenges and Work Directions

(Invited Paper)

Joseph Sifakis
Verimag

Abstract

The development of a satisfactory Embedded Systems Design Science provides a timely challenge and opportunity for reinvigorating Computer Science.

Embedded systems are components integrating software and hardware jointly and specifically designed to provide given functionalities, which are often critical. They are used in many applications areas including transport, consumer electronics and electrical appliances, energy distribution, manufacturing systems, etc.

Embedded systems design requires techniques taking into account extra-functional requirements regarding optimal use of resources such as time, memory and energy while ensuring autonomy, reactivity and robustness.

Jointly taking into account these requirements raises a grand scientific and technical challenge: extending Computer Science with paradigms and methods from Control Theory and Electrical Engineering. Computer Science is based on discrete computation models not encompassing physical time and resources which are by their nature very different from analytic models used by other engineering disciplines.

We summarize some current trends in embedded systems design and point out some of their characteristics, such as the chasm between analytical and computational models, and the gap between safety critical and best-effort engineering practices. We call for a coherent scientific foundation for embedded systems design, and we discuss a few key demands on such a foundation: the need for encompassing several manifestations of heterogeneity, and the need for design paradigms ensuring constructivity and adaptivity.

We discuss main aspects of this challenge and associated research directions for different areas such as modelling, programming, compilers, operating systems and networks.

Formal Verification of an ASIC Ethernet Switch Block

B. A. Krishna
Chelsio Communications Inc.
bkrishna@chelsio.com

Anamaya Sullerey
Chelsio Communications Inc.
anamaya@chelsio.com

Alok Jain
Cadence Design Systems, Inc.
alokj@cadence.com

***Abstract*— Traditionally, validation at the ASIC block level relies primarily upon simulation based verification. Specific components that are "hot spots" are then considered as candidates for Formal Verification. Under this usage model, the hurdles to Formal Verification are intractability and poor specifications. In this paper, we outline an alternate approach, where we used Formal Verification as the "first line of defense" in the course of validating a Packet Switch. This block had several components that were complex and hard to verify, including components that required liveness guarantees, where responses are event bound, and not cycle bound. To surmount typical hurdles, an early collaboration was formed between design and verification engineer, both to influence the design as well as to identify relevant manual abstraction techniques upfront. All significant components were formally verified at the module level.**

This approach was successful in identifying most bugs during the design phase itself and drastically minimized bugs during verification/emulation phases of the project. This paper illustrates the strengths of such an approach. It describes our overall methodology and the proof techniques utilized. The overall effort yielded a total of 55 bugs found (52 during the design phase and only 3 bugs during the verification phase). No bugs were found subsequently during emulation. As a result, this block was deemed "tape out ready" 2 months prior to other blocks of similar complexity.

I. INTRODUCTION

The complexity of modern designs has been increasing at a rapid pace. Modern design blocks are made up of modules that have very complex behaviors and interactions. Verification of such blocks poses a serious challenge. The conventional approach is to verify through simulations at the block level. However, simulation has the inherent limitation that one can simulate only a limited set of patterns in any reasonable amount of time. As design sizes grow, it is becoming increasingly difficult to maintain a high level of confidence purely based on simulation coverage. A possible solution is to use Formal Verification to verify some of complex modules in your design. Formal Verification performs exhaustive verification by exploring the entire state space of the design.

In this paper, the design block under consideration is a switch with around 650k gates and with multiple ports. Most of the modules inside this design block have high complexity both in terms of the control oriented behavior and data path operations. Based on previous experiences, it was estimated that simulation based verification techniques of such designs would require more than a year for a dedicated engineer to fully verify.

In addition, the design in question also had several components for which *liveness* guarantees were required, which were not possible to validate using simulation based verification. Thus, it was therefore concluded that the most cost-effective approach would be to utilize Formal Verification techniques to prove correctness of all significant components of the design at the module level. Conventional simulation based design verification (DV) was also done, but at the block level.

Our overall approach was inspired by the following quotation from *"Mythical Man Month"* [1]:

"The use of clean, debugged components saves much more time in system testing than that spent on scaffolding and thorough component test."

Our FV efforts commenced very early during the design phase and consisted of the following methodology (which took place alongside conventional DV efforts at the block level):

1) Partition the design into minimally sized pieces and generate specifications at the module level. Use the compositional verification technique of proving properties of a system by checking the properties of its components, using "assume-guarantee" style reasoning.

2) Aim to prove "black-box" (end-to-end module level) properties and use the tractability results to both influence design re-partitioning as well as to gain insights about RTL complexity.

3) Study cones of influence in order to deduce possibilities for manual abstractions. Once identified, these abstractions were then used to replace stateful RTL components within the design.

In a few cases where all other options failed, we resorted to proving "white-box" properties (based on RTL internal state). We used this approach as a last resort since rigorous specifications of RTL internals are hard to come by, and further, such specifications often change in the course of the design cycle.

This paper will focus on the techniques used to verify two of the modules in the design, namely the *Synchronizer* and the *Page Manager* modules. The first case study is a control &

datapath block that consists of 20k gates and the second is a datapath block that consists of 25k gates.

In subsequent sections, we describe each module, the Device Under Test (DUT) operational details, the Formal Verification strategy utilized in each case as well as the verification results. Later, we also present our overall results (number of bugs found etc.) and our high level conclusions. The model checkers Incisive Formal Verifier (IFV)[2] and SMV[5] were used over the course of this project.

II. DESCRIPTION OF THE PACKET SWITCH

The design block under consideration was a packet switch with multiple ports that accepted packets, stored them in memory, and later forwarded them to various output ports, allowing for the possibilities of switching and replicating packets.

In order to accomplish this functionality, the block had various types of complex components, components that were responsible for storing incoming packets to memories, components that were responsible for managing pages in memory over which packets were stored, components that maintained caches, etc.

The goal here was to a) design specifically with Formal Verification in mind (keep modules small, keep interfaces crisp) as well as to b) formally verify as many elements of the design as possible. In total, 14 modules of the design were formally verified. The design consisted of 18 modules in its entirety.

The following design principles were utilized to ensure FV tractability:

- Careful design partitioning with exhaustive invariants of module interfaces.
- Isolation of modules that exhibit FIFO-ness.
- Significant parameterization of modules, to allow abstraction/reduction of bus widths, etc.
- Significant reuse of common modules, e.g., arbiters, aligners, etc.
- Decomposition of all architectural invariants into micro-architectural invariants.

III. FORMAL VERIFICATION OF THE SYNCHRONIZER

The synchronizer module has two inputs, a) packet data is sent across *in_{valid,sop,eop,data[63:0]}* , where *sop* and *eop* are start/end packet delimitors and b) address of a valid page is specified across *in_addr, in_addr_valid*. Its purpose is to place the arriving data, which arrives in units of 8 bytes, into various slots within the specified page. The interface for this module is shown in Fig 1.

The input packet data bus adheres to the following protocol: *in_valid* is asserted whenever there is new data presented across the input. During the first 8 byte data chunk within a packet, *in_sop* will be asserted, and during the last 8 byte chunk, *in_eop* will be asserted.

Each page is of size 128 bytes, which is broken down into 16 x 8 byte slots. This module receives an input, *sync_cnt[3:0]*, which is an external counter that increments every cycle. The output consists of: *rf_write, rf_write_sop, rf_write_eop, rf_write_data[63:0]*. If, at any point in time, we see *rf_write==1* and *sync_cnt==i (where i:=0...15)*, then it means that *rf_write_data[63:0]* is being written into slot *i* within the page.

The rules determining when/what data is written into a particular slot in a page are described in the *Operational Details* section. All data arriving over *in_data* goes into an internal *skid_fifo*. The data that is at the head of the *skid_fifo* is written out into a page only when various design rules are satisfied.

This module is called the *synchronizer* because it synchronizes when and where an incoming data segment is written into a page. It is part of a larger system that is responsible for accumulating various 8 byte chunks of data within a register file so that it can later generate an atomic memory write operation for a half page worth of data.

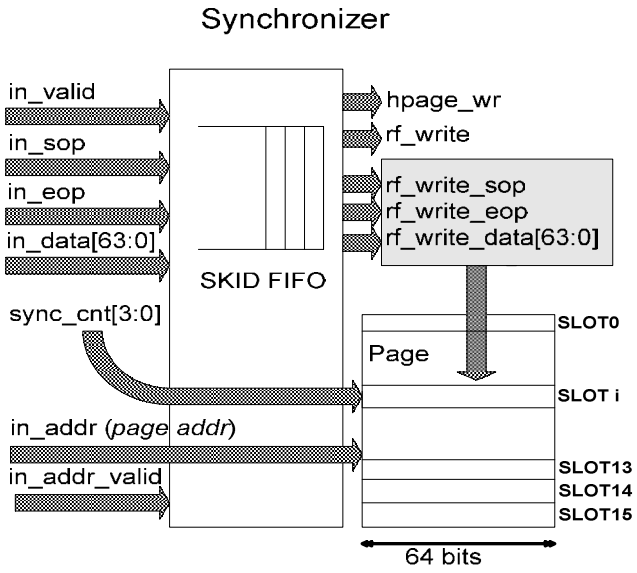


Figure 1 – Block Diagram of the Synchronizer

IV. OPERATIONAL DETAILS OF THE SYNCHRONIZER

Following are rules governing the *Synchronizer* module:

- Across the datapath between *in_{valid,sop,eop,data}* & *rf_write,rf_write_{sop,eop,data}*, FIFO-ness needs to hold. Note that the input bus has no backpressure

capability (i.e., the input interface should *always* be able to sink data and cannot throttle the input bus).

- For a given page (presented on: *in_addr), rf_writes* should occur to *slot=0....15* in a monotonically increasing fashion.

- For a given page, if a non-EOP data word was written into *slot=i*, then the next data word for this packet *must* be written into *slot=i+1*.

- If we are at the lower half of a page (*slot=7*) and a) there's an *rf_write* or b) we are not within a packet and have seen an *rf_write* in the past to the lower half of this page, then at the next cycle *hpage_wr* will be asserted and not otherwise.

- If we are at the upper half of a page (*slot=15*) and a) there's an *rf_write* or b) we are not within a packet and have seen an *rf_write* in the past to the upper half of this page, then at the next cycle *hpage_wr* will be asserted and not otherwise.

V. DESIGNER'S INVARIANTS FOR THE *SYNCHRONIZER*

Apart from the rules that were identified by the verification engineer, we also proceeded to prove the following invariants put forth by the designer. The intent here was to prove invariants that emerged after interface study by the verification engineer, as well as those that were deemed important by the designer.

- If there is an *rf_write* to some slot *x (where x=0…15)*, then there will be no write to slot *y (y<=x)* until there is an assertion of output signal *hpage*_wr.

VI. SYNCHRONIZER VERIFICATION STRATEGY

We could visually establish that this module demonstrated *data independence*. The circuit accepted data and shuffled it around, but no control signals were derived from data. This could be done relatively easily, by examining the fan-out cone associated with the data-path elements.

Further, the design also dealt exclusively in terms of 8 byte (64 bit chunks) and didn't reorder data bytes within each incoming double word. In order to prove that the unit fulfilled the specification of a FIFO, it was possible to utilize *Wolper's Theorem* [3], abstract the data width to just 2 bits, inject a regular expression consisting of *A*BA*CA** over the input data interface and expect that the data showing up at the output also conformed to this regular expression.

A packet generator was written to inject packets that a) conformed to SPI4 framing conventions and b) had a minimum length of 64 bytes, over the input bus: *in_{valid,sop,eop,data}*. This packet generator data words consisting of just 4 types: *{A,B,C,D}, where A=64'h0, B=64'h1, C=64'h2, D=64'h3*. A auxiliary fsm was written to monitor the outputs: *rf_write,rf_write_{sop,eop,data}*.

Three critical proofs, pertaining to packet data-integrity and framing, were then cast using the packet generator and auxiliary FSM.

Proof Obligation1*:* To prove data integrity across the FIFO's data-path.

This proof asserted that if we injected packets conforming to the regular expression *A*BA*CA** over *in_data[1:0]*, then we are guaranteed to see outputs that also conform to the regular expression *A*BA*CA** over *rf_write[1:0]*. Note that this regular expression is injected and expected across all valid input and output data words This proves that no input data word is dropped, duplicated or reordered.

Proof Obligation2*:* To prove that SOPs are preserved intact across the internal FIFO.

For this proof, the regular expression *A*BA*CA** was injected into *in_data[1:0]* for SOP input words, and *D* was injected into *in_data[1:0]* for non-SOP input words. The expectation was that the regular expression *A*BA*CA** will always be seen on *rf_write[1:0]*, for SOP output words and *D* will always be seen on *rf_write[1:0]*, for non SOP output words.

Any corruption of an input SOP word (with data values: *{A,B,C}*) into an output non-SOP word, would result in an output non-SOP with a value of *{A,B,C}*, which will be detected as a violation of *Proof Obligation2.*

Any corruption of an input non-SOP word (with data value: *D*) into an output SOP word, would result in an output SOP word with a value of *D*, which will be detected as a violation of *Proof Obligation2.*

Proof Obligation3*:* To prove that EOPs are preserved intact across the internal FIFO.

For this proof, the regular expression *A*BA*CA** was injected into *in_data[1:0]* for EOP input words, and *D* was injected into *in_data[1:0]* for non-EOP input words. The expectation was that the regular expression *A*BA*CA** will always be seen on *rf_write[1:0]*, for EOP output words and *D* will always be seen on *rf_write[1:0]*, for non EOP output words.

Any corruption of an input EOP word (with data values: *{A,B,C}*) into an output non-EOP word, would result in an output non-EOP with a value of *{A,B,C}*, which will be detected as a violation of *Proof Obligation3.*

Any corruption of an input non-EOP word (with data value: *D*) into an output EOP word, would result in an output EOP word with a value of *D*, which will be detected as a violation of *Proof Obligation3.*

In order to prove that writes within a page were to monotonically increasing slots, a tracking FSM was written. This FSM did the following: Every time a new page was presented over *in_addr, in_addr_valid*, it recorded the slot into which it first saw an *rf_write*, storing both the value of *sync_cnt* into *last_wr_ptr* as well as *rf_write_{sop,eop}* into *last_wr_{sop,eop}*.

Properties were then written to monitor the behavior of *rf_write*. The two most important assertions were:

1. If we are performing an *rf_write* to some slot=*sync_cnt* and if this is *not* the first write to the page, then *sync_cnt* will be greater than *last_wr_ptr*.

2. If we are performing an *rf_write* and if this is *not* the first write to the page and if the previous write was a non-EOP data word (i.e., *last_wr_ptr=i && last_wr_eop=0*), then this write *will* be to slot=*(i+1)*.

This tracking FSM also monitored writes to upper/lower halves of a page such that later, when *sync_cnt={7,15}* (i.e., write pointer is at the upper/lower half boundaries), if any writes had occurred to a half, the output *hpage_wr* would be asserted.

VII. SYNCHRONIZER VERIFICATION RESULTS

A critical bug was found in the implementation of *hpage_wr*. The failing counterexample consisted of a scenario where there was a write to the upper half of a page for which there was a valid *hpage_wr* assertion. However, this signal continued to be asserted for 8 extra cycles indicating a write to the lower half of the page inspite of the fact that the lower half was not written into. This was found very early in the design stage.

Another critical bug was found in the FIFO size required. The property corresponding to *Proof Obligation1* failed. Our analysis showed us that the minimum FIFO depth should have been 18 and not 16. The depth had to account for the internal FIFO latency. This bug was found very early in the design stage. While *sync_cnt* is a primary input to this module, it is an internal signal within the larger block. Since conventional simulation based DV was being performed at the block level, precise control over this signal is difficult to realize in simulation, making this bug an improbable event within block level DV. The designer estimates that debugging this issue would have required ~ 2 hours within a block level verification test failure, but within the module level FV framework, this debugging took just a few minutes.

VIII. FORMAL VERIFICATION OF THE *PAGE MANAGER* MODULE

The *Page Manager* module's block diagram is shown in Figure 2. It is responsible for managing all pages on the receive path of our Ethernet Switch. This module's interface supports four types of requests: *Allocate, Enqueue, Dequeue* and *Dealloc*. It also has an output bus, *Page Free*.

IX. PAGE MANAGER OPERATIONAL DETAILS

Data passing through the switch from input to output ports is stored in pages. A list of pages defines a packet. The *Page Manager* maintains the state of the page, from the time it is allocated until the time it is relinquished. Internally, the *Page Manager* consists of a) *Free List Manager* and b) *Life Count Memory*. These two sub-units together maintain the state of a page, which consists of its allocation state as well its reference count (i.e., the number of packets utilizing that page).

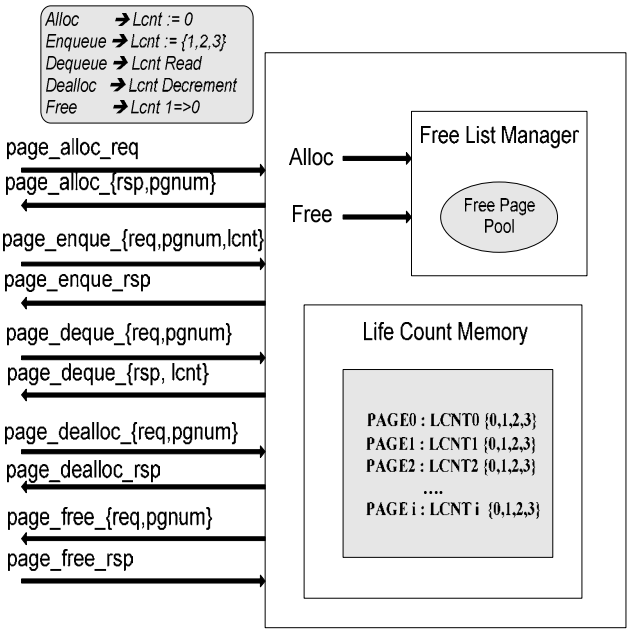


Figure 2 – Block Diagram of Page Manager

The *Free List Manager* sub-unit maintains a list of free pages and its interface allows pages to be allocated and freed. The *Life Count Memory* sub-unit maintains a reference count (also called *Life count* or *lcnt*) on a per-page basis, representing the number of packets present on a single page. The legal *lcnt* values are: 0...3.

The life cycle of any page consists of the following event sequence:

- The unit first receives a *Page Allocate.* This request is fielded by the *Free List Manager*, and a free page is handed to out to the requestor. Coincident with that, the page's *lcnt* is initialized to 0 in the *Life Count Memory*..

- Once a page has been successfully allocated, an *Enqueue* reque*st* will be received along with a specified initial *lcnt.* The legal values for *lcnt* are: {0,1,2,3}. This information is then stored alongside the page within the LCNT complex.

- After a page has been *Enqueue'ed*, it will then receive (at arbitrary points in time), various *Page Dealloc* requests.

During each *Dealloc* request, this page's *lcnt*, will be decremented in the *Life Count Memory* complex.

- The design assumes that once a page has been *Enqueue'ed* with some *lcnt* (1,2 or 3), it will only field those many *Dealloc* requests. After the last *Dealloc* request (in the course of which a particular page's *lcnt* goes from 1 to 0), the *Free List Manager* should free the relevant page and the*Page Free* output signal will be asserted.

- Between the time a particular page has been *Enqueue'ed*, and the time it is freed up, its *lcnt* can be read any number of times over the *Page Dequeue* interface. Each *Dequeue* request extracts the *lcnt* and return this value in the *Dequeue* response.

X. PAGE MANAGER VERIFICATION STRATEGY

The design was responsible for managing a total of 1024 pages. When an attempt was made to cast proofs against the DUT, it was found that the proofs did not converge due to *state space explosion*. The biggest contributor to the state space was the *Free List Manager* (with 1024 state bits).

The *Free List Manager*'s interface definition is shown in Table I. This module has a page allocation interface *alloc_{srdy,drdy,num}* as well as a page free interface *dlloc_{srdy,drdy,num}*.

Table I (Free List Manager Interface)

```
/*
 *      alloc_srdy => alloc page available
 *      alloc_drdy => alloc page consumed by client
 *      alloc_num => alloc page number
 *      dlloc_srdy => dlloc page requested by client
 *      dlloc_drdy => dlloc page request accepted
 *      dlloc_num => dlloc page number
 */
module fl_mgr(
  Clk,
  Rst_,
  alloc_srdy,
  alloc_drdy,
  alloc_num,
  dlloc_srdy,
  dlloc_drdy,
  dlloc_num
);
input          Clk;
input          Rst_;
output         alloc_srdy;
input          alloc_drdy;
output [9:0]   alloc_num;
input          dlloc_srdy;
output         dlloc_drdy;
input [9:0]    dlloc_num;
...
endmodule
```

Our abstraction reasoning hinged on a single observation: *If you focus on the life of a single page, every other page's activity (and state) should be orthogonal to this page's life.* We utilized this observation in constructing a manual abstraction for the *Free List Manager* that maintains state only for a single page of interest thereby cutting down the size of the cone-of-influence significantly. This technique is based on the *Refinement strategy* described in [4].

The *Free List Manager* abstraction had the following characteristics:

- It was aware of the address of a *magic page* and maintained state only for that page.

- It operates in two modes, depending upon whether this *magic page* is allocated or not:
  - If the *magic page* was already allocated, during subsequent allocation requests, it would non-deterministically allocate a page whose address!= *magic page.*

  - If the *magic* page wasn't already allocated, during subsequent allocation requests, it would non-deterministically allocate any page (including one whose address == *magic page*).

This *Free List Manager* abstraction SMV code is shown in Table II. This abstraction was coded in both SMV (for abstraction soundness proofs) as well as in verilog (for the *Page Manager* proofs, which were run within IFV).

As can be seen in the abstraction's code, a single state variable, *magicPageAllocated*, was used to record whether or not the *magic page* was allocated, and this state is then used in determining the page handed out during allocation requests.

Aside from this state, the notion of *magic page* was maintained within a *rigid variable* that was set non-deterministically by the external environment at the time of reset, and kept constant during each path. By virtue of maintaining just 1 bit of state (*magic page*'s allocation state), the number of bits of state was reduced by 1023 bits within the cone of influence. This abstraction was then used to replace the *Free List Manager* instance within the DUT.

The intent here, in the construction of the *Free List Manager* abstraction, was to provide ourselves with a light-weight stub that allowed completely non-deterministic allocation and freeing of pages, with arbitrary latencies, with a single restriction that it would never reallocate the *magic page*, if someone else already have it allocated – which are characteristics required for this abstraction to be "sound".

Table II (Free List Manager Abstraction)

```
layer abstract : {
  alcVld                 : boolean;
  dlcVld                 : boolean;
  magicPageAllocated     : boolean;
  magicPageAllocatedNxt  : boolean;

  alcVld := (alloc_srdy & alloc_drdy);
  dlcVld := (dlloc_srdy & dlloc_drdy);

  init  (magicPageAllocated) := 0;
  next (magicPageAllocated) := magicPageAllocatedNxt;

  /* magicPageAllocatedNxt generation */
  default {
    magicPageAllocatedNxt := magicPageAllocated;
  } in {
    if (~Rst_)
      magicPageAllocatedNxt := 0;
    else {
      if (alcVld & ~dlcVld){
        /* Only Alloc */
        if ((alloc_num=magicPage) | magicPageAllocated)
          magicPageAllocatedNxt := 1;
      }
      else
      if (~alcVld & dlcVld){
        /* Only Dlloc */
        if (magicPageAllocated & dlloc_num=magicPage)
          magicPageAllocatedNxt := 0;
      }
      else
      if (alcVld & dlcVld){
        /* Both Alloc & Dlloc */
        if (alloc_num=magicPage)
          magicPageAllocatedNxt := 1;
        else
        if (dlloc_num=magicPage)
          magicPageAllocatedNxt := 0;
      }
    }
  }

  /* alloc_num generation */
  default {
    /* any page whatsoever */
    alloc_num := {0..MAX_NPAGES-1};
  } in {
    if (alloc_drdy & magicPageAllocated){
      /* any page other than magicPage */
      alloc_num := { i : i=0..MAX_NPAGES-1, i~=magic Page };
    }
  }
```

The FV framework additionally maintained an auxiliary non-deterministic "tracking state" FSM (*trkState*) to both exhaustively generate requests sequences while tracking the life of the *magic page* as well as to help predict the DUT's responses. This FSM's state diagram is shown in Figure 3

The *trkState* FSM starts off in IDLE state and transitions into ALCD state if *magic page* is allocated. Once it is in ALCD state, it non-deterministically generates an *Enqueue* request with *lcnt={1,2,3}* and transitions to states LCNT1, LCNT2, LCNT3 respectively. After it moves into an LCNT state, it then non-deterministically generates as many *Dealloc* requests as is permissable.

During the last *Dealloc* request generation (which occurs while in LCNT1) state, this FSM expects to see a *Page Free* event for the *magic page*. If this event occurs, the FSM transitions to IDLE. On the other hand, during this last *Dealloc*, a *Page Free* event is not observed for the *magic page*, it transitions to and forever remains in ERROR state. In addition, any unexpected output event also caused a transition to ERROR state.

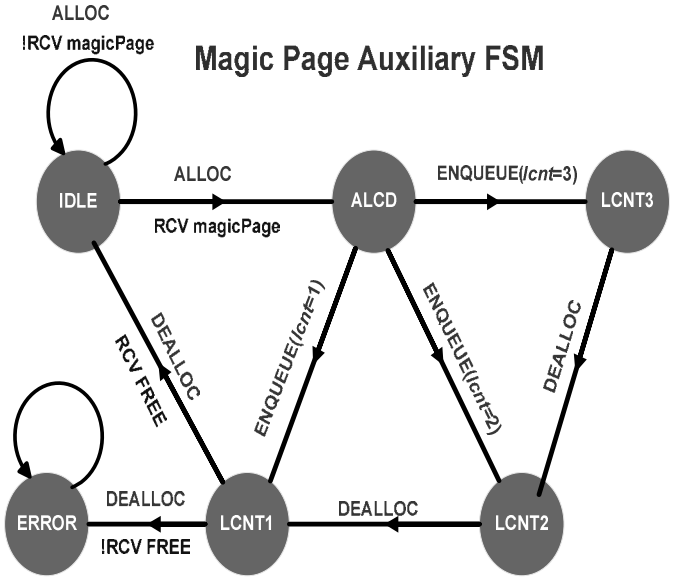


Figure 3 – trkState FSM state diagram

There are two modes of operation within the FV framework, based on whether or not *magicPageAllocated* is set:

1. If *magicPageAllocated* is 0, the *trkState* FSM will be in IDLE and the FV framework will non-deterministically generate requests (for any page), to the DUT.

2. If *magicPageAllocated* is 1, the *trkState* FSM will generate legal/exhaustive requests (for *magic page*) while other input constraints non-deterministically generate requests (for any page other than *magic page*).

In addition to generating exhaustive and legal inputs, the purpose of the FSM's state variable was to predict the DUT's responses while in various states.

We now describe some important assertions governing the DUT's behavior (These were coded in System Verilog):

- While in non-IDLE states (i.e., *magic page* has already allocated), the DUT should not reallocate *magic page* to any other requesting agent.

- After the Allocate phase, during the *Enqueue* phase for the *magic page,* the specified *lcnt* should be initialized.

- After the *Enqueue* phase for the *magic page,* during each *Dealloc* phase, its *lcnt* should be properly decremented in the LCNT memory.

- The output *Page Free* should be generated for the *magic page* if and only if the last *Dealloc* request has been issued for this page.

- While in non-IDLE states, for any *Dequeue* request, the response *lcnt* should match what we expect based on the FSM state (0 if in ALCD, 1 if in LCNT1, 2 if in LCNT2, 3 if in LCNT3).

Table III (Example SV Assertions)

```
/*
 *  If we're in non-IDLE state, magic page is already in use and
 * should not be reallocated to any other requestor
 */
assert_page_no_realloc: assert property(
  @(posedge Clk) disable iff  (!Rst_)(
    (trkState!=IDLE) |-> !(page_alloc_req && page_alloc_rsp
&& page_alloc_pgnum==magic_page)
  )
);

/*
 *  If  in LCNT1 state and there is a dealloc of the magic page,
 * then we should see a freeing of the magic page
 */
assert_page_free_valid: assert property(
  @(posedge Clk) disable iff  (!Rst_)(
    (trkState==LCNT1 && page_dealloc_req &&
page_dealloc_rsp && page_dealloc_pgnum==magic_page) |->
    (page_free_req && page_free_pgnum==magic_page
  )
);

/*
 * If  in !(LCNT1 state and there is a dealloc of the magic page),
 * then we should not see a freeing of the magic page
 */
assert_page_free_invalid: assert property(
  @(posedge Clk) disable iff  (!Rst_)(
    ! (trkState==LCNT1 && page_dealloc_req &&
page_dealloc_rsp && page_dealloc_pgnum==magic_page) |->
    ! (page_free_req && page_free_pgnum==magic_page
  )
);
```

We provide some example SV assertions in Table III. The first property, *assert_page_no_realloc*, asserts that if *trkState* is not IDLE, that is if the *magic page* is already allocated, it will not be reallocated to any other requestor.

The second and third properties that are shown here, *assert_page_free_{valid,invalid}*, describe the necessary and sufficient condition required for the *magic page* to be freed ("*magic page should be freed if and only if trkState is in LCNT1 and magic page is deallocated"*).

By maintaining a rigid variable that determined *magic page* and by having a *Free List Manager* abstraction that maintained state for just this one page, the design was rendered tractable. The properties outlined earlier were all proven against the life of this single *magic page*, and since this page address was non-deterministically generated (to have any page address), the proofs hold for all pages.

In the interest of completeness, the *Free List Manager* was separately formally verified within an SMV framework. Two properties were proven against the actual *Free List Manager*:

- A page, once allocated, will never be reallocated until it is deallocated *(safety property)*
- All page allocation requests will eventually be fulfilled *(liveness property)*

It is worth noting that the last property mentioned above required the following *fairness* constraint: *"Every allocated page will always eventually be relinquished"* in order to eliminate invalid counter-examples.

In addition, the soundness of (an SMV version of) the *Free List Manager* abstraction was also proven within this framework.

XI. OVERALL VERIFICATION RESULTS

During this project, 14 modules within this block were formally verified by a single FV engineer, over a period of 6 months. A total of 55 bugs were found during this effort; 52 bugs were found in the design phase and 3 bugs were found in the verification phase. It is also worth noting that during the verification phase, 3 other bugs slipped through FV and were found in block level simulation (2 were due to missing properties and 1 was due to an overly tight constraint).

The 3 bugs found in simulation were recreated within FV by adding new properties and correcting an overly constrained input. In addition, the fixes were formally verified.

During emulation, this formally verified block was the first to successfully withstand data integrity type testing. As a consequence, this block was deemed *tape-out ready* two months prior to other blocks, of similar complexity that exclusively underwent simulation based verification.

During ASIC "bring-up", no issues were found in any of the design components that were formally verified.

XII. CONCLUSIONS

Based on our experience, we come to the conclusion that it is possible to significantly address block level verification needs

by breaking down the design into minimally sized modules and then formally verifying each of them.

Our methodology also helped yield the following benefits over the course of this project:

- Overcoming state space explosion during proof runs within the model checker.
- Generating rigorous specifications upfront at the module level, something that is often overlooked while embarking on "block level" DV.
- Providing SVA assertions and assumptions which could also be used in simulation.
- Creating FV frameworks within which we could verify design changes/bug fixes with a high degree of confidence alleviating the need to rerun all simulation tests.

While re-partitioning of design based on FV tractability can sometimes lead to added design latency, this tradeoff was worthwhile overall because the more minimally sized design modules were easier to maintain.

We also observed that debugging of counter-examples was very efficient since we specified a large number of module level invariants that helped isolate root-causes fairly quickly.

We believe there is value in some amount of overlap between FV efforts and conventional simulation based verification. Such a parallel/overlapping approach reduces the risks posed by overly tight constraints and inadequate (or missing) properties. This overlapping effort is justified by the fact that almost all bugs were found in the design phase itself and the FV proof frameworks provided us with a vehicle within which the fixes could be formally verified.

While the techniques outlined here, to render modules tractable under FV, are well known in the research world, they are seldom applied in the course of ASIC formal verification efforts and are hence worth emphasizing.

XIII. LIMITATIONS AND FUTURE WORK

Our approach relies on the verification engineer using design insights to come up with the right manual abstractions. This approach does risk bias particularly in light of the fact that commercial model checkers (that we know of) lack the means to prove ***soundness*** of abstractions or the means to express ***refinement maps*** (as can be done with SMV[5]).

To alleviate this risk, we made a deliberate attempt to keep our abstractions very simple (less than half a screen worth of verilog code per abstraction), and as a result have a high degree of confidence in our abstractions' soundness.

For the specific case of the ***Free List Manager*** abstraction, we reimplemented this abstraction within an SMV "layer" and proved its soundness, ensuring that for every path taken within the RTL component replaced, there exists at least one identical path within the abstract definition.

Most commercial model checkers do not possess the ability to verify data-independence in any automated way. We look forward to such features so that we can utilize them in the interest of completeness.

However, to put these concerns into practical perspective, we observe that these risks are no worse than other concerns, such as ensuring that DUT inputs are not over-constrained, ensuring that assertions correctly capture the specification's intent, etc.

ACKNOWLEDGMENT

We are very grateful to Ásgeir Eiríksson, the CTO of Chelsio Communications, for providing us with valuable insights pertaining to abstractions and to Jon Michelson for sharing his numerous detailed observations. We are also grateful to Vigyan Singhal, Prashant Arora and Paul Everhardt for providing feedback regarding various drafts of this paper.

REFERENCES

[1] Frederick P. Brooks, Jr. *The Mythical Man-Month*, Addison-Wesley Publications, 1995.

[2] *Incisive Formal Verifier User Guide*, Cadence Design Systems.

[3] Pierre Wolper. *Expressing interesting properties of program in propositional temporal logic*. In *Proceedings of the 13th ACM SIGACT-SIGPLAN symposium on Principles of programming languages, pages 184-193.* ACM Press, 1986.

[4] Ásgeir Eiríksson. *The Formal Design of 1M-gate ASICs*. In *Formal Methods in System Design, Vol 16, Issue 1 (Jan 2000), Special issue on formal methods for computer-aided system design*. Kluwer Academic Publishers.

[5] K. L. McMillan, *Getting started with SMV*, Cadence Berkeley Labs, 1999.

Formal Verification of Arbiters using Property Strengthening and Underapproximations

Gadiel Auerbach Fady Copty
IBM Haifa Research Laboratory, Haifa, Israel. e-mail: *gadiel,fadyc@il.ibm.com*
Viresh Paruthi
IBM Systems and Technology Group, Austin, TX, USA. e-mail: *vparuthi@us.ibm.com*

***Abstract*—Arbiters are commonly used components in electronic systems to control access to shared resources. In this paper, we describe a novel method to check starvation in random priority-based arbiters. Typical implementations of random priority-based arbiters use pseudo-random number generators such as linear feedback shift registers (LFSRs) which makes them sequentially deep precluding a direct analysis of the design. The proposed technique checks a stronger bounded-starvation property; if the stronger property fails, we use the counter-example to construct an underapproximation abstraction. We next check the original property on the abstraction to check for its validity. We have found the approach to be a very effective bug hunting technique to reveal starvation issues in LFSR-based arbiters. We describe its successful application on formal verification of arbiters on a commercial processor design.**

I. Introduction

Arbiters [4] are widely used in electronic systems such as microprocessors and interconnects. Arbiters restrict access to shared resources when the number of requests exceeds the maximum number of requests that can be satisfied concurrently. For example, an arbiter that regulates access to a bus selects which requestors would be granted access to the bus if there are more concurrent requests than the bus can handle. Arbiters use various arbitration schemes in the form of a priority function to serialize access to the shared resource by the requestors. The priority function decides which requestor to grant next. Examples of priority functions include round robin (rotate priority amongst requestors), queue-based (first-in first-out), or random priority (select next requestor randomly).

Random priority-based arbiters [8] have been gaining in popularity because of their high potential for fair arbitration, unlike other techniques such as round robin or queue-based which can be unfair because of their fixed order of arbitration. This arbitration scheme allows any request to have the highest priority at random. A random priority-based arbiter uses a pseudo-random number generator to select or influence the selection of the next requestor. A common implementation of such arbiters uses a Linear Feedback Shift Register (LFSR) [7] to generate a pseudo-random sequence of numbers. An LFSR is a cyclic shift register whose current state is a linear function of its previous state, and it generates a sequence of numbers which is statistically similar to a truly-random sequence. In this paper we focus on formal verification of such LFSR-based random priority arbiters.

The main concern in verification of an arbiter is checking for starvation. Starvation is a special case of liveness properties, in which any request must have a grant eventually. Liveness properties are often computationally hard to verify even on medium-sized designs. To alleviate this, it is common to check for starvation by replacing liveness properties with bounded properties – "request will be granted within N cycles", for some constant N. If a bounded property passes, it implies the correctness of the original liveness property. Even so, the sheer size of LFSR-based industrial arbiters may preclude an exhaustive analysis of the bounded property.

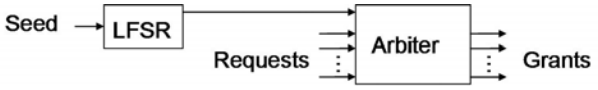


Figure 1. LFSR-based arbiter

We describe a method to uncover bugs leading to long latencies before requestors are granted in such complex arbiters. If the bounded property fails, we study the counter-example and attempt to either fix the problem by increasing the bound, or to use the information from the counter-example to underapproximate the original design. The concepts presented in this paper can be easily generalized to other schemes (besides LFSRs) to implement a random priority function. The presented technique can, in fact, be generalized to model checking of general-purpose systems, and we briefly present such a generalization.

II. LFSR-based arbiters

An LFSR-based arbiter grants access to a pending request based on the random number generated by the LFSR at any given point in time. Figure 1 shows a schema of an LFSR-based arbiter. An LFSR of length N generates a deterministic cyclic sequence whose period is $2^N - 1$, where all numbers from 1 to $2^N - 1$ are visited. The initial value of an LFSR is called the seed, and the sequence of numbers generated by the LFSR is completely determined by the value of its seed. An LFSR of length N may be used to arbitrate between M requestors, where $M \ll 2^N$, by sampling a subset $log(M)$ bits of the LFSR to select the next request to be granted. Such a scheme helps to amortize the cost of implementing an LFSR in hardware by way of the same LFSR serving multiple arbiters with different tap points. E.g., N may be 16, while M is 8 requiring 3-bits of the 16-bits of the LFSR to be tapped.

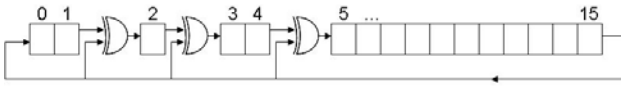


Figure 2. 16-bit LFSR

Figure 2 depicts a 16-bit LFSR from one of our case studies, the I_arbiter. The register shifts bits from left to right with some bits XORed with the most significant bit. The LFSR seed is configurable, and may be assigned any value between 1 and $2^{16} - 1 = 65535$. Formal verification environments typically assign a non-deterministic value to the seed.

III. Formal verification of LFSR-based arbiters

Verification of arbiters entails checking for starvation, which may be formulated as a liveness property. E.g., the following PSL [6] property specifies that whenever signal `request` is asserted, signal `grant` is asserted some time in the future.

```
always request -> eventually! (grant)
```

A counter-example for such a property is a trace showing a path leading to an infinite loop. In an LFSR-based arbiter, this constitutes a cycling through of all the valuations of the LFSR. The LFSR minimal loop length is $2^N - 1$, thus any loop showing a counter-example of the liveness property must be at least of that length. Hence, finding a trace for such a property of an LFSR-based arbiter is very hard. An easier yet more useful alternative to the above correctness property is to check for a request to be granted within a specified number of cycles, determined by the arbiter specification. In other words, we check to see if the request is granted within k cycles [8], [5]. In addition to verifying that a request is granted, such a formulation gives insights into the performance aspects of the arbiter, which is quite useful given the critical role arbiters play in the overall performance of electronic systems. The following property expresses a bounded-starvation condition.

```
always request-> next_e[1..k](grant)
```

Exhaustive verification of above properties to guarantee lack of bugs on is becoming increasingly challenging, if not impossible, for arbiters in real-world systems due to their sheer size and complexity. This calls for bug hunting methods to detect as many bugs as possible using scalable underapproximate techniques and (semi-) formal analysis. Such methods are more practical and provide concrete traces, rather than a suspicious bounded pass due to suspect abstractions.

Related work

As stated above, typical approaches to verify arbiters check for eventual grant of resources to the requests without much attention to performance aspects. Krishnan et. al. [8] studied starvation and performance of random priority-based arbiters extensively. They proposed a three-step verification process for computing an upper bound on the request-to-grant delay. In the first step they compute the maximum length Complete Random Sequence (CRS) comprising all random numbers (in the context of the sampled bits) the LFSR can assume. Next they compute the maximum number of CRSes needed for a request to be granted by the arbiter standalone with replacing the LFSR with a random-number generator. In the third step, the two values computed are combined to give the worst-case request-to-grant delay in clock cycles. A drawback of this method is the decoupling of the LFSR from the arbiter in the second step; a CRS can complete without being sampled by the arbiter. This produces a theoretical worst-case request-to-grant delay yielding very high bounds at times, much higher than the bounds stated in the model specification to be useful. Moreover, the trace produced by this technique is not representative of the overall system comprising the LFSR and the arbiter.

Our proposed technique compliments the above solutions by providing an effective bug hunting method for the actual LFSR-based arbiter, without any simplification thereof which may render the treatment (results) removed from the real logic. The effectiveness of the method has been proven on highly complex arbitration systems where it was leveraged to find real bugs. The method dynamically chooses between property strengthening and underapproximations in order to find a failure faster. The method can be easily generalized to create property-based underapproximations.

IV. Bug hunting in LFSR-based arbiters

The complexity of property checking is a function of the property and the design-under-test (DUT). Our bug hunting approach considers both the property and the DUT. In this section we describe how we construct easier-to-check under-approximate abstractions of LFSR-based arbiters.

Underapproximation and overapproximation techniques are commonly used to falsify properties or prove their correctness [3]. An abstract system is easier to check than the concrete system because it has fewer states and fewer transitions. Since our focus is bug hunting of safety properties we leverage underapproximations to obtain traces falsifying the property, which are then validated on the concrete/original model.

The seed of an N-bit LFSR may range between 1 and $2^N - 1$. The seed fully determines the LFSR sequence, so a run of the arbiter is based on one of $2^N - 1$ possible seeds. To underapproximate the arbiter we fix the LFSR seed by assigning it a constant N-bit number. A fixed-seed arbiter underapproximates the nondeterministic-seed arbiter as every run of a fixed-seed arbiter corresponds to a single LFSR sequence. If a bounded-starvation property fails in a fixed-seed arbiter then it definitely fails in the nondeterministic-seed arbiter; additionally, a counter-example that demonstrates a fail of a safety property in a fixed-seed arbiter is valid in the nondeterministic-seed arbiter. If a bounded-starvation property holds in a fixed-seed arbiter we cannot ascertain if it holds in the concrete system.

Falsification of a k-cycle-starvation property in an N-bit LFSR arbiter requires checking runs of depth k in a model that allows $2^N - 1$ possible LFSR sequences. Our method addresses the inherent hardness by alternating checking easier-to-check properties on the original system, and checking the original property on abstract systems. We iteratively check starvation with lesser bounds on the original system, and starvation with

the original bound on fixed-seed arbiters. We use property strengthening to seek interesting seeds that generate sequences that are likely to cause long starvation.

We define the following properties that express lower request-to-grant delays

$p_j \doteq$ `request -> next_e [1..j] (grant)`

for $1 \leqq j < k$. It is obvious that checking any of the properties p_j can be done in a shorter period of time than the original property. Clearly, every run that starves a request for k cycles starts with a starvation of j cycles, but a starvation of j cycles does not necessarily end with a starvation of k cycles. If a property p_j fails in the concrete system and a counter-example is generated, we underapproximate the arbiter by restricting it to the very same LFSR sequence that the counter-example reveals. Since LFSR sequences are determined by their seed it is enough to confine the arbiter's non-deterministic seed to the same seed that is exposed by the counter-example. Checking the fixed-seed arbiter is easier and likely to uncover a k-cycle long starvation.

Our method is outlined in Algorithm 1. We denote the original nondeterministic-seed LFSR arbiter by M, the maximal number of cycles allowed between a request and a grant as determined by the specification by k, and for some constant number c, we denote by $M[seed \leftarrow c]$ the arbiter M whose seed is the constant number c.

Algorithm 1 Checking bounded starvation on LFSR-based arbiters

1) check $M \stackrel{?}{\models} p$
2) **if** pass or fail **then return** result
3) $j_{min} \leftarrow 1; j_{max} \leftarrow k$
4) **while** ($j_{min} \lneq j_{max}$) **do**
   a) $j \leftarrow \lfloor \frac{j_{min}+j_{max}}{2} \rfloor$
   b) check $M \stackrel{?}{\models} p_j$
   c) **if** pass **then return** "pass"
   d) **if** timeout **then** $j_{max} \leftarrow j$
   e) **if** fail **then**
      i) $M_j \leftarrow M[seed \leftarrow seed_j]$; $j_{min} \leftarrow j$;
      ii) check $M_j \stackrel{?}{\models} p$
         A) **if** fail **then return** "fail"

The algorithm checks bounded starvation with different bounds and creates underapproximations of the original arbiter by initializing it with different seeds. We iteratively check property p_j with arriving at the next value of j using a binary search. If checking of a bounded-starvation property p_j times out, we next check another bounded-starvation property with a lower bound. If a property p_j fails, we extract the LFSR seed from the counter-example, denoted by $seed_j$. Next we restrict the arbiter's seed to $seed_j$, and check if the original property fails in the fixed-seed arbiter. If the property does not fail we narrow the seed space by checking a weaker property with a higher bound.

The algorithm halts after $\log(k)$ steps at the most. Let us examine an extreme case where all runs of the strengthened properties on the concrete model, $M \models p_j$, time out. This indicates that the arbiter is extremely complex and beyond the capabilities of our formal-verification tools. We note that the method is an effective bug hunting heuristic, but does not guarantee a bug free design, nor does it cover all LFSR seeds.

V. BUG HUNTING METHOD – A GENERALIZATION

We generalize the presented heuristic to general purpose model checking. The rationale is straightforward – check strengthened properties on the original model to aid in finding an efficient underapproximation for bug hunting on the original model. If any of the strengthened properties pass on the original model, it implies that the original property passes as well. If it fails then, heuristically, it has some information leading to a fail of the original property. This information can be extracted, and used to guide the search for a failure on the original property. This is achieved by defining an underapproximation of the model and checking for the validity of the property on it.

Intuitively, a safety property asserts that something bad never happens, while a strengthened property asserts that something "not-as-bad" never happens. Formally, for two properties p and q we say that property p is *stronger* than property q if $p \rightarrow q$. Consequently, given system M and two properties p and q such that p is stronger than q, we have $M \models p \rightarrow M \models q$, i.e., if p holds in M then q holds in M.

Falsification of a strengthened property tends to be easier than falsification of the original property because it defines more bad states in the system. If falsification of the original property is infeasible then we check a strengthened version of the property. If the strengthened property fails, we restrict the concrete system to the valuations provided by the obtained counter-example, and see if the original property fails.

It is not easy to determine how to strengthen a property in a useful manner. Hence, we restrict the discussion to a subset of properties whose strengthened versions enable an efficient and exhaustive search. A straightforward example for such properties is PSL parameterized properties that have a single parameter that serves as a sequence consecutive-repetition operator or as a bound of the next_e or next_a families of operators (formal definitions can be found in [1]). These widely-used operators are similar to the next_e operator used in our test case, and the practice of binary search over a bounded range of integers readily applies to them.

VI. EXPERIMENTAL RESULTS

The bug hunting method described in section IV has been used to verify several random priority-based arbiters used in an interconnect unit, and a router of a complex commercial processor. Table I shows the experimental results on 3 such industrial designs that use different types of random priority-based arbiters, and different LFSR sizes to generate pseudo-random numbers. The first arbiter, referred to as C_arbiter, is a command arbiter using a 32-bit LFSR. It arbitrates 27 requestors going to a single target. Its specification states the starvation bound to be 600 cycles. It uses a compound priority scheme combining LFSR-based arbitration and round robin to combinatorially compute the next granted requestor.

Design	*Random seed run time (h:m)*	*Fixed seed run time (h:m)*	*Vars before Redn*	*Gates before Redn*	*Vars after Redn*	*Gates after Redn*
C_arbiter	48:00 (Timeout)	8:56	2361	90397	812	7883
I_router	48:00 (Timeout)	21:09	104575	4223285	34070	1413519
I_arbiter	21:34	19:50	104575	4223285	30766	876328

Table I
RUN TIMES AND MEMORY USAGE FOR DIFFERENT ARBITERS

The second design, referred to as I_router, is a router of 56 requestors to 56 targets. The router is a more complex case of arbitration. It cannot starve an input from getting a request, and it cannot block an output from receiving a request. This router has a 16-bit LFSR, and it uses three of its bits for arbitration. It is a very large design with hundreds of thousands of variables (inputs and Flip-Flops) with multiple arbitration stages. The third arbiter, I_arbiter, is a simpler case of this router, with only one target available, thus checking arbitration only. The specification of I_router and I_arbiter requires a starvation bound of 1000 cycles.

All experiments were run on a 2x2.4GHz AMD dual core processor with 8 GB RAM memory, using IBM's RuleBase PE [2] and SixthSense [9] state-of-the-art industrial formal verification tools. The problem size is in term of gates and variables as reported by the RuleBase PE tool, shown before and after running RuleBase PE automatic model-size reductions. *Vars* denotes the numbers of registers and inputs.

For each of the designs we first applied the CRS technique [8]. The results yielded request-to-grant bounds higher than the starvation bounds in the specification. E.g., for the router arbiter it showed that the max length of CRS is 95 cycles; and we found that the request-to-grant delay is at least 50 CRSes – while trying to find a higher bound of 100, the tool timed out, implying a best case request-to-grant upper bound to be at least 4750 cycles.

Table I shows the run time of runs of the original property on fixed-seed arbiters that yielded traces (the last step in Algorithm 1). The various runs to compute an initial LFSR seed took anywhere from few minutes to 4 hours. We used parallel capabilities of our toolset to run a large number of rules with different starvation bounds, with a total run-time of 8 hours. The highest bounds on which the properties p_j failed were 375 for the C_arbiter and 687 for the I_arbiter. We gathered all LFSR seed values from the failing traces, seeded the LFSR of the original design with those, and ran the original formula. For benchmark purposes, the results above show the run time of RuleBase PE without using the parallel feature.

The verification timed out on the nondeterministic-seed runs of the C_arbiter, while a specification violation with a fixed seed was found in 9 hours. For the I_router design, the nondeterministic-seed runs timed out as well, while a trace for a fixed seed was obtained after 21 hours. As for the I_arbiter, the nondeterministic-seed finished in 21-1/2 hours while the fixed seed finished in 20 hours. In the I_router and I_arbiter designs the trace was found after the first run of algorithm 1, while on the C_arbiter the algorithm ran more than once and timeout increased for the run of stronger properties on the original model.

Clearly the fixed seed method shows a significant advantage on the more complex designs. It was able get past the huge complexity barrier of these designs. Note that even if the nondeterministic-seed runs were to finish easily, the initial state of the LFSR from these runs can be used as a seed for future runs that try to falsify proposed fixes. Another interesting fact was that the initial LFSR seed for the I_router and I_arbiter traces was different. In addition to finding the bounded starvation traces, our method was able to give us a large number of interesting traces which provided insights into the relationship between the LFSR and the arbiter.

VII. Conclusion and Future Work

We presented an effective method for computing smart property-based underapproximations. The technique dynamically converges on underapproximations which yield useful results in the form of bugs or interesting insights into the workings of the logic. This method has been successfully applied to LFSR-based arbiters and provided results which otherwise would not have been obtained with other techniques.

The described approach can be further generalized to other types of properties. Other directions include developing more general ways to construct underapproximations from counter-examples. The search for underapproximations can be improved by considering additinal seeds provided by the underlying decision procedure. The method can be enhanced further to be a proof-oriented approach by extracting reasons for pass results of the strengthened properties from the solving engines.

Acknowledgments

The authors would like to thank Alexander Ivrii and Hana Chockler for their helpful comments.

References

[1] *IEEE Standard for Property Specification Language (PSL). IEEE Std 1850*. 2010.
[2] S. Ben-David, C. Eisner, D. Geist, and Y. Wolfsthal. Model checking at ibm. *Formal Methods in System Design*, 22(2):101–108, 2003.
[3] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM*, 50(5):752–794, 2003.
[4] W. Dally and B. Towles. *Principles and Practices of Interconnection Networks*. Morgan Kaufmann Publishers, San Francisco, 2003.
[5] N. Dershowitz, D.N. Jayasimha, and S. Park. Bounded Fairness. *Lecture Notes in Computer Science*, 2772:304–317, 2004.
[6] C. Eisner and D. Fisman. *A Practical Introduction to PSL*. Integrated Circuits and Systems. Springer-Verlag, 2006.
[7] P. Horowitz and W. Hill. *The art of electronics*. Cambridge University Press, 2nd edition, 1989.
[8] K. Kailas, V. Paruthi, and B. Monwai. Formal verification of correctness and performance of random priority-based arbiters. In *FMCAD*, 2009.
[9] H. Mony, J. Baumgartner, V. Paruthi, R. Kanzelman, and A. Kuehlmann. Scalable automated verification via expert-system guided transformations. In *FMCAD*, pages 159–173, 2004.

SAT-Based Semiformal Verification of Hardware

Sabih Agbaria, Dan Carmi, Orly Cohen, Dmitry Korchemny, Michael Lifshits and Alexander Nadel
Intel Corporation, P.O. Box 1659, Haifa 31015 Israel
Email: (sabih.agbaria,dan.carmi,orly.cohen,dmitry.korchemny,michael.lifshits,alexander.nadel)@intel.com

***Abstract*—Semiformal, or hybrid, verification techniques are extensively used in pre-silicon hardware verification. Most approaches combine simulation and formal verification (FV) algorithms to achieve better design coverage than conventional simulation and scale better than FV. In this paper we introduce a purely SAT-based semiformal verification (SFV) method that is based on new algorithms for generating multiple heterogeneous models for a propositional formula. An additional novelty of our paper is the extension of the SFV algorithm to liveness properties. The experimental data presented in this paper clearly shows that the proposed method can effectively find bugs in complex industrial designs that neither simulation nor FV reveal.**

I. Introduction

Traditionally, Register Transfer Logic (RTL) level design validation is carried out by applying simulation techniques throughout the design and formal verification in certain high risk areas. In simulation, design behavior is checked with a large number of mostly random tests which cover just a small fraction of the design space. FV resolves the coverage issue by exhaustively checking all possible scenarios. It usually requires building a restricted environment and reduced model, as it cannot be directly applied on typical industrial-size designs. One of today's most efficient FV methods, SAT-based bounded model checking (BMC) [1], verifies the lack of bugs in scenarios of bounded length. The maximal reachable BMC bound is not sufficient in many cases to address structures with long latency, such as deep queues or counters.

Semiformal verification approaches developed throughout the last decade trade the completeness of FV for effectiveness. They aim to detect bugs in larger designs rather than to prove their correctness. Being incomplete, these approaches are sound — all reported violations of the properties are true bugs. SFV approaches that simultaneously apply multiple verification techniques in a complementary fashion are referred as *hybrid* approaches. Bhadra et al. [2] provide a comprehensive survey of recent advances in hybrid approaches to functional verification. A major challenge for hybrid tools is their practical applicability to a wide range of industrial designs and the soundness of the integration of the individual technique. As opposed to hybrid approaches, our SFV method is based on a single FV algorithm – SAT-based BMC.

Previous SFV approaches using a single FV algorithm suggested heuristics to search in a fraction of the original state space. This allowed reducing the binary decision diagrams (BDD) [3] to a manageable size in semiformal symbolic reachability analysis [4]–[6]. BDD-based algorithms, whose capacity is limited to hundreds of variables, are unsuitable for verifying properties in today's industrial designs, which often comprise tens of thousands of state elements. Cabodi et al [7] restrict the BMC SAT engine during the search based on dynamically computed simplified BDD-based image and preimage computations. The work in [8] suggests a rarity-based metric to identify states of a particular depth, searching from which leads to better coverage.

We use a different approach that utilizes user guidance to restrict the search within the state space - an idea extending the "lighthouses" used in the SIVA tool [9]. The user guides the search by providing a series of waypoints – describing design behavior throughout the desired scenario. The idea is similar to [10], but is used in the context of property verification rather than post-silicon debugging. Some works have suggested ways to automate the guiding algorithm, as they consider user guidance as a major drawback. For example, see probabilistic state ranking in [11] and lighthouse generation automation in [12], [13]. However, our experience shows that because verification engineers are well versed in the design, they can easily specify the required waypoints. Moreover, they usually prefer to encounter events they are familiar with when analyzing the resulting counterexamples.

There are other hybrid techniques that augment simulation with formal searches, as is done in KETCHUM [14], SIVA [9] and other systems [15], [16]. The biggest challenge for these tools is the synchronization of the simulation and FV environments. Random simulation needs to take into account the FV environment, which is usually modeled with complex sequential assumptions. Although this problem was partially addressed in [17], eventuality assumptions, assumptions involving internal or output signals, and assumptions requiring a *lookahead* (e.g. $G(a \rightarrow past(b))$) are very difficult or impossible to account for, thus resulting in false negative results. Another approach applies multiple shallow FV searches starting from selected cycles in simulation, a technique known as dynamic FV. Dynamic FV approaches suffer from an inherent drawback – they require tight coordination between the FV and simulation environments, which is extremely difficult to achieve, since in most cases FV is applied at a lower level of hierarchy than simulation. Moreover, the FV environment is usually restricted, allowing only a subset of functionalities, a fact which makes many simulation tests unusable.

Our SFV technique uses user guidance to compose several applications of purely SAT-based model checking, and explores the system state space in parts. It can be applied to all LTL properties, including liveness properties. We address the known problem that some waypoint states may not be extendable to the next waypoint. We introduce two new

highly configurable SAT-based algorithms for model sampling to generate different traces towards waypoints – necessary for achieving sufficient coverage and detecting corner-case bugs. This differs from previously suggested approaches, e.g. periodically tunneling or backtracking between shallow and deeper waypoints [13]. Our experimental results show the superior bug-finding ability of our approach, which detected critical bugs in industrial-scale designs that were "clean" from FV and simulation perspectives.

The rest of the paper is organized as follows. Section II describes the proposed BMC-based SFV algorithm. Section III introduces SAT-based algorithms for model sampling. Section IV is dedicated to semiformal verification of liveness properties. Our experiments are described in Sections V and VI, the first reviewing the test cases and the second summarizing the results. Conclusions and future work directions follow in Section VII.

We use a standard LTL notation for temporal properties: X for *next*, U for *until*, G for *always*, and F for *eventually* (see [18]). Instead of repeating X n times we use a shortcut notation X^n.

II. SAT-Based Semiformal Verification

A. Basic Algorithm

The verification time in BMC grows exponentially with the bound, and as a result it cannot explore scenarios that require many clock cycles to execute. The proposed semiformal verification algorithm applies multiple shallow BMC runs, trading the exhaustiveness of a search for speed. The user provides an ordered set of *waypoints* which direct the search engine towards the desired deep design state. The algorithm searches for a path from one waypoint to the next starting from the initial state, the BMC engine being restarted at each waypoint. Being familiar with the design behavior, users naturally direct the search towards the desired area by encoding the waypoints with cover points. For example, consider a queue that requires 200 clock cycles to be filled. To verify the design in a risky *"full queue"* state, possible waypoints could be *"1/4 full queue"*, *"1/2 full queue"*, *"3/4 full queue"*, each waypoint being easily reached and the overall verification time being but a fraction of the original BMC verification time.

The high-level SFV algorithm below is based on the fact that the properties may be represented with finite automata [18]. Another possibility for handling properties is to generate the satisfiability formula directly by the syntactic structure of the temporal assertion [19]. However, this algorithm is much less efficient than semantic translation based on automata [18], as shown in [20]; therefore we do not consider syntactic translation here.

Given a series of cover points $\xi_1, \xi_2, \ldots, \xi_n$ and the property φ, the algorithm performs the following steps:

1) Calculate the set of relevant assumptions for $\xi_1, \xi_2, \ldots, \xi_n$ and run BMC targeting ξ_1 from the set of initial states W_0.
2) If a witness has been found, the property automata are simulated along this witness. BMC and simulation are repeated each time using the end point of the last simulation as the new initial state, targeting consequent cover points $\xi_2, \ldots, \xi_n$. If a witness is not found for some ξ_i, an indeterminate result is reported.
3) Run BMC to determine whether φ holds. If there is a failure, append the counterexample to the concatenation of witnesses $\xi_1, \ldots, \xi_n$. If a timeout or required BMC bound is reached, report a lack of failure.

$$\begin{array}{l}\mathbf{init}\ q_1, \ldots, q_4 \leftarrow 0 \\ \mathbf{next}(q_1) \leftarrow a;\ \mathbf{next}(q_2) \leftarrow q_1;\ \ldots; \mathbf{next}(q_4) \leftarrow q_3 \\ fail \leftarrow \neg b \wedge q_4\end{array}$$

Fig. 1: RTL for assumption $G(a \rightarrow X^4 b)$

B. Calculation of New Initial States for Safety Properties

Since a safety property automaton can be synthesized into RTL [21], it may be simulated on the waypoint witness using a conventional RTL simulator. As an example, consider an assumption $G(a \rightarrow X^4 b)$. Its automaton may be synthesized as shown in Fig. 1.

If $a = 1$ in the witness appears in the next to last step, the initial state of the next BMC run should have $q_2 = 1$. Simulating the property automaton is important: blindly reusing the initial property condition $\mathbf{init}\ q_1, \ldots, q_4 \leftarrow 0$ would have led to the discontinuity of the adjacent BMC runs, and potentially to false negatives and bogus witnesses and counterexamples.

III. Using Multiple SAT Models to Enhance Coverage

A. Motivation and Related Work

The experiments conducted, described in Section VI, show that the proposed basic algorithm will likely miss corner-case bugs. The reason for this is that a randomly chosen path, constructed from a series of witnesses each of which satisfies the corresponding intermediate waypoint, does not exhibit sufficient coverage of the design space. Greater coverage may be achieved by advancing towards the desired deep state along multiple paths in parallel. For each intermediate waypoint, a heterogeneous set of witnesses is generated instead of a single witness, and for each such witness a separate verification process towards the next waypoint is launched. Consider Fig. 2 which illustrates a scenario where using two witnesses for the waypoints resulted in bug detection, whereas the chances of detecting the bug would have been much smaller otherwise.

A number of approaches to generating random witnesses (or solutions, or models) exist in literature. BDD-based, local-search-based, and arithmetic-based approaches such as [22], [23], and [24], respectively, are not applicable for our domain, since our test-cases are too complex for BDD-based and local-search-based algorithms, and they contain more bit-vector operations than arithmetical operations.

Modern efficient SAT solvers are able to solve complex formulas that arise in FV. SAT-based methods can also be used to sample the solutions of a given formula. One such method,

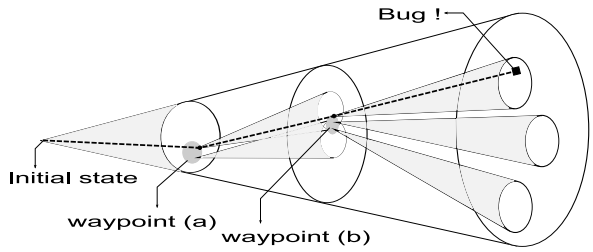


Fig. 2: Multiple witnesses

called XORSample, was proposed in [25]. XORSample invokes the SAT solver at least k times to generate k models. For each invocation, the initial formula is augmented with random XOR constraints. A sampling is not rejected only if the augmented formula has one and only one model. This requirement was relaxed in [26], whose version of XORSample does not reject samplings. Another SAT-based method, called DPLL-based sampling, was mentioned in [24] (we did not find any reference to a work introducing it). DPLL-based sampling invokes a SAT solver k times to generate k models on the same input formula. Model diversification is achieved by making the first boolean value assignment to a variable random for each invocation of the SAT solver.

Literature on the AllSAT problem (that is, the problem of finding all the models for a formula) is also relevant for our purposes. Most AllSAT engines are built on top of a SAT solver. When a model is found, a typical AllSAT solver [27], [28] adds a blocking clause which prevents the solver from rediscovering the same model in a subsequent search and restarts the search. Unlike DPLL-based sampling, AllSAT invokes a SAT solver only once.

B. SAT-Based Algorithms for Generating Multiple Witnesses

In this section we describe two new algorithms for generating heterogeneous models (witnesses) to a given formula: Rand-k-SAT and Guide-k-SAT. Both our algorithms surpass existing approaches in terms of both diversification quality (formally defined below) and performance. We also present two modifications to Rand-k-SAT and Guide-k-SAT, called AllSAT-sampling and BCP-aware Guide-k-SAT, which allow the user to trade diversification quality for performance.

Given a propositional formula F in conjunctive normal form (CNF) over variables $V = \{v_1, \ldots, v_n\}$, a SAT solver either finds a complete satisfying assignment (model) for F or proves that no model for F exists. We define the *distance* $D(\mu_1, \mu_2)$ between two partial assignments μ_1 and μ_2 to be the number of variables that are assigned in both μ_1 and μ_2 and have different values in μ_1 and μ_2. Note that our definition yields that the distance between two models is the Hamming distance. We define the *diversification quality* of k models $\mu_1 \ldots \mu_k$ $Q(\mu_1 \ldots \mu_k)$ to be the average distance between each pair of models, normalized by the number of variables: $Q(\mu_1 \ldots \mu_k) = (\sum_{i=1}^{k} \sum_{j=i+1}^{k} D(\mu_i, \mu_j))/(n(k^2 - k)/2)$.

For example, consider a formula $F = (a \vee b \vee c) \wedge (\neg a \vee b)$ and three models $\mu_1 = \{a = 1,\ b = 1,\ c = 0\}$, $\mu_2 = \{a = 1,\ b = 1,\ c = 1\}$, and $\mu_3 = \{a = 0,\ b = 0,\ c = 1\}$. Then, $D(\mu_1, \mu_2) = 1$, $D(\mu_1, \mu_3) = 3$, $D(\mu_2, \mu_3) = 2$, and $Q(\mu_1, \mu_2, \mu_3) = (1+2+3)/(3\times((3^2-3)/2)) = 2/3$. Note that since the diversification quality is normalized by the number of variables, it must lie between 0 and 1.

Given a propositional formula F in CNF and an integer number $k > 0$, we are interested in finding k models for F with the optimization goal of increasing the diversification quality of the models. We do not intend to guarantee a certain quality in a theoretical sense, but rather to combine solid performance with a good model quality for the practical needs of efficient semiformal verification.

Both our approaches, Rand-k-SAT and Guide-k-SAT, invoke the SAT solver only once, like AllSAT solvers do. However, we do not add blocking clauses when models are discovered. Instead, the solver restarts the search after a model is discovered. Diversification is achieved solely by changing the phase selection heuristic for variables.

The decision stage of a modern SAT solver chooses a variable and its phase at each decision point during the search. The variable decision heuristic selects a variable. The phase selection heuristic selects a boolean value for the selected variable. Most modern SAT solvers use RSAT solver's phase selection heuristic [29], which tries to refocus the search on subspaces that the solver has knowledge about. This heuristic keeps a saved-phase array, indexed by variables. The array contains boolean values and is initialized with 0's. The solver stores the last assignment given to a variable in the saved-phase array. The phase selection heuristic for variable v always chooses the value of v from the saved-phase array.

Both Rand-k-SAT and Guide-k-SAT override the traditional phase selection heuristics. However, they differ from one another conceptually in their phase selection strategies. Rand-k-SAT selects the phase randomly on all occasions. Guide-k-SAT selects the polarity in a non-random manner: explicitly guides the solver to extend its partial assignment σ so that the distance between σ and previous models $\mu_1, \ldots, \mu_{n-1}$ will be as large as possible. We designed this strategy keeping in mind the goal of making the distance between the next model μ_n and the previous models as large as possible. More specifically, Guide-k-SAT uses the following greedy approach. Suppose a variable v is selected by the variable decision heuristic. Let $p(v)/n(v)$ be the number of times v was assigned 1/0 in previous models. If $p(v) > n(v)$, v is assigned 0; if $p(v) < n(v)$, v is assigned 1; if $p(v) = n(v)$ (including the case where no models have yet been identified), v is assigned a random value.

The ideas behind Rand-k-SAT and Guide-k-SAT are very simple and straightforward to implement, yet they turn out to be powerful and efficient for finding heterogeneously distributed models on well-structured problems, with an acceptable performance overhead compared to a modern SAT solver. On the one hand, we continue using all the modern SAT strategies, whose goal is to achieve solid performance on structured instances. On the other, we achieve sufficient diversification quality, either by selecting the phase randomly

TABLE I: Comparing Approaches to Generating Heterogeneous Models.

	DbS	Rand-k-SAT	Guide-k-SAT
Mean Quality	0.215	0.313	0.339
Overall Run-Time	47456	30307	28450

TABLE II: Trading Quality for Run-Time in Heterogeneous Model Generation.

	AllSAT-sampling	BaG; T=100	BaG; T=100000
Mean Quality	0.124	0.342	0.353
Overall Run-Time	8211	33392	183857

or by explicitly guiding the solver away from previous models.

DPLL-based sampling (DbS) is the best previous SAT-based approach to finding heterogeneous models. We implemented DPLL-based sampling as well as our algorithms Rand-k-SAT and Guide-k-SAT, and compared them experimentally on 66 benchmarks. The number of propositional clauses in the benchmarks varies from eight thousand to more than three million. In all the experiments, the required number of models was 10. All experiments were carried out on a machine with 4Gb of memory and two Intel Xeon CPU 3.60 processors. All the algorithms were implemented in the latest version of Intel's Eureka SAT solver. Eureka's default phase selection heuristic is RSAT's heuristic.

Table I compares DPLL-based sampling (DbS), Rand-k-SAT, and Guide-k-SAT in terms of mean diversification quality and overall run-time. Two scatter plots, comparing our best algorithm, Guide-k-SAT, and DPLL-based sampling in terms of run-time and quality are provided in Fig. 3. Similar scatter plots, comparing Guide-k-SAT and Rand-k-SAT, appear in Fig. 4. Our experiments yield two main conclusions.

First, both our algorithms are clearly preferable to DPLL-based sampling in terms of both quality and run-time. Table I confirms the overall advantage. Consider now the the right-hand scatter plot of Fig. 3 comparing the quality of Guide-k-SAT and DPLL-based sampling. A significant number of dots appear near the x-axis, far away from the diagonal, hinting that the gap is significant for some of the benchmarks. Now consider the run-time comparison scatter plot to the left. Guide-k-SAT outperforms DPLL-based sampling on most of the most difficult instances.

Second, Guide-k-SAT outperforms Rand-k-SAT in terms of both quality and run-time. The gap in run-time is not so significant: it stands at 6.5% overall. Also, the run-time comparison scatter plot in Fig. 4 shows that Guide-k-SAT is not always preferable to Rand-k-SAT. Now consider diversification quality. While the gap between average quality is not large, the quality comparison scatter plot clearly shows that Guide-k-SAT yields better diversification quality on every one of the benchmarks. Hence, for our examples, to achieve better performance and model diversification it is preferable to explicitly guide the SAT solver away from previous models (using Guide-k-SAT) than to use randomness (using Rand-k-SAT).

It is also possible to modify our algorithms to trade quality for run-time. Consider a variation of Rand-k-SAT, called *AllSAT-sampling*, that invokes the SAT solver once, but assigns random values only to variables selected for the first time or for the first time after a restart. Note that the solver is expected to keep assigning the same values to the variables for some restricted time after the beginning of the search or a restart due to RSAT's phase selection heuristic. A comparison of Table I and Table II shows that AllSAT-sampling is much faster than both Guide-k-SAT and Rand-k-SAT; however, the distribution quality is significantly worse. Accordingly, AllSAT-sampling can be recommended when the problem is computationally very complex.

Consider now a variation of Guide-k-SAT, called *BCP-aware Guide-k-SAT*. BCP-aware Guide-k-SAT tries to take into consideration the impact of Boolean Constraint Propagation (BCP) on the distance between the current partial assignment and the previous models. It performs BCP for both polarities, and measures the distance between the resulting partial assignments σ and previous models. Eventually, it picks the polarity that yielded the larger distance.

Specifically, the algorithm operates as follows. Suppose a variable v is selected by the variable decision heuristic. Let $p(v)/n(v)$ be the number of times v was assigned 1/0 in previous models. The variable v is assigned a value p as follows: if $p(v) > n(v)$, p is 1; otherwise p is 0. Then, BCP is carried out. Suppose that the set of variables V_p is assigned as a result of BCP. The algorithm saves the distance D_p between the partial assignment, induced by $\{v\} \cup V_p$, and the previous models. Afterwards, the algorithm unassigns $\{v\} \cup V_p$, assigns v the value $\neg p$, and propagates it using BCP. Suppose now that the set of variables $V_{\neg_p}$ is assigned as a result of BCP. The algorithm calculates the distance $D_{\neg p}$ between the partial assignment, induced by $\{v\} \cup V_{\neg p}$, and the previous models. If $D_{\neg p} > D_p$, the algorithm continues to the next decision. Otherwise, it unassigns $\{v\} \cup V_{\neg p}$, assigns v the value p, propagates using BCP, and continues to the next decision. Note that the algorithm first tries the polarity p that is less likely to result in better distance. The reasons is that if $\neg p$ is preferable, BCP is performed only twice; otherwise it is performed three times.

BCP-aware Guide-k-SAT is a costly algorithm, since it has to perform BCP two or three times per decision. Hence we limit its usage as follows. BCP-aware Guide-k-SAT is used until a certain number of conflicts T is encountered by the SAT solver. In addition, BCP-aware Guide-k-SAT is reinvoked after each model is discovered until T conflicts are encountered. The algorithm then uses plain Guide-k-SAT until the next model is encountered. Table II shows that BCP-aware Guide-k-SAT (BaG) improves distribution quality, but deteriorates run-time. Observe that it is possible to trade quality for run-time by changing T.

We also implemented XORSample [25] as well as the modified XORSample of [26]. We tried a variety of distribution quality values $(0.1, 0.01, \ldots, 0.0000001)$ and the number of generated XOR constraints $(1000, 10000, \ldots)$. Our results show that, depending on the configuration, XORSample is

either slower by an order of magnitude compared to Rand-k-SAT and Guide-k-SAT (it timed-out on most of the instances), or its distribution quality is worse by approximately 10 times compared to Rand-k-SAT and Guide-k-SAT. Hence, although XORSample is useful on randomly generated instances and on small real-world formulas when a large number of models needs to be generated, it is inferior to other methods on difficult benchmarks when a small number of models needs to be generated.

Our experience shows that the best approach for generating multiple counterexamples in the framework of semiformal verification is to allow the user some control over the algorithm used within the tool. As our experimental results demonstrate, Guide-k-SAT is preferable as the default algorithm, since it exhibits the most attractive trade-off between run-time and solution diversification quality (which translates to efficient verification). However, we encountered a number of especially difficult cases where AllSAT-sampling was mandatory in order to satisfy performance requirements. In those cases, AllSAT-sampling was 25X faster than Guide-k-SAT (1 hour versus 25 hours to generate 10 models), although the diversification quality was 1.7X worse (0.181 versus 0.307). For easy test cases we recommend using BCP-aware Guide-k-SAT, where the trade-off between run-time and solution diversification quality is controlled by the threshold T.

IV. Checking Liveness Properties

A. Motivation

To the best of our knowledge, no attempt at semiformal verification of liveness properties has ever been described in the literature. We do not restrict our consideration to pure liveness properties, and by "liveness" we understand everywhere general liveness. Verifying liveness properties is required when the exact timing in end-to-end properties is not specified, and to check the absence of starvation. FV of liveness properties without prior aggressive abstraction is challenging: the complexity of their BMC-based verification is significantly more expensive than the verification of safety properties. Therefore the ability to perform semiformal verification of liveness properties is important.

One possible way of handling liveness properties would to convert them to equivalent safety properties, as explained in [30]. However, this approach is problematic in the semiformal verification context for the following reasons: 1) The number of property variables doubles when transforming a liveness property into a safety property, and 2) This translation makes sense when the resulting safety property is exhaustively checked. Therefore we did not explore this option in our work.

It is well known [31] that a violated liveness property always has a lasso-shaped counterexample: a state path consisting of a linear prefix and a loop. As explained in [32], in BMC of liveness properties these lasso-shapes paths are described with Boolean formulas parameterized by the size of the prefix and of the loop. SFV may help get to a design state close to the beginning of the loop, and/or to a neighborhood of a smaller loop. For example, to check starvation, it is necessary to bring the system into a state where resources have been requested by several clients. Applying BMC directly from the initial state is useless if the greatest feasible bound is insufficient to bring the system to such a state.

To apply classical algorithms based on semantic translations to check liveness properties in semiformal verification, the main challenge is to simulate their automata along the waypoint witness. Application of the algorithm proposed below is not limited to BMC-based semiformal verification; it may also be combined with other semiformal methods such as those described in [14], [15], [33].

B. Simulation of Non-deterministic Büchi Automata

Liveness properties cannot be represented as finite automata on finite words, and for their representation a finite automaton on infinite words (a so called Büchi automaton) is needed [18]. In practice it is more convenient to represent LTL properties with a more general form of Büchi automata — alternating Büchi automata [18]. For the sake of simplicity we describe our algorithm for regular (nondeterministic) Büchi automata only, but with minimal changes the same method may be applied to alternating Büchi automata as well. Unlike safety property automata, Büchi automata representing liveness properties are simulated symbolically, as described below.

In our algorithm we use a symbolic representation of the transition relation as a Boolean function of two sets of variables, current (unprimed) and next (primed) [19]: $\delta(w, w')$. We also introduce a map $\beta : w' \mapsto w$ to convert functions of next variables to functions of current variables. For example, $\beta(a' \wedge b') = a \wedge b$.

Let U_i be a symbolic representation of the states reachable at step i (*active states*) from one of the initial states while respecting the given witness. For the witness of the first waypoint, $U_0 = Q_0$ — the set of initial states of the automaton. For other witnesses U_0 is the symbolic representation of the endpoint of the automaton simulation along the previous waypoint witness. Let V_i be the set of pairs (w, w'), where $w \in U_i$ is a current active state, and w' is the next state reachable from w according to the transition relation δ, respecting the limitations imposed by the witness a_i at step i: $V_i = U_i \wedge \delta \wedge a_i$. The next variables computed this way become current variables for the next step, and the process is repeated: $U_{i+1} = \beta(\exists w.V_i)$. In this formula the existential quantifier selects the member w' of the pair $(w, w') \in V_i$.

We will illustrate this algorithm on the Büchi automaton in Fig. 5 for a 4-cycle long witness trace shown in Table III. The symbolic transition relation $\delta = \bigwedge_{i=0}^{4} \delta_i$, where

$$\delta_0 = q_0 \rightarrow q'_0 \vee \neg a \wedge q'_1 \vee a \wedge q'_2$$
$$\delta_1 = q_1 \rightarrow \neg a \wedge q'_1 \vee a \wedge q'_2$$
$$\delta_2 = q_2 \rightarrow q'_3$$
$$\delta_3 = q_3 \rightarrow \neg b \wedge q'_3 \vee b \wedge q'_4$$
$$\delta_4 = (q_4 \rightarrow q'_4)$$

The values of U_i and V_i are shown in Table III. As expected, the values of U_i contain symbolic representation of the active states of the automaton at each simulation step. The initial state of the next BMC run should have $q_0 \vee q_2 = 1$.

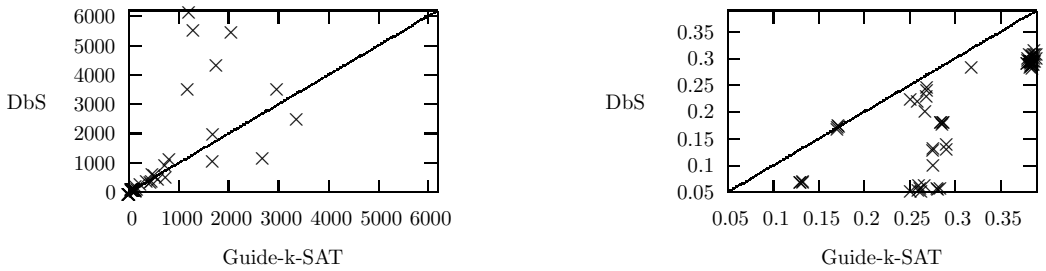


Fig. 3: Comparing the Run-Time in Seconds (on the Left) and the Quality (on the Right) of Guide-k-SAT (x-axis) vs. DPLL-based Sampling (y-axis)

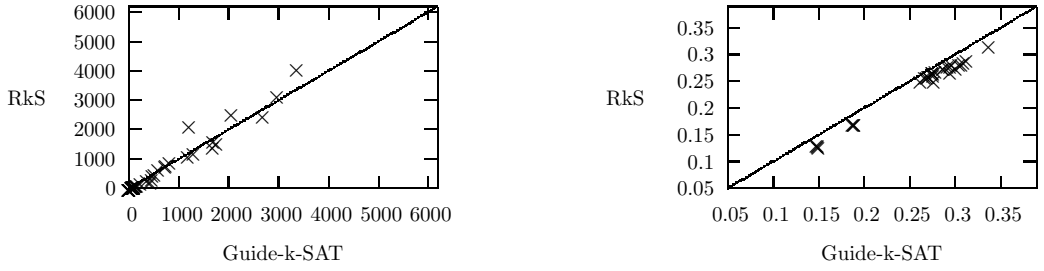


Fig. 4: Comparing the Run-Time in Seconds (on the Left) and the Quality (on the Right) of Guide-k-SAT (x-axis) vs. Rand-k-SAT (y-axis)

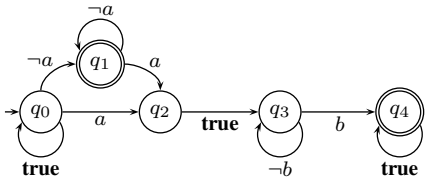


Fig. 5: Büchi automaton with accepting states q_1 and q_4

TABLE III: Simulation of Büchi automaton

Time	a	b	U_i	V_i
0	0	0	q_0	$q_0 \wedge \neg a \wedge \neg b \wedge (q_0' \vee q_1') \wedge \bigwedge_{i=1}^{4} \delta_i$
1	0	0	$q_0 \vee q_1$	$(q_0 \vee q_1) \wedge \neg a \wedge \neg b \wedge (q_0 \rightarrow q_0' \vee q_1') \wedge (q_1 \rightarrow q_1') \wedge \bigwedge_{i=2}^{4} \delta_i$
2	1	0	$q_0 \vee q_1$	$(q_0 \vee q_1) \wedge a \wedge \neg b \wedge (q_0 \rightarrow q_0' \vee q_2') \wedge (q_1 \rightarrow q_2') \wedge \bigwedge_{i=2}^{4} \delta_i$
3	0	0	$q_0 \vee q_2$	—

V. Test Cases

We implemented the algorithm in Intel's proprietary FV tool and chose three CPU design blocks for our experiments. These design blocks had been extensively tested in simulation and the design was believed to be mature. The blocks were modeled in SystemVerilog and included novel features carrying high risk. The properties were captured using SystemVerilog Assertions (SVA). We chose blocks of sizes that SAT-based FV engines could handle — the full cone of influence of a typical assertion comprised 1K inputs, 5K state elements, and 75K gates. As a result, the FV confidence level was not high enough in all test cases, as the BMC bound reached by the traditional BMC approach was not sufficient. In most cases, after reducing the models using both manual and automatic techniques, design scenarios requiring more than forty clock cycles could not be addressed. It is worth noting that our attempts to apply industrial semiformal verification tools yielded no tangible results. This was due to complex environments that needed to be synchronized and to the unique properties and large size of the CPU design blocks.

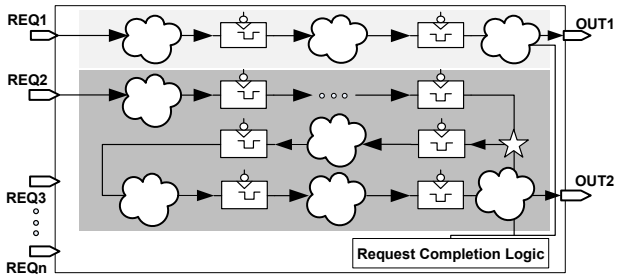


Fig. 6: Request Tracker

The first block, a Request Tracker, is responsible for managing various request types and ensuring the correct execution order of the requests, giving preference to high-priority requests while not starving low-priority requests. Requests arrive from various sources, and each is associated with a unique identifier (ID). A high-level diagram of Request Tracker is shown in Fig. 6.

The different request types vary in the time needed to process them, e.g. a $REQ1$ request (path $REQ1$ — $OUT1$) requires considerably fewer clock cycles than a $REQ2$ request (path $REQ2$ — $OUT2$). We chose to experiment with $REQ2$, which had not been properly addressed in FV due to BMC bound limitations.

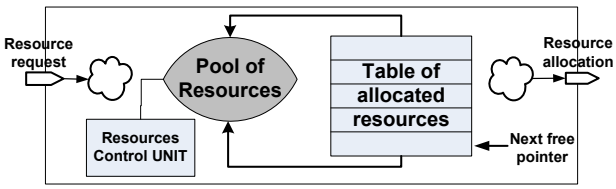


Fig. 7: Resource Manager

The second block, a Resource Manager, is responsible for controlling resources and making sure that no resource is allocated twice and that none are lost. The resources are kept in the pool and allocations/deallocations are recorded using a cyclic table. See Fig. 7 for a high-level diagram of the block.

The third block, a Flow Manager, implements a mechanism to control a complex flow involving many agents. It comprises a central FSM with additional smaller FSMs around it, each being responsible for a specific flow scenario or sending and receiving data from a certain agent. The central FSM controls the flow, supervising all the smaller FSMs around it. This block was used for algorithm experiments with liveness properties, as the main concern is that the flow will eventually finish successfully without getting stuck in live-lock due to a bug in one of the FSMs.

VI. Results

In this section we describe the results of applying the proposed SFV algorithm to the RTL blocks described in Section V. The most important result was the exposure of three real corner-case bugs described in Section VI-A. We also inserted several artificial corner-case bugs described in Section VI-B. We sought to corroborate different characteristics of the algorithm, namely its ability to adequately cover design state space using the multiple witness approach.

A. Real Corner-Case Bugs in Mature Designs

Our SFV algorithm revealed the following bugs in the Resource Manager, two of them critical. These bugs could be revealed neither in simulation, nor using traditional FV, nor using SFV with a single witness.

- Incorrect STALL calculation in a very specific combination of allocation requests, which causes resources to be lost.
- A bug in recovery/restart event handling which results in not all of the allocated resources being correctly sent back to the resource pool.
- Corruption, in a scenario involving extremely high allocation traffic, of a mechanism which validates resource integrity in the Resource Control Unit.

B. Testing the Ability to Adequately Cover Design State

We inserted an artificial corner-case bug in the Request Completion Logic sub-block which causes a failure when multiple $REQ2$ type requests from particular sources and ID ranges arrive in a particular order. The bug results in one of the requests being incorrectly marked as completed. This bug could not be revealed with the simulation regression.

TABLE IV: Resource Manager Verification Results

CP/Asrt	BMC		SFV, single		SFV, multiple	
	Result	Bound	Result	Bound	Result	Bound
Line 4	covered	69	covered	69	covered	69
Line 8	covered	71	covered	77	covered	77..83
Line 12	uncov.	24	covered	89	covered	85..95
Line 16	uncov.	26	covered	99	covered	93..107
Line 19	uncov.	26	covered	113	covered	99..119
Line 0	N/A	N/A	covered	129	covered	107..133
Asrt	TO	38	TO	42	failed	142

We used nine different waypoints modeling several $REQ2$ requests in various pipe stages on a path $REQ2$ — $OUT2$; for example, the one marked by a star in Fig. 6. In this and other experiments we used *general* waypoints (waypoints previously defined by validation engineers for other purposes) in order to eliminate the possibility that prior knowledge about the bugs might lead us unconsciously to craft waypoints leading directly to them. For each waypoint we calculated 5 witnesses, targeting each of the twelve assertions from 9×5=45 different initial states defined by these witnesses. The cover points occurred at bounds 64–70, and verification took 1406–3379 seconds (on a machine with 4Gb memory and two Intel Xeon CPU 3.60 processors). A failure was detected by one out of 12 assertions from only one initial state, whereas runs from the other 44 initial states missed the problematic scenario. It occurred at bound 34 (70+34=104 clock phases from the original initial state) after 14707 seconds.

We inserted an artificial corner-case bug into the Resource Manager logic which calculates the condition for next request $STALL$. This caused *Next free pointer* to wrap around early due to illegal allocation, thereby running over other resources in the table. We used general cover points as waypoints asserting that table lines were allocated, and the table was incrementally filled up until the wraparound. We ran traditional BMC and SFV with single as well as multiple witnesses. The assertion verified that resources were not being lost in the system. In all cases a timeout of 20 hours was used. Results are summarized in Table IV.

A wraparound happens after the 19th table line is allocated, as the cyclic allocation table size is 20. BMC could not get beyond the allocation of line 8, and the multiple witness approach was needed in order to come across the problematic combination of resource requests. The total number of verification runs was 3(witnesses)$^{6(\text{waypoints})}$=729. Note that the SFV algorithm does not necessarily produce the shortest counterexample — line 8 was reached with bound 71 using BMC whereas using SFV it was reached with bound 77 to 83.

We experimented with liveness properties in the Flow Manager block. The properties validate forward progress with the control FSM (dispatcher), eventually reaching predefined control points without getting stuck, e.g. due to a bug in one of the FSMs. The proof assumes the legal behavior of the surrounding agents. We used waypoints describing the

state transitions of the dispatcher FSM. Although we did not find any real design bugs, we validated the correctness of the algorithm by properly detecting a known deep bug using our approach. The failure was detected faster: 1575 seconds (509 seconds towards the waypoint and 1064 seconds to get a counterexample) vs. 5470 seconds for traditional BMC (3.5X faster). This is due to the run-time reduction phenomenon described in Section II-A.

VII. Conclusion and Future Work

The method suggested in this paper for pure SAT-based semiformal verification is very simple to grasp and straightforward to implement, yet it exhibits a superior ability to achieve good design coverage and detect deep, corner-case bugs in industrial-scale designs. The experimental results confirm this by exposing both real and artificial design bugs missed by simulation (due to coverage limitations) and classic FV (due to bound limitations). These encouraging results were achieved with a relatively small amount of work on the part of the validation engineers, much less than the effort required by the traditional FV and simulation approaches applied prior to our experiments. Moreover, the suggested method can save the substantial effort usually invested in reducing designs to fit the capacity limitations of FV tools, as it can replace such activities.

As a by-product, we developed two SAT-based algorithms, Rand-k-SAT and Guide-k-SAT, that are able to efficiently find a number of heterogeneous models for a given problem. We also discuss variations of Rand-k-SAT and Guide-k-SAT that allow the user to achieve the desired balance between performance and solution diversification quality. We have also proposed an extension of the semiformal verification algorithm for liveness properties.

In our future work we intend to study how different diversification techniques affect bug detection capabilities and to collect more experimental data on semiformal verification of liveness properties to better understand the practical utility of this technique.

Acknowledgments

The authors would like to thank Vadim Ryvchin, Paul Inbar, Roy Frank, Tamir Salus, Yael Zbar, Zurab Khasidashvili, Asi Sapir, Haim Kerem, Dani Even-Haim, Amit Palti, Eli Singerman, and Alon Flaisher for their valuable suggestions, ideas, experimental results, reviews, and support of our work.

References

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

[2] J. Bhadra, M. S. Abadir, L.-C. Wang, and S. Ray, "A survey of hybrid techniques for functional verification," in *IEEE Design and Test of Computers*, vol. 24, 2007, pp. 112–123.

[3] K. L. McMillan, *Symbolic Model Checking*. Norwell, MA, USA: Kluwer Academic Publishers, 1993.

[4] K. Ravi and F. Somenzi, "High density reachability analysis," in *ICCAD*, 1995.

[5] J. Yuan, J. Shen, J. A. Abraham, and A. Aziz, "On combining formal and informal verification," in *CAV*, 1997, pp. 376–387.

[6] R. Fraer, G. Kamhi, B. Ziv, M. Y. Vardi, and L. Fix, "Prioritized traversal: Efficient reachability analysis for verification and falsification." in *CAV*, 2000, pp. 389–402.

[7] G. Cabodi, S. Nocco, and S. Quer, "Improving sat-based bounded model checking by means of bdd-based approximate traversals," in *DATE*, 2003, pp. 10 898–10 905.

[8] M. K. Ganai and A. Aziz, "Rarity based guided state space search," in *GLSVLSI*, 2001, pp. 97–102.

[9] M. Ganai, P. Yalagandula, A. Aziz, A. Kuehlmann, and V. Singhal, "Siva: A system for coverage-directed state space search," *J. Electron. Test.*, vol. 17, no. 1, pp. 11–27, 2001.

[10] C. R. Ho, M. Theobald, B. Batson, J. Grossman, S. C. Wang, J. Gagliardo, M. M. Deneroff, R. O. Dror, and D. E. Shaw, "Post-silicon debug using formal verification waypoints," in *DVCon*, 2009.

[11] A. Kuehlmann, K. L. McMillan, and R. K. Brayton, "Probabilistic state space search," in *ICCAD*, 1999, pp. 574–579.

[12] P. Yalagandula, V. Singhal, and A. Aziz, "Automatic lighthouse generation for directed state space search," in *DATE*, 2000, pp. 237–242.

[13] P. Bjesse and J. H. Kukula, "Using counterexample guided abstraction refinement to find complex bugs," in *DATE*, 2004, pp. 156–161.

[14] P. H. Ho, T. Shiple, K. Harer, J. Kukula, R. Damiano, V. Bertacco, J. Taylor, and J. Long, "Smart simulation using collaborative formal and simulation engines," in *ICCAD*, 2000, pp. 120–126.

[15] A. Aziz, J. Kukula, and T. Shiple, "Hybrid verification using saturated simulation," in *DAC*, 1998, pp. 615–618.

[16] D. L. Dill and C. H. Yang, "Validation with guided search of the state space," in *DAC*, 1998, pp. 599–604.

[17] E. Cerny, A. Dsouza, K. Harer, P.-H. Ho, and T. Ma, "Supporting sequential assumptions in hybrid verification," in *ASP-DAC*, 2005, pp. 1035–1038.

[18] M. Y. Vardi, "An automata-theoretic approach to linear temporal logic," in *Proceedings of the VIII Banff Higher order workshop conference on Logics for concurrency: structure versus automata*, 1996, pp. 238–266.

[19] A. Biere, A. Cimatti, E. M. Clarke, M. Fujita, and Y. Zhu, "Symbolic model checking using SAT procedures instead of BDDs," in *DAC*, 1999, pp. 317–320.

[20] E. M. Clarke, D. Kroening, J. Ouaknine, and O. Strichman, "Computational challenges in bounded model checking," *Journal of Software Tools and Technology Transfer*, vol. 7, no. 2, pp. 174–183, 2005.

[21] R. Armoni, S. Egorov, R. Fraer, D. Korchemny, and M. Vardi, "Efficient LTL compilation for SAT-based model checking," in *ICCAD*, 2005.

[22] J. Yuan, K. Albin, A. Aziz, and C. Pixley, "Simplifying boolean constraint solving for random simulation-vector generation," in *ICCAD '02*, 2002, pp. 123–127.

[23] W. Wei, J. Erenrich, and B. Selman, "Towards efficient sampling: Exploiting random walk strategies," in *AAAI*, 2004, pp. 670–676.

[24] N. Kitchen and A. Kuehlmann, "Stimulus generation for constrained random simulation," in *ICCAD*, 2007, pp. 258–265.

[25] C. P. Gomes, A. Sabharwal, and B. Selman, "Near-uniform sampling of combinatorial spaces using XOR constraints," in *NIPS*, B. Schölkopf, J. C. Platt, and T. Hoffman, Eds., 2006, pp. 481–488.

[26] S. Plaza, I. L. Markov, and V. Bertacco, "Random stimulus generation using entropy and XOR constraints," in *DATE*, 2008, pp. 664–669.

[27] K. L. McMillan, "Applying SAT methods in unbounded symbolic model checking," in *CAV*, 2002, pp. 250–264.

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

[29] K. Pipatsrisawat and A. Darwiche, "A lightweight component caching scheme for satisfiability solvers," in *SAT*, 2007, pp. 294–299.

[30] A. Biere, C. Artho, and V. Schuppan, "Liveness checking as safety checking." *Electr. Notes Theor. Comput. Sci.*, vol. 66, no. 2, 2002. [Online]. Available: http://dblp.uni-trier.de/db/journals/entcs/entcs66.html#BiereAS02

[31] E. M. Clarke, O. Grumberg, and D. A. Peled, *Model Checking*, 6th ed. MIT Press, 2008.

[32] M. K. Ganai, A. Gupta, and P. Ashar, "Beyond safety: customized sat-based model checking," in *DAC '05: Proceedings of the 42nd annual Design Automation Conference*. New York, NY, USA: ACM, 2005, pp. 738–743.

[33] M. K. Ganai, A. Aziz, and A. Kuehlmann, "Enhancing simulation with BDDs and ATPG," in *DAC*, 1999, pp. 385–390.

DFT Logic Verification through Property Based Formal Methods – SOC to IP

Lopamudra Sen, Amit Roy, Supriya Bhattacharjee,
Bijitendra Mittra
Interra Systems India Pvt. Ltd.
Bangalore, India

Subir K. Roy
Texas Instruments (India) Pvt. Ltd.,
Bangalore, India

***Abstract* —** System On Chips (SOCs) are being increasingly deployed in large number of applications and systems as they allow automation to be implemented to render ease and convenience in many human activities, a prime example being smart mobile phones. This renders their design implementation a fairly difficult task - with larger product space and product revisions, comes the requirement for larger feature integration in smaller die-sizes, smaller design turnaround times and lower power consumption. To address these issues, SOCs are being designed by integrating existing in house Intellectual Properties (IPs), or third party IPs provided by external vendors.

DFT logic integration is an important design activity in any SOC design implementation, which gets carried out almost as a background activity, while not being accorded the due importance given to the prominent front-end design activity related to implementing functional features in the design of any SOC. Integration of DFT logic and the verification of this integration to other functional sub-systems and IPs in a SOC constitutes a significant portion of the overall design and verification effort. Any savings in this component helps in reducing the overall chip design and verification time and therefore, the cost. This is achievable through automation. The predominantly canonical and regular nature of the structures and behavior of most DFT IPs facilitates this, leading to the kind of convergence presently seen towards standardized configurable DFT logic architectures. Such standardized configurable DFT logic architectures lend themselves to auto-generation of their RTLs with ease. In addition, this feature enables high re-usability at different levels of hierarchy in any SOC design because similar DFT functionalities are needed, whether it be at the IP level, sub-system level or at the SOC level, albeit with increasing complexities in their functionality. Re-use further reduces the complexity, time and cost associated with verification. In this paper, while we emphasize the verification task of DFT logic in an SOC at the RTL level, which constitutes a significant portion of the entire DFT logic verification task, there are several gate level DFT Logic verification tasks which are better suited to simulation (through TDLs). Even for such gate level verification tasks, ensuring a clean DFT logic integration at the RTL level helps in reducing the overall effort, as many errors at this level of hierarchy, using earlier approaches, are attributable to RTL level integration errors.

The principal objective of the proposed approach has been to 1). Reduce simulation based DFT logic integration verification at the RTL level, 2). Improve robustness of Silicon quality by complete elimination of any bugs related to DFT logic, and 3). Enable re-use of DFT logic verification infrastructure across different SOCs and across different hierarchies within each SOC. These objectives have been achieved by taking the formal verification route with auto-generation of formal properties and the formal tool set up, on which the proof of these properties are executed. In this paper we give several examples which highlight our contributions to the above objectives across different hierarchies within an SOC and across different SOCs.

Keywords - Formal Verification, DFT Logic, SOC Integration

SLAM2: Static Driver Verification with Under 4% False Alarms

Thomas Ball
Microsoft Research
Redmond, USA

Ella Bounimova
Microsoft Research
Redmond, USA

Rahul Kumar
Microsoft
Redmond, USA

Vladimir Levin
Microsoft
Redmond, USA

***Abstract*—In theory, counterexample-guided abstraction refinement (**CEGAR**) uses spurious counterexamples to refine overapproximations so as to eliminate provably false alarms. In practice,** CEGAR **can report false alarms because: (1) the underlying problem** CEGAR **is trying to solve is undecidable; (2) approximations introduced for optimization purposes may cause** CEGAR **to be unable to eliminate a false alarm; (3)** CEGAR **has no termination guarantee - if it runs out of time or memory then the last counterexample generated is provably a false alarm.**

We report on advances in the SLAM **analysis engine, which implements** CEGAR **for C programs using predicate abstraction, that greatly reduce the false alarm rate.** SLAM **is used by the Static Driver Verifier (**SDV**) tool. Compared to the first version of** SLAM **(**SLAM1**, shipped in** SDV **1.6), the improved version (**SLAM2**, shipped in** SDV **2.0) reduces the percentage of false alarms from 25.7% to under 4% for the WDM class of device drivers. For the** KMDF **class of device drivers,** SLAM2 **has under 0.05% false alarms. The variety and the volume of our experiments of** SDV **with** SLAM2**, significantly exceed those performed for other CEGAR-based model checkers.**

These results made it possible for SDV **2.0 to be applied as an automatic and required quality gate for Windows 7 device drivers.**

I. Introduction

A decade ago, the SLAM project [BR02b] introduced the concept of counterexample-guided abstraction refinement (CEGAR) for the analysis of temporal safety properties of C programs. This work resulted in the Static Driver Verifier (SDV) tool that Microsoft applies internally to its device drivers and ships with the Windows Driver Development Kit (WDK) for use by third-party device driver writers [BBC+06].

As shown in Figure 1, the essential points of the CEGAR process, as implemented by SLAM, are: (1) the automated creation of a Boolean program *abstraction* of an instrumented C program that contains information relevant to the property under consideration; (2) *model checking* of the Boolean program to determine the absence or presence of errors; (3) the *validation* of a counterexample *trace* to determine whether or not it is a feasible trace of the C program. The last step can either produce a validated counterexample trace or a proof that the trace is invalid (a provably false alarm), in which case information is added to the abstraction to rule out the false alarm.

The CEGAR process has three distinct attributes: first, it may terminate with either a proof of correctness ("verified") or a validated counterexample trace; second, if CEGAR proves a counterexample trace is invalid then, in theory, it can rule out at least this trace from the abstraction (the so-called *progress property*); third, even if CEGAR always makes progress it still has no guarantee of terminating [BPR02].

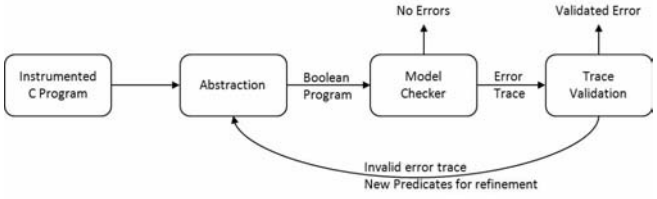


Fig. 1. The SLAM realization of the CEGAR loop.

Theoretically, the lack of a termination guarantee appears to be the death knell for CEGAR: most program analyses typically have termination guarantees despite having the problem of false alarms. However, we can set a time limit on a CEGAR run. If the run is aborted, we have the result that the last counterexample trace considered by CEGAR was invalid (provably a false alarm). So, CEGAR with a time limit has a three-valued outcome: (1) verified; (2) validated error trace; (3) not-useful result (NUR) due to lack of progress or timeout/spaceout. In the second case, the result still could be a false alarm due to bugs in the environment model, temporal safety property, or the SLAM engine itself. In the results reported in the abstract and here in the introduction, we count such cases as well as NURs as "false alarms".

In order to improve the chances for CEGAR to terminate with useful results and fewer false alarms, we explored four main ideas in SLAM2, which was derived from SLAM1.

First, we increase the precision of the predicate transformer over statement sequences. SLAM1 abstracts each C program statement (such as an assignment or **assume** statement representing a conditional branch) to a corresponding Boolean program statement. Thus, if the C program contains the statement sequence $(S_1; S_2)$ then the Boolean program abstraction computed by SLAM1 contains the statement sequence $({S_1}^{\#}; {S_2}^{\#})$, where $S^{\#}$ is the abstraction of statement S. We call this approach *fine-grained* abstraction. Our contribution here is to show how to construct the Cartesian/Boolean program abstraction [BPR01] for sequences of assignments and assume statements, so that the statement sequence $(S_1; S_2)$ abstracts to $(S_1; S_2)^{\#}$. We call this approach *coarse-grained* abstraction, which SLAM2 implements.

Second, we use diverse strategies for exploring counterexample traces. SLAM1 uses a "depth-first" strategy: it symbolically executes a counterexample trace in the C program forward from the initial state. As soon as it finds a trace prefix that is inconsistent, it generates a set of refinement predicates and a refined Boolean program abstraction. The SLAM1 symbolic execution step is complicated because of its use of symbolic (Skolem) constants, which must be tracked and eliminated in order to later generate properly scoped predicates [BR02a].

In contrast, SLAM2 uses both forward and backward symbolic execution. Forward symbolic execution is a simple interpreter that maintains a symbolic store. Backward symbolic execution is based on preconditions, decomposed and cached per program point in order to make predicate generation very simple. The combination of forward and backwards symbolic execution allows SLAM2 to detect inconsistencies near the beginning of a counterexample trace as well as near the end or in the middle, giving it more flexibility over SLAM1.

The third major difference is in how the two engines react to the lack of progress, which can occur because SLAM computes approximations to the best Boolean abstraction in order to speed the search for both proofs and counterexamples. Upon finding lack of progress (identified when none of the predicates generated in the current iteration of CEGAR is new), SLAM1 refines the Boolean program transition relation [BCDR04]. We call this the CONSTRAIN module of SLAM, which is common to both SLAM1 and SLAM2. In contrast, SLAM2 detects multiple inconsistencies in the same counterexample trace when a lack of progress stops it; it interleaves the discovery of new predicates with application of the CONSTRAIN module so that it is less likely to get stuck.

Fourth, SLAM2 uses information computed during forward symbolic execution to optimize backward symbolic execution in several ways. In particular, the value of pointers computed by the forward execution is critical to the optimization of the precondition calculation for assignment statements and procedure calls.

In addition to these four main ideas, SLAM2 has a completely re-implemented and more efficient pointer analysis. To optimize predicate evaluation, SLAM2 uses the Z3 state-of-the-art SMT solver [MB08] with two major improvements in the interface between SLAM and Z3: an efficient encoding of the predicates given to Z3 and a new set of axioms that express the SLAM memory model, in particular, relations between pointers and locations [BBdML10].

As the saying goes, "the proof is in the pudding": compared to SLAM1, SLAM2 reduces the percentage of false alarms from 25.7% to under 4% for the WDM class of device drivers. For the KMDF class of device drivers, SLAM2 has under 0.05% false alarms.[1] These figures come from 5727 unique checks using both SLAM1 and SLAM2 on 69 device drivers from the WDK against 83 temporal safety properties.

A common question about verification tools is "who verifies the verifier?". The typical answer is that one uses lots of benchmarks and testing, as well as cross comparison to other tools. In the development of SLAM2, we found numerous deficiencies in SLAM1, including its overconstraining of the abstract transition relation, which leads to "false verification", a real but little acknowledged problem with verification tools.

So, we also compared SLAM2 to the YOGI analysis engine [NRTT09] on the same benchmarks. For WDM, SLAM2 provides 7% fewer NURs, fewer false defects (2 versus 18), while finding 18 true defects that YOGI misses (YOGI finds 2 true defects that SLAM2 misses), and is two times faster than YOGI. For KMDF, SLAM2 produces 58 times fewer NURs (2 versus 117), and is 8 times faster than YOGI.

SLAM2 moves closer to the CEGAR promise to "abstract-and-refine" until it produces a proof of correctness or a validated trace. The false alarm rate of SLAM2 is so low that SLAM2 empowers a truly push-button software model checking experience for users of the SDV tool, which resulted in the technology being required as quality gate for shipping of Microsoft-produced Windows 7 device drivers.

The rest of this paper is organized as follows: Section II presents the coarse-grained abstraction; Section III describes the forward and backwards symbolic interpreters; Section IV describes how SLAM2 uses these interpreters to optimize the CEGAR loop; Section V presents the treatment of preconditions for assignments and procedure calls in the presence of pointers; Section VI presents experiments results; Section VII reviews related work, and Section VIII concludes the paper.

II. COARSE-GRAINED BOOLEAN ABSTRACTION

Given a C program P, a set of Boolean expressions E, SLAM's predicate abstraction step produces the Boolean program abstraction $BP(P, E)$ containing variables $V = \{b_1, b_2, \ldots, b_n\}$. Each variable b_i in V corresponds to the Boolean expression (predicate) ϕ_i in E. Boolean programs contain all the control-flow constructs of C, including procedures and procedure calls. We will focus here on the abstraction of a procedure with no procedure calls, as the handling of procedure calls and returns remain unchanged compared to SLAM1 [BMR05].

Each procedure of a C program is represented by a control-flow graph with basic blocks, where each basic block is a sequence of assignments, skips, and **assume** statements. The **assume** statements are used to model the semantics of **if-then-else** statements as well as assumptions about data (non-nullness of pointers).

SLAM2 generalizes the abstraction step compared to SLAM1 by abstracting sequences of statements as opposed to single statements:

$$S \rightarrow S_1; S_2 \mid \textbf{skip} \mid x := e \mid *x := e \mid \textbf{assume}(e)$$

The main advantage of coarse-grained abstraction compared to fine-grained is increased precision [CC77].

[1] The Windows Driver Model (WDM) is a widely-used kernel-level API that provides access to low-level kernel routines as well as routines specific to driver's operation and life-cycle. The Kernel-mode Driver Framework (KMDF) is a new kernel-level API which provides higher-level abstractions of common driver actions.

S	$pre(S,Q)$	$wp(S,Q)$
skip	Q	Q
$x := e$	$Q[e/x]$	$Q[e/x]$
$*x := e$	$(x = \&y_1 \wedge Q[e/y_1]) \vee \ldots \vee (x = \&y_k \wedge Q[e/y_k])$	same as $pre(S,Q)$
assume(e)	$e \wedge Q$	$e \implies Q$
$S_1; S_2$	$pre(S_1, pre(S_2, Q))$	$wp(S_1, wp(S_2, Q))$

Fig. 2. Predicate transformers pre and wp.

A. Transformation

We use the standard precondition (pre) and weakest precondition (wp) predicate transformers to assign meaning to C programs as well as to perform the abstraction to Boolean programs. Figure 2 shows the predicate transformers for the statements S under consideration. Recall that $wp(S,Q) = \neg pre(S, \neg Q)$.

We use a source-to-source transformation on the C program to simplify the abstraction process. Any statement sequence S is equivalent to **assume**$(pre(S,true)); sub(S)$, where the function $sub(S)$ is defined to be the maximal subsequence of S containing only assignment statements of S (and is defined to be the **skip** statement in the case that S contains no assignment statements).

Lemma 1 (*Correctness of transformation*). For all statement sequences S and predicates Q:

$$wp(S,Q) \iff wp(\textbf{assume}(pre(S,true)); sub(S), Q)$$

Proof. By induction on length of statement sequence S, show that

$$wp(S,Q) \iff (pre(S,true) \implies wp(sub(S),Q))$$

[The proof is straightforward but omitted due to lack of space]

B. Abstraction

A *cube* over V is a conjunction $c_{i_1} \wedge c_{i_2} \wedge \ldots \wedge c_{i_k}$, where each $c_{i_j} \in \{b_{i_j}, \neg b_{i_j}\}$ for some $b_{i_j} \in V$. For a variable $b_i \in V$, let $\mathcal{E}(b_i)$ denote the corresponding predicate φ_i, and let $\mathcal{E}(\neg b_i)$ denote the predicate $\neg\varphi_i$. Extend $\mathcal{E}$ to cubes and disjunctions of cubes in the natural way.

For any predicate φ and set of Boolean variables V, let $\mathcal{F}_V(\varphi)$ denote the largest disjunction of cubes c over V such that $\mathcal{E}(c)$ implies φ. The predicate $\mathcal{E}(\mathcal{F}_V(\varphi))$ represents the weakest predicate over $\mathcal{E}(V)$ that implies φ. The corresponding weakening of a predicate is also defined similarly. Let $\mathcal{G}_V(\varphi)$ be $\neg\mathcal{F}_V(\neg\varphi)$. The predicate $\mathcal{E}(\mathcal{G}_V(\varphi))$ represents the strongest predicate over $\mathcal{E}(V)$ that is implied by φ.

Following Lemma 1 and the definition of Cartesian/Boolean abstraction [BPR01], Figure 3 shows the translation of a statement S to a *guarded parallel assignment* in the Boolean program. Here the $*$ value represents a value non-deterministically chosen from $\{true, false\}$. The computation of the predicate abstraction of a formula ϕ, as represented by $\mathcal{F}_V(\phi)$, typically relies on an automated theorem prover [GS97]. SLAM1 and SLAM2 both rely on a specialized algorithms for predicate abstraction [LBC05].

assume$(\mathcal{G}_V(pre(S,true)))$;

$b_1 :=$ **if** $(\mathcal{F}_V(wp(sub(S),\varphi_1))$ **then** $true$ **else**
if $(\mathcal{F}_V(wp(sub(S),\neg\varphi_1)))$ **then** $false$ **else** $*$,

$\ldots$

$b_n :=$ **if** $(\mathcal{F}_V(wp(sub(S),\varphi_n))$ **then** $true$ **else**
if $(\mathcal{F}_V(wp(sub(S),\neg\varphi_n)))$ **then** $false$ **else** $*$;

Fig. 3. Cartesian/Boolean abstraction of statement sequence S.

III. Counterexample Trace Validation

In this section, we explain the two symbolic interpreters that SLAM2 uses to perform counterexample trace validation on C programs and predicate discovery. The first is a forward interpreter and the second a backwards interpreter (SLAM1 only performs forward symbolic execution). The next section will discuss more about how the two interpreters are used together.

The language of compound statements introduced in the previous section for the abstraction of basic blocks also serves as the basis for our discussion of symbolic execution of an *execution trace*. An execution trace is simply a sequence of basic blocks through the control-flow graph, whose code can be modeled by a sequence of assignment and **assume** statements (one very long basic block). For the rest of this section, let $S_1 \ldots S_n$ represent the sequence of statements in the execution trace under consideration.

A. Forward Symbolic Execution

Forward Symbolic Execution (FSE) processes the *entire* trace $S_1 \ldots S_n$ with two goals: (1) to find an invalid execution trace prefix of the form $S_1 \ldots S_i$; (2) to populate a "trace database" that maps each statement S_j to the store computed by FSE just before execution of S_j. The main use of the trace database is to resolve pointer-aliasing questions in a trace-sensitive manner, as detailed in Section V.

Operationally, forward symbolic execution is an interpreter that computes the strongest post-condition ($sp(P,S)$) of a statement sequence S with respect to the initial predicate $P = true$. Recall that

$$\begin{array}{lcl} sp(P, \textbf{skip}) & = & P \\ sp(P, \textbf{assume}(e)) & = & P \wedge e \\ sp(P, x := e) & = & \exists\theta_x.P[x/\theta_x] \wedge (x = e[x/\theta_x]) \\ sp(P, S_1; S_2) & = & sp(sp(P,S_1), S_2) \end{array}$$

C-like Program	Precondition Vectors

1: void main(){
2: int x, y, a;
3: x := y;
4: x := x+1;
5: if(a>0)
6: a := a+1;
7: if(x = y+2){
8: SLIC_ERROR:0;
9: }
10: }
	↑

(a)	(b)

	0	1	2
3-4	$true$	$y+1=y+2$	$\neg(a>0)$
5	$true$	$x=y+2$	$\neg(a>0)$
7	$true$	$x=y+2$	
8	$true$		

Fig. 4. Backwards Symbolic Execution

(for brevity, we omit the rule for $*x := e$).

FSE maintains a store mapping locations to values and processes the statements $S_1 \ldots S_n$ in order from S_1 to S_n. Symbolic evaluation of an assignment ($x := e$ or $*x := e$) involves: (1) evaluation of the RHS expression e in the context of the current store to get a value v; (2) evaluation of the LHS expression in the context of the current store to get a location l; (3) mapping location l to value v in the store (possibly overwriting the previous mapping for location l). During symbolic execution, if a location l (such as the address of variable x) doesn't have a mapping in the store then a fresh symbolic value θ_l for the value of l is created and l is mapped to θ_l in the store.

Execution of a statement $S_i =$ **assume**(e_i) first evaluates the Boolean expression e_i in the current store, which results in an expression ϕ_i solely over constants of the programming language (such as 1, 42, ...) and symbolic constants (such as θ_l). FSE maintains a trace condition ϕ (initially $true$), which is the conjunction of the ϕ_i. A call to the theorem prover Z3 [MB08] determinines the satisfiability of the formula $\exists\theta.\phi \wedge e_i$. If the formula is satisfiable, then there is an assignment of values to the symbolic constants θ (the primary inputs to the execution trace) that witness the validity of the execution trace. If it is unsatisfiable then the trace prefix $S_1 \ldots S_i$ is inconsistent/invalid.

B. Backwards Symbolic Execution

Operationally, backwards symbolic execution (BSE) computes $pre(S_1 \ldots S_k, true)$, $k \leq n$, but decomposes and caches the representation of each application of pre in order to enable predicate generation if the counterexample is determined to be invalid. The benefits of symbolic execution with pre are: (1) there is no need to introduce symbolic constants; (2) assignments to variables that don't appear in the postcondition Q have no effect. An issue with the use of pre is a blow-up in the size of the precondition formula due to pointer aliasing (see the rule for $*x := e$ in Figure 2), which we will return to later.

The decomposition of pre is based on the simple observation that $pre($**assume**$(e), Q) = (e \wedge Q)$. If Q is a conjunction $(q_0 \wedge \ \ldots \ \wedge q_r)$, represented implicitly by the vector $< q_0, \ \ldots, \ q_r >$, then we represent $(e \ \wedge \ Q)$ by $< q_0, \ \ldots, \ q_r, \ e >$, which preserves the positions of the q_i in the vector.

BSE starts with the one element vector $Q = \ < true >$. Processing of an **assume** statement lengthens the vector by one element, as described above. For an assignment statement, the pre computation for the assignment is applied point-wise to the input vector, resulting in a new vector of the same length.

We can visualize the computation of pre as creating an upper-left-triangular matrix of row vectors, where the first column contains $true$ everywhere and each subsequent column represents the history of a subformula introduced by an **assume** statement. The last row (k+1) of the matrix represents the starting point where $Q_k =< true >$. The i^{th} row of the matrix ($1 \leq i < k$) represents $Q_i = pre(S_i \ldots S_k, true)$.

For each new precondition vector Q_i computed, Z3 is called to query if the conjunction of formulas in the vector is satisfiable. If it is unsatisfiable then the trace $S_i \ldots S_k$ is invalid and the predicate discovery algorithm starts, as described in the next subsection. Otherwise, BSE proceeds to consider statement S_{i-1} in the trace. If BSE determines that Q_1 is satisfiable then the execution trace is valid.

Figure 4 illustrates BSE on a simple C program (a). Consider the false counterexample trace `2-3-4-5-7-8`. Figure 4(b) shows the vector-based computation of pre on this trace, with the corresponding trace step numbers in the left-most column (only the steps where the preconditions change are shown).

Columns 0-2 in the table show the precondition computation for each step of the trace, going backwards from the error step 7. For example, at step 6 a new vector element $x = y + 2$ is added, which corresponds to the **then** branch of the conditional. At steps 3 and 4, which correspond to the sequence of assignments $y := x; x := x + 1$, the precondition in column 1 is computed as $pre(y := x; x := x+1, x = y+2) = (y+1 = y+2)$, whereas the precondition in column 2 is not affected.[2]

C. Predicate Discovery

Given an invalid execution trace $S_i \ldots S_k$, the goal of predicate discovery is very simple: find a set of predicates

[2]Note that the two assignment statements occupy the same basic block, so are treated together, just as they are during the abstraction step. This reduces the number of predicates generated.

E such that the abstract version of pre induced by E (pre_E) can prove $S_i \ldots S_k$ is an invalid execution trace.

More formally, let $\alpha_E(\phi)$ be the weakest formula ϕ' (in the implication ordering) such that ϕ' is a Boolean combination of the predicates in E and ϕ' implies ϕ. Then for a basic block S, $pre_E(S, Q) = \alpha_E(pre(S, Q))$ and for a sequence of two basic blocks S_1 and S_2, $pre_E(S_1; S_2, Q) = pre_E(S_1, pre_E(S_2, Q))$. Suppose that $pre(S_i \ldots S_k, true) = false$, where the S_x are basic blocks, then we wish to find a *sufficient* set of predicates E such that $pre_E(S_i \ldots S_k, true) = false$.

Once BSE has discovered that a precondition vector Q_i is unsatisfiable, it is clear that the set of predicates in the precondition matrix $M_{i+1} =< Q_{i+1} \ \ldots \ Q_k >$ are sufficient. Of course, we can do much better: the underlying theorem prover can provide us an unsatisfiable core of Q_i, a small subset of the elements of Q_i whose conjunction is unsatisfiable. This subset identifies a set of "inconsistent" columns in M_{i+1}. Again, it is clear that the set of predicates from this set of columns are sufficient. In our example at line 3, the formula

$$\exists y. \exists a. true \wedge (y + 1 = y + 2) \wedge \neg(a > 0)$$

is unsatisfiable. An unsatisfiable core is $\{(y+1 = y+2)\}$. So, a sufficient set E includes predicates from the second column: $\{x = y + 2\}$.

IV. Optimizing the CEGAR Loop: Multiple Inconsistencies

Optimizations of the CEGAR loop are based on analysis of the cases when SDV fails on Windows device drivers with "not-useful results" (NURs, in SDV terminology). In theory, for a CEGAR run, the set of predicates strictly increases as the iterations of CEGAR increase. Let E_i be the set of predicates discovered by iteration i of CEGAR. In practice, both SLAM1 and SLAM2 may discover predicates E_j such that $E_j \subseteq \bigcup_{0 \leq i < j} E_i$. This lack of progress condition can arise due to approximations introduced in the abstraction step, which can result in the same counterexample trace being produced in consecutive iterations.

Upon finding lack of progress, SLAM1 employs a tool called CONSTRAIN to refine the Boolean program abstraction computed for the current set of predicates [BCDR04]. Our experiments indicated that CONSTRAIN was a bottleneck in SLAM1, so we experimented with techniques in SLAM2 to reduce the need to use CONSTRAIN.

The optimized CEGAR loop makes use of both FSE and BSE, as well as the CONSTRAIN module. Given a counterexample trace $S_1 \ldots S_n$, SLAM2 first applies FSE. If FSE finds an invalid trace prefix $S_1 \ldots S_i$ then BSE is applied to the trace $S_1 \ldots S_i$ to discover new predicates.

The approach outlined above is similar to SLAM1: predicates are discovered based on invalid trace *prefixes*. However, an invalid trace can have several invalid subtraces. So, SLAM2 also uses BSE in two new ways to discover more invalid subtraces. First, if there is lack of progress on invalid trace prefix $S_1 \ldots S_i$, SLAM2 will apply BSE to the entire trace $S_1 \ldots S_n$ to try to find an invalid trace suffix $S_k \ldots S_n$. Second, if there is lack of progress on invalid trace suffix $S_k \ldots S_n$, SLAM2 will perform a *partial reset* of the pre computation and continue BSE, as follows. Suppose that the set of inconsistent columns of the precondition matrix after processing $S_k \ldots S_n$ are $k_1, k_2, \ldots, k_m$. The partial reset removes these columns from the precondition matrix and resumes BSE at statement S_{k-1}. The partial reset can be done multiple times to find multiple invalid traces.[3]

The above approach is interleaved with the application of the CONSTRAIN module, which is applied just once when a lack of progress is first identified. SLAM1 does not attempt to find multiple invalid subtraces. Upon lack of progress, it attempts to resolve the issue using CONSTRAIN. If lack of progress continues, SLAM1 terminates with a "GiveUp" result, whereas SLAM2 will continue to analyze the trace to find new predicates. If SLAM2 finishes exploring $S_1 \ldots S_n$ with no new predicates, it too will terminate with a "GiveUp" result.

V. Procedure Calls and Pointers

A key aspect of the SLAM approach to CEGAR is that the Boolean program abstraction contains procedures and procedure calls. Thus, Boolean variables introduced by predicate discovery can be locally scoped to a procedure, which reduces the cost of model checking.

SLAM2 remains unchanged with respect to SLAM1 regarding Boolean program abstractions with procedures. BSE performs precondition evaluation at procedure return and procedure call steps by converting the precondition from the scope of the caller into the scope of the callee (for returns) and back (for calls). This is done by using relations between actual and formal parameters of the call/return, and between the return value of the procedure call (if any) and the return variable of the callee.

As discussed before, the precondition computation applied during BSE has the potential to blow up in size because of pointers. But, in fact, SLAM2 eliminates this possibility by making the pre computation trace-sensitive for BSE, using the pointer aliasing information computed by FSE. Consider a statement $S_i : *x := e$ in the trace. Recall that $pre(*x := e, Q)$ is

$$(x = \&y_1 \wedge Q[e/y_1]) \vee \ldots \vee (x = \&y_k \wedge Q[e/y_k])$$

To reduce the size of this formula, BSE looks up the location pointed to by x in the store computed by FSE on entry to statement S_i. Suppose that in this store x maps to $\&y_j$. Then the above equation reduces to $Q[e/y_j]$.

VI. Experimental Results

We now present a comparison of SLAM2, SLAM1 and YOGI by running SDV on two large test suites developed and maintained by Microsoft quality assurance teams for testing SDV. We first describe our evaluation platform and

[3] One could also perform a full reset of the precondition matrix to the initial vector $< true >$ - we did not experiment with this approach.

Metric	SDV 1.6 (SLAM1)	SDV 2.0 (SLAM1)	SDV 2.0 (SLAM2)
Drivers	69	69	69
Rules	68	83	83
Total checks	4692	5727	5727
LightweightPass results	-	2477	2477
Pass results	-	2563	2551
NUR results	6% (285/4692)	2.1% (123/5727)	3.3% (187/5727)
Defects reported	157	564	512
GiveUp results only	-	0.52% (30/5727)	0.3% (16/5727)
False defects	19.7% (31/157)	9.04% (51/564)	0.4% (2/512)
Time for identical pass	-	39922	65800
Time for identical defect	-	4440	2669

TABLE I
COMPARISON OF SLAM1 AND SLAM2 FOR WDM DRIVER CHECKS.

criteria. At Microsoft, SDV is used for verification of device drivers built in multiple driver development models. For our analysis, we have chosen test suites developed for WDM and KMDF drivers. These comprehensive test suites include drivers of different sizes (1-30K LOC), with a mix of test drivers written to test SDV rules (with injected defects), sample drivers that are shipped in WDK to provide guidance to driver developers, and drivers that are shipped as part of the Windows operating system. Note that all the data presented in this section has been extracted from test runs performed by the test team.

Most of the metrics used in this section were explained in previous sections. New to this section are the following metrics. A "check" is a run on one driver for one rule. A "LIGHTWEIGHTPASS" result refers to the fact that before starting the CEGAR loop, SDV first applies property instrumentation, pointer analysis, and function pointer resolution to show that the error state of a rule is not reachable in the call-graph of the C program. An "out of resource" (OOR) result refers to checks that exceeded the allocated time or memory resources. The NUR results include both the OOR and GiveUp results.

SDV can report a false defect for a number of reasons: a bug in the verification engine, a bug in the rule, or a bug in the environment model (the C code that calls into a driver and provides stubs of kernel routines used by drivers). Hence, improvements to any of those components can result in the reduction in the number of false defects.

SDV can report a Pass result which is actually a "false verification", due to overconstraining of the abstract transition relation. This problem can be revealed by comparing SDV runs with different engines, for example, SLAM1 versus SLAM2. In particular, we observed that some Pass results with SLAM1 turn into Defect or OOR results with SLAM2. The OOR result would mostly occur on the runs for large drivers and/or hard rules. Specific data for such cases are presented in Tables I and II.

For the purposes of profiling SDV and comparing the analysis engines, we use the two official releases of SDV, SDV 1.6 and 2.0, and also runs of SDV 2.0 with SLAM1, for a more accurate comparison.

Table I compares the data for the WDM drivers for SLAM1 as part of both SDV 1.6 and SDV 2.0, and for SLAM2 as part of SDV 2.0. Dashes in the table indicate that the data is not available for that particular metric.

SDV 2.0 (SLAM1)	SDV 2.0 (SLAM2)	COUNT	CHANGE
OOR	Pass	31	$\surd$
Defect (false)	Pass	5	$\surd$
Defect (true)	Pass	2	$\times$
GiveUp	Pass	15	$\surd$
OOR	Defect (true)	2	$\surd$
Defect (false)	OOR	36	$\surd$
GiveUp	OOR	13	$\surd$
Pass	OOR	64	$\sim$
OOR	GiveUp	2	$\sim$
Defect (false)	GiveUp	11	$\surd$
Defect (true)	GiveUp	1	$\times$

TABLE II
BREAKDOWN OF CHANGES OBSERVED BETWEEN SLAM1 AND SLAM2 USING SDV 2.0 FOR WDM DRIVERS.

Table I shows significant reduction in the number of false defects and GiveUp results for SLAM2. This is due to the better precision of coarse-grained abstraction, as well as to the improved trace validation and predicate discovery. All three factors play a role in these improvements. In particular, better predicate discovery helps make progress (discover new predicates) in the cases where SLAM1 couldn't; more precise abstraction reduces the need for additional predicates in the first place. The number of NURs significantly decreased between SDV 1.6 and SDV 2.0 for both engines. This is mostly due to the improvements in SDV environment and rules, in particular, NULL pointer dereference bugs. Those bugs have been found by running SDV with SLAM2 (but not with SLAM1). Finally, SLAM2 is faster in finding defects, but takes more time to prove Pass results. The time difference for the Pass results is due to the problem of overconstraining of the abstract transition relation in SLAM1, i.e., "false verification".

According to Table I, for WDM drivers, SLAM2 provides a useful result 96.7% of the time, and upon discovery of a defect, provides a 99.6% guarantee that this is a true defect.

Table II shows the breakdown of the individual results and changes observed between SDV 2.0 with SLAM1 and with SLAM2 for WDM drivers. The leftmost column is the result reported by SLAM1, followed by the result reported by SLAM2 and the count for such changes. The rightmost column indicates whether the changes are in favor ($\surd$), against

Metric	SDV 2.1 (SLAM2)	SDV 2.1 (YOGI)
LightweightPass results	2457	2457
Pass results	2556	2538
NUR results	3.3% (194/5727)	3.65% (209/5727)
Defects reported	520	523
False/reported defects	0.4% (2/520)	3.4% (18/523)
Missed defects	2	18
Time for identical pass	76922s	147189s (∼2x)
Time for identical defect	1795s	9984s (∼6x)

TABLE III
COMPARISON OF SLAM2 WITH YOGI USING SDV 2.1 FOR WDM DRIVERS.

SDV 2.1 (YOGI)	SDV 2.1 (SLAM2)	COUNT	CHANGE
NUR	Pass	114	√
Defect (false)	Pass	8	√
NUR	Defect (true)	10	√
Pass	Defect (true)	8	√
Defect (false)	OOR	1	√
Pass	OOR	94	×
NUR	GiveUp	4	∼
Defect (false)	GiveUp	10	√
Defect (true)	GiveUp	2	×
Pass	GiveUp	2	×

TABLE IV
BREAKDOWN OF CHANGES OBSERVED BETWEEN SDV 2.1 WITH SLAM2 AND SDV 2.1 WITH YOGI FOR WDM DRIVERS.

(×), or neutral (∼), for SLAM2 with respect to SLAM1.

There are 28 cases where GiveUp results by SLAM1 changed into Pass (15 cases) or OOR (13 cases) for SLAM2. The change from GiveUp to OOR indicates that progress has been made beyond the GiveUp point (but not until a definite result, due to insufficient resources). Out of 14 cases where SLAM2 produces a GiveUp, there are 11 cases for which SLAM1 produces a (false) defect. There are 36 cases where false defects reported by SLAM1 changed into OOR for SLAM2, which is clearly favorable for SLAM2. Finally, we mark the changes from the Pass result for SLAM1 into the OOR result for SLAM2 (64 cases) as neutral, because we have a strong evidence that SLAM1 was able to prove the Pass result by overconstraining, but it is unrealistic to investigate each case to validate this claim. Note that the two defects found by SLAM1 but not by SLAM2 are being investigated.

Table III presents a comparison of SLAM2 with YOGI [NRTT09] for WDM drivers. SLAM2 provides 7% fewer NURs, fewer false defects (2 versus 18), while finding 18 true defects that YOGI misses (the respective number for YOGI is 2), and is two times faster than YOGI. Note that YOGI does not report GiveUp results in the same way as SLAM does, so this analysis is not performed - instead, the GiveUp cases are included into the NUR cases. Notably, YOGI takes 6 times longer for finding the same defects as SLAM2, but only 2 times longer for finding the same proofs as SLAM2.

According to Table III, for WDM drivers, YOGI provides a useful result 96.3% of the time, and upon discovery of a defect, provides a 96.6% guarantee that this is a true defect. SLAM2 provides a useful result 96.6% of the time and a true defect guarantee of 99.8%.

Table IV provides a breakdown of the changes observed between SLAM2 and YOGI using SDV 2.1 on WDM drivers. The format is the same as in Table II. The table shows that in general, SLAM2 provides a higher rate of useful results: 114 Pass results and 10 defect reports for which YOGI reports NUR. There are 8 Pass results for SLAM2 for which YOGI reports false defects. There are 11 cases where SLAM2 finishes with an NUR result, and YOGI reports a false defect.

On the other hand, there are two cases where YOGI finds a defect which SLAM2 is unable to find (GiveUp) - those proved to be useful in identifying limitations of SLAM2.

Table V compares SLAM1, SLAM2, and YOGI using SDV on KMDF drivers. Note that KMDF drivers are significantly smaller than WDM drivers, due to the higher level of the APIs provided by the KMDF model. This explains the comparable results for both SLAM1 and SLAM2. There is a significant improvement in the number of NURs (1% to 0.04%) and false defects (25% to 0%) between SDV 1.6 and SDV 2.0, regardless of the SLAM version. This improvement is primarily due to the improvements in the KMDF environment model and rules between the two releases. Comparing SLAM2 to YOGI, we observe significantly larger number of NURs for YOGI: 117 vesus 2 for SLAM2. Additionally, YOGI takes 8 times longer than SLAM2 for checks with the identical results. Note that the defect analysis (true versus false defects) for comparing YOGI to SLAM2 has not been performed for KMDF drivers.

Table V shows the comparison of SLAM1, SLAM2, and YOGI for KMDF drivers. SLAM2 provides a useful result 99.8% of the time, and upon discovery of a defect, provides a 100% guarantee that this is a true defect. Comparatively, YOGI provides a useful result 97.8% of the time.

In summary, our comprehensive analysis of the realistic empirical data confirms that SLAM2 provides highly reliable results by reporting defects with a high degree of confidence that those are true defects, or finding proofs when there's no defect. Our comparison involves two driver models and three verification engines and is based on the data obtained in an industrial setting by independent testers.

VII. RELATED WORK

Coarse-grained Abstraction. After the development of SLAM1, it became clear that we were underutilizing the power of automated theorem provers such as Z3 to cope with complex Boolean formulae, relying instead on the Boolean program model checker to deal with arbitrary Boolean combinations of predicates. With coarse-grain abstraction, we give Z3 a little bit more work to do and increase the precision of the abstraction. However, one can do much more, as explored by Beyer and colleagues in their work on "software model checking via large-block encoding" [BCG+09]. They show that one can abstract over loop-free fragments of code such as sequences of **if-then-else** statements. They compared their large-block approach to the approach where each single

Metric	SDV 1.6 (SLAM1)	SDV 2.0 (SLAM1)	SDV 2.0 (SLAM2)	SDV 2.1 (SLAM2)	SDV 2.1 (YOGI)
Driver	51	51	51	51	51
Rules	61	102	102	103	103
Total checks	3111	5202	5202	5253	5253
NUR results	1% (31/3111)	0.04% (2/5202)	0.04% (2/5202)	0.04% (2/5253)	2.2% (117/5253)
Defects reported	300	271	271	271	-
False defects	25% (75/300)	0% (0/271)	0% (0/271)	0% (0/271)	-
Total time for identical checks	-	-	-	8414s	63645s (∼8x)

TABLE V
COMPARISON OF SLAM1, SLAM2 AND YOGI USING SDV FOR KMDF DRIVERS.

statement is abstracted in isolation. It would be interesting to compare their approach to the presented approach.

Multiple Inconsistencies Per Trace. We are not aware of other work that explores the idea of finding multiple invalid subtraces of a single counterexample trace. We found this technique to be very valuable for making more progress, but it does come at an increased cost in model checking, as more predicates are introduced. The ability to recover from "irrelevant refinements" (retracting predicats that are not useful) would be valuable in order to explore multiple inconsistencies during CEGAR. McMillan explores how to give CEGAR such a flexibility, which would be very helpful for the case of detecting multiple inconsistencies. [McM10]

Path/Trace-Sensitive Pointer Aliasing. SLAM2's use of pointer aliasing information, computed by forward symbolic execution, to refine the precondition computation is very similar to that used by the DASH algorithm [BNRS08], that forms the basis of the the YOGI tool we compare against. However, SLAM2 only uses this technique during symbolic execution and not the abstraction process, as YOGI does.

VIII. CONCLUSION

We have described major improvements in the SLAM verification engine, shipped with SDV 2.0 in September, 2009 as a part of the Windows 7 WDK. SLAM2 significantly improved the reliability, robustness and precision of SDV. SDV adoption inside Microsoft proved to be very successful, with "SDV clean" being a requirement for Microsoft drivers to be shipped with Windows 7.

Our results show that SDV 2.0 with SLAM2 is an industrial quality static analysis tool, compared to previous versions of SDV based on SLAM1, which was in many respects a research prototype. The SDV tool has benefited greatly from a multi-engine approach, allowing us to easily compare SLAM2 to YOGI.

REFERENCES

[BBC+06] T. Ball, E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, B. Ondrusek, S. K. Rajamani, and A. Ustuner. Thorough static analysis of device drivers. In *EuroSys 06*, pages 73–85, 2006.

[BBdML10] T. Ball, E. Bounimova, L. de Moura, and V. Levin. Efficient evaluation of pointer predicates with Z3 SMT Solver in SLAM2. Technical Report MSR-TR-2010-24, Microsoft Research, 2010.

[BCDR04] T. Ball, B. Cook, S. Das, and S. K. Rajamani. Refining approximations in software predicate abstraction. In *TACAS 04: Tools and Algorithms for the Construction and Analysis of Systems*, pages 388–403, 2004.

[BCG+09] D. Beyer, A. Cimatti, A. Griggio, M. E. Keremoglu, and R. Sebastiani. Software model checking via large-block encoding. In *FMCAD 09: Formal Methods in Computer Aided Design*, pages 25–32, 2009.

[BMR05] T. Ball, T. D. Millstein, and S. K. Rajamani. Polymorphic predicate abstraction. *ACM Trans. Program. Lang. Syst.*, 27(2):314–343, 2005.

[BNRS08] N. Beckman, A. V. Nori, S. K. Rajamani, and R. J. Simmons. Proofs from tests. In *ISSTA 08: International Symposium on Software Testing and Analysis*, pages 3–14, 2008.

[BPR01] T. Ball, A. Podelski, and S. K. Rajamani. Boolean and cartesian abstractions for model checking C programs. In *TACAS 01: Tools and Algorithms for Construction and Analysis of Systems*, pages 268–283, 2001.

[BPR02] T. Ball, A. Podelski, and S. K. Rajamani. On the relative completeness of abstraction refinement. In *TACAS 02: Tools and Algorithms for Construction and Analysis of Systems*, pages 158–172, April 2002.

[BR02a] T. Ball and S. K. Rajamani. Generating abstract explanations of spurious counterexamples in C programs. Technical Report MSR-TR-2002-09, Microsoft Research, January 2002.

[BR02b] T. Ball and S. K. Rajamani. The SLAM project: Debugging system software via static analysis. In *POPL 02: Principles of Programming Languages*, pages 1–3, January 2002.

[CC77] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for the static analysis of programs by construction or approximation of fixpoints. In *POPL 77: Principles of Programming Languages*, pages 238–252, 1977.

[GS97] S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In *CAV 97: Computer Aided Verification*, pages 72–83. 1997.

[LBC05] S. K. Lahiri, T. Ball, and B. Cook. Predicate abstraction via symbolic decision procedures. In *CAV 05: Computer-Aided Verification*, pages 24–38, 2005.

[MB08] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *TACAS 08: Tools and Algorithms for the Construction and Analysis of Systems*, 2008.

[McM10] K. L. McMillan. Lazy annotation for program testing and verification. In *CAV 10: Computer-Aided Verification*, 2010.

[NRTT09] A. V. Nori, S. K. Rajamani, S. Tetali, and A. V. Thakur. The Yogi project: Software property checking via static analysis and testing. In *TACAS '09: Tools and Algorithms for the Construction and Analysis of Systems*, pages 178–181, 2009.

Precise Static Analysis of Untrusted Driver Binaries

Johannes Kinder
Technische Universität Darmstadt
Darmstadt, Germany
Email: kinder@cs.tu-darmstadt.de

Helmut Veith
Technische Universität Wien
Vienna, Austria
Email: veith@forsyte.at

***Abstract*—Most closed source drivers installed on desktop systems today have never been exposed to formal analysis. Without vendor support, the only way to make these often hastily written, yet critical programs accessible to static analysis is to directly work at the binary level. In this paper, we describe a full architecture to perform static analysis on binaries that does not rely on unsound external components such as disassemblers. To precisely calculate data and function pointers without any type information, we introduce Bounded Address Tracking, an abstract domain that is tailored towards machine code and is path sensitive up to a tunable bound assuring termination.**

We implemented Bounded Address Tracking in our binary analysis platform Jakstab and used it to verify API specifications on several Windows device drivers. Even without assumptions about executable layout and procedures as made by state of the art approaches [1], we achieve more precise results on a set of drivers from the Windows DDK. Since our technique does not require us to compile drivers ourselves, we also present results from analyzing over 300 closed source drivers.

I. Introduction

Software model checking and static analysis are successful methods for finding certain bugs or proving their absence in critical systems software such as drivers. Source code analysis tools like SDV [2] are available for developers to statically check their software for conformance to specifications of the Windows driver API. For instance, if a driver calls the API method IoAcquireCancelSpinLock, it is required to call IoReleaseCancelSpinLock before calling IoAcquireCancelSpinLock again [3]. The vendors, however, are not forced to use these analysis tools in development, and they are unwilling to submit their source code and intellectual property to an external analysis process. Without source code, certification programs such as the Windows Hardware Quality Labs (WHQL) have to rely on testing only, which cannot provide guarantees about all possible executions of a driver. A solution to this problem is to relocate the static analysis to the level of the compiled binary. If the analysis does not require source code or debug symbols, an analysis infrastructure can be created independently of active vendor support.

Working with binaries poses several specific challenges. In general, code cannot be easily identified in x86 executables such as Windows device drivers. Data can be arbitrarily interleaved with code, and bytes representing code can be interpreted as multiple different instruction streams depending on the alignment at which decoding starts [4]. Therefore, a major challenge in analyzing binaries is to reliably extract those instructions that are actually executed at runtime and to build a control flow graph that accurately represents the possible targets even of indirect jumps. Existing approaches to static analysis of binary executables rely on a preprocessing step performed by a dedicated, heuristics based disassembler such as IDA Pro [5] to produce a plain text assembly listing [6]. This decouples the analysis infrastructure from disassembly itself and makes it difficult to use results from static analysis towards improving the control flow graph. Furthermore, since the analysis builds on an external disassembler, soundness can only be guaranteed with respect to the (error prone) output produced by the disassembler.

To overcome this problem, we propose an architecture for single pass disassembly and analysis, which does not discriminate between disassembly and analysis stages (Figure 1). Its integrative design is based on the following key insight: Following the control flow of a binary in order to decode the executed instructions is already an analysis of reachable locations. This is non-trivial in presence of indirect control-flow and should not be left to heuristic algorithms.

Another challenge in statically analyzing binaries is that the lack of types and the a priori unknown control flow make a cheap points-to analysis impossible. Every dereference of an unknown pointer can mean an access to any memory address, be it the stack, global memory, or the heap. A write access then causes a *weak update* to the entire memory: After the write, every memory location *may* contain the written value, which dramatically impacts the precision of the analysis. Worst of all, weak updates potentially overwrite return addresses stored on the stack (or function pointers anywhere in memory), which can cause spurious control flow to locations that are never executed at runtime. The goal of a sound and precise analysis on binaries is thus to achieve *strong updates* wherever possible: If a pointer can only point to one specific address in a state, the targeted memory location *must* contain the written value after a write access [7].

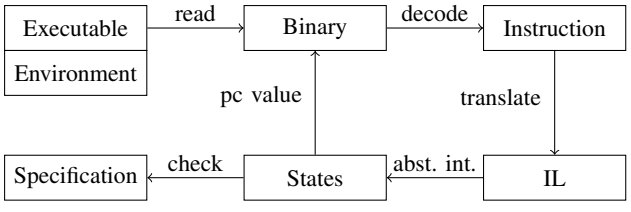


Fig. 1. Disassembly and analysis architecture.

In essence, an analysis capable of dealing with the lack of types in binaries needs to be precise enough to represent addresses without over-approximation that might introduce spurious control flow into non-code regions. On the other hand, high precision analyses are known not to scale to larger programs, so abstraction has to be introduced where possible. In this paper, we present our approach to dealing with these challenges without sacrificing soundness. In particular, our paper makes the following contributions:

- We describe an architecture for checking specifications on binary executables without access to source code and without a heuristics based, separate disassembly step. The control flow of the binary is reconstructed in a single pass with static analysis, following the approach presented in [8]. Abstractions of the execution environment can be written in C and are compiled into a separate module.
- We introduce *Bounded Address Tracking*, an abstract domain based on tracking a selection of register and memory values up to a given bound (inspired by [9]). The path sensitivity of our analysis allows strong updates to allocated heap regions. Since path sensitivity subsumes context sensitivity, we do not require assumptions about a separate call stack or well-structured procedures.
- In our path-sensitive analysis, nondeterminism in the program (e.g., from modeling input) is especially expensive. To address this issue, we offer two different constructs for nondeterminism, *havoc* and *nondet*, which cause explicit enumeration of variable values or their abstraction to an unknown value, respectively.

II. Background

We extended our own iterative disassembler JAKSTAB [10] to implement the integrated analysis architecture for single pass disassembly and static analysis (Figure 1). Using the entry point of the executable as the initial program counter (*pc*) value, our tool decodes one instruction at a time from the file offset that corresponds to *pc*. This instruction is then translated into one or more statements of the intermediate language (IL). Depending on the abstract domain chosen for the analysis, JAKSTAB calculates successor states by interpreting the abstract semantics of the IL. If a newly reached state is an error state according to the specification, an abstract error trace is generated. Otherwise, JAKSTAB concretizes new *pc* values from the states and uses these to decode the next instructions to be interpreted.

A. Low Level Intermediate Language

CISC architectures such as x86 offer very rich instruction sets, in which a single instruction can affect multiple registers and status flags and can even represent non-trivial operation sequences including loops. To avoid dealing with hundreds of different concrete and abstract state transformers when analyzing machine code, we translate each instruction into a sequence of IL statements using specifications of the instruction semantics. For instance, the instruction push eax, which pushes the contents of register eax to the stack and decrements the stack pointer, is specified to translate to the IL code $m[esp] := eax; esp := esp-4$. Note that for simplicity of the exposition, in this paper we assume all memory accesses and all bit vectors to be 32 bit. The actual implementation allows arbitrary word lengths using bit masking expressions.

The IL uses a finite set of bit vector type registers $V = \{v_0, \ldots, v_n\}$, a store $m[\cdot]$, and the program counter pc. The set $\mathbf{Exp}$ of expressions of the IL contains common arithmetic, Boolean, and bit-manipulation operations. All expressions are of the 32-bit bit vector type $\mathbb{I}_{32}$; Boolean *true* and *false* are represented by the bit vectors 1 and 0, respectively. To model input from the hardware, expressions can contain the keyword $nondet$, which nondeterministically evaluates to some bit vector value in its concrete semantics.

A program is made up of IL statements of the form $[stmt]^{\ell}_{\ell'}$, where $\ell \in \mathbb{I}_{32}$ is the address of the statement, $\ell' \in \mathbb{I}_{32}$ is the address of the next statement, and $stmt \in \mathbf{Stmt}$ is one of nine types of statements:

- Register assignments $v := e$, with $v \in V$ and $e \in \mathbf{Exp}$, assign the value of expression e to register v.
- Store assignments $m[e_1] := e_2$, with $e_1, e_2 \in \mathbf{Exp}$, assign the value of expression e_2 to the memory location at the address computed by evaluating e_1.
- Guarded jumps if e_1 jmp e_2, with $e_1, e_2 \in \mathbf{Exp}$, transfer control to the target address resulting from evaluating e_2 if the guard expression e_1 does not evaluate to 0.
- A halt statement terminates execution.
- Allocation statements alloc v, e, with $v \in V$ and $e \in \mathbf{Exp}$, reserve a block of memory of the size determined by evaluating e and write the address to register v.
- Deallocation statements free v release the block of memory pointed to by $v \in V$ for reallocation.
- Statements assume e terminate execution if $e \in \mathbf{Exp}$ evaluates to 0, and do nothing otherwise.
- Assertions assert e are similar to assume statements, but signal an error on termination.
- Statements havoc $v <_u n$, with $v \in V, n \in \mathbb{I}_{32}$, nondeterministically assign a value x with $0 \leq_u x <_u n$ to v, where $\leq_u$ denotes unsigned comparison. The same effect can be achieved using $v := nondet$; assume $v <_u n$. The point of having two different sources of nondeterminism becomes apparent in Section III-C, where they will be used for selective abstraction.

The statements alloc, free, assert, and havoc are never generated from regular instructions, but are encoded in our abstracted model of the operating system (Section IV-C).

Note that call and return instructions receive no special treatment in our IL but are translated to assignments and jumps. In x86 assembly, these instructions simply store the current program counter on the stack and jump to a target, or read an address from the stack and jump to it, respectively. There is no fixed concept of procedures in x86 assembly, so relying on binary code to respect high level procedural structuring can introduce unsoundness into the analysis.

The concrete IL semantics is defined in terms of states $S = \mathbf{Loc} \times \mathbf{Val} \times \mathbf{Store} \times \mathbf{Heap}$, consisting of the location

$$\mathbf{post}[\![[v := e]^{\ell}_{\ell'}]\!](s) := s[v \mapsto \mathbf{eval}[\![e]\!](s)][pc \mapsto \ell']$$
$$\mathbf{post}[\![[m[e_1] := e_2]^{\ell}_{\ell'}]\!](s) := s[m[\mathbf{eval}[\![e_1]\!](s)] \mapsto \mathbf{eval}[\![e_2]\!](s)][pc \mapsto \ell']$$
$$\mathbf{post}[\![[\mathsf{if}\ e_1\ \mathsf{jmp}\ e_2]^{\ell}_{\ell'}]\!](s) := \begin{cases} s[pc \mapsto \ell'] & \text{if } \mathbf{eval}[\![e_1]\!](s) = 0 \\ s[pc \mapsto \mathbf{eval}[\![e_2]\!](s)] & \text{otherwise} \end{cases}$$
$$\mathbf{post}[\![[\mathsf{halt}]^{\ell}_{\ell'}]\!](s) := \bot$$
$$\mathbf{post}[\![[\mathsf{alloc}\ v, e]^{\ell}_{\ell'}]\!](s) := s[v \mapsto h][pc \mapsto \ell'],\ \text{min. } h > h_0 \text{ s.t.}$$
$$\forall (h', z') \in s(H).h \geq h' + z' \vee h + z \leq h', \text{where } z = \mathbf{eval}[\![e]\!](s)$$
$$\mathbf{post}[\![[\mathsf{free}\ v]^{\ell}_{\ell'}]\!](s) := s[H \mapsto H \setminus (v, \cdot)][pc \mapsto \ell']$$
$$\mathbf{post}[\![[\mathsf{assume}\ e]^{\ell}_{\ell'}]\!](s) := \begin{cases} \bot & \text{if } \mathbf{eval}[\![e_1]\!](s) = 0 \\ s[pc \mapsto \ell'] & \text{otherwise} \end{cases}$$
$$\mathbf{post}[\![[\mathsf{assert}\ e]^{\ell}_{\ell'}]\!](s) := \begin{cases} \bot(\text{raise error}) & \text{if } \mathbf{eval}[\![e_1]\!](s) = 0 \\ s[pc \mapsto \ell'] & \text{otherwise} \end{cases}$$
$$\mathbf{post}[\![[\mathsf{havoc}\ v <_u n]^{\ell}_{\ell'}]\!](s) := s[v \mapsto x][pc \mapsto \ell'],\ \text{with some } x \leq n$$

Fig. 2. Concrete semantics of the intermediate language.

valuation $\mathbf{Loc} := \{pc\} \to \mathbb{I}_{32}$, the register valuation $\mathbf{Val} := V \to \mathbb{I}_{32}$, the store valuation $\mathbf{Store} := \mathbb{I}_{32} \to \mathbb{I}_{32}$, and a heap set $\mathbf{Heap} := \mathbb{I}_{32} \to \mathbb{I}_{32}$, which maps addresses of allocated heap objects to their corresponding sizes. Allocation of heap objects starts above some constant h_0 in the address space. We denote access to parts of the state by $s(pc)$, $s(v_i)$, $s(m[\cdot])$, $s(H(p))$. The syntax $s[\cdot \mapsto \cdot]$ denotes the state obtained by updating part of state s with a new value. The concrete semantics is then given by the concrete $\mathbf{post}$ operator from states and statements to states in Figure 2. It uses the operator $\mathbf{eval} :: \mathbf{Exp} \to \mathbb{I}_{32}$ to concretely evaluate IL expressions.

B. Control Flow Reconstruction

In [8], we proposed an integrated theoretical framework for building the most precise control flow graph of a low level program while calculating data flow facts, akin to control flow analysis in functional programming languages. The basic idea of the framework is to translate low level statements into edges ($\mathbb{I}_{32} \times \mathbf{Stmt} \times \mathbb{I}_{32}$) of the control flow automaton (a control flow graph where edges instead of vertices carry the statements). The edges over-approximate the concrete control flow of the program, eliminating any indirect jumps.

In particular, every guarded jump $[\mathsf{if}\ e_1\ \mathsf{jmp}\ e_2]^{\ell}_{\ell'}$ is transformed into a set E of edges labeled with assume statements: If $e_1 = 0$, E contains the fall-through edge $(\ell, \mathsf{assume}\ (e_1 = 0), \ell')$. If $e_1 \neq 0$, E also contains all of the possible target edges $\{(\ell, \mathsf{assume}\ (e_1 \neq 0 \wedge e_2 = \ell''), \ell'') \mid \ell'' \in \widehat{\mathbf{eval}}[\![e_2]\!](\widehat{\mathbf{post}}[\![\mathsf{assume}\ (e_1 \neq 0)]\!](s))\}$, where $\widehat{\mathbf{post}}$ and $\widehat{\mathbf{eval}}$ denote the abstract $\mathbf{post}$ and $\mathbf{eval}$ operator of a suitable abstract domain, respectively. The key feature that allows this approach to produce the most precise control flow automaton is that the conditions for taking a particular edge from a guarded jump, i.e., the jump condition and the jump target, are encoded into the assumption.

As a result, an abstract domain used with this framework only needs to supply implementations of the $\widehat{\mathbf{post}}$ (for statements other than jmp) and $\widehat{\mathbf{eval}}$ operators and does not need to deal specifically with indirect jumps.

III. Precise Pointer and Value Analysis

The translation of guarded jumps to labeled edges requires a precise evaluation of the target expression, otherwise spurious control flow edges can be introduced that point into code or data sections never meant to be executed, causing a cascading loss of precision. Furthermore, the lack of types in binaries prohibits a limited over-approximation of points-to sets. While in regular source based static analysis an unknown pointer may point to all variables of the matching type, an unknown pointer in untyped assembly code may point to any location in the entire memory, including code.

We have therefore devised a highly precise abstract domain for tracking states as valuations of registers and memory locations that supports pointer arithmetic and the ambiguity between integer values and addresses (there is no distinction between pointers and regular values in machine code).

A. Memory Model

The virtual memory available to a process is organized as one large, continuous array. The stack, the heap, and global variables all share this address space. The runtime environment initializes the stack and heap locations to reasonable values such that they do not interfere, and it uses buffer pages between these logical memory regions to detect overflows. Correct implementations of malloc (and its kernel-level equivalents available to drivers) guarantee that allocated memory blocks in the heap do not overlap. Therefore, we use a concrete memory model based on a set $\mathbf{R}$ of separate *memory regions*:

- The global region, containing code, global variables, and static data,
- a single stack, holding local variables, parameters, and return addresses at runtime,
- and zero or more allocated heap regions, which correspond to memory blocks allocated using malloc.

We thus treat every memory address as a pair of memory region and offset from $\mathbf{R} \times \mathbb{I}_{32}$. Pointers into the global region are denoted by $(\mathsf{global}, \mathit{offset})$; the stack pointer is assumed to be initialized to a value of $(\mathsf{stack}, 0)$. Subsequent modifications to the stack pointer then change the offset, but let it stay within the stack region. In x86, the stack grows downward, so the stack pointer will always have negative offsets within valid code. The number of heap regions is unbounded, and a fresh heap region is created by any call to malloc. A fresh identifier tags the individual heap region, creating pointers such as $(\mathsf{alloc}_{id}, \mathit{offset})$.

Strictly speaking, this memory model presents an abstraction of the actual x86 memory layout, since it ignores the relative position of regions to each other. If for whatever reason the memory region model is too imprecise for the kind of code being analyzed, it can be effectively turned off by initializing the stack pointer and any newly allocated memory into the global address space.

Our memory model combines integer and pointer values similarly to Value Set Analysis [6]; it does not make the assumption of separated procedure stack frames, however, but uses a single region for the entire stack instead.

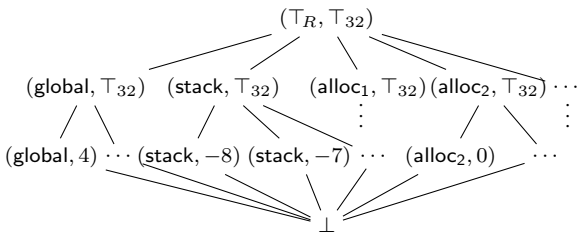
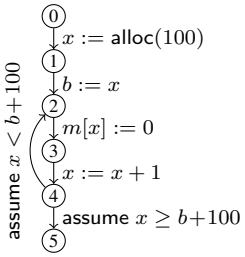


Fig. 3. Diagram of the lattice of abstract addresses $\hat{A}$.



ℓ	# x	# b
0	1	1
1	1	1
2	6	1
3	6	1
4	6	1
5	1	1

x, # b: Number of unique values for x and b.

Fig. 4. Example code fragment and final value counts.

B. Bounded Address Tracking

To build our abstract domain, we extend the model of memory addresses to a lattice that includes a top element $(\top_R, \top_{32})$ representing a memory address with the unknown region $\top_R$ and unknown offset $\top_{32}$. We further introduce an intermediate level of pointers with known region but unknown offset of the form $(region, \top_{32})$, which represents the join of different addresses within the same region (e.g., $(r, 4) \sqcup (r, 8) = (r, \top_{32})$). We thus define the set of abstract memory addresses as $\hat{A} = \{(\top_R, \top_{32})\} \cup (\mathbf{R} \times \{\top_{32}\}) \cup (\mathbf{R} \times \mathbb{I}_{32})$. The resulting lattice for $\hat{A}$ is sketched in Figure 3.

Our analysis over-approximates the set of reachable concrete states of the program by calculating a fixpoint over the abstract states. Abstract states form the set $\hat{S} = \mathbf{Loc} \times \widehat{\mathbf{Val}} \times \widehat{\mathbf{Store}}$, consisting of an abstract register valuation $\widehat{\mathbf{Val}} := V \to \hat{A}$ and an abstract store $\widehat{\mathbf{Store}} := \hat{A} \to \hat{A}$. The initial state at the entry point of the executable is initialized to $(\ell_{\mathbf{start}}, \{esp \to (\mathsf{stack}, 0)\}, \{(\mathsf{stack}, 0) \to \ell_{\mathbf{end}}, (\mathsf{global}, a_0) \to d_0), \ldots, (\mathsf{global}, a_n) \to d_n)\})$, where $a_0, \ldots, a_n$ denote static data locations in the executable (e.g., initial values for global variables, integer or string constants) and $d_0, \ldots, d_n$ their respective values. Location $\ell_{\mathbf{end}}$ points to a halt statement that catches control flow when the main procedure returns, the esp register is initialized to point to this return address on the stack. All registers and memory locations (including all offsets in all heap regions) not shown are implicitly set to $(\top_R, \top_{32})$.

Our analysis is path sensitive, i.e., it does not join abstract states when control flow recombines after a conditional block or loop. To ensure termination, we introduce bounds on the number of values tracked for each register and memory location (hence the name *Bounded Address Tracking*). In particular, the analysis bounds the number of abstract addresses *per variable per location* that it explicitly tracks and performs widening in two steps. Before calculating abstract successors for a state s at location ℓ, the analysis checks for each register or memory location x whether the total number of unique abstract values for x in all reached states at ℓ exceeds the configurable bound k. If it does, then the value of x is widened to $(r, \top_{32})$, where r is the memory region of x in s. If the number of unique memory regions also exceeds the bound k, then x is widened to $(\top_R, \top_{32})$ (see BOUND rule in Figure 5).

Consider the example code in Figure 4. The single initial abstract state is $(0, \{x \to (\top_R, \top_{32}), b \to (\top_R, \top_{32})\}, \emptyset)$, so there is one unique value per variable. We choose to set the bound k to 5. After creating a new abstract heap region and copying the pointer into b, the analysis enumerates states in the loop $2, 3, 4$ while the edge $(4, \mathsf{assume}\ x \geq b + 100, 5)$ remains infeasible. When the state $(2, \{x \to (\mathsf{global}, 5), b \to (\mathsf{global}, 0)\}, \{(\mathsf{alloc}_1, 0) \to (\mathsf{global}, 0), \ldots\})$ is reached, the analysis counts 6 unique values for x in location 2, and widens x to $(\mathsf{alloc}_1, \top_{32})$. This causes a weak update to alloc_1 once x is dereferenced. At the end of the loop, both assume edges are now feasible, and the analysis reaches a fixpoint.

The abstract semantics of Bounded Address Tracking is given using the abstract evaluation operator $\widehat{\mathbf{eval}} :: \mathbf{Exp} \to \hat{S} \to \hat{A}$, the bounding operator $\mathbf{bound} :: \hat{S} \to (V \cup \hat{A}) \to \hat{S}$, and the abstract transfer function $\widehat{\mathbf{post}} :: \mathbf{Stmt} \to \hat{S} \to 2^{\hat{S}}$ from statements and abstract states to sets of abstract states defined in Figure 5. A worklist algorithm extended to apply and adapt precision information [11] (in our case bounds over the number of abstract values) enforces the bound for all registers and memory locations before calculating the abstract transfer function.

Global addresses (global, n) are absolute integers and thus expressions over them are calculated concretely (first case of EVALOP). Addresses for other regions have no statically known absolute value, so only additions of positive or negative integers to their offset can be precisely modeled (second and third case); if pointers to different regions are added or pointers are involved in other types of expressions (including comparisons), the resulting abstract value is safely over-approximated to $(\top_R, \top_{32})$ (fourth case). Other operations (bit extraction, sign extension, etc.) are interpreted analogously. Explicit nondeterminism in expressions evaluates to $(\top_R, \top_{32})$, and memory reads are interpreted by joining the values stored at the addresses in the concretization of the abstract pointer.

A register assignment is interpreted concretely and replaces an existing mapping in the new abstract state. For an assignment to a memory location (i.e., an assignment to a dereferenced pointer), we distinguish three cases depending on the abstract value of the pointer. We can perform a:

- *Strong update*, if both region and offset of the pointer are known. A strong update allows to replace the old value of the memory location in the new state.
- *Weak update to a single region*, if the region of the pointer is known but the offset is $\top_{32}$. Since the precise offset is not known, all memory locations in the region *may* hold the new value, so the existing values have to be joined

$$
\begin{array}{rll}
\textsc{EvalOp} & \widehat{\mathbf{eval}}[\![e_1 \odot e_2]\!](s) & := \mathsf{let}(r_1,o_1) := \widehat{\mathbf{eval}}[\![e_1]\!](s), (r_2,o_2) := \widehat{\mathbf{eval}}[\![e_2]\!](s) \\
 & & \quad \begin{cases} (\mathsf{global}, o_1 \odot o_2) & \text{if } \odot \text{ not } + \text{ and } r_1 = \mathsf{global} \wedge r_2 = \mathsf{global} \\ (r_1, o_1 + o_2) & \text{if } \odot \text{ is } + \text{ and } r_2 = \mathsf{global} \\ (r_2, o_1 + o_2) & \text{if } \odot \text{ is } + \text{ and } r_1 = \mathsf{global} \\ (\top_R, \top_{32}) & \text{otherwise} \end{cases} \\
\textsc{EvalNondet} & \widehat{\mathbf{eval}}[\![nondet]\!](s) & := (\top_R, \top_{32}) \\
\textsc{EvalMem} & \widehat{\mathbf{eval}}[\![m[e]]\!](s) & := \mathsf{let}(r,o) := \widehat{\mathbf{eval}}[\![e]\!](s) \begin{cases} s(\hat{m}[r,o]) & \text{if } r \neq \top_R \wedge o \neq \top_{32} \\ \bigsqcup_{i \in \mathbb{I}_{32}} s(\hat{m}[r,i]) & \text{if } r \neq \top_R \wedge o = \top_{32} \\ (\top_R, \top_{32}) & \text{if } r = \top_R \wedge o = \top_{32} \end{cases} \\
\textsc{Bound} & \mathbf{bound}(s,x) := \begin{cases} s & \text{if } |\{s(x) \mid s \in \{s' | s'(pc) = \ell\}\}| \leq k \\ s[x \mapsto (\top_R, \top_{32})] & \text{if } |\{r \mid (r,o) = s'(x).s' \in \{s'' | s''(pc) = \ell\}\}| > k \\ \mathsf{let}(r,o) = s(x).s[x \mapsto (r, \top_{32})] & \text{otherwise} \end{cases} & \\
\textsc{AssignReg} & \widehat{\mathbf{post}}[\![[v := e]^{\ell}_{\ell'}]\!](s) & := \left\{ s[v \mapsto \widehat{\mathbf{eval}}[\![e]\!](s)][pc \mapsto \ell'] \right\} \\
\textsc{AssignMem} & \widehat{\mathbf{post}}[\![[m[e_1] := e_2]^{\ell}_{\ell'}]\!](s) & := \mathsf{let}(r,o) := \widehat{\mathbf{eval}}[\![e_1]\!](s),\ a := \widehat{\mathbf{eval}}[\![e_2]\!](s),\ s' := s[pc \mapsto \ell'] \\
 & \textit{strong update} & \quad \{s'[\hat{m}[r,o] \mapsto a]\} \quad \text{if } r \neq \top_R \wedge o \neq \top_{32} \\
 & \textit{weak update single region} & \quad \{s'[\hat{m}[r,i] \mapsto s(\hat{m}[r,i]) \sqcup a][\ldots] \text{ for all } i \in \mathbb{I}_{32}\} \quad \text{if } r \neq \top_R \wedge o = \top_{32} \\
 & \textit{weak update all regions} & \quad \{s'[\hat{m}[r,i] \mapsto s(\hat{m}[j,i]) \sqcup a][\ldots] \text{ for all } j \in \mathbf{R}, i \in \mathbb{I}_{32}\} \quad \text{if } r = \top_R \\
\textsc{Alloc} & \widehat{\mathbf{post}}[\![[\mathsf{alloc}\ v, e]^{\ell}_{\ell'}]\!](s) & := \left\{ s[v \mapsto (r,0)][pc \mapsto \ell'] \text{ where } r \text{ is a fresh region identifier} \right\} \\
\textsc{Free} & \widehat{\mathbf{post}}[\![[\mathsf{free}\ v]^{\ell}_{\ell'}]\!](s) & := \mathsf{let}(r,o) := s(v),\ s' := s[pc \mapsto \ell'] \\
 & & \quad \begin{cases} \emptyset \text{ (raise error)} & \text{if } r = \top_R \vee o \neq 0 \\ \{s'[\hat{m}[r,i] \mapsto (\top_R, \top_{32})][\ldots] \text{ for all } i \in \mathbb{I}_{32}\} & \text{otherwise} \end{cases} \\
\textsc{Assume} & \widehat{\mathbf{post}}[\![[\mathsf{assume}\ e]^{\ell}_{\ell'}]\!](s) & := \begin{cases} \emptyset & \text{if } \widehat{\mathbf{eval}}[\![e]\!](s) = (\mathsf{global}, 0) \\ \{s[pc \mapsto \ell']\} & \text{otherwise} \end{cases} \\
\textsc{Assert} & \widehat{\mathbf{post}}[\![[\mathsf{assert}\ e]^{\ell}_{\ell'}]\!](s) & := \begin{cases} \emptyset \text{ (raise error)} & \text{if } \widehat{\mathbf{eval}}[\![e]\!](s) = (\mathsf{global}, 0) \\ \{s[pc \mapsto \ell']\} & \text{otherwise} \end{cases} \\
\textsc{Havoc} & \widehat{\mathbf{post}}[\![[\mathsf{havoc}\ v <_u n]^{\ell}_{\ell'}]\!](s) & := \left\{ s[v \mapsto (\mathsf{global}, i)][pc \mapsto \ell'] \mid i <_u n, i \in \mathbb{I}_{32} \right\}
\end{array}
$$

Fig. 5. Definition of abstract evaluation and abstract post operators for Bounded Address Tracking.

with the new value (with respect to the lattice of abstract addresses shown in Figure 3).
Note that this rule makes the assumption that a memory write to a specific region never exceeds the bounds to write to an adjacent heap regions, since the goal of this work is not to prove memory safety but check API specifications. For full soundness, however, we would have to perform a weak update to all regions.

- *Weak update to all regions*, if neither region nor offset of the pointer are known. All memory locations in all regions have to be joined with the new value.
  In practice, the state becomes too imprecise to continue analysis. In particular, all return addresses will be affected by the weak update. Our implementation thus signals an error for writing to an unknown (possibly also null) pointer in this case.

Besides the fact that region and offset have to be known, there is another prerequisite for performing strong updates: The region of the pointer must not be a *summary region*, i.e., on all execution paths, the abstract region corresponds only to one concrete memory region [7]. Our analysis never creates summary regions, which can be seen from the ALLOC rule in Figure 5. New regions are tagged with fresh, unique identifiers. The only way the abstract region value of a pointer can represent multiple regions is if the number of regions for the pointer exceeds the value bound k and is joined to $\top_R$. In this case, a weak update to all regions will be performed when the pointer is dereferenced, which is a sound over-approximation for an assignment to a summary region.

The abstract post operator for free sets all memory locations in the freed region to $(\top_R, \top_{32})$. The abstract semantics for assume and assert is similar to the concrete case and only adapted to the abstract address model. The abstract post for havoc is the only implementation that returns a non-singleton set: It splits abstract states by enumerating absolute integer values for the given register.

C. Abstraction of Nondeterminism

Abstraction by approximating multiple concrete program states with abstract states is the key to achieving scalability of an analysis. In static analysis, abstraction is introduced by choosing a suitable abstract domain for the program to be analyzed. In software model checking, an iterative refinement finds a suitable abstraction by adding new predicates over program variables. Control flow reconstruction from binaries requires concrete values for jump targets, however, and the

lack of types requires precise values for pointer offsets. Therefore, existing mechanisms for abstraction are not well-suited for a precise analysis of binaries. Still, abstraction has to be introduced to make the analysis feasible.

Even though Bounded Address Tracking resembles software model checking in the way that states from different paths are not merged, it allows registers and memory locations to be *unknown*, i.e., set to $(\top_R, \top_{32})$. This is especially useful when representing nondeterminism in the execution environment (e.g., input, unspecified behavior). Setting parts of the state to *unknown* avoids an exponential enumeration of value combinations. When designing the environment model for a program, we often have a good idea of what needs to be precisely modeled and where we can safely over-approximate. For instance, the standard calling convention of the Windows API specifies that upon return the contents of registers eax, ecx, and edx are undefined. Enumerating all possible values for the registers in a full explicit state exploration would require creating 2^{96} states. By abstracting the nondeterministic choice of values to $(\top_R, \top_{32})$ for all three registers, we only need a single abstract state. It is extremely unlikely to produce a spurious counterexample from this abstraction, since code should not depend on undefined side-effects.

On the other hand, there are occasions when abstracting to an unknown value increases the requirements for the abstract domain. Consider the following code, which is a stub for the Windows API function IoCreateSymbolicLink:

```
int choice = nondet32;          mov eax, nondet32
if (choice == 0)                neg eax
 return STATUS_SUCCESS;         sbb eax, eax
else                            and eax, 0xC0000001
 return STATUS_UNSUCCESSFUL;    ret
```

Here, the compiler replaced the if-statement with bit-manipulation of the return value. Our abstract domain can only deduce that eax is $(\top_R, \top_{32})$ at the return statement, even though eax actually can be only either `0` or `0xC0000001`. Therefore we added the havoc statement to the IL; it causes the analysis to generate multiple successor states with different integer values for a register (HAVOC in Figure 5). With it, we can change the first line of the stub to `int choice; havoc(choice, 1)`. This causes the analysis to create two states; one with eax set to 0, and one with eax set to 1. From these states it can easily compute the two possible states at the return statement: In the first case eax becomes `0`, in the second case `0xC0000001`.

IV. IMPLEMENTATION

We have implemented the architecture and approach described in this paper in our binary analysis platform JAKSTAB (**Ja**va tool**k**it for **sta**tic analysis of **b**inaries). As input, JAKSTAB is able to process Windows PE files (the format used in 32-bit Windows for `.exe`, `.dll`, `.sys`, and more), unlinked COFF object files, and Linux ELF executables. It can load an executable in combination with multiple dynamic libraries and will resolve dependencies between the files.

A. Instruction Sets

JAKSTAB currently supports only the x86 architecture, but can be extended to other architectures by supplying an opcode table and a description of instruction semantics. Instructions are specified using the semantic specification language of the Boomerang decompiler [12], [13]. We used Boomerang's existing x86 specifications as a starting point, which we rewrote and extended heavily.

Our current description of x86 instruction semantics covers over 500 instructions, which includes all instructions that we encountered in the executables analyzed during the experiments. Large parts of the floating point instruction set and the various SSE extensions are supported. The instruction semantics are specified on the level of registers and flags, I/O instructions are specified to read nondeterministic values.

B. Analysis Architecture

JAKSTAB's analysis architecture is based on the Configurable Program Analysis API by Beyer et al. [9], [11], which allows to seamlessly combine state splitting and state joining analyses such as predicate abstraction and interval analysis, respectively. For the work described in this paper, we used only our Bounded Address Tracking domain combined with the trivial location domain that expands the state space of the program to at least one state per IL statement.

C. OS Abstraction and Driver Harness

Executables in general and drivers in particular frequently interact with the operating system. As in source based analyses, we abstract system calls using stubs, which model the relevant side effects such as memory allocation or the execution of callback routines. Following the approach of the source code software model checker SDV [2], we load the driver into JAKSTAB together with a separate *harness* module, that includes system call abstractions relevant to drivers and contains a main function that nondeterministically exercises the driver's initialization and dispatch routines. The harness is written in C and compiled into a dynamic library (DLL) for loading; it is based on SDV's `osmodel.c` and follows SDV's invocation scheme for plug&play drivers. For our experiments, we manually encoded specifications in the harness by inserting state variables and assertions at the locations where SDV places hooks into its specification files.

Several parts of the SDV harness and rules had to be modified to make it suitable for binary analysis. For example, the preprocessor macro IoMarkIrpPending, which sets a bit in the control word of interrupt request packets (IRPs), is intercepted by SDV to change the state for the *PendedCompletedRequest* rule. Since macro invocations are no longer explicit in the binary, we had to modify the rule's assertion to check the bit directly instead of a separate state variable. Furthermore, we replaced SDV's statement for nondeterminism by either havoc or *nondet*, depending on the context.

The IL statements alloc, free, havoc, and assert are exclusively generated by the harness, since they do not correspond to any real x86 instructions. These statements are encoded

	DDA/x86			JAKSTAB					
Driver	**Instr**	**Time**	**Result**	**k**	**k_h**	**States**	**Instr**	**Time**	**Result**
vdd/dosioctl/krnldrvr/krnldrvr.sys	2824	14s	✓	28	5	378	413	2s	OK
general/ioctl/sys/sioctl.sys	3504	13s	✓	28	5	3947	630	7s	✓
general/tracedrv/tracedrv/tracedrv.sys	3719	16s	✓	28	5	486	439	2s	✓
general/cancel/startio/cancel.sys	3861	12s	✓	28	5	633	759	2s	✓
general/cancel/sys/cancel.sys	4045	10s	✓	28	5	600	780	2s	✓
input/moufiltr/moufiltr.sys	4175	3m 3s	×	28	5	3830	722	9s	×
general/event/sys/event.sys	4215	20s	✓	28	5	663	690	2s	✓
input/kbfiltr/kbfiltr.sys	4228	2m 53s	×	28	5	3834	726	8s	×
general/toaster/toastmon/toastmon.sys	6261	4m 1s	✓	28	25	4853	977	9s	✓
storage/filters/diskperf/diskperf.sys	6584	3m 17s	✓	28	5	19772	1409	46s	✓
network/modem/fakemodem/fakemodem.sys	8747	11m 6s	✓	28	5	13994	1887	24s	$\times_m$
storage/fdc/flpydisk/flpydisk.sys	12752	1h 6m	FP	100	35	186543	1782	39m34s	✓
input/mouclass/mouclass.sys	13380	40m 26s	FP	28	28	3055	1763	8s	FP_c
input/mouser/sermouse.sys	13989	1h 4m	FP	28	28	1888	1293	4s	FP
kernel/serenum/SerEnum.sys	14123	19m 41s	✓	28	25	5213	1503	8s	✓
wdm/1394/driver/1394diag/1394DIAG.sys	23430	1h33m	FP	28	28	2181	2426	4s	FP_m
wdm/1394/driver/1394vdev/1394VDEV.sys	23456	1h38m	FP	28	28	2837	2872	5s	FP_m

Fig. 6. Comparison of experimental results on Windows DDK drivers between DDA/x86 (on a 3GHz Xeon) and JAKSTAB (on a 3GHz Opteron).

into the compiled harness object file using illegal instructions, which are directly mapped to the corresponding IL statements during disassembly. For instance, an alloc statement can be generated from the C source of the harness by inlining the assembly instruction `lock rep inc eax`.

V. EXPERIMENTS

For direct comparison with the IDA Pro and CodeSurfer/x86 based binary driver analyzer DDA/x86 described in [1], we ran JAKSTAB on the same set of drivers from the Windows Driver Development Kit (DDK) release 3790.1830 and checked the same specification *PendedCompletedRequest*. The rule specifies that a driver must not call IoCompleteRequest and return STATUS_PENDING unless it invokes the IoMarkIrpPending macro on the IRP being processed. We compiled the drivers without debug information using default settings. Note that unlike [1], we did not compile and link the driver source code against the harness; our approach is directly applicable to drivers without access to source code.

Our experimental results are listed alongside those reported in [1] in Figure 6. The number of instructions include instructions from the harness in both cases. Note that the tools report very different numbers of instructions for the same binaries; this is due to the fact that JAKSTAB disassembles instructions only on demand, i.e., if they are reachable by the analysis. In contrast, CodeSurfer/x86 uses IDA Pro as front end, which heuristically disassembles all likely instructions in the executable. Since for DDA/x86 the entire harness was compiled and linked with the driver, IDA Pro disassembled all code from the harness, including code that is unreachable from the driver under analysis. Conversely, it is possible that some driver code is unreachable from the harness. For the experiments we used two value bounds which we determined empirically; k shows the value bound for registers and stack locations, k_h the value bound for memory locations in allocated heap regions.

For `flpydisk.sys`, JAKSTAB was able to verify the specification, while DDA/x86 found a false positive (FP). This is due to the only limited degree of path sensitivity in DDA/x86, which follows the ESP approach [14] for differentiating paths based on states of a property automaton. In [1], the property automaton is extended to track updates to the variable holding the return value, but it can miss updates due to its heuristic for detecting interprocedural dependencies for the return value.

In `fakemodem.sys`, JAKSTAB encountered a potentially unsafe memory access (marked as $\times_m$), where an uninitialized value, i.e., $(\top_R, \top_{32})$, is used as the index for a write to an array. We manually confirmed the feasibility of the error trace for the execution environment simulated by the harness. DDA/x86 does not check for memory safety due to the large number of false positives [1], so it did not detect this bug. As mentioned in Section III-B, our analysis signals an error on weak updates to all regions. This amounts to implicitly checking for write accesses to uninitialized pointers, which allows JAKSTAB to detect the error. As a consequence of building on the SDV harness, which is not designed for checking memory safety and often omits proper pointer allocation, our analysis yielded false positives where the result shows FP_m in Figure 6. In `mouclass.sys`, a switch jump could not be resolved because the switch variable was over-approximated leading to a false positive of invalid control flow (FP_c). Currently, we manually investigate abstract error traces and extend the harness if necessary to eliminate false positives. We leave a partial or full automation for future work.

The comparison of execution times should be taken with a grain of salt, since both prototypes were run on different machines. DDA/x86 was run on a 64-bit Xeon 3GHz processor with 4GB of memory per process, while the experiments with

JAKSTAB were conducted on a 64-bit AMD Opteron 3GHz processor with 4GB of Java heap space (we report the average time of 10 runs per driver). Still, it is possible to see that execution times for JAKSTAB appear favorable overall.

We do not have to recompile and link drivers with the harness, so we were able to extend our experiments beyond the Windows DDK. We ran our prototype on all 322 drivers from the `system32\drivers` directory of a regular 32-bit Windows XP desktop system, using $k = 28$ and $k_h = 5$. Besides the *PendedCompletedRequest* rule, we also checked the *CancelSpinLock* rule, which enforces that a global lock is acquired and released in strict alternation. Note that this set of drivers also includes classes of drivers which are not even supported by the SDV harness in source code analysis, such as graphics drivers. Nonetheless, we were able to successfully analyze 28% of these drivers. For 41% of the drivers, analysis failed because of weak global updates, mostly due to missing information about pointer allocation in the harness. In 31% of the cases, the analysis failed due to unknown or erroneous control flow, which can be again caused by unknown side effects of API functions not supported by the harness, or by coarse abstraction of variables used in switch jumps. Two drivers timed out after 1 hour; in three drivers the analysis found potential assertion violations. To our knowledge, this is the first time static analysis was successfully applied to real world, closed source, binary driver executables.

VI. Related Work and Discussion

Bounded Address Tracking was inspired by the Explicit Analysis of Beyer et al. [11], which tracks explicit values of integer variables of C programs up to a certain bound. In their work, explicit analysis is used for cheap enumeration of values for a variable before it is modeled by the computationally more expensive predicate abstraction.

As pointed out already, the CodeSurfer/x86 project is most closely related to our work and faces similar challenges. The major differences in approach are that CodeSurfer/x86 is implemented on top of the heuristics based IDA Pro, and that its analyses (in particular Value Set Analysis (VSA) [6]) are based on more "classic" static analyses such as interval analysis. VSA is path insensitive and thus requires the use of call strings for reasonable results. Call strings, however, are tied to the concept of procedures (which is unreliable in x86 assembly) and assume the existence of a separate call stack. This issue lead us to the design of the bounded path sensitive analysis presented in this paper.

Balakrishnan and Reps generally rely on summary nodes for representing heap objects. They reduce the number of weak updates by introducing a *recency abstraction* [15] of heap nodes. Their approach extends the common paradigm of using one summary node per allocation site (i.e., address of the call to malloc), by splitting this summary node into (i) the region most recently allocated in the current execution path and (ii) a summary node for the remaining regions. In contrast, our approach instead explicitly discriminates allocated regions up to the value bound.

VII. Summary

In this paper, we presented a framework for precise static analysis of driver binaries. Compared to existing approaches, it significantly reduces the sources of unsoundness by eliminating the separate, error-prone disassembly step. We introduced Bounded Address Tracking, an abstract domain which allows strong updates to memory locations on the heap, as long as the number of different pointer values stays below a definable bound. Experiments on several driver binaries confirm the feasibility of our approach on small, but real world code and demonstrate its improved performance compared to state of the art approaches in spite of increased precision. Moreover, we tried our approach on all drivers of a regular desktop system and achieved encouraging results.

For scaling up to larger programs, however, we will attempt to reduce precision where it is not required. One approach is to reduce the value bound individually for variables not involved with control flow or specifications. Starting from a generally low bound, an iterative refinement loop can help to identify memory locations and function stubs in the harness where increased precision is required. Furthermore, we will investigate the use of summaries that do not require assumptions on procedure structure or calling conventions.

Acknowledgments

The authors would like to thank Vlad Levin for discussing SDV, Gogul Balakrishnan for feedback on DDA/x86, and Peter Bokor and the anonymous reviewers for their detailed comments on the paper. This work was supported by CASED.

References

[1] G. Balakrishnan and T. Reps, "Analyzing stripped device-driver executables," in *TACAS*, ser. LNCS. Springer, 2008, pp. 124–140.
[2] T. Ball, E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, B. Ondrusek, S. K. Rajamani, and A. Ustuner, "Thorough static analysis of device drivers," in *Proc. 2006 EuroSys Conf.* ACM, 2006, pp. 73–85.
[3] Microsoft. Windows Driver Kit documentation. [Online]. Available: http://msdn.microsoft.com/en-us/library/ff557573(VS.85).aspx
[4] B. Schwarz, S. Debray, and G. Andrews, "Disassembly of executable code revisited," in *WCRE*. IEEE Computer Society, 2002, pp. 45–54.
[5] Hex-Rays SA. IDA Pro. [Online]. Available: http://www.hex-rays.com/idapro/ [Accessed: July 26, 2010]
[6] G. Balakrishnan and T. W. Reps, "Analyzing memory accesses in x86 executables." in *CC*, ser. LNCS, vol. 2985. Springer, 2004, pp. 5–23.
[7] D. R. Chase, M. N. Wegman, and F. K. Zadeck, "Analysis of pointers and structures," in *PLDI*, 1990, pp. 296–310.
[8] J. Kinder, H. Veith, and F. Zuleger, "An abstract interpretation-based framework for control flow reconstruction from binaries," in *VMCAI*, ser. LNCS, vol. 5403. Springer, 2009, pp. 214–228.
[9] D. Beyer, T. Henzinger, and G. Théoduloz, "Configurable software verification: Concretizing the convergence of model checking and program analysis," in *CAV*, ser. LNCS, vol. 4590. Springer, 2007, pp. 504–518.
[10] J. Kinder and H. Veith, "Jakstab: A static analysis platform for binaries," in *CAV*, ser. LNCS, vol. 5123. Springer, 2008, pp. 423–427.
[11] D. Beyer, T. Henzinger, and G. Théoduloz, "Program analysis with dynamic precision adjustment," in *ASE*. IEEE, 2008, pp. 29–38.
[12] M. van Emmerik and T. Waddington, "Using a decompiler for real-world source recovery." in *WCRE*. IEEE Computer Society, 2004, pp. 27–36.
[13] C. Cifuentes and S. Sendall, "Specifying the semantics of machine instructions." in *IWPC*. IEEE Computer Society, 1998, pp. 126–133.
[14] M. Das, S. Lerner, and M. Seigle, "ESP: Path-sensitive program verification in polynomial time," in *PLDI*. ACM, 2002, pp. 57–68.
[15] G. Balakrishnan and T. Reps, "Recency-abstraction for heap-allocated storage," in *SAS*, ser. LNCS, vol. 4134. Springer, 2006, pp. 221–239.

Verifying SystemC: a Software Model Checking Approach

Alessandro Cimatti Andrea Micheli Iman Narasamdya Marco Roveri
Fondazione Bruno Kessler – FBK-irst – Embedded System Unit – Italy
{cimatti,amicheli,narasamdya,roveri}@fbk.eu

***Abstract*—SystemC is becoming a de-facto standard for the development of embedded systems. Verification of SystemC designs is critical since it can prevent error propagation down to the hardware. SystemC allows for very efficient simulations before synthesizing the RTL description, but formal verification is still at a preliminary stage. Recent works translate SystemC into the input language of finite-state model checkers, but they abstract away relevant semantic aspects, and show limited scalability.**

In this paper, we approach formal verification of SystemC by reduction to software model checking. We explore two directions. First, we rely on a translation from SystemC to a sequential C program, that contains both the mapping of the SystemC threads in form of C functions, and the coding of relevant semantic aspects (e.g. of the SystemC kernel). In terms of verification, this enables the "off-the-shelf" use of model checking techniques for sequential software, such as lazy abstraction.

Second, we propose an approach that exploits the intrinsic structure of SystemC. In particular, each SystemC thread is translated into a separate sequential program and explored with lazy abstraction, while the overall verification is orchestrated by the direct execution of the SystemC scheduler. The technique can be seen as generalizing lazy abstraction to the case of multi-threaded software with exclusive threads and cooperative scheduling.

The above approaches have been implemented in a new software model checker. An experimental evaluation carried out on several case studies taken from the SystemC distribution and from the literature demonstrate the potential of the approach.

I. Introduction

The development of System-on-Chips (SoCs) is often started by writing an executable model, using high-level languages such as SystemC [1]. Verification of SystemC designs is an important issue, since errors identified in such models can reveal errors in the specification and prevent error propagation down to the hardware.

SystemC is arguably becoming a de-facto standard, since it allows for high-speed simulation before synthesizing the RTL hardware description. However, formal verification of SystemC is still at a preliminary stage. In fact, a SystemC design is a very complex entity, that can be thought of as multi-threaded software, where scheduling is cooperative and carried out according to a specific set of rules [2], and the execution of threads is mutually exclusive.

There have been several works that have tried to apply model checking techniques to complement simulation [3]–[7]. These approaches map the problem of SystemC verification to some kind of model checking problem, but suffer from severe limitations. Some of them disregard significant semantic aspects, e.g., they fail to precisely model the SystemC scheduler or the communication primitives. Others show poor scalability of model checking, because of too many details included in the model.

In this paper we present an alternative approach to the verification of safety properties (in the form of program assertions) of SystemC designs, based on software model checking techniques [8]–[11]. The primary motivation is to investigate the effectiveness of such techniques, that have built-in abstraction capabilities, and have shown significant success in the analysis of sequential software.

We explore two directions. First, we rely on a translation from SystemC to a sequential C program, that contains both the mapping of the SystemC threads in form of C functions, and the coding of relevant semantic aspects (e.g. of the SystemC kernel). In terms of verification, this enables the "off-the-shelf" use of model checking techniques for sequential software.

However, the exploration carried out during software model checking treats in the same way both the code of the threads and the kernel model. This turns out to be a problem, mostly because the abstraction of the kernel is often too aggressive, and many refinements are needed to re-introduce abstracted details.

Thus, we propose an improved approach, that exploits the intrinsic structure of SystemC. In particular, each SystemC thread is translated into a separate sequential program and explored with lazy abstraction, i.e. by constructing an abstract reachability tree as in [8], [12]. The overall verification is orchestrated by the direct execution of the SystemC scheduler, with techniques similar to explicit-state model checking. This technique, in the following referred to as Explicit-Scheduler/Symbolic Threads (ESST) model checking, is not limited to SystemC: it lifts lazy abstraction to the general case of multi-threaded software with exclusive threads and cooperative scheduling.

We have implemented our approaches into a tool chain that includes a SystemC front-end derived from PINAPA [13], and a new software model checker, called SYCMC, using several extensions built on top of NUSMV and MATHSAT [14]–[16]. We have been experimenting the two approaches on a set of benchmarks taken and adapted from the SystemC distribution, and from other works that are concerned with the verification of SystemC designs. First, we have run several software model checkers on the sequential C programs resulting from the translation of SystemC designs. Finally, we have experimented with the new ESST model checking algorithm. The results, although preliminary, are promising. In particular, the ESST algorithm demonstrates dramatic speed ups over the first approach based on the verification of sequentialized C programs.

The structure of this paper is as follows. In Section II we introduce SystemC. In Section III we reduce model checking

of SystemC designs to model checking of sequential C. In Section IV we present ESST model checking. In Section V we discuss some related work. We present the results of the experimental evaluation in Section VI. Finally, in Section VII we draw conclusions and outline some future work.

II. Background on SystemC

SystemC is a C++ library that consists of (1) a core language that allows one to model a SoC by specifying its components and architecture, and (2) a simulation kernel (or scheduler) that schedules and runs processes (or threads) of components. SoC components are modeled as SystemC modules (or C++ classes). Channels abstract communication between modules, while ports in a module are used to bind the modules with channels. The SystemC library provides primitive channels such as signal, mutex, semaphore and queue.

A module contains one or more threads describing the parallel behavior of the SoC design. The SystemC library also provides general-purpose events used for the synchronization between threads. A thread can suspend itself by waiting for an event `e`, i.e. by calling `wait(e)`, or by waiting for some specified time, i.e. by calling `wait(t)`, for some time unit `t` ≥ 0. A thread can perform immediate notification of an event `e`, by calling `e.notify()`, or delayed notification, by calling `e.notify(t)` for some time unit `t`.

The SystemC scheduler is a cooperative non-preempting scheduler that runs at most one thread at a time. During a simulation, the state of a thread changes from sleeping, to runnable, and to running. A running thread will only give control back to the scheduler by suspending itself. The scheduler runs all runnable threads, one at a time, in a single delta cycle, while postponing the channel updates made by the threads. When there are no more runnable threads, the scheduler materializes the channel updates, and wakes up all sleeping threads that are sensitive to the updated channels. If, after this step, there are some runnable threads, then the scheduler moves to the next delta cycle. Otherwise, it accelerates the simulation time to the nearest time point where a sleeping thread or an event can be woken up. The scheduler quits if there is no more runnable thread after time acceleration.

Listing 1 depicts an excerpt of a simple producer-consumer example in SystemC. It defines the `producer` that has two threads, `write` and `read`. The thread `write` sends the value stored in the variable `d` to the `consumer` by calling the function `put` in the `consumer`, and then wait for the event `e` to be notified. The method thread `read` reads from the channel bound to the input port `p_in` and notify the event `e`. It is sensitive to the input port `p_in`. A method thread only suspends itself by exiting the function and becomes runnable when the channel bound to the port is updated. The function `dont_initialize` makes the thread `read` not runnable at the beginning of simulation. The `consumer` consists of two threads: `compute` and `write_b`. The thread `compute` is triggered by the event `f` notified by the function `put` that was called by the `producer`. The interface `write_if` contains the signature of `put` and is derived from the SystemC interface. The thread `compute`

```
1  SC_MODULE(producer) {
2   private:
3    int d;
4    sc_event e;
5   public:
6    sc_in<int> p_in;
7    sc_port<write_if> p_w;
8    SC_HAS_PROCESS(producer);
9
10   producer(sc_module_name name) : sc_module(name) {
11    SC_THREAD(write);
12    SC_METHOD(read); sensitive << p_in; dont_initialize();
13   }
14
15   void write() {
16    int t;
17    wait(SC_ZERO_TIME);
18    while (1) {
19     t = d;          // Save old value of d.
20     p_w->put(d);    // Write d's value to consumer.
21     wait(e);
22     assert(d == t+1);
23    }
24   }
25
26   void read() { d = p_in.read(); e.notify(); }
27 }
28
29 SC_MODULE(consumer), public write_if {
30  private:
31   int data;
32   sc_event f, g;
33  public:
34   sc_out<int> p_out;
35   sc_export<write_if> ex_w;
36   SC_HAS_PROCESS(consumer);
37
38   consumer(sc_module_name name) : sc_module(name) {
39    ex_w(*this);
40    SC_THREAD(compute);
41    SC_THREAD(write_b);
42   }
43
44   void put(int d) { data = d; f.notify(); }
45
46   void compute() {
47    while (1) {
48     wait(f); ++data; g.notify();
49    }
50   }
51
52   void write_b() {
53    while (1) {
54     wait(g); p_out.write(data);
55    }
56   }
57 };
58
59 int main() {
60   sc_signal<int> s;
61   // Create producer and consumer instances.
62   produce  * p = new producer("P");
63   consumer * c = new consumer("C");
64   // Interconnect signal.
65   p->p_in(s); c->p_out(s);
66   // Interconnect modules.
67   p->p_w(c->ex_w);
68   // Start simulation.
69   sc_start();
70 }
```

Listing 1. Definition of a producer/consumer design in SystemC.

increments the value sent by the producer, and then notifies the event `g` that subsequently activates the thread `write_b`. The thread `write_b` then writes the incremented value to the channel connecting the producer and the consumer through the port `p_out`. Finally, the `main` function shows that the producer `p` and the consumer `c` are connected via the signal channel `s`. The export construct of SystemC allows communication between components without any intermediate channel, as shown by the binding of port `p_w` of `producer` and port `ex_w` of `consumer`.

III. Model Checking SystemC via Sequentialization

In this section we describe the translation from SystemC designs into an equivalent sequential C programs by using the producer-consumer example introduced in the previous section.

A. Translating SystemC to C

In our translation each thread in the SystemC design is translated into a C function. Members of module instances,

```
1  int d;    /* Global variable for producer. */
2  /* Events in the design */
3  int event_e; /* Status of event e. */
4  int event_f; /* Status of event f. */
5  int event_g; /* Status of event g. */
6  /* Local to thread producer::write() */
7  int write_pc;     /* Program counter. */
8  int write_state;  /* Status of thread. */
9  int event_write;  /* Status of thread event. */
10 int t_write;      /* Local variable t */
```

Listing 2. Excerpt of the C preamble.

channels, and events are translated into a set of global variables. We assume that the SystemC design does not contain any dynamic creation of such components. We also assume that each function call in the SystemC thread code can be inlined statically.

To model context switches that occurs during the SystemC simulation, for each thread t, we introduce the following supporting variables: (1) t_pc keeps track of the program counter of the thread; (2) t_state keeps track of the status of the thread, whose possible values are SLEEP, RUNNABLE, or RUNNING; (3) event_t describes the status of the event associated with the thread, whose possible values are DELTA, FIRED, TIME, or NONE; and (4) event_time_t keeps track of the notification time of the event associate with the thread. The status DELTA indicates that the event will be triggered at the transition from current delta cycle to the next one. The status TIME indicates that the event will be triggered at some time in the future. The status FIRED indicates that the event has just been triggered, while the status NONE indicates there is no notification or triggering applied to the event. Similarly, for each event e occurring in the design, we introduce a variable event_e whose values range over event status and a variable time_event_e that keeps track of the notification time. For succinctness of presentation, we do not prefix the above variables with the names of module instances that own the threads. Moreover, when the design has no time notification we omit the TIME status and the variable that keeps track of the notification time.

Member variables of a module instance are visible by all its threads. Thus, they are translated into global variables in the C program. For variables local to some thread, as context switches require saving and restoring such variables, we introduce for every local variable l of thread t a global variable l_t of the same type as l. Saving the value of l means assigning its value to l_t, while restoring the value of l means assigning l_t's value to l. Listing 2 shows the variables introduced for the thread write and for all the events of Listing 1.

Listing 3 shows the result of translating the thread write of producer into a C function. First, the function is annotated with program labels indicating the locations of context switches. The function starts with a jump table whose targets depend on the values of the program counter write_pc that points to the location at which the thread has to resume its execution. Second, we model calls to wait functions and their variants by the following instructions: (1) an assignment setting the thread's status to SLEEP; (2) an assignment setting the thread's program counter to the location where the thread has to resume its execution once it is woken up; (3) assignments saving the values of thread's local variables into the corresponding global variables introduced above; (4) a return statement; (5) a program label representing the location where the thread has to resume its execution; and (6) assignments restoring the values of thread's local variables. For example, for wait(e) in the thread write, we introduce the program label WRITE_WAIT_2 and set the program counter write_pc to WRITE_WAIT_2 before the function returns (see lines 25–32 of Listing 3). In the case of wait(SC_ZERO_TIME) in the thread write, the thread is suspended and will be woken up at the delta-cycle transition. To model this, we set the variable event_write to DELTA.

```
1  void write() {
2    int t;
3    /* Local jump table */
4    if      (write_pc == WRITE_ENTRY)  goto WRITE_ENTRY;
5    else if (write_pc == WRITE_WAIT_2) goto WRITE_WAIT_2;
6    else if (write_pc == WRITE_WAIT_1) goto WRITE_WAIT_1;
7  WRITE_ENTRY:
8    /* wait(SC_ZERO_TIME); _BEGIN_ */
9    write_state = SLEEP;
10   write_pc = WRITE_WAIT_1;
11   event_write = DELTA;
12   t_write = t; /* Save t. */
13   return;
14 WRITE_WAIT_1:
15   t = t_write; /* Restore t. */
16   /* wait(SC_ZERO_TIME); _END_ */
17   while (1) {
18     t = d;
19     /* inline consumer::put _BEGIN */
20     data = d;
21     event_f = FIRED;       /* f.notify() _BEGIN_ */
22     activate_threads();
23     event_f = NONE;        /* f.notify() _END_ */
24     /* inline consumer::put _END_ */
25     /* wait(e) _BEGIN_ */
26     write_state = SLEEP;
27     write_pc = WRITE_WAIT_2;
28     t_write = t; /* Save t. */
29     return;
30 WRITE_WAIT_2:
31     t = t_write; /* Restore t. */
32     /* wait(e) _END_ */
33     assert(d==t+1);
34   }
35 }
```

Listing 3. Sequential thread write of producer.

An event e can be specified to be notified at immediate time or at some time in the future. In the former case, every thread that depends on the notified event has to be triggered. To this end, we introduce for each thread t a function is_t_triggered that returns 1 if the thread is triggered, 0 otherwise. Now immediate notifications can be modeled by the following instructions: (1) an assignment setting the event's status to FIRED; (2) a list of queries checking if threads are triggered, and if they are triggered, their status are set to RUNNABLE; this list is represented by the function activate_thread; and (3) an assignment setting the event's status to NONE. Lines 21–23 of Listing 3 shows the translation of f.notify(). Listing 4 shows the code for thread activation. The notification by e.notify(SC_ZERO_TIME) is modeled similarly to wait(SC_ZERO_TIME), that is, we set the variable event_e to DELTA. To model general time delayed notification, one needs a statement that assigns the delayed notification time to the variable associated with the event that keeps track of the notification time.

Next, we inline the function calls in the SystemC code. For instance, the inlining of the call p_w->put(d) in write is shown in lines 19–24 of Listing 3. As we will discuss later, function inlining can give advantages to the application of software model checking techniques, particularly in the encoding of the threads.

```
int is_write_triggered() {
 if ((write_pc == WRITE_WAIT_1)
     && (event_write == FIRED)) return 1;
 if ((write_pc == WRITE_WAIT_2)
     && (event_e == FIRED)) return 1;
 return 0;
}

void activate_threads() {
 if (is_write_triggered())    write_state = RUNNABLE;
 if (is_compute_triggered()) compute_state = RUNNABLE;
 if (is_write_b_triggered()) write_b_state = RUNNABLE;
 if (is_read_triggered())     read_state = RUNNABLE;
}
```

Listing 4. Thread activation.

A signal channel s is represented by two global variables `s_old` and `s_new`. Writing to and reading from a port bound to the channel is modeled as, respectively, an assignment to `s_new` and an assignment from `s_old`. For each channel, we include the update function of the channel in the resulting C program. For a signal s, the update function simply assigns `s_old` with the value of `s_new` if their values are different.

The SystemC scheduler is included in the C program resulting from the translation. The scheduler is shown in Listing 5. It consists of five phases: the initial phase, the evaluation phase, the update phase, the delta-notification phase, and the time phase. (We based the definition of the scheduler on [2])

In the initial phase all channels are updated by calling the corresponding update functions. The function `init_thread` changes the status of a thread to `SLEEP` if `dont_initialize` is specified for the thread. The function `fire_delta_events` simply changes the status of an event to `FIRED` if it was previously `DELTA`, while the function `reset_events` changes the status to `NONE`. Similarly for the function `fire_time_events`. In the evaluation phase, denoted by function `eval`, all runnable threads are run one at a time. Unlike the original SystemC scheduler that explores only one schedule, in the verification we want to explore all possible schedules. To this end, we use the function `nondet()` that returns a non-deterministic value.

The scheduler enters the update phase when there is no more runnable thread. In the update phase all channels are updated. The scheduler then moves to the delta-notification phase. This phase signifies the transition from the current delta phase to the next one. In this phase the scheduler triggers all events whose status are `DELTA`, and subsequently wakes up triggered events. The time phase is entered if there is no runnable thread after the delta-notification phase. In this phase the scheduler simply accelerates the simulation time. The scheduler quits if there are no more runnable threads. Note that, this encoding of the scheduler admits one impossible schedule where no runnable threads are selected to run. However, the existence of such a schedule is benign given we are focusing on the verification of safety properties.

To complete the translation, all variables related to threads and events must be initialized. The program counter is initialized to the entry label, for example, `write_pc` is initialized to `WRITE_ENTRY`. All variables whose values represent thread status are initialized to `RUNNABLE`, and all variables whose values represent event status are initialized to `NONE`. These initializations are performed in the function `init_model` called by the `main` function.

This translation from SystemC to sequentialized C preserves the behavior of the original SystemC design.

```
void eval() {
  while (exists_runnable_thread()) {
    if (write_state == RUNNABLE && nondet())
      { write_state = RUNNING; write(); }
    if (compute_state == RUNNABLE && nondet())
      { compute_state = RUNNING; compute(); }
    if (write_b_state == RUNNABLE && nondet())
      { write_b_state = RUNNING; write_b(); }
    if (read_state == RUNNABLE && nondet())
      { read_state = RUNNING; read(); }
  }
}

void start_simulation() {
  update_channels();  /* Initialization phase. */
  init_threads();
  fire_delta_events();
  activate_threads();
  reset_events();
  do {
    eval(); /* Evaluation phase. */
    update_channels(); /* Update phase. */
    fire_delta_events(); /* Delta-notification phase. */
    activate_threads();
    reset_events();
    if (!exists_runnable_thread()) {
      fire_time_events(); /* Time-notification phase. */
      activate_threads();
      reset_events();
    }
  } while (exists_runnable_thread());
}

int main() {
  init_model(); start_simulation();
}
```

Listing 5. Sequential SystemC scheduler and `main`.

B. Model Checking (SystemC as) C

The translation from SystemC to C presented above opens up the possibility to reduce the verification of a SystemC design to the problem of verifying the translated C program. Verification of C programs is possible by using existing software model checkers, such as SATABS [17], BLAST [8], and CPACHECKER [18]. We notice that these model checkers implement approaches that are complementary to the ones that have been proposed in the past to verify SystemC.

Among the above approaches, one particularly promising is the idea of model checking via lazy abstraction [10]. The approach is based on the construction and analysis of an abstract reachability tree (ART) using predicate abstraction. The approach can be seen as combining an exploration of the control flow automaton (CFA) of the program with explicit-state techniques, while the data path is analyzed by means of predicate abstraction. (See also [8]–[11], [18] for a more thorough discussion) The ART represents reachable abstract states obtained by unwinding the CFA of the program. An ART node typically consists of a control flow location, a call stack, and a formula representing a region or data states (i.e. assignments to each variable of the program of a value in its domain).

An ART node is expanded by applying the strongest post operator followed by predicate abstraction to the region and the outgoing CFA edge of the location labelling the node [12], [18]. When the expansion reaches an error location, if the path from the root to the node with the error location is feasible, then the path is a counter-example witnessing the error (or assertion violation). Otherwise, the path is analyzed to discover new predicates to track and to determine the point in the ART where to start the refinement to discard the spurious behavior.

Predicate abstraction can benefit from advanced SMT techniques like [15] and [16]. Large block encoding (LBE) for lazy-

abstraction has been proposed in [12] to reduce the number of paths (and nodes) in the ART that have to be explored independently. Intuitively, in LBE each edge in the CFA corresponds to a rooted directed acyclic graph (DAG) in the original CFA. Such an edge can be thought of as a summarization of the corresponding rooted DAG in the original CFA. In LBE function calls and loops in a program require block splitting. As we want to keep the number of blocks as small as possible, one can complementary apply function inlining to calls to non-recursive functions and loop unrolling to the loops whose bounds are known. The refinement can benefit from proof of unsatisfiability and from interpolation based techniques. For instance, in [11] it has been described an interpolation based refinement approach where the relevant predicates at each location of the infeasible path are inferred from the interpolant between the two formulas that define the prefix and the suffix of the path.

The idea of applying software model checking techniques to the C program resulting from the translation of SystemC is, to the best of our knowledge, novel. The hope is that the various abstraction techniques may provide some leverage to tackle the state explosion problem.

However, we remark that the exploration of the ART carried out during software model checking will treat in the same way both the code of the threads and the kernel model. In a sense, a general purpose technique is being applied to programs that have a very specific structure, resulting from the sequentialization of concurrency. In the next section, we propose a generalization to software model checking that exploits this feature of the analyzed problems.

IV. Explicit Scheduler + Symbolic Threads

In this section we propose a novel approach to the verification of SystemC designs. First, unlike the previous approach, here we decouple the scheduler from the threads. That is, the scheduler will no longer be part of the program, but is embedded in the model checking algorithm. Second, we combine the explicit model checking technique with the symbolic one based on lazy predicate abstraction. In this combination we still represent the state of each thread as a formula describing a region. But, unlike the classical lazy abstraction, we keep track of the states of scheduler explicitly. In the following, we refer to this technique as Explicit-Scheduler/Symbolic Threads (ESST) model checking. Fig. 1 shows an overview of this new approach.

We introduce several primitive functions to model SystemC synchronization mechanism and for interacting with the model checking algorithm. For example, the SystemC's wait functions `wait(`t`)` and `wait(`e`)` are modeled by primitive functions `wait(`t`)` and `wait_event(`e`)`, respectively. These primitive functions perform synchronization by updating the state of the scheduler. In the proposed algorithm the scheduler requires precise information about its state in order to schedule the threads. To this end, we assume that in the SystemC design the values of t and e in `wait(`t`)` and `wait_event(`e`)` can be determined statically. This assumption does not limit the applicability of our technique since, to the best of our knowledge, most real SystemC designs satisfy the assumption.

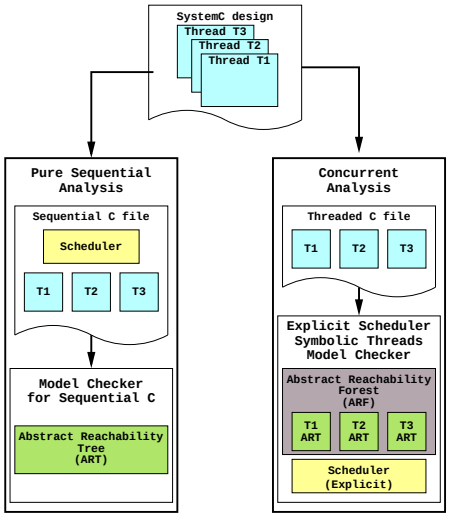


Fig. 1. An overview of the ESST approach.

A. Abstract Reachability Forest

We build an *abstract reachability forest* (ARF) to describe reachable abstract states. An ARF consists of some ART's, each of which is obtained by unwinding the running thread. The connections between one ART with the others in an ARF describe context switches.

For a model with n threads, each node in the ARF is a tuple $(\langle q_1, s_1, \varphi_1\rangle, \ldots, \langle q_n, s_n, \varphi_n\rangle, \varphi, S)$, where (1) q_i, s_i, and φ_i are, respectively, the program location, the call stack, and the region of thread i, (2) φ is the region describing the data state of global variables, and (3) S is the state of scheduler. The state S does all the book keeping necessary to model the behavior of the scheduler. For example, it keeps track of the status of threads and events, the events that sleeping threads are waiting for, and the delays of event notifications.

To expand the ARF, we need to execute primitive functions and to explore all possible schedules. To this end, we introduce the function SEXEC that takes as inputs a scheduler state and a call to a primitive function f, and returns the updated scheduler state obtained from executing f. For example, the state $S' = \text{SEXEC}(S, \texttt{wait\_event(}e\texttt{)})$ is obtained from the state S by changing the status of running thread to sleep, and noting that the now sleeping thread is waiting for an event e.

We also introduce the function SCHED that implements the scheduler. This function takes as an input a scheduler state and returns a set of scheduler states, each of which has exactly one running thread. These resulting states represent all possible schedules.

To describe the expansion of a node in ARF, we assume that there is at most one running thread in the scheduler state of the node. The rules for expanding a node $(\langle q_1, s_1, \varphi_1\rangle, \ldots, \langle q_n, s_n, \varphi_n\rangle, \varphi, S)$ are as follows:

E1. If there is a running thread i in S such that the thread performs an operation *op*, then the successor node is obtained in the following way:

- If op is *not* a primitive function, then the successor node is $(\langle q'_1, s'_1, \varphi'_1\rangle, \ldots, \langle q'_n, s'_n, \varphi'_n\rangle, \varphi', S')$ where $\varphi'_i = SP^\pi(\varphi_i \wedge \varphi, op)$, $\varphi'_j = SP^\pi(\varphi_j \wedge \varphi, \text{HAVOC}(op))$ for $j \neq i$, $\varphi' = SP^\pi(\varphi, op)$, $s'_k = s_k$ for all $k = 1, \ldots, n$, and $S' = S$. $SP^\pi(\varphi, op)$ computes the abstract strongest post condition w.r.t. precision π. In our case of predicate abstraction the precision π can contain (1) a set of predicates that are tracked for the global region φ, and (2) for all i, a set of predicates that are tracked for each thread region φ_i. HAVOC is a function that collects all global variables that are possibly updated by the operation op, and builds a new operation where these variables are assigned with new fresh variables. We do this since we do not want to leak variables local to the running thread in order to update the region of other threads.
- If op is a primitive function, then the new node is $(\langle q_1, s_1, \varphi_1\rangle, \ldots, \langle q_n, s_n, \varphi_n\rangle, \varphi, S')$ where $S' = \text{SEXEC}(S, op)$.

E2. If there is no more running thread in S, then for each scheduler's state $S' \in \text{SCHED}(S)$ we create a node $(\langle q_1, s_1, \varphi_1\rangle, \ldots, \langle q_n, s_n, \varphi_n\rangle, \varphi, S')$ such that the node becomes the root node of a new ART that is then added to the ARF. This represents the context switch that occurs when a thread gives the control back to the scheduler.

In the same way as the classical lazy abstraction, one stops expanding a node if the node is covered by other nodes. In our case we say that a node $(\langle q_1, s_1, \varphi_1\rangle, \ldots, \langle q_n, s_n, \varphi_n\rangle, \varphi, S)$ is *covered* by a node $(\langle q'_1, s'_1, \varphi'_1\rangle, \ldots, \langle q'_n, s'_n, \varphi'_n\rangle, \varphi', S')$ if (1) $q_i = q'_i$ and $s_i = s'_i$ for $i = 1, \ldots, n$, (2) $S = S'$, and (3) $\varphi \rightarrow \varphi'$ and $\bigwedge_{i=1,\ldots,n}(\varphi_i \rightarrow \varphi'_i)$ are valid. We also stop expanding a node if the conjuction of all thread regions and the global region is unsatisfiable.

We say that an ARF is *complete* if it is closed under the expansion rules described above and there is no node that can be expanded. An ARF is *safe* if it is complete and, for every node $(\langle q_1, s_1, \varphi_1\rangle, \ldots, \langle q_n, s_n, \varphi_n\rangle, \varphi, S)$ in the ARF such that $\varphi \wedge \bigwedge_{i=1,\ldots,n} \varphi_i$ is satisfiable, none of the locations $q_1, \ldots, q_n$ are error locations.

B. ARF construction

The construction of an ARF starts with a single ART representing reachable states of the main function. In the root node of that ART all regions are initialized with $True$, all thread locations are set to the entries of the corresponding threads, all call stacks are empty, and the only running thread in the scheduler's state is the main function. The main function then suspends itself by calling a primitive function that starts the simulation.

We expand the ARF using the rules E1 and E2 until either the ARF is complete or we reach a node where one of the thread's location is an error location. In the latter case we build a counterexample consisting of paths in the trees of the ARF and check if the counterexample is feasible. If it is feasible, then we have found a real counterexample witnessing that the program is unsafe. Otherwise, we use the spurious counterexample to discover predicates to refine the ARF.

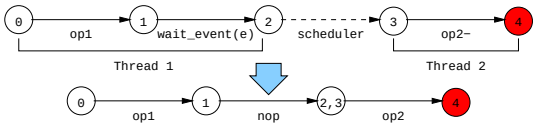


Fig. 2. An example error path.

C. Counterexample analysis and predicate discovery

The counterexample in our proposed technique is built in a similar way to that of in the classical lazy abstraction for sequential programs. In our case each call to a primitive function is replaced with a *nop* (no operation). The connections between trees induced by SCHED is removed and the two connected nodes are collapsed into one.

Let us consider the path represented in Fig. 2. There are two threads in this example. First, thread 1 moves from node 0 to node 1 with operation op_1, and then moves from node 1 to node 2 with `wait_event(e)` that makes thread 1 sleep and wait for the event `e` to be notified. The scheduler SCHED is then executed, and this execution creates a connection from node 2 to node 3, and also makes thread 2 as the running thread. Finally, thread 2 moves from node 3 to error node 4 with operation op_2. The counterexample is built by replacing the call to `wait_event(e)` labeling the transition from node 2 to 3 with *nop* and by collapsing nodes 2 and 3 into a new node 2,3. We thus obtain the path depicted in the lower part of Fig. 2. This final path corresponds to a "standard" path in the pure sequential software model checker, and is the path we consider for the counterexample analysis.

When the formula corresponding to the error path built above is unsatisfiable, then the proof of unsatisfiability is analyzed in order to extract new predicates. These predicates are then used to refine the abstraction in order to rule out this unfeasible error path in the expansion of ARF. For this purpose we re-use the same techniques used in the sequential case, e.g. Craig interpolants and unsatisfiable core. The newly discovered predicates are then used to update the precision. Depending on the nature of the predicates, they can be associated to all threads globally, which is the precision of the global region, or to a specific thread, which is the precision of the thread region. Due to lack of space, we refer the reader to [19] for a more thorough discussion of the refinement process.

D. Parametric Summarization of Control Flow Automata

CFA summarization based on the large block encoding (LBE) has been introduced in [12]. The encoding can also be applied to summarize the CFA representing a thread.

We define a parameterized version of the LBE w.r.t. a set $\Gamma \subseteq Ops$ of operations that is used to prevent the creation of a large block. The algorithm to compute parametric LBE is a variant of the algorithm described in [12]. First, a CFA is a tuple (L, G) where L is a set of control locations and $G \subseteq L \times Ops \times L$ is a set of edges. Without loss of generality, we assume that the CFA has at most one error location denoted by l_E. The LBE of Γ-*CFA Summarization* consists of the application of the rules we

describe below: we first apply the rule R1, and then repeatedly apply the rule R2 and R3 until none of them are applicable.

R1. We remove all edges $(l_E, *, *)$ from G. This rule transforms the error location into a sink location.

R2. If $(l_1, op, l_2) \in G$ such that $l_1 \neq l_2$, $op \notin \Gamma$, l_2 has no other incoming edges, and for all $(l_2, op_i, l_i) \in G$ we have $op_i \notin \Gamma$, then $L = L \setminus \{l_2\}$ and $G = (G \setminus \{(l_1, op, l_2)\}) \cup \{(l_1, op; op_i, l_i) | \text{for all } i\}$. If the current operation op, or one of the outgoing operations is in Γ, then we stop summarizing the current block.

R3. If $(l_1, op_1, l_2) \in G$, $(l_1, op_2, l_2) \in G$, and none of op_1, op_2 are in Γ then $G = (G \setminus \{(l_1, op_1, l_2), (l_1, op_2, l_2)\}) \cup \{(l_1, op_1 \| op_2, l_2)\}$. Intuitively, if there is a choice and none of the two outgoing operations are in Γ, then we join the operations.

Since the parameter of summarization only prevents the creation of large blocks, the correctness of summarization as stated in [12] still holds for the above rules.

V. Related Work

There have been some works on the verification of SystemC designs. Scoot [20] is a tool that extracts from a SystemC design a flat C++ model that can be analyzed by SATABS [17]. The SystemC scheduler itself is included in the flat model. Scoot, to the best of our knowledge, has only been used for race analysis [21], and for synthesizing a static scheduler to speed up simulation [22]. Our work on embedding the scheduler into the model-checking algorithm can benefit from the techniques described in [21] for reducing the number of schedules to explore.

CheckSyC [4] is a tool used for property and equivalence checking, and for simulation of SystemC designs. It relies on SAT based bounded model checking (BMC) and thus does not support unbounded loops. Moreover CheckSyC does not support SystemC constructs that have no correspondence in RTL, like channels.

Lussy [3] is a toolbox for the verification of SystemC designs at TLM. The tool extracts from a SystemC design a set of parallel automata that captures the semantics of the design, including the SystemC scheduler. These automata are then translated into Lustre or SMV model for the verification. The results reported in [23] show that the approach does not scale. An extension for the use of Spin is discussed in [6]. However, this translation is manual. Moreover, it is bound to not scale-up when the SystemC design requires to model nondeterministic signals with a large domain like e.g. an integer. For us, this is not a problem since we model them symbolically.

In [7] the SystemC design is encoded as a network of timed automata where the synchronization mechanism is modeled through channels. The execution semantics is specified through a pre-determined model of the scheduler, and by means of templates for events and threads. The resulting network of automata is verified using the UPPAAL model checker. This approach only supports bounded integer variables.

Formal verification of SystemC designs by abstracting away the scheduler, that is encoded in each of the threads, has been reported in [5]. This work does not handle channel updates and time delays. Our translation from SystemC to C can adopt the technique in the paper to simplify the resulting C program.

Works on the verification of multi-threaded C programs are related to our work. Software model checkers for multi-threaded programs such as Verisoft [24] and Zing [25] explore states and transitions using explicit enumeration. Although several state space reduction techniques (e.g. partial order reduction [26] and transaction based methods [27]) have been proposed, they do not scale well because of the state explosion caused by the thread interleaving. Extensions of the above techniques by using symbolic encodings [28] combined with bounded context switches [29] and abstraction [30] have been proposed. In [31] an asynchronous modeling is used to reduce the size of BMC problems. All of these techniques can be applied to the verification of SystemC designs by properly encoding the semantics of the SystemC scheduler. Our approach can benefit from these optimizations. In particular we expect that partial-order reduction that can reduce the number of schedules to explore will lead to dramatic improvements, but we leave it as future work.

VI. Experimental Evaluation

We have implemented a tool chain that supports the SystemC verification approaches presented in this paper. The front-end for handling SystemC is an extended version of PINAPA [13] modified to generate the flattened pure sequential C program described in Section III, and the output suitable for the new algorithm described in Section IV.

To deal with the sequential C program, we have implemented a new software model checker for C that we call SYCMC. Inspired by BLAST [8], SYCMC implements lazy predicate abstraction. Furthermore, SYCMC also provides LBE and the Γ-CFA summarization described before. SYCMC is built on top of an extended version of NUSMV [14] that integrates the MathSAT [32] SMT solver and provides advanced algorithms for performing predicate abstraction by combining BDDs and SMT formulas [15], [16]. The new ESST model-checking algorithm is implemented within SYCMC. In SYCMC as well as in ESST each time we expand an ART (ARF) node we perform the check to verify whether the newly generated node is covered by another ART (ARF) node. Thus, it is fundamental to perform this check as efficiently as possible. Similarly to CPACHECKER, in SYCMC as well as in ESST we use BDDs to represent the regions, and we exploit them for efficiently checking whether a node is covered by another node.

A. Results

We used benchmarks taken and adapted from the SystemC distribution [1], from [23], and from [33] to experiment with our approaches. To the best of our knowledge, none of the tools used in [3], [4], [7] is available for comparison. We first experimented with the translation of SystemC models to C programs, by running the benchmarks on the following model checkers: SATABS [17], BLAST [8], CPACHECKER presented in [12], and SYCMC. We then experimented the ESST algorithm of

SYCMC on the same set of benchmarks. As the model checkers feature a number of verification options, we only consider what turned out to be the best options for the benchmarks. For BLAST we used the `-foci` option, while for CPACHECKER and for SYCMC we applied LBE, depth first node expansion with global handling of predicates, and restarting ART from scratch when new predicates are discovered. We have experimented the tools on an Intel-Xeon DC 3GHz running Linux, equipped with 4GB of RAM. We fixed the time limit to 1000 seconds, and the memory limit to 2GB.

The results of experiments are shown on Table I. In the second column we report S, U, or - to indicate that the verification status of the benchmark is safe, unsafe, or unavailable respectively. The unavailability of the status is due to time or memory out. In the remaining columns we report the running time in seconds. We use T.O. for time out, M.O. for memory out, and N.A. for not available.

The results show that the translation approach is feasible, but the model checkers often reached timeout. This is because the presence of the scheduler in the C program enlarges the search space that has to be explored by the model checkers. Moreover, we noticed that several iterations of refinement are needed to discover predicates describing the status of the scheduler in order to rule out spurious counterexamples. We notice that, as far as these benchmarks are concerned, CPACHECKER outperforms BLAST, while we have cases where SYCMC performs better than CPACHECKER, and others where it performs worse. This is explained by the fact that the search in the two model checkers, although similar may end-up exploring paths in a different order and thus discovering different sets of predicates.

The table clearly shows that the ESST algorithm outperforms the other four approaches in most cases. In the case of `pipeline` design CPACHECKER and SYCMC outperform the ESST algorithm. It turns out that for the verification of this design precise details of the scheduler are not needed. CPACHECKER and SYCMC are able to exploit this characteristic and thus they end up exploring less abstract states than ESST. Indeed, for this design the ESST algorithm needs to explore many possible schedules that can be reduced by using techniques like partial-order reduction. For the `mem-slave` design SATABS outperforms other model checkers. SYCMC and ESST employ a precise Boolean abstraction in the expansion of the ART. Such an abstraction is expensive when there are a large number of predicates involved. For this design, SYCMC and ESST already discovered about 70 predicates in the early refinement steps. SATABS also discovered a quite large number of predicates (51 predicates). However, it performs a cheap approximated abstraction that turns out to be sufficient for the verification of this design.

All the benchmarks and the executable to reproduce the results reported in this paper are available at http://es.fbk.eu/people/roveri/tests/fmcad2010.

B. Limitations

The approaches presented in this paper assume that the SystemC design does not contain any dynamic creation of threads, channels, and module instances. In particular, in the sequentialization approach the encoding of the scheduler requires those components to be known a priori. For example, to encode the evaluation and the channel update phases (the functions `eval` and `update_channels`, respectively) one needs to know all threads and channels that are involved in the simulation. In the threaded C approach we assume the values of t and e in `wait(`t`)` and `wait_event(`e`)` can be determined statically. Similarly for the translation to threaded C and in the ESST algorithm, at the moment we do not support dynamic creation of threads, channels, and module instances. It turns out that also the SystemC front-end we use for our translator suffers of these limitations. Indeed, PINAPA parses the SystemC design and executes it until the point just before the simulation begins. At that point PINAPA gives access to the abstract syntax tree (AST) of the design and to all the ground SystemC objects (i.e. module instances, channels, and threads) of the design. We remark that, these limitations do not affect the applicability of the proposed techniques since, to the best of our knowledge, most real SystemC design satisfy this assumption.

		Sequentialized				ESST
Name	V	SATABS	BLAST	CPAC.	SYCMC	SYCMC
toy1	S	22.790	T.O.	282.230	57.300	1.990
toy2	U	28.050	T.O.	621.120	35.300	0.690
toy3	U	20.290	T.O.	141.780	22.390	0.190
token-ring1	S	16.520	97.2000	14.590	36.990	0.010
token-ring2	S	62.240	888.2900	30.330	540.160	0.090
token-ring3	S	152.360	T.O.	141.860	T.O	0.190
token-ring4	S	602.300	T.O.	911.300	T.O	0.400
token-ring5	S	T.O.	T.O.	T.O.	T.O	1.000
token-ring6	S	T.O.	T.O.	T.O.	T.O	2.500
token-ring7	S	T.O.	T.O.	T.O.	T.O	6.390
token-ring8	S	T.O.	T.O.	T.O.	T.O	18.400
token-ring9	S	T.O.	T.O.	T.O.	T.O	54.290
token-ring10	S	T.O.	T.O.	T.O.	T.O	201.980
token-ring11	-	T.O.	T.O.	T.O.	T.O	M.O
transmitter1	U	2.230	1.2700	11.850	6.200	0.010
transmitter2	U	26.920	29.4000	18.210	640.750	0.010
transmitter3	U	61.460	501.3500	44.320	176.290	0.010
transmitter4	U	190.620	T.O.	113.490	T.O	0.090
transmitter5	U	472.180	T.O.	296.580	T.O	0.190
transmitter6	U	T.O.	T.O.	969.530	T.O	0.500
transmitter7	U	T.O.	T.O.	T.O.	T.O	1.390
transmitter8	U	T.O.	N.A.	T.O.	T.O	3.690
transmitter9	U	T.O.	N.A.	T.O.	T.O	11.690
transmitter10	U	T.O.	T.O.	T.O.	T.O	40.590
transmitter11	U	T.O.	T.O.	T.O.	T.O	150.480
transmitter12	-	T.O.	T.O.	T.O.	T.O	M.O
pipeline	S	T.O.	T.O.	130.610	178.490	T.O
kundu1	S	139.440	T.O.	232.310	T.O	2.900
kundu2	U	41.500	245.8500	57.160	T.O	0.900
kundu3	U	110.550	T.O.	129.370	T.O	2.900
bistcell	S	36.600	T.O.	10.560	38.000	1.090
pc-sfifo1	S	4.260	46.6500	13.110	7.690	0.300
pc-sfifo2	S	5.210	300.3800	28.490	34.790	0.300
mem-slave	S	77.210	T.O.	T.O.	T.O	677.010

TABLE I: RESULTS FOR EXPERIMENTAL EVALUATION.

The PINAPA SystemC front-end at the current stage of development suffers of many other limitations. For example, as far as we know, it does not recognize all SystemC transaction-level modeling (TLM) constructs and does not fully support function pointers. Because of these limitations, our translator from SystemC to sequential C (and also to threaded C) does not handle such constructs either. For the experiments presented in this paper we extended PINAPA to handle simple TLM

constructs like `sc_export`. Support for additional SystemC constructs can be added to PINAPA with a reasonable effort.

As far as the limitations of our translator are concerned, we do not yet support rich C++ features like standard template library (STL) data structures and respective constructs, and we do not yet support pointers, arrays, and dynamic creation of objects. To this end, we remark that most of the software model checkers currently available are not able to fully support all of them. We remark that, our translator can be extended to support such constructs wit a reasonable effort.

Finally, the new SYCMC and ESST model checkers are not yet able to handle designs that use complex data types (like e.g. records), pointers, arrays, dynamic creation of objects, and recursive function. However, support for all these constructs is currently argument of future extensions of the tools.

VII. CONCLUSIONS AND FUTURE WORK

In this paper we have presented two novel approaches aiming at lifting software model checking techniques to the verification of SystemC designs. We first presented a conversion of a SystemC design into an C program that can be verified by any off the shelf software model checker for C. Second, we presented a novel model checking algorithm that combines an explicit model checking technique to model the states of SystemC scheduler with lazy abstraction. Both approaches have been implemented in a tool set and an experimental evaluation was carried out showing the potential of the approach and the fact that the new algorithm outperforms the first approach.

As future work, we will investigate the applicability of static and dynamic partial order techniques to reduce the number of paths to explore. We will extend the set of primitives to interact with the scheduler to better handle TLM constructs. Moreover, we will investigate the possibility to handle the scheduler semi-symbolically by enumerating possible next states exploiting SMT techniques as to eliminate the current limitations of the ESST approach. Finally, we will also extend our back-end to support richer data like e.g. arrays [34], [35].

ACKNOWLEDGMENTS

We thank A. Griggio for the fruitful discussions on LBE and on the behavior of CPACHECKER. This work is sponsored by the EU grant FP7-2007-IST-1-217069 COCONUT.

REFERENCES

[1] "IEEE 1666: SystemC language Reference Manual," 2005.
[2] D. Tabakov, G. Kamhi, M. Y. Vardi, and E. Singerman, "A Temporal Language for SystemC," in *FMCAD*. IEEE, 2008, pp. 1–9.
[3] M. Moy, F. Maraninchi, and L. Maillet-Contoz, "LusSy: A Toolbox for the Analysis of Systems-on-a-Chip at the Transactional Level," in *ACSD*. IEEE, 2005, pp. 26–35.
[4] D. Große and R. Drechsler, "CheckSyC: An Efficient Property Checker for RTL SystemC Designs," in *ISCAS (4)*. IEEE, 2005, pp. 4167–4170.
[5] D. Kroening and N. Sharygina, "Formal Verification of SystemC by Automatic Hardware/Software Partitioning," in *MEMOCODE*. IEEE, 2005, pp. 101–110.
[6] C. Traulsen, J. Cornet, M. Moy, and F. Maraninchi, "A SystemC/TLM Semantics in Promela and its Possible Applications," in *SPIN*, ser. LNCS, vol. 4595. Springer, 2007, pp. 204–222.
[7] P. Herber, J. Fellmuth, and S. Glesner, "Model Checking SystemC Designs using Timed Automata," in *CODES+ISSS*. ACM, 2008, pp. 131–136.
[8] D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar, "The Software Model Checker Blast," *STTT*, vol. 9, no. 5-6, pp. 505–525, 2007.
[9] K. L. McMillan, "Lazy Abstraction with Interpolants," in *CAV*, ser. LNCS, vol. 4144. Springer, 2006, pp. 123–136.
[10] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre, "Lazy Abstraction," in *POPL*, 2002, pp. 58–70.
[11] T. A. Henzinger, R. Jhala, R. Majumdar, and K. L. McMillan, "Abstractions from Proofs," in *POPL*. ACM, 2004, pp. 232–244.
[12] D. Beyer, A. Cimatti, A. Griggio, M. E. Keremoglu, and R. Sebastiani, "Software Model Checking via Large-Block Encoding," in *FMCAD*. IEEE, 2009, pp. 25–32.
[13] M. Moy, F. Maraninchi, and L. Maillet-Contoz, "Pinapa: An Extraction Tool for SystemC Descriptions of Systems-on-a-Chip," in *EMSOFT*. ACM, 2005, pp. 317–324.
[14] A. Cimatti, E. M. Clarke, F. Giunchiglia, and M. Roveri, "NuSMV: A New Symbolic Model Checker," *STTT*, vol. 2, no. 4, pp. 410–425, 2000.
[15] R. Cavada, A. Cimatti, A. Franzén, K. Kalyanasundaram, M. Roveri, and R. K. Shyamasundar, "Computing Predicate Abstractions by Integrating BDDs and SMT Solvers," in *FMCAD*. IEEE, 2007, pp. 69–76.
[16] A. Cimatti, J. Dubrovin, T. Junttila, and M. Roveri, "Structure-aware Computation of Predicate Abstraction," in *FMCAD*. IEEE, 2009, pp. 9–16.
[17] E. M. Clarke, D. Kroening, N. Sharygina, and K. Yorav, "SATABS: SAT-Based Predicate Abstraction for ANSI-C," in *TACAS*, ser. LNCS, vol. 3440. Springer, 2005, pp. 570–574.
[18] D. Beyer, T. A. Henzinger, and G. Théoduloz, "Configurable Software Verification: Concretizing the Convergence of Model Checking and Program Analysis," in *CAV*, ser. LNCS, vol. 4590. Springer, 2007, pp. 504–518.
[19] A. Cimatti, A. Micheli, I. Narasamdya, and M. Roveri, "Verifying SystemC: a Software Model Checking Approach," FBK-irst, Tech. Rep., 2010, http://es.fbk.eu/people/roveri/tests/fmcad2010.
[20] N. Blanc, D. Kroening, and N. Sharygina, "Scoot: A Tool for the Analysis of SystemC Models," in *TACAS*, ser. LNCS, vol. 4963. Springer, 2008, pp. 467–470.
[21] N. Blanc and D. Kroening, "Race Analysis for SystemC using Model Checking," in *ICCAD*. IEEE, 2008, pp. 356–363.
[22] ——, "Speeding Up Simulation of SystemC using Model Checking," in *SBMF*, ser. LNCS, vol. 5902. Springer, 2009, pp. 1–16.
[23] M. Moy, "Techniques and Tools for the Verification of Systems-on-a-Chip at the Transaction Level," INPG, Grenoble, Fr, Tech. Rep., Dec 2005.
[24] P. Godefroid, "Software Model Checking: The VeriSoft Approach," *F. M. in Sys. Des.*, vol. 26, no. 2, pp. 77–101, 2005.
[25] T. Andrews, S. Qadeer, S. K. Rajamani, J. Rehof, and Y. Xie, "Zing: A Model Checker for Concurrent Software," in *CAV*, ser. LNCS, vol. 3114. Springer, 2004, pp. 484–487.
[26] C. Flanagan and P. Godefroid, "Dynamic Partial-Order Reduction for Model Checking Software," in *POPL*. ACM, 2005, pp. 110–121.
[27] S. D. Stoller and E. Cohen, "Optimistic Synchronization-based State-Space Reduction," *F. M. in Sys. Des.*, vol. 28, no. 3, pp. 263–289, 2006.
[28] V. Kahlon, A. Gupta, and N. Sinha, "Symbolic Model Checking of Concurrent Programs using Partial Orders and On-the-Fly Transactions," in *CAV*, ser. LNCS, vol. 4144. Springer, 2006, pp. 286–299.
[29] S. Qadeer and J. Rehof, "Context-Bounded Model Checking of Concurrent Software," in *TACAS*, ser. LNCS, vol. 3440. Springer, 2005, pp. 93–107.
[30] I. Rabinovitz and O. Grumberg, "Bounded Model Checking of Concurrent Programs," in *CAV*, ser. LNCS, vol. 3576. Springer, 2005, pp. 82–97.
[31] M. K. Ganai and A. Gupta, "Efficient Modeling of Concurrent Systems in BMC," in *SPIN*, ser. LNCS, vol. 5156. Springer, 2008, pp. 114–133.
[32] R. Bruttomesso, A. Cimatti, A. Franzén, A. Griggio, and R. Sebastiani, "The MathSAT 4 SMT Solver," in *CAV*, ser. LNCS, vol. 5123. Springer, 2008, pp. 299–303.
[33] S. Kundu, M. K. Ganai, and R. Gupta, "Partial Order Reduction for Scalable Testing of SystemC TLM Designs," in *DAC*. ACM, 2008, pp. 936–941.
[34] A. Armando, M. Benerecetti, and J. Mantovani, "Abstraction Refinement of Linear Programs with Arrays," in *TACAS*, ser. LNCS, vol. 4424. Springer, 2007, pp. 373–388.
[35] R. Jhala and K. L. McMillan, "Array Abstractions from Proofs," in *CAV*, ser. LNCS, vol. 4590. Springer, 2007, pp. 193–206.

Coping with Moore's Law *(and More)*: Supporting Arrays in State-of-the-Art Model Checkers

Jason Baumgartner Michael Case Hari Mony
IBM Systems & Technology Group

***Abstract*—State-of-the-art hardware model checkers and equivalence checkers rely upon a diversity of synergistic algorithms to achieve adequate scalability and automation. While higher-level decision procedures have enhanced capacity for problems of amenable syntax, little prior work has addressed (1) the generalization of many critical synergistic algorithms beyond bit-blasted representations, nor (2) the issue of bridging higher-level techniques to problems of complex circuit-accurate syntax. In this paper, we extend a variety of bit-level algorithms to designs with memory arrays, and introduce techniques to rewrite arrays from circuit-accurate to verification-amenable behavioral syntax. These extensions have numerous motivations, from scaling formal methods to verify ever-growing design components, to enabling hardware model checkers to reason about software-like systems, to allowing state-of-the-art model checkers to support temporally-consistent function- and predicate-abstraction.**

I. Introduction

Contemporary hardware designs are often of substantial complexity, comprising a diversity of bit-level control logic, datapaths, and performance-related artifacts including pipelining, multi-threading, out-of-order execution, and power-saving techniques. While reference models expressing the correctness of such designs may be specified at a higher abstraction level, it is often necessary to directly reason about the circuit-accurate implementation. For example, equivalence checkers *must* reason about the circuit-accurate implementation. If the designer-specified implementation closely matches the circuit, combinational equivalence checking (CEC) may scalably solve the equivalence-checking problem – leaving a formidable correctness check of the circuit-accurate implementation vs. the reference model. If in contrast the implementation more closely matches the higher-level specification, functional verification becomes simpler, leaving a formidable sequential equivalence check between the implementation vs. the circuit.

Numerous automated transformations have been developed to alleviate the challenges of verifying contemporary hardware designs. For example, *phase abstraction* eliminates verification complexities of designs with intricate clocking and latching schemes [1]. *Retiming* reduces the verification overhead associated with pipelined designs [2]. *Redundancy removal* and *rewriting* eliminate numerous design artifacts which may dramatically hurt verification scalability [3], [4], [5]. Such techniques have become key components of state-of-the-art model checkers and equivalence checkers [6], [1], [7], without which such solvers often fail to yield a conclusive result on industrial designs. However, these techniques have hitherto largely been developed assuming a *bit-blasted* representation.

Substantial recent research has focused upon enhanced reasoning scalability for designs expressed at a higher-level of abstraction. For example, numerous techniques have been established to enhance the verification scalability of designs containing *arrays*: storage devices arranged as a set of addressable rows of a specific width, accessed through atomic *write* and *read* operations. Example techniques include the *efficient memory model* which preserves data consistency within temporally-bounded reasoning using a modeling whose complexity grows sub-linearly with respect to array size [8], [9], and the abstraction-refinement technique of [10] which reduces an array to a small number of consistently-modeled rows. Additionally, a *large number* of dedicated decision procedures have been developed around theories of arrays [11].

While extremely powerful for amenable problems, such techniques have not yet delivered their full impact in industrial hardware verification for several reasons. First, such techniques are often applicable only to designs with behavioral syntax, not to designs of intricate circuit-accurate syntax. Manual creation of behavioral models may alleviate this concern for property checking – though at an often-prohibitive expense to the overall design flow. Furthermore, these behavioral models must be equivalence-checked to the circuit-accurate implementation to ensure the soundness of such an approach; while property checking may become simpler, the equivalence check may be intractable. Second, techniques which are incompatible with bit-level transformations are of limited utility on industrial designs, given capacity limitations in reasoning about the logic *adjacent to* the arrays. In our experience, the logic around the dataflow often contains the most subtle flaws; the dataflow itself poses a bottleneck to verification algorithms which often necessitates manual guidance to expose these flaws and ultimately establish correctness.

In this paper, we address the issue of efficient formal reasoning about industrial hardware designs which include arrays. Our contributions include: **(1)** algorithmic extensions to a variety of traditionally bit-level transformation algorithms to support designs with arrays, including redundancy removal (Section III), phase abstraction (Section IV), temporal decomposition and retiming (Section V); **(2)** techniques to simplify array syntax, enabling efficient array reasoning upon designs which may otherwise lack a suitable behavioral representation (Section III-C); **(3)** enhancements to the robustness and scalability of known array abstraction techniques (Section VI). Experiments are provided in Section VII to confirm the profound verification benefits enabled by these techniques.

There are numerous motivations for this work.

- As per Moore's Law, increasing array size (caches, main memory, lookup tables, ...) is one prevalent way in which growing transistor capacity is used to increase design performance [12]. While bit-blasted analysis suffers substantial overhead with doubled array size, native reasoning techniques often entail sub-linear complexity growth – e.g., merely requiring an additional address-comparison bit.
- There are numerous problem domains which are practically infeasible for bit-blasted techniques without manual abstraction, such as formally verifying logic that interacts with main memory or large caches. Large arrays already constitute a substantial scalability differential between formal and informal industrial verification efforts, as most hardware simulators and accelerators represent arrays without bit-blasting.
- Increasing the scalability of automated solutions *mandates* enabling the applicability of as large a set of algorithms as possible, to leverage algorithmic synergies to eliminate implementation characteristics which otherwise may pose a bottleneck to, if not outright inapplicability of, otherwise well-suited algorithms. This is particularly true for *satisfiability modulo theories* solvers, which tend to be highly sensitive to the type of logic which may be efficiently handled by a given combination of theories (e.g., [13]).
- Randomly-initialized read-only arrays may be used to abstract complex combinational functions in a temporally-consistent manner. In particular, the data output of such an array, addressed by the arguments to the function being abstracted, may be used to replace the logic associated with that function. This uninterpreted modeling may simulate the original function, hence is sound for verification – and maintains the necessary invariant for arbitrary model checking algorithms that applying identical arguments to the abstracted function at different points in time yields identical results [14]. Our techniques thus constitute a method to utilize uninterpreted functions in a state-of-the-art model checker.

II. Preliminaries

We represent the design under verification as a *netlist*.

Definition 1: A *netlist* comprises a directed graph with vertices representing gates, and edges representing interconnections between gates. Gates have associated functions, such as constants, primary inputs (termed *RANDOM* gates), combinational logic of various functionality, and single-bit sequential elements termed *registers*. Registers have associated *initial values* defining their time-0 or *reset* behavior, and next-state functions defining their time-$(i+1)$ behavior.

The *And / Inverter Graph (AIG)* is a commonly-used netlist representation where the only combinational primitives are single-bit inverters and two-input AND gates [3], [4]. This implies a bit-blasting of all higher-level constructs. Our netlist format is an AIG which also includes *array* primitives.

Definition 2: An *array* is a gate representing a two-dimensional grid of registers (referred to as *cells*), arranged as rows vs. columns. Cells are accessed via *read* and *write ports*.

```
reg [COLS-1 : 0] ram[ROWS - 1 : 0]; // array declaration
always @(posedge clk) begin
                    // write port:
  if (wr_en)        // enable is  "(posedge clk AND wr_en)"
    ram[wr_addr] <= // address is "wr_addr"
      wr_data;      // data is    "wr_data"
end
                    // read port:
assign rd_data =    // data is    "rd_data"
     rd_en ?        // enable is  "rd_en"
     ram[rd_addr] : // address is "rd_addr"
     {(COLS){1'bX}};
```

Fig. 1: Verilog array example

Ports have three types of *pins*: an enable, an address vector, and a data vector: refer to Figure 1. The *enable* indicates whether the given port is actively accessing the array cells. The *address* indicates which row is being accessed. The *data* represents the values to be stored to (read from) the given row for a write (read) port. A *column* refers to a one-dimensional vector: the ith cell of each row. All pins are inputs of the array gate, aside from read data pins which are outputs.

Arrays have a defined number of r rows, q columns, and p address pins per port; a default *initial value* (in case an unwritten row is read); and an indication of *read-before-write* vs *write-before-read* behavior. The latter is relevant in case of a concurrent read and write to the same address: write-before-read will return the concurrent write data, whereas read-before-write will not. Read data is conservatively randomized when the read enable is de-asserted, or when the read is "out-of-bounds" – i.e., its address exceeds the number of array rows. Write ports have a specified precedence (e.g., reflecting the order of *if, else if* statements in Verilog), defining which will persist in case of concurrent stores to the same address.

We refer to the read ports as $R_1, \ldots, R_m$, and the write ports in order of increasing precedence as $W_1, \ldots, W_n$. For a given port P_i, let $P_i.e$ represent its enable pin, $P_i.a(0 \ldots, p-1)$ its address pins, and $P_i.d(0, \ldots, q-1)$ its data pins.

Definition 3: A *merge* is a reduction technique which effectively eliminates a gate from a netlist by replacing its fanout references with references to a semantically-equivalent gate.

It is highly desirable to be able to merge array outputs if it can be determined that the referenced array cells exhibit redundancy. However, the nondeterminism exhibited at an array output when its read port is disabled or out-of-bounds often precludes a direct merge from being a semantically-consistent transformation.

Definition 4: An *array output merge* is a specialized merge to achieve the desired netlist reduction while preserving necessary nondeterminism. This operation consists of replacing the array output to be merged by a multiplexor which selects the merged-onto gate when the corresponding read port is enabled and in-bounds, else selects a unique RANDOM gate.

A. Temporal Unfolding and the Efficient Memory Model

Many algorithms reason about netlist behavior over a specific number of timesteps. *Unfolding* is commonly used for this purpose, replicating the netlist for the desired number of timesteps to allow valuations to propagate through next-state functions. Depending upon the application for which

unfolding is performed, the time-0 unfolding of the sequential elements will differ. For Bounded Model Checking (denoted as unfold$_b$), the time-0 value will be the initial value of the array or register [15]. For induction, the time-0 value will be a RANDOM gate [16]. For a sequential transformation such as phase abstraction (denoted as unfold$_p$), the time-0 value will be a reference to an existing array or register in the netlist [1].

The efficient memory model (EMM) represents data consistency for arrays within unfoldings using sub-linear modeling size vs. the number of array cells [9]: the data at an array output at time i for an enabled, in-bound read must be the highest-priority, most-recently-written data for the corresponding address. This may be modeled in unfolding using a sequence of *if-then-else* constructs, one per write port and timestep, selected by the corresponding write being enabled and address-matching the read being synthesized [9]. Because each read must be compared to each write, the size of *each* synthesized read for time t is $O(t \cdot |W|)$, resulting in overall quadratic unfolding size with respect to depth as a multiple of the number of write ports $|W|$ and read ports $|R|$.

A technique to further reduce array unfolding size is proposed in [17], re-encoding array references given upper-bounds on the number of distinct referenced addresses. Rewriting rules are used to minimize the number of memory references, e.g., synthesizing *if-then-else* constructs for reads as with EMM. While highly effective for suitable problems, we have not yet found a method to advantageously leverage this technique in a model checking framework: these rewriting rules shadow the complexity of an EMM unfolding, and since arrays are generally interconnected by arbitrary bit-level logic it is challenging to improve upon the effectiveness of standard logic optimization techniques upon such unfoldings, or to *a priori* meaningfully upper-bound a desired unfolding depth.

B. Symbolic Row Abstraction

While EMM is highly effective to boost the efficiency of temporally-bounded reasoning, many alternate algorithms are critical to a robust model checker. For example, BDD-based reachability analysis is often necessary to prove properties of extremely temporally deep netlists. For such temporally-unbounded algorithms, EMM is not directly applicable.

A related technique has been proposed in [10] as a *sequential netlist* abstraction that is applicable for arbitrary model checking algorithms. This abstraction bit-blasts an array into a small set of symbolic rows. This set begins empty and rows are added during refinement in response to spurious counterexamples. In addition to modeling *data* contents for represented rows, the *address* correlating to each modeled row is represented using nondeterministically-initialized registers; reads and writes to modeled rows are performed precisely, whereas writes to unmodeled rows are ignored and reads of unmodeled rows are randomized. To prevent trivial failures merely due to reading unmodeled rows, *antecedent conditioning* of properties is performed: given a spurious counterexample caused by a read from port R_i which occurred k timesteps prior to the property failure, resulting in a new row being modeled with symbolic address r_i^k, property $always(p)$ is replaced by $always\big(prev^k(R_i.a \equiv r_i^k) \rightarrow p\big)$. This abstraction is sound because the abstract netlist may simulate the original, and the antecedent conditioning forms a complete temporal case-split. While very effective for certain types of problems, the abstraction risks exceeding the size of a bit-blasted netlist due to the need to represent modeled addresses, and due to a potentially large number of temporal read dependencies.

III. Logic Optimization Techniques

A vast collection of logic optimization techniques have been developed over the past decades, which reduce netlist size while preserving the behavior of sequential elements. Examples include redundancy removal [3], [5] as well as extensions under observability don't cares [18], and syntactic combinational rewriting [4]. Many of these techniques operate on local logic windows treating sequential elements as unconstrained *cutpoints*, hence are directly applicable to netlists with arrays. Some require bounded / inductive reasoning, possibly to derive *invariants* with which to constrain local analysis, for which the *efficient memory model* provides a suitable extension to netlists with arrays. There are however several optimization techniques which have required substantial customization to achieve an adequate level of scalability and optimality, which we detail in this section.

A. Ternary-Simulation Based Analysis and Reduction

Ternary simulation-based reduction is a method to identify and eliminate a subset of semantically-equivalent gates. Initially, the registers are assigned their initial values and the inputs are assigned an *unknown* ternary *X* value. Next-state functions are then simulated, overapproximating an image computation. These next-states values are propagated through the registers, and another overapproximate image is computed. This iteration continues until a repeated ternary state is detected, indicating that an overapproximation of the reachable states have been explored. Pairs of gates which always evaluate to the same deterministic values in these states may be *merged* to reduce netlist size [1]. This technique is highly overapproximate and able to identify a relatively small subset of truly redundant gates, though is remarkably scalable and often able to yield substantial reductions on industrial netlists [1]. This analysis may also be used to detect oscillating clocks for phase abstraction (Section IV), and transient behavior for temporal decomposition (Section V). Ternary simulation has thus found a role in many state-of-the-art model checkers.

Unlike with Boolean simulation, each three-valued address may resolve to *multiple* existing simulated array value entries. Numerous commercial simulators support multi-valued reasoning, though to avoid the computational overhead entailed by multiple-entry address resolution they take shortcuts such as mapping *X* values on enables or addresses to Boolean constants, or *X*'ing array contents in such cases, as also was noted in [8]. Such shortcuts render unacceptable suboptimalities and even unsoundness in model checking applications.

Building upon the work of [8], which uses three-valued write lists for precise read resolution in *symbolic trajectory*

Algorithm 1 Ternary simulation *write* function

```
1: function write(enable, addr, data)
2:   if (enable ≡ 0) then return
3:   nodesToWrite = deepest nodes whose address intersects addr
4:   for all node in nodesToWrite do
5:     subAddr = intersection of addr and node.address
6:     if (subAddr ≡ node.address) then
7:       node.data = (enable ≡ 1) ? data : resolve(node.data, data)
8:     else add child to node with address subAddr and data data
9:   end for
10:  for all ( address cube subAddr in addr not written above ) do
11:    add new child to tree root with address subAddr and data data
12:  end for
13:  subsume children with data equal to parent
14: end function
```

evaluation, we have developed the following algorithm for precise and efficient three-valued array simulation. Our framework uses a tree structure, where each node represents an ⟨address, data⟩ tuple and edges satisfy the following relationships, maintained during *writes* to enable efficient *reads*:

- A child's address cube is *contained* in its parent's address.
- Child addresses are *exceptions* to parent addresses. E.g., given parent $\langle XX1, D_0\rangle$ with child $\langle X11, D_1\rangle$, addresses $\{XX1 \setminus X11\}$ have data D_0 and $\{X11\}$ has data D_1.
- For any parent, the addresses of all children are disjoint.

Read operations traverse the tree to identify nodes with addresses intersecting the referenced address. A *resolve* function is used to compute the tightest cube that contains all associated three-valued data, similar to resolution across list entries in [8]. Accordingly, *X*-saturated data may be returned without traversing all relevant nodes. Write operations, detailed in Algorithm 1, similarly traverse nodes with intersecting addresses. These nodes are updated if the write address covers their address, else a new child node is created. We employ a more efficient data structure with more aggressive subsumption rules than used in [8], since simulation applications may entail 1000s of timesteps of analysis. The need to continually retraverse lists often degrades to quadratic runtime over simulation depth, whereas the use of a tree enables analysis to be limited to the subset of nodes relevant to a given operation.

Figure 1 illustrates the tree resulting from an array initialized to 000, after a write of ⟨*1*XX, 1XX⟩, then ⟨*X*X0, XX0⟩, then ⟨*X*1X, 01X⟩.

B. Sequential Redundancy Identification and Removal

Arrays are composed of columns comprising one cell per row. It is possible for two array columns (within the same or across different arrays) to evaluate identically in all reachable states. This is particularly common when equivalence checking netlists with arrays; the arrays themselves may be unaltered (merely the logic *adjacent to* the arrays may be altered), or they may reflect a column-equivalence-preserving transformation such as partitioning. The overall equivalence check nonetheless often requires reasoning about array contents, if e.g. the logic adjacent to the arrays was optimized using *don't care* conditions inherent in the array data, precluding their elimination via *black-boxing* [19]. Solving the equivalence checking problem requires efficient methods to identify and

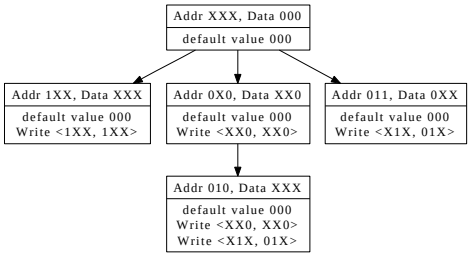


Fig. 1: Three-valued array simulation example

eliminate such column redundancy. More generally, column equivalence is a form of netlist redundancy whose removal significantly benefits the scalability of all types of verification.

Induction is a scalable technique which may be used to identify sequential redundancy [5]. An inductive unfolding instantiates a distinct RANDOM gate for each sequential element to represent an arbitrary state. If it is desired to prove equivalences among sequential elements, the corresponding induction hypotheses constrain the values of these RANDOM gates and thereby often enable inductive redundancy identification. With arrays, however, it is desirable to *not* require explicit correlation of individual cells or even rows, as their cardinality may render such reasoning intractable – basically degrading to the overhead of redundancy identification on a bit-blasted netlist. Directly attempting to establish array output or column equivalence without cell correlation is a highly-noninductive problem, since each unfolding timestep may reference a distinct row, hence induction hypotheses over earlier timesteps do not meaningfully constrain later timesteps.

One approach that we have found useful to enable inductive redundant column identification is to move the proof obligation from array outputs to inputs: two columns are equivalent if they have the same number of rows, they initialize equivalently and any value written to one column is concurrently written to the other column. This proof obligation may be decomposed into a bidirectional check that each enabled, inbound irredundant write to one column has an equivalent write to the other column. This check may be formalized as follows, where it is suspected that columns i and j of arrays A_i and A_j, respectively, are equivalent. Predicate $\mathtt{oob}(W_i.a)$ indicates that W_i has an out-of-bounds address. Predicate $\mathtt{rdt}(W_i)$ indicates that W_i is superseded by a higher-precedence write to the same address, and may be strengthened to check that $W_i.d(i)$ differs from the current value of the addressed cell.[1]

$$\begin{aligned}\forall\,\text{ports}\, W_i \,\text{of}\, \mathrm{A_i} : W_i.e \wedge \neg\mathtt{oob}(W_i.a) \wedge \neg\mathtt{rdt}(W_i).\\ \exists\,\text{port}\, W_j \,\text{of}\, \mathrm{A_j} : W_j.e \wedge (W_i.a \equiv W_j.a) \wedge \neg\mathtt{rdt}(W_j)\\ \wedge \big(W_i.d(i) \equiv W_j.d(j)\big)\end{aligned}$$

Speculative reduction is a technique to enable the benefit of a merge even before the corresponding suspected redundancy has been proven, yielding orders of magnitude speedup to redundancy identification [5]. This technique simplifies the netlist while retaining a proof obligation to identify whether the postulated redundancy is accurate. Speculative reduction

[1]This *irredundant* write-data condition is often necessary in practice, to enable column equivalence detection despite *don't care* optimizations used to minimize the redundant writing of values already present in the array.

may be extended for column equivalences by modifying each read port that references a potentially-redundant column. If it is suspected that columns i and j of arrays A_i and A_j, respectively, are equivalent, each read port R_i referencing column i may be modified to derive values from column j. This is accomplished by synthesizing a new read port R^*_{ij} of array A_j with $R^*_{ij}.e = R_i.e$ and $R^*_{ij}.a = R_i.a$, and replacing references to the redundant column of R_i by references to the representative column of R^*_{ij}. Note that speculative reduction of array outputs reduces the number of RANDOM gates in the inductive unfolding, which is *essential* to overall inductivity.

Column equivalence conditions may be verified directly on the speculatively-reduced netlist. Any identified column equivalences may be eliminated from the netlist, replacing R_i by the corresponding R^*_{ij} as in the speculatively-reduced netlist. The array representation may then be simplified using the techniques introduced in the following section.

C. Array Simplification Techniques

In addition to simplifying logic *around* the arrays, it is advantageous to simplify the arrays themselves: the number of columns, rows, ports, and even the number of distinct arrays. All simplifications tend to enhance algorithmic scalability, and these particular simplifications are often practically *necessary* to enable the efficient use of array reasoning techniques. For example, "content-addressable memories" often have one read port per row, using downstream logic to select which reads are actually relevant. Additionally, industrial arrays often entail circuit-oriented characteristics which may entail fragmenting wide arrays into numerous narrow arrays, implementing one write port per row with orthogonal address-related enables, or intertwining test- or initialization-logic with the array.

Such circuit-accurate arrays pose numerous challenges to verification, which often render them substantially *less* efficient to verify in their native vs. bit-blasted form.

- The efficient memory model entails large unfoldings for netlists containing many arrays with many read and write port (refer to Section II-A).
- As will be discussed in Section VI, the abstraction approach of [10] may run into suboptimalities or even inapplicability given such circuit-accurate syntax.
- Logic simulators are significantly burdened by such representations, and accelerators may be *unable* to model such arrays without bit-blasting – motivating manual creation of behavioral representations for enhanced validation, and using equivalence checking to establish their correctness.

We have found the following transformations essential to *automatically* convert circuit-accurate array representations to behavioral representations for enhanced property checking and equivalence checking. These techniques also are useful to simplify ports created through other transformations such as phase abstraction, and generally to simplify arrays to as efficient of representations as possible.

1. If a given data pin is disconnected from every read port, the corresponding column may be eliminated from the array.
2. Read ports with no connected data pins may be eliminated.
3. Arrays with no read ports may be eliminated.
4. If the enable pin of a given port is semantically equivalent to 0, that port may be eliminated. If that port is a *read*, its outputs may be replaced by RANDOM gates.
5. If a given address pin is an identical constant across every read port, some rows are un-readable hence the array's address space may be reduced. Each write port may conjunct its enable with the condition that its corresponding address pin evaluates to this constant value, then the number of rows and address pins may be reduced accordingly.
6. If a pair of ports P_i, P_j for $i < j$ have identical addresses, and these ports are *compatible*,[2] then these ports may be coalesced to eliminate P_i. Coalescing of write ports consists of multiplexing data: *if* $P_j.e$ *then* $P_j.d$ *else* $P_i.d$. Read data may be directly merged as per Definition 4. The enable pin of P_j is finally replaced by $(P_i.e \vee P_j.e)$.
7. Similar to item 6, if compatible ports P_i, P_j for $i < j$ have orthogonal enables, then these ports may be coalesced to eliminate P_i. Data and address pins on P_j are multiplexed by enables, then P_j's enable is disjuncted with that of P_i.
8. If every data pin of read port R_i has the same *observability don't care* condition O_i, then $R_i.e$ may be optimized using O_i as a don't care – e.g. conjuncting $R_i.e$ with O_i. This often enables the orthogonal-enable port coalescing of item 7.
9. For a write-before-read array, if read port R_i and write port W_j have semantically-equivalent addresses, $W_j.e$ implies $R_i.e$, and no higher-precedence write port may address-match R_i, then $R_i.d$ may be merged onto $W_j.d$ as per Definition 4.
10. If each write port has a semantically-equivalent data pin for two different columns m and n, array outputs for columns m and n may be merged.
11. If arrays A_i and A_j have an identical number of rows and read-before-write vs. write-before-read type, and they have an identical number of ports of each type with semantically-equivalent enable and address pins, the columns of A_j may be concatenated onto A_i, eliminating A_j.
12. If arrays A_i and A_j have identical size and type, identical deterministic initial values, and an identical number of write ports with semantically-equivalent enable, address, and data pins, the read ports of A_i may be migrated to A_j.
13. A write-before-read array may be converted to a read-before-write array, by creating a multiplexor for each read port which selects the highest-precedence concurrent write port data, else the array output itself if no such write exists. This may enable array elimination as per item 11 or 12.

These simplifications are synergistic in that one reduction may enable the applicability of another, and we have found it useful to iterate the above transformations until no further reduction is achieved. It is also useful to iterate these reductions with other logic optimization and abstraction techniques because simplifying the logic *around* the arrays may greatly enhance the applicability of these reductions and vice-versa.

[2] All read ports are compatible. Write ports are compatible if no port P_k for $i < k < j$ may concurrently write to the same address.

IV. Phase Abstraction

Phase abstraction is a temporal abstraction which unfolds next-state functions for a specific number of timesteps c. The resulting netlist represents a c-accelerated variant of the original netlist, such that each state transition of the abstracted netlist correlates to c consecutive transitions of the original netlist. This unfolding results in c copies of every combinational gate in the original netlist, correlating to different modulo-c timesteps. Safety property checking is preserved by disjuncting over each copy of the property gate [1].

Phase abstraction has been demonstrated to yield dramatic speedups to the verification of *clocked* netlists where most registers toggle at most once every c consecutive timesteps. This transformation eliminates the need to model an oscillating clock in the netlist, and often eliminates many registers from the cone of influence as their values become irrelevant to the unfolded next-state functions. Additionally, phase abstraction greatly enhances the reduction capability of techniques such as retiming and redundancy removal [1] and enhances a variety of verification algorithms such as reachability analysis, interpolation [20], and induction [16]. This technique has thus become an essential component of many industrial-strength hardware model checkers [6], [1], [7]. In this section, we extend phase abstraction to netlists with arrays.

Phase abstracted arrays intuitively must have the following characteristics: **(1)** Abstracted write ports must be replicated to reflect all updates that may occur during the c consecutive unfolded timesteps. **(2)** Abstracted read ports must be replicated to support all data fetches which may occur during the c consecutive unfolded timesteps. It is nonetheless essential to ensure that data consistency is maintained during this transformation: read ports for "older" unfolded timesteps cannot be allowed to return write data from "newer" unfolded timesteps. Algorithm 2 yields the necessary semantics-preserving transformation through creation of new array ports.

To ensure data consistency, function unfoldReadPort$_p$ synthesizes data-forwarding paths for read ports unfolded within unfold$_p$, to capture the most-recent applicable unfolded write data. This data may be concurrent for a write-before-read array, else must be strictly earlier. If no such write occurs (the *if-the-else* returns line 24), or if the read enable is deasserted or its address is out-of-bounds (line 31), the read is satisfied by a reference to the newly-created read port from line 8. Note also that the type of the array is converted to read-before-write to ensure that unfoldings for "newer" write ports will not satisfy "older" reads.

V. Temporal Decomposition and Retiming

Transient simplification is a technique to reduce a netlist with respect to *transient signals* which behave arbitrarily for a fixed number of timesteps after reset, and thereafter settle to a reducible behavior. The prefix timesteps, before the transient signals settle to their reducible behavior, may be verified with Bounded Model Checking. The netlist may then be *time-shifted* to represent its post-prefix behavior, decomposing the verification task such that unbounded analysis may focus only upon timesteps after which the transient signals have settled and hence may be eliminated [21]. Such decomposition may reduce the overhead associated with *initialization logic* in a verification testbench. A subset of transients may be efficiently detected using ternary simulation. Given efficient techniques for ternary simulation and Bounded Model Checking, the extension necessary to support temporal decomposition for netlists with arrays is that of time-shifting the arrays.

Time shifting replaces initial values by the set of states reachable in a specific number of timesteps. For registers, a temporal unfolding of their values may be used as their new initial values [21]. Like registers, arrays have initial values that must be modified to reflect writes that occur within the time-shifted prefix. Algorithm 3 illustrates the overall time-shifting transformation. To ensure data consistency, this algorithm places the unfolded prefix write ports lower in precedence than the existing ports, which are used to reflect post-transient writes. These prefix ports are prioritized in order of increasing unfolding time, following the precedence order of the original write ports within each timestep.

Retiming is a technique which moves registers across other types of gates in a netlist, reducing their cardinality while preserving overall netlist behavior. Each retiming step moves one register from each input of a gate to each of its outputs, or vice-versa. The number of registers moved fanin-wise across a gate is referred to as its *lag*, representing the number

Algorithm 2 Array-compatible phase abstraction algorithm

```
 1: function phaseAbstract(netlist, unfoldDegree)
 2:   for all array in netlist do
 3:     writePorts = set of write ports in original array
 4:     readPorts = set of read ports in original array
 5:     for all time in 0 to unfoldDegree-1 do
 6:       for all R in readPorts do
 7:         // port syntax: ⟨enable, address, data⟩
 8:         create shell read port ⟨∅, ∅, R_time⟩
 9:       end for
10:     end for
11:     for all time in 0 to unfoldDegree-1 do
12:       for all R in readPorts do
13:         fill in ⟨unfold_p(R.e, time), unfold_p(R.a, time), R_time⟩ for R
14:       end for
15:       for all W in writePorts via increasing precedence do
16:         append ⟨unfold_p(W.e, time), unfold_p(W.a, time), unfold_p(W.d,
            time)⟩ as highest-precedence write port
17:       end for
18:     end for
19:   end for
20:   perform traditional phase abstraction over non-array gates [1]
21:   convert all arrays to type read-before-write
22: end function

23: function unfoldReadPort_p(port, time)
24:   readData = R_time
25:   time′ = (port's array is write-before-read) ? time : time-1
26:   for all time″ in 0 to time′ do
27:     for all W in writePorts via increasing precedence do
28:       readData = if (unfold_p(W.e, time″) ∧ (unfold_p(W.a, time″) ≡
          unfold_p(R.a, time))) then unfold_p(W.d, time″) else readData
29:     end for
30:   end for
31:   readData = if (¬unfold_p(R.e, time) ∨ (unfold_p(R.a, time) is out-of-
      bounds)) then R_time else readData
32:   return readData
33: end function
```

Algorithm 3 Array-compatible time-shifting algorithm

```
 1: function timeShift(netlist, timeSteps)
 2:   for all register in netlist do
 3:     initialValue[register] = unfold_b(register, timeSteps)
 4:   end for
 5:   init = new register with initial value 1, next-state function 0
 6:   for all time in 0 to timeSteps-1 do
 7:     for all array in netlist do
 8:       newPorts = ∅
 9:       for all writePort of array via increasing precedence do
10:         append newPorts with ⟨(init ∧ unfold_b(writePort.e, time)),
            unfold_b(writePort.a, time), unfold_b(writePort.d, time)⟩
11:       end for
12:       inject newPorts in appended precedence order as lowest-priority
          write ports for array
13:     end for
14:   end for
15: end function
```

of timesteps its behavior has been delayed. Coupled with *peripheral retiming*, in which registers may be borrowed or discarded across RANDOM gates or properties, retiming has been demonstrated to enable orders of magnitude speedup to numerous verification algorithms [2], [6]. *Normalized retiming*, in which all lags are negative, is often used in verification to ensure that retimed initial values may be consistently computed through unfolding. Computing of retimed initial values is analogous to that for time-shifting, aside from the distinction that the lag of each gate may differ hence unfolding is performed at a finer level of granularity.

The following customizations enable the retiming of arrays.

1. All pins associated with a given port must have an identical lag to ensure that each port may be evaluated atomically.

2. No write port may be lagged to a more-negative degree than any read port for a given array. This is to ensure that a read cannot return data from a *later* write, similar in justification to the need to convert write-before-read to read-before-write arrays for phase abstraction in Algorithm 2.

3. For every array with a lagged write port, we use a mechanism similar to Algorithm 3 to reflect its prefix writes. For each array, we iterate from 0 to the maximum negative lag of any write port. For each write port, if its lag is more-negative than the current time iteration, we enqueue a port reflecting the time-iteration unfolding of that port, conjuncting the corresponding enable with an *init* register. We finally inject this queue as the lowest priority write ports.

4. For every read port R_i, a bypass path is constructed to capture data consistency constraints, similar to lines 24-31 of Algorithm 2. Specifically, for any write port lagged to a less-negative degree than a given read port, we build a multiplexor chain that selects the appropriate unfolded write which is more-recent than what is reflected by the array representation, fetching the array contents only if there is no such more-recent write or if the read was not enabled or was out-of-bounds.

VI. Symbolic Row Abstraction

The array abstraction technique described in Section II-B is capable of substantially reducing verification complexity for certain classes of properties [10], though faces several limitations which we have found extensions to ameliorate.

First, in *content-addressable memory* style arrays, all rows are read every timestep, using logic downstream of the array to select which reads are actually relevant. Antecedent-conditioning properties with respect to a particular read port address-matching a modeled address is thus basically meaningless. In [10] it is instead proposed to search for a vector of registers of the width of the address, which evaluates to the address appearing at the read port referenced in the counterexample trace being refined. If found, the equality of that vector of registers (vs. the address of the read port) to the modeled address is used to antecedent-condition the properties.

This approach tends to be fragile in practice. For example, some arrays use arbitrary signals, not only registers, in their read-selection logic. Additionally, given arbitrary design styles, it may not be the case that a dedicated vector exists representing the address of relevance. We have found our array simplification techniques from Section III-C able of eliminate this concern, in reducing the number of read ports in content-addressable memory arrays and thus obviating the need for heuristics to identify useful antecedent addresses.

Second, it is often suboptimal to model a distinct address per refinement step, as doing so fails to explicitly reflect address correlation in the abstract netlist. Consider the equivalence checking of two netlists, each containing an array to abstract. The testbench itself may ensure that equivalent addresses are presented to these arrays, even if design optimizations such as retiming are used to change the timing with which relevant reads occur across these arrays. Additionally, for arrays which are fragmented to reflect circuit characteristics, *many* arrays may have correlated addresses.

A correlated-address optimization may be implemented as follows. Instead of immediately modeling a fresh address upon refinement, we first attempt to assess a relationship between the address to be refined and a previously-refined address. If a correlation is found, the newly modeled row will have its address defined as the postulated correspondence with respect to the previously-modeled address, and no antecedent-conditioning is performed for this refinement step – else this optimization would not be sound. Only if this modeling fails to block the spurious counterexample is a fresh address modeled.

Regarding postulated equivalences: often *identity* between the address of a current refinement and that of a previously-modeled row is an adequate relation. Alternatively, we have encountered equivalence checking problems where an array with a large number of rows in one netlist is replaced with multiple arrays of a smaller number of rows in another. In such cases, postulating a correspondence between an address of the larger array to an address *identical modulo the number of rows in the smaller array* is often effective.

Failure to directly model address correlation in the abstract netlist poses several verification suboptimalities. First, the abstract netlist is larger, requiring more logic to represent more modeled addresses. Second, because distinct addresses are being modeled, this lack of *address* correlation entails a loss of any *data* correlation which holds in the original netlist. For example, in equivalence checking, array data may be identical

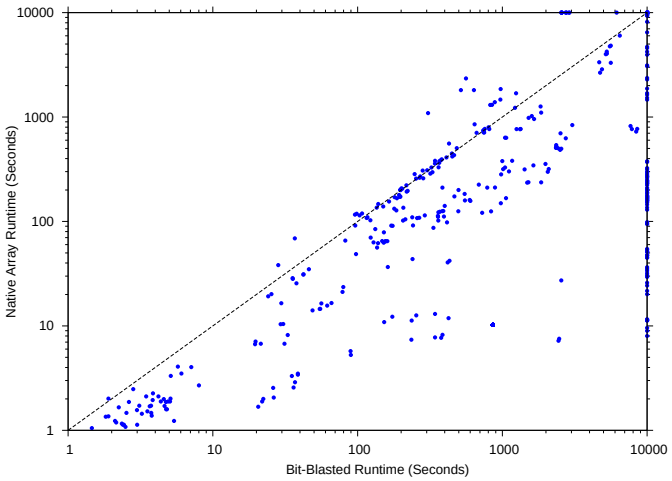


Fig. 2: Cumulative verification runtime experiments

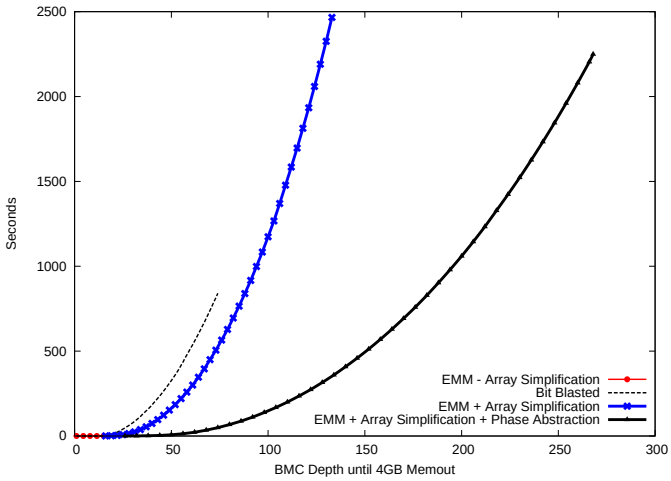


Fig. 3: Bounded Model Checking experiments

on a per-address basis across two arrays, and hence logic optimization techniques may be able to merge those arrays as being redundant. However, if a distinct address is used to abstract each array, the modeled data will differ in states for which modeled addresses differ, precluding such reductions. While such states may be irrelevant due to antecedent conditioning, it is computationally expensive to need to identify such irrelevance through the sequential observability don't care condition of the antecedent vs. being able to identify such redundancy natively using arbitrary logic optimizations.

Furthermore, this lack of modeled address correlation tends to inherently limit the subsequent effectiveness of localization abstraction [22], which eliminates irrelevant gates through replacing them with RANDOMs. E.g., the localized netlist must include enough logic in the fanin of the array addresses to establish the correlation conditions that otherwise would be natively reflected in the correlated-address abstraction.

VII. Experimental Results

In this section we experimentally demonstrate the utility of our techniques to reduce verification resources. All experiments were run on a 1.9 GHz POWER5 Processor, using the IBM internal verification toolset *SixthSense* [6].

Cumulative Impact: Given the numerous techniques presented in this paper, and their ability to synergistically enable solutions to complex problems for which standalone or bit-blasted techniques would fail, our first set of experiments in Figure 2 demonstrates their *cumulative* impact across a large set of complex non-falsifiable industrial property checking and sequential equivalence checking problems. We used a set of often-effective algorithm sequences including the simplification and abstraction techniques presented in this paper, followed by either interpolation or inductive redundancy removal, assessing their effectiveness on bit-blasted netlists vs. ones with arrays within a 10000 second timeout.

Most runtimes become significantly faster without bit-blasting, and many (108 of 810) complete that otherwise timeout. Only a small percentage witness significant slowdown with arrays; almost all of these may be turned to an advantage by fine-tuning algorithm parameters. While these experiments illustrate the profound cumulative benefit of our techniques in enhancing the capacity of state-of-the-art verification solutions, this high-level overview offers little insight into the merit of particular techniques, which we focus on below.

Array Simplification: Figure 3 shows Bounded Model Checking (BMC) performance for several runs: efficient memory model (EMM) with and without our array simplification techniques, vs. a bit-blasted representation, each run until memout. This netlist has 430373 AND gates and 21429 registers, in addition to 444 1-column, 128-row arrays, each with 128 read and write ports: a *content-addressable memory*. Our array simplification techniques from Section III-C reduce these to three 148-column 128-row arrays with one read and one write port each, using 1.3 seconds of runtime. The bit-blasted netlist has 599381 AND gates and 78209 registers.

EMM run *without* array simplification quickly completes 15 timesteps of BMC, after which a formidable resource spike is encountered due to the large number of arrays and ports – ultimately resulting in memout. The bit-blasted approach fares considerably better, completing 74 timesteps before memout. Array simplification enables EMM to yield substantially better results, completing 133 timesteps before memout. These results clearly illustrate the utility of automated techniques to convert circuit-accurate arrays to behavioral representations, without which bit-blasting may be a superior solution.

Phase Abstraction: Recall from Section IV that phase abstraction multiplies the number of read and write ports by its unfolding depth. However, for every netlist we have encountered for which phase abstraction reduced clocking complexity, array simplifications eliminate these duplicated ports as irrelevant (e.g., enables being conjuncted with a clock signal) or redundant (e.g., identical reads / writes occur across consecutive clock phases). Phase abstraction plus array simplification may thus *quarter* (or better) the size of EMM modelings through halving (or better) the number of read ports and write ports compared across a specific unfolding depth. This benefit is illustrated in Figure 3, where modulo-2 phase abstraction enabled the completion of 268 BMC timesteps before memout, requiring only 0.5 seconds of reduction time.

Correlated Row Abstraction: This netlist also illustrates the value of the correlated-address abstraction techniques discussed in Section VI. We focused on a single parity-style property. If applying the technique directly from [10], each of the 444 1-column arrays requires the modeling of a single row. This yields a substantial reduction; seven registers to represent each modeled address, and one for the modeled data, per array – vs. 128 registers for a precise bit-blasting. However, the large number of arrays entails a large collective abstraction size. Furthermore, the failure to model address correlation hampered subsequent verification: localization could not reduce the resulting netlist below 3241 registers, which we could not verify within an eight hour timeout.

In contrast, using our address-correlation optimization, only three abstract addresses need to be modeled across *all* 444 arrays. Localization and logic optimizations were able to reduce this address-correlated abstraction to only 32 registers, which interpolation solved within one second of runtime.

Localization: We have noted numerous additional benefits of applying localization without bit-blasting: **(1)** BMC tends to be much more efficient; **(2)** far fewer refinements need to be performed given fewer gates in the netlist; and **(3)** fewer necessary refinements entails fewer inevitable *mistakes* which unnecessary bloat the abstract netlist.

Sequential Redundancy Identification: To illustrate the benefit of identifying redundancies without operating on a bit-blasted netlist, we detail a sequential equivalence checking (SEC) problem involving a DRAM. This DRAM implementation and its redundancy scheme (used for fault-tolerance) was altered, yet in a way that preserved input-to-output behavior. One netlist has sixty-four 9-column, 128-row arrays; the other has four 144-column, 128-row arrays. The overall SEC problem additionally has 95786 AND gates and 5286 registers surrounding these arrays. Our redundancy identification framework from Section III-B is able to automatically identify 572 column equivalences and 1238 register equivalences inductively in 851 seconds. However, given changes in the fault-tolerance scheme, four columns and 2810 registers did not correspond hence the SEC problem remained unsolved; a combination of localization and interpolation on the redundancy-eliminated netlist was necessary to complete the overall SEC problem with a total runtime of 34 minutes. The bit-blasted variant has 770215 AND gates and 152710 registers, for which we were unable to even prove the equivalent sequential elements (without tedious *manual* correlation of array cells [19], [23]) given 48 hours of runtime.

Overall, redundancy identification substantially benefits without bit-blasting due to **(1)** speedups to BMC and simulation used to filter invalid candidate equivalences, and to induction used in proofs, and **(2)** requiring far fewer computations at the granularity of columns vs. cells.

VIII. Conclusion

Arrays are ubiquitous in industrial hardware designs, along with many control- and performance-related artifacts which practically mandate the availability of a large set of synergistic algorithms to enable automated verification. In this paper, we extend numerous traditionally bit-level state-of-the-art model checking and equivalence checking algorithms to support designs with arrays, and introduce automated techniques to transform arrays of circuit-accurate to behavioral syntax, enabling the use of higher-level reasoning techniques on problems of otherwise-unsuitable syntax. Nearly all algorithms used in a state-of-the-art model checker (simulators, logic optimization and abstraction techniques, isomorphism detection, ...) tend to significantly benefit from operating on the smaller non-bit-blasted netlist, in addition to the even more profound benefits that dedicated array reasoning techniques may offer. These techniques have collectively enabled dramatic scalability enhancements to our model checking and equivalence checking solutions, enabling automation for verification tasks that otherwise would have required significant manual guidance.

Acknowledgments: The authors wish to thank Per Bjesse for helpful feedback on this work.

References

[1] P. Bjesse and J. Kukula, "Automatic generalized phase abstraction for formal verification," in *ICCAD*, Nov. 2005.
[2] A. Kuehlmann and J. Baumgartner, "Transformation-based verification using generalized retiming," in *CAV*, July 2001.
[3] A. Kuehlmann, V. Paruthi, F. Krohm, and M. Ganai, "Robust Boolean reasoning for equivalence checking and functional property verification," *TCAD*, vol. 21, Dec. 2002.
[4] A. Mishchenko, S. Chatterjee, and R. Brayton, "DAG-aware AIG rewriting: A fresh look at combinational logic synthesis," in *DAC*, 2006.
[5] H. Mony, J. Baumgartner, A. Mishchenko, and R. Brayton, "Speculative-reduction based scalable redundancy identification," in *DATE*, Apr. 2009.
[6] H. Mony, J. Baumgartner, V. Paruthi, R. Kanzelman, and A. Kuehlmann, "Scalable automated verification via expert-system guided transformations," in *FMCAD*, Nov. 2004.
[7] Berkeley Logic and Synthesis Group, *ABC: A System for Sequential Synthesis and Verification*. http://www.eecs.berkeley.edu/alanmi/abc.
[8] M. N. Velev and R. E. Bryant, "Efficient modeling of memory arrays in symbolic ternary simulation," in *TACAS*, March 1998.
[9] M. Ganai, A. Gupta, and P. Ashar, "Verification of embedded memory systems using efficient memory modeling," in *DATE*, March 2005.
[10] P. Bjesse, "Word-level sequential memory abstraction for model checking," in *FMCAD*, Nov. 2008.
[11] J. McCarthy, "Towards a mathematical theory of computation," in *IFIP Congress*, 1962.
[12] G. E. Moore, "Cramming more components onto integrated circuits," *Electronics*, vol. 38, Apr. 1965.
[13] L. de Moura and N. Bjorner, "Generalized and efficient array decision procedures," in *FMCAD*, Nov. 2009.
[14] K. McMillan, "A methodology for hardware verification using compositional model checking," *Cadence Technical Report*, April 1999.
[15] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu, "Symbolic model checking without BDDs," in *TACAS*, March 1999.
[16] M. Sheeran, S. Singh, and G. Stalmarck, "Checking safety properties using induction and a SAT-solver," in *FMCAD*, Nov. 2000.
[17] P. Manolios, S. Srinivasan, and D. Vroon, "Automatic memory reductions for RTL model verification," in *ICCAD*, Nov. 2006.
[18] Q. Zhu, N. Kitchen, A. Kuehlmann, and A. Sangiovanni-Vincentelli, "SAT sweeping with local observability don't-cares," in *DAC*, July 2006.
[19] A. Koelbl, J. Burch, and C. Pixley, "Memory modeling in ESL-RTL equivalence checking," in *DAC*, June 2007.
[20] K. McMillan, "Interpolation and SAT-based model checking," in *CAV*, July 2003.
[21] M. Case, H. Mony, J. Baumgartner, and R. Kanzelman, "Enhanced verification through temporal decomposition," in *FMCAD*, Nov. 2009.
[22] P. Chauhan et al., "Automated abstraction refinement for model checking large state spaces using SAT based conflict analysis," in *FMCAD*, 2002.
[23] Z. Khasidashvili, M. Kinanah, and A. Voronkov, "Verifying equivalence of memories using a first order logic theorem prover," in *FMCAD*, 2009.

CalCS: SMT Solving for Non-Linear Convex Constraints

Pierluigi Nuzzo, Alberto Puggelli, Sanjit A. Seshia and Alberto Sangiovanni-Vincentelli
Department of Electrical Engineering and Computer Sciences, University of California, Berkeley
Berkeley, California 94720
Email: {nuzzo, puggelli, sseshia, alberto}@eecs.berkeley.edu

***Abstract*—Certain formal verification tasks require reasoning about Boolean combinations of non-linear arithmetic constraints over the real numbers. In this paper, we present a new technique for satisfiability solving of Boolean combinations of non-linear constraints that are convex. Our approach applies fundamental results from the theory of convex programming to realize a satisfiability modulo theory (SMT) solver. Our solver, CalCS, uses a lazy combination of SAT and a theory solver. A key step in our algorithm is the use of complementary slackness and duality theory to generate succinct infeasibility proofs that support conflict-driven learning. Moreover, whenever non-convex constraints are produced from Boolean reasoning, we provide a procedure that generates conservative approximations of the original set of constraints by using geometric properties of convex sets and supporting hyperplanes. We validate CalCS on several benchmarks including formulas generated from bounded model checking of hybrid automata and static analysis of floating-point software.**

I. Introduction

The design and verification of certain systems requires reasoning about nonlinear equalities and inequalities, both algebraic and differential. Examples range from mixed-signal integrated circuits (e.g., [1]) that should operate correctly over process-voltage-temperature variations, to control design for biological or avionics systems, for which safety must be enforced (e.g., [2]). In order to extend the reach of formal verification methods such as bounded model checking (BMC) for such systems [3], [4], it is necessary to develop efficient satisfiability modulo theories (SMT) solvers [5] for Boolean combinations of non-linear arithmetic constraints. However, SMT solving for arbitrary non-linear arithmetic over the reals, involving, e.g., quantifiers and transcendental functions, is undecidable [6]. There is therefore a need to develop efficient solvers for special cases that are also useful in practice.

In this paper, we address *the satisfiability problem for Boolean combinations of convex non-linear constraints*. We follow the lazy SMT solving paradigm [7], where a classic David-Putnam-Logemann-Loveland (DPLL)-style SAT solving algorithm interacts with a theory solver based on fundamental results from convex programming. The theory solver needs only to check the feasibility of conjunctions of theory predicates passed onto it from the SAT solver. However, when all constraints are convex, a satisfying valuation can be found using interior point methods [8], running in polynomial time.

A central problem in a lazy SMT approach is for the theory solver to generate a compact explanation when the conjunction of theory predicates is unsatisfiable. We demonstrate how this can be achieved for convex constraints using duality theory for convex programming. Specifically, we formulate the convex programming problem in a manner that allows us to easily obtain the subset of constraints responsible for unsatisfiability.

Additionally, even when constraints are restricted to be convex, it is possible that, during Boolean reasoning, some of these constraints become negated, and thus the theory solver must handle some non-convex constraints. We show how to handle such constraints by set-theoretic reasoning and approximation with affine constraints.

The main novel contributions of our work can be summarized as follows:

- We present the first SMT solver for a Boolean combination of convex non-linear constraints. Our solver exploits information from the solution of convex optimization problems to establish satisfiability of conjunctions of convex constraints;
- We give a novel formulation that allows us to generate certificates of unsatisfiability in case the conjunction of theory predicates is infeasible, thus enabling the SMT solver to perform conflict-directed learning;
- Whenever non-convex constraints originate from convex constraints due to Boolean negation, we provide a procedure that can still use geometric properties of convex sets and supporting hyperplanes to generate approximations of the original set of constraints;
- We present a proof-of-concept implementation, CalCS, that can deal with a much broader category than linear arithmetic constraints, also including conic constraints, as the ones in quadratic and semidefinite programs, or any convex relaxations of other non-linear constraints [8]. We validate our approach on several benchmarks including formulas generated from BMC for hybrid systems and static analysis of floating-point programs, showing that our approach can be more accurate than current leading non-linear SMT solvers such as iSAT [9].

The rest of the paper is organized as follows. In Section II, we briefly review some related work in both areas on which this work is based, i.e. SMT solving for non-linear arithmetic constraints and convex optimization. In Section III, we describe background material including the syntax and semantics of the SMT problems our algorithm handles. Section IV introduces to the convex optimization concepts that our development builds on and provides a detailed explanation of our algorithm. In Section V we report implementation details on integrating convex and SAT solving. After presenting some benchmark results in Section VI, we conclude with a summary of our work and its planned extensions.

II. Related Work

An SMT instance is a formula in first-order logic, where some function and predicate symbols have additional interpretations related to specific theories, and SMT is the problem

of determining whether such a formula is satisfiable. Modern SAT and SMT solvers can efficiently find satisfying valuations of very large propositional formulae, including combinations of atoms from various decidable theories, such as lists, arrays, bit vectors [5]. However, extensions of the SMT problem to the theory of non-linear arithmetic constraints over the reals have only recently started to appear. Since our work combines both SAT/SMT solving techniques with convex programming, we briefly survey related works in both of these areas.

A. SMT solving for non-linear arithmetic constraints

Current SMT solvers for non-linear arithmetic adopt the lazy combination of a SAT solver with a theory solver for non-linear arithmetic. The ABsolver tool [10] adopts this approach to solve Boolean combinations of polynomial non-linear arithmetic constraints. The current implementation uses the numerical optimization tool IPOPT [11] for solving the non-linear constraints. However, without any other additional property for the constraints, such as convexity, the numerical optimization tool will necessarily produce incomplete results, and possibly incorrect, due to the local nature of the solver (all variables need upper and lower bounds). Moreover, in case of infeasibility, no rigorous procedure is specified to produce infeasibility proofs.

A completely different approach is adopted by the iSAT algorithm that builds on a unification of DPLL SAT-solving and interval constraint propagation [9] to solve arithmetic constraints. iSAT directly controls arithmetic constraint propagation from the SAT solver rather than delegating arithmetic decisions to a subordinate solver, and has shown superior efficiency. Moreover, it can address a larger class of formulae than polynomial constraints, admitting arbitrary smooth, possibly transcendental, functions. However, since interval consistency is a necessary, but not sufficient condition for real-valued satisfiability, spurious solutions can still be generated.

To reason about round-off errors in floating point arithmetic an efficient decision procedure (CORD) based on precise arithmetic and CORDIC algorithms has been recently proposed by Ganai and Ivancic [12]. In their approach, the non-linear part of the decision problem needs first to be translated into a linear arithmetic (LA) formula, and then an off-the-shelf SMT-LA solver and DPLL-style interval search are used to solve the linearized formula. For a given precision requirement, the approximation of the original problem is guaranteed to account for all inaccuracies.

B. Convex Programming

An SMT solver for the non-linear convex sub-theory is motivated by both theoretical and practical reasons. On the one hand, convex problems can be solved very efficiently today, and rely on a fairly complete and mature theory. On the other hand, convex problems arise in a broad variety of applications, ranging from automatic control systems, to communications, electronic circuit design, data analysis and modeling [8]. The solution methods have proved to be reliable enough to be embedded in computer-aided design or analysis tool, or even in real-time reactive or automatic control systems. Moreover, whenever the original problem is not convex, convex problems can still provide the starting point for other local optimization methods, or a cheaply computable lower bounds via constraint or Lagrangian relaxations. A thorough reference on convex programming and its applications can be found in [8].

As an example, convex optimization has been used in electronic circuit design to solve the sizing problem [13]–[15]. Robust design approaches based on convex models of mixed-signal integrated circuits have also been presented in [16], [17]. While, in these cases, there was no Boolean structure, Boolean combinations of convex constraints arise when the circuit topology is not fixed, or for cyber-physical systems where continuous time dynamics need to be co-designed with discrete switching behaviors between modes. It is therefore necessary to have solvers that can reason about both Boolean and convex constraints.

In the context of optimal control design for hybrid systems, the work in [18], [19] proposes a combined approach of mixed-integer-programming (MIP) and constraint satisfaction problems (CSP), and specifically, convex programming and SAT solvers, as in our work. The approach in [18], [19] is, in some respects, complementary to ours. A SAT problem is first used to perform an initial logic inference and branching step on the Boolean constraints. Convex relaxations of the original MIP (including Boolean variables) are then solved by the optimization routine, which iteratively calls the SAT solver to ensure that the integer solution obtained for the relaxed problem is feasible and infer an assignment for the logic variables that were assigned to fractional values from the MIP. However, the emphasis in [18], [19] is more on speeding up the optimization over a set of mixed convex and integer constraints, rather than elaborating a decision procedure to verify feasibility of Boolean combinations of convex constraints, or generate infeasibility proofs. Additionally, unlike [18], [19], by leveraging conservative approximations, our work can also handle disjunctions of convex constraints.

III. Background and Terminology

We cover here some background material on convexity and define the syntax of the class of SMT formulae of our interest.

Convex Functions. A function $f : \mathbb{R}^n \rightarrow \mathbb{R}$ is termed *convex* if its domain $\mathbf{dom} f$ is a convex set and if for all $x, y \in \mathbf{dom} f$, and θ with $0 \leq \theta \leq 1$, we have

$$f(\theta x + (1-\theta)y) \leq \theta f(x) + (1-\theta) f(y). \qquad (1)$$

Geometrically, this inequality means that the *chord* from x to y lies above the graph of f. As a special case, when (1) always holds as an equality, then f is *affine*. All linear functions are also affine, hence convex. It is possible to recognize whether a function is convex based on certain properties. For instance, if f is differentiable, then f is convex if and only if $\mathbf{dom} f$ is convex and $f(y) \geq f(x) + \nabla f(x)^T (y - x)$ holds for all $x, y \in \mathbf{dom} f$, and $\nabla f(x)$ is the gradient of f. The above inequality states that if f is convex, its first-order Taylor approximation is always a global underestimator. The converse result can be also shown to be true. If f is twice differentiable, then f is convex if and only if $\mathbf{dom} f$ is convex and its Hessian $\nabla^2 f(x)$ is positive semidefinite matrix for all $x \in \mathbf{dom} f$. In addition to linear, affine, and positive semi-definite quadratic forms, examples of convex functions may include exponentials (e.g. e^{ax}), powers (e.g. x^a when $a \geq 1$), logarithms (e.g. $-\log(x)$), the $\max$ function, and all norms.

Convex Constraint. A convex constraint is of the form $f(x)\ \{<, \leq, >, \geq\}\ 0$ or $h(x) = 0$, where $f(x)$ and $h(x)$ are convex and affine (linear) functions, respectively, of their real variables $x \in \mathcal{D} \subseteq \mathbb{R}^n$, with $\mathcal{D}$ being a convex set. In the

following, we also denote a constraint in the form $f(x) \leq 0$ ($f(x) < 0$) as a *convex* (*strictly convex*) constraint (CC), where $f(x)$ is a convex function on its convex domain. A convex constraint is associated with a set $\mathcal{C} = \{x \in \mathbb{R}^n : f(x) \leq 0\}$, i.e. the set of points in the space that satisfy the constraint. Since $\mathcal{C}$ is the 0-sublevel set of the convex function $f(x)$, $\mathcal{C}$ is also convex. We further denote the negation of a (strictly) convex constraint, expressed in the form $f(x) > 0$ ($f(x) \geq 0$), as *reversed convex* (*reversed strictly convex*) constraint (RCC). An RCC is, in general, non-convex as well as its satisfying set $\mathcal{N} = \{x \in \mathbb{R}^n : f(x) > 0\}$. The complement $\bar{\mathcal{N}}$ of $\mathcal{N}$ is, however, convex.

Syntax of Convex SMT Formulae. We represent SMT formulae over convex constraints to be quantifier-free formulae in conjunctive normal form, with atomic propositions ranging over propositional variables and arithmetic constraints. The formula syntax is therefore as follows:

$$\begin{array}{rcl} formula & ::= & \{clause\wedge\}^* clause \\ clause & ::= & (\{literal\vee\}^* literal) \\ literal & ::= & atom \mid \neg atom \\ atom & ::= & conv\_constraint \mid bool\_var \\ conv\_constraint & ::= & equation \mid inequality \\ equation & ::= & affine\_function = 0 \\ inequality & ::= & convex\_function\ relation\ 0 \\ relation & ::= & < \mid \leq \end{array}$$

In the grammar above, *bool_var* denotes a Boolean variable, and *affine_function* and *convex_function* denote affine and convex functions respectively. The terms *atom* and *literal* are used as is standard in the SMT literature. Note that the only theory atoms are convex or affine constraints. Even though we allow negations on convex constraints (hence allowing non-convex constraints), we will term the resulting SMT formula as a *convex SMT formula*.

Our constraint formulae are interpreted over valuations $\mu \in (BV \rightarrow \mathbb{B}) \times (RV \rightarrow \mathbb{R})$, where *BV* is the set of Boolean and *RV* the set of real-valued variables. The definition of satisfaction is also standard: a formula ϕ is satisfied by a valuation μ ($\mu \models \phi$) iff all its clauses are satisfied, that is, iff at least one atom is satisfied in any clause. A literal l is satisfied if $\mu_{\mathbb{B}}(l) = \texttt{true}$. Satisfaction of real constraints is with respect to the standard interpretation of the arithmetic operators and the ordering relations over the reals.

Based on the above definitions, here is an example of a convex SMT formula:

$$\begin{array}{l} (x + y - 3 = 0 \vee a \vee -\log(\mathrm{e}^x + \mathrm{e}^y) + 10 \geq 0) \\ \wedge\, (\neg b \vee ||(x-2, z-3)||_2 \leq y - 5) \wedge (x^2 + y^2 - 4x \leq 0) \\ \wedge\, \left(\neg a \vee y < 4.5 \vee \max\{2x + z, 3x^2 + 4y^4 - 4.8\} < 0\right), \end{array} \tag{2}$$

where $a, b \in BV$, $x, y, z \in RV$, and $||\cdot||_2$ is the Euclidean norm on $\mathbb{R}^2$.

If the SMT formula does not contain any negated convex constraint, the formula is termed a *monotone convex SMT formula*.

IV. Theory Solver for Convex Constraints

In optimization theory, the problem of determining whether a set (conjunction) of constraints are consistent, and if so, finding a point that satisfies them, is a *feasibility problem*. The feasibility problem for convex constraints can be expressed in the form

$$\begin{array}{ll} \text{find} & x \\ \text{subject to} & f_i(x) \leq 0, \quad i = 1, \ldots, m \\ & h_j(x) = 0, \quad j = 1, \ldots, p \end{array} \tag{3}$$

where the single (vector) variable $x \in \mathbb{R}^n$ represents the n-tuple of all the real variables $(x_1, \ldots, x_n)^T$, the f_i functions are convex, and the h_j functions are affine. As in any optimization problem, if x is a feasible point and $f_i(x) = 0$, we say that the i-th inequality constraint $f_i(x) \leq 0$ is *active* at x. If $f_i(x) < 0$, we say the constraint $f_i(x) \leq 0$ is *inactive*. The equality constraints are active at all feasible points. For succinctness of presentation, we make the assumption that inequalities are non-strict (as listed in (3)), but our approach extends to systems with strict inequalities as well.

In this section, we describe how we construct a theory solver for a convex SMT formula that generates explanations when a system of constraints is infeasible. In general, the system of constraints can have both convex constraints and negated convex constraints (which can be non-convex). We will first consider the simpler case where all constraints in the system are convex, and show how explanations for infeasibility can be constructed by a suitable formulation that leverages duality theory (Section IV-A). We later give an alternative formulation (Section IV-B) and describe how to deal with the presence of negated convex constraints (Section IV-C).

Although it is possible to directly solve feasibility problems by turning them into optimization problems in which the objective function is identically zero [8], no information about the reasons for inconsistency would be propagated with this formulation, in case of infeasibility. Therefore, we cast the feasibility problem (3) as a combination of optimization problems with the addition of slack variables. Each of these newly generated problems is an equivalent formulation of the original problem (and it is therefore in itself a feasibility problem), while at the same time being richer in informative content. In particular, given a conjunction of convex constraints, our framework builds upon the following equivalent formulations of (3), namely the *sum-of-slacks* feasibility problem (*SSF*), and the *single-slack* feasibility (*SF*) problem, both detailed below.

A. Sum-of-Slacks Feasibility Problem

In the *SSF* problem, a slack variable s_i is introduced for every single constraint, so that (3) turns into the following

$$\begin{array}{ll} \text{minimize} & \sum_{k=1}^{m+2p} s_k \\ \text{subject to} & \tilde{f}_k(x) - s_k \leq 0, \quad k = 1, \ldots, m + 2p \\ & s_k \geq 0 \end{array} \tag{4}$$

where $\tilde{f}_k(x) = f_k(x)$ for $k = 1, \ldots, m$, $\tilde{f}_{m+j}(x) = h_j(x)$, and $\tilde{f}_{m+p+j}(x) = -h_j(x)$ for $j = 1, \ldots, p$. In other words, every equality constraint $h_j(x) = 0$ is turned into a conjunction of two inequalities, $h_j(x) \leq 0$ and $-h_j(x) \leq 0$ before applying the reduction in (4). The *SSF* problem can be interpreted as trying to minimize the infeasibilities of the constraints, by pushing each slack variable to be as much as

possible close to zero. The optimum is zero and is achieved if and only if the original set of constraints (3) is feasible.

Based on *duality theory* [8], a *dual problem* is associated with (4), which maximizes the *Lagrange dual function* associated with (4), under constraints on the *dual variables* or *Lagrange multipliers*. While the dual optimal value always provides a lower bound to the original (*primal*) optimum, an important case obtains when this bound is tight and the two primal and dual optima coincide (*strong duality*). As a simple sufficient condition, Slater's theorem states that strong duality holds if the problem is convex, and there exists a strictly feasible point, such that the non-linear inequality constraints hold with strict inequalities. As a consequence of duality theory, the following result holds for (4) at optimum:

Proposition IV.1. *Let $(x^*, s^*) \in \mathbb{R}^{n+m+2p}$ be a primal optimal and $z^* \in \mathbb{R}^{m+2p}$ be a dual optimal point for* (4)*. Then: (i) if* (3) *is feasible, x^* provides a satisfying assignment; (ii) moreover, we obtain:*

$$z_k^*(\tilde{f}_k(x^*) - s_k^*) = 0 \quad k = 1, \ldots, m + 2p. \tag{5}$$

Proof sketch: The first statement trivially follows from the solution of problem (4). Since x^* is the optimal point, it also satisfies all the constraints in (4) with $s_k = s_k^* = 0$, therefore it is a satisfying assignment for (3). The second statement follows from *complementary slackness*. In fact, under the assumptions in Section III, (4) is a convex optimization problem. Moreover, it is always possible to find a feasible point which strictly satisfies all the nonlinear inequalities since, for a any given x, the slack variables s_k can be freely chosen, hence Slater's conditions hold. As a result, strong duality holds as well, i.e. both the primal and dual optimal values are attained and equal, which implies complementary slackness, as in (5). □

We use complementary slackness to generate infeasibility certificates for (3). In fact, if a constraint k is strictly satisfied (i.e. $s_k^* = 0$ and $\tilde{f}_k(x^*) < 0$) then the relative dual variable is zero, meaning that the constraint $\tilde{f}_k(x^*) \leq 0$ is actually non-active. Conversely, a non-zero dual variable will necessary correspond to either an unfeasible constraint ($s_k^* > 0$) or to a constraint that is non strictly satisfied ($s_k^* = 0$). In both cases, the constraint $\tilde{f}_k(x^*) \leq s_k$ is active at optimum and it is one of the reasons for the conflict. We can therefore conclude with the following result:

Proposition IV.2. *The subset of constraints in* (4) *that are related to positive dual variables at optimum represents the active subset, and therefore provides a succinct reason of infeasibility (certificate).* □

Numerical issues must be considered while implementing this approach. When (3) is feasible, the optimization algorithm in practice will terminate with $|\sum_{k=1}^{m+2p} s_k| \leq \epsilon_t$, thus producing an ϵ_t-suboptimal point for arbitrary small, positive ϵ_t. Accordingly, to enforce strict inequalities such as $\tilde{f}_k(x) < 0$, we modify the original expression with an additional user-defined positive slack constant ϵ_s as $\tilde{f}_k(x) + \epsilon_s \leq 0$, thus requiring that the constraint be satisfied with a desired margin ϵ_s. All the above conclusions valid for (3) can then be smoothly extended to the modified problem.

B. Single-Slack Feasibility Problem

While the *SSF* problem is the workhorse of our decision procedure, we also present an alternative formulation of the feasibility problem, which will be useful in the approximation of RCCs.

The *SF* problem minimizes the maximum infeasibility s of a set of convex constraints as follows

$$\begin{aligned} &\text{minimize} \quad s \\ &\text{subject to} \quad \tilde{f}_k(x) - s \leq 0, \quad k = 1, \ldots, \ m + 2p \end{aligned} \tag{6}$$

where inequalities are pre-processed as in Section IV-A. The goal is clearly to drive the maximum infeasibility below zero. At optimum the sign of the optimal value s^* provides feasibility information. If $s^* < 0$, (6) has a strictly feasible solution; if $s^* > 0$ then (6) is infeasible; finally, if $s^* = 0$ (in practice $|s^*| \leq \epsilon_t$ for some small $\epsilon_t > 0$) and the minimum is attained, then the set of inequalities is feasible, but not strictly feasible. As in (4), complementary slackness will hold at optimum, i.e.

$$z_k^*(\tilde{f}_k(x^*) - s^*) = 0 \quad k = 1, \ldots, m + 2p.$$

Therefore, even when the problem is feasible, whenever a constraint k is not active, then $(\tilde{f}_k(x^*) - s^*) \neq 0$ will be strictly satisfied, and imply $z_k = 0$. Conversely, if $z_k \neq 0$, then the constraint $(\tilde{f}_k(x^*) - s^*)$ is certainly active and $\tilde{f}_k(x)$ contributes to determine the maximum infeasibility for the given problem, in the sense that if s^* was further pushed to be more negative, $\tilde{f}_k(x)$ would be no longer satisfied.

C. Dealing with Reversed Convex Constraint

A negated (reversed) convex constraint (an RCC) is non-convex and defines a non-convex set $\mathcal{N}$. Any conjunction of these non-convex constraints with other convex constraints results in general in a non-convex set. To deal with such non-convex sets, we propose heuristics to compute convex over- and under-approximations, which can then be solved efficiently. This section describes these techniques.

Our approximation schemes are based on noting that the complementary set $\bar{\mathcal{N}}$ is convex. Therefore geometric properties of convex sets, such as strict or weak separation [8], can still be used to approximate or bound $\mathcal{N}$ via a supporting hyperplane. Once a non-convex constraint is replaced with a bounding hyperplane, the resulting *approximate problem* (AP) will again be convex, and all the results in Section IV-A will be valid for this approximate problem.

For simplicity, we assume in this section that we have exactly one non-convex constraint (RCC), and the rest of the constraints are convex. We will describe the general case in Sec. IV-D. Let $g(x)$ be the convex function associated with the RCC. Our approach proceeds as follows:

1) Solve the sum-of-slacks (SSF) problem for just the convex constraints. Denote the resulting convex region by $\mathcal{B}$.
   If the resulting problem is UNSAT, report this answer along with the certificate computed as described in Sec. IV-A.
   Otherwise, if the answer returned is SAT, denote the optimal point as x_b^* (satisfying assignment) and proceed to the next step.

2) Add the negation of the RCC (a convex constraint) and solve the SSF problem again, which we now denote as *reversed problem* (RP). There are two cases:
   (a) If the answer is UNSAT, then the RCC region $\bar{\mathcal{N}}$ does not intersect the convex region $\mathcal{B}$. This implies that $\mathcal{B} \subset \mathcal{N}$, and hence the RCC is a redundant constraint. This situation is illustrated in Fig. 1(a). Thus, the solver can simply return SAT (as returned in the previous step).
   (b) On the other hand, if the answer is SAT, we denote as x_c^* the optimal point of the RP and check whether the negated RCC is now redundant, based on the shift induced in the optimal point x_b^*. In particular, if both x_c^* and x_b^* are inside $\bar{\mathcal{N}}$, we solve two single-slack feasibility (*SF*) problems, and we denote as $\tilde{x}_b^*$ and $\tilde{x}_c^*$ the two optimal points, for the problem having just the convex constraints and for the the RP, respectively. Similarly, we denote the two optimal values as $\tilde{s}_b^*$ and $\tilde{s}_c^*$.
   As also observed in Section IV-B, for a set of satisfiable constraints, $\tilde{x}_b^*$, $\tilde{x}_c^*$, $\tilde{s}_b^*$ and $\tilde{s}_c^*$ may contain more information than the optimal points x_b^* and x_c^* (and their slack variables) for the *SSF* problem. In fact, since $\tilde{s}_b^*$ and $\tilde{s}_c^*$ are also allowed to assume negative (hence different) values at optimum, they can provide useful indications on how the RCC has changed the geometry of the feasible set, and which constraints are actually part of its boundary, thus better driving our approximation scheme. In particular, if we verify that $\tilde{s}_b^* = \tilde{s}_c^*$, $\tilde{x}_b^* = \tilde{x}_c^*$, and $\mathcal{B} \subset \bar{\mathcal{N}}$, then we imply $\mathcal{B} \cap \mathcal{N} = \emptyset$. Hence, the solver can return UNSAT. Techniques to detect if a conjunction of convex constraints generates sets that are (exactly or approximately) contained in a convex set are reported in [20], [21]. For instance, when both $\mathcal{B}$ and $\bar{\mathcal{N}}$ are spheres, the condition $\mathcal{B} \subset \bar{\mathcal{N}}$ is equivalent to checking that the slack constraint related to the RCC is not active at optimum in the *SF* problem. This case is illustrated in Fig. 1(b) for the following conjunction of constraints:

$$(x_1^2 + x_2^2 - 1 \leq 0) \wedge (x_1^2 + x_2^2 - 4 > 0)$$

   where $(x_1^2+x_2^2-4 > 0)$ is the non-convex constraint defining region $\mathcal{N}$. If set containment cannot be exactly determined the procedure returns UNKNOWN.

   If none of the above cases hold, we proceed to the next step. For example, this is the case whenever x_b^* is outside $\bar{\mathcal{N}}$, or on its boundary (i.e. $g(x_b^*) \geq 0$). This implies that the negated RCC is not redundant, and we can move to the next step without solving the two *SF* problems.
3) In this step, we generate a convex under-approximation of the original formula including the convex constraints and the single non-convex RCC. If the resulting problem is found satisfiable, the procedure returns SAT. Otherwise, it returns UNKNOWN.

We now detail the under-approximation procedure in Step 3. As an illustrative example, we use a 2-dimensional region defined by the following SMT formula:

$$(x_1^2+x_2^2-1 \leq 0) \wedge (x_1^2+x_2^2-4x_1 \leq 0) \wedge (x_1^2+x_2^2-2x_2 > 0). \quad (7)$$

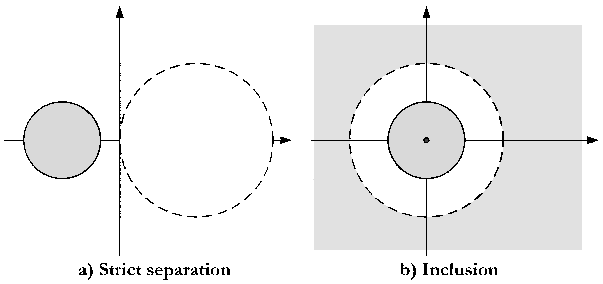


Fig. 1. Two special cases for handling non-convex constraints: (a) by adding a negated RCC a new set is generated that is strictly separated from the previous convex set; (b) the negated RCC generates a set that totally includes the previous convex set.

As apparent from the geometrical representation of the sets in Fig. 2 a), the problem is clearly satisfiable and a satisfying valuation could be any point in the grey region $\mathcal{A}$.

First, we note for this example the results obtained before the under-approximation is performed. We solve the *SSF* problem for the convex set $\mathcal{B} = \{(x_1, x_2) \in \mathbb{R}^2 : (x_1^2 + x_2^2 - 1 \leq 0) \wedge (x_1^2 + x_2^2 - 4x_1 \leq 0)\}$, obtained from $\mathcal{A}$ after dropping the RCC N. The problem is feasible, as shown in Fig. 2 (b), and the optimal point $x_b^* = (0.537, 0)$ is returned.

Next, the RCC is negated to become convex and the *SSF* problem is now solved on the newly generated formula

$$(x_1^2+x_2^2-1 \leq 0) \wedge (x_1^2+x_2^2-4x_1 \leq 0) \wedge (x_1^2+x_2^2-2x_2 \leq 0)$$

which represents the previously defined (RP). The RP will provide useful information for the approximation, thus acting as a "geometric probe" for the optimization and search space. Since the RCC is reversed, the RP is convex and generates the set $\mathcal{C}$, shown in Fig. 2 (c).

Let us assume, at this point, that the RP is feasible, as in this example. Then $\mathcal{C} \neq \emptyset$, and an optimal point $x_c^* = (0.403, 0.429) \in \mathcal{C}$ is provided. Moreover, $\mathcal{A}$ can be expressed as $\mathcal{B} \setminus \mathcal{C}$, and x_b^* is clearly outside the convex set $\bar{\mathcal{N}}$ generated by the negated RCC, meaning that we can go to the under-approximation step without solving the SF problems since the negated RCC is certainly non-redundant.

The key idea for under-approximation is to compute a hyperplane that we can use to separate the RCC region $\mathcal{N}$ from the remaining convex region. This "cut" in the feasible region is performed by exploiting the perturbation of the optimal point from x_b^* to x_c^* induced by the negated RCC $\bar{N} : (x_1^2 + x_2^2 - 2x_2) \leq 0$. At this point, we examine a few possible cases:

Case (i): Suppose that $x_b^* \neq x_c^*$, and x_b^* is outside $\bar{\mathcal{N}}$ (as in our example). In this case, we find the orthogonal projection $p = \mathcal{P}(x_b^*)$ onto $\bar{\mathcal{N}}$, which can be performed by solving a convex, L_2-norm minimization problem [8]. Intuitively, this corresponds to projecting x_b^* onto a point p on the boundary of the region $\bar{\mathcal{N}}$. Finally, we compute the supporting hyperplane to $\bar{\mathcal{N}}$ in p. The half-space defined by this hyperplane that excludes $\bar{\mathcal{N}}$ provides our convex (affine) approximation $\tilde{\mathcal{N}}$ for $\mathcal{N}$.

For our example, $\bar{\mathcal{N}} = \{x \in \mathbb{R}^n : x_1^2 + x_2^2 - 2x_2 \leq 0\}$. The affine constraint resulting from the above procedure is $\tilde{N} : -0.06x_1 + 0.12x_2 + 0.016 < 0$. On replacing the RCC N with $\tilde{N}$, we obtain a new set $\mathcal{D}$, as shown Fig. 2(d), which is now our approximation for $\mathcal{A}$.

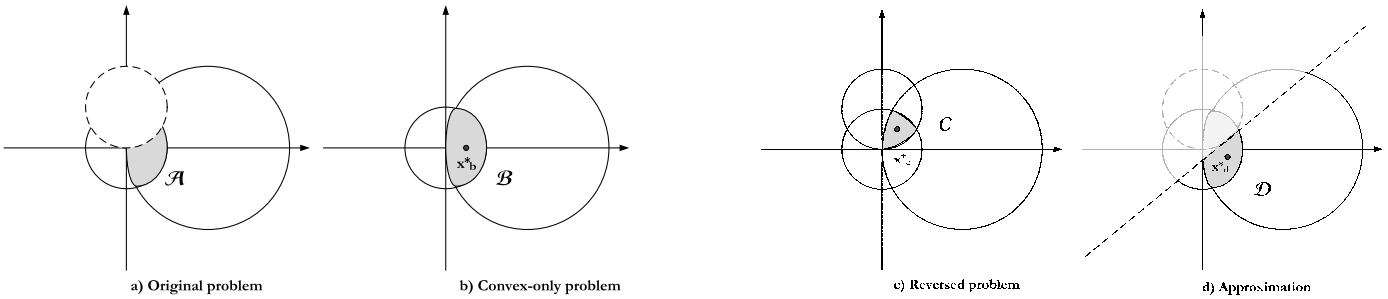


Fig. 2. Geometrical representation of the sets used in Section IV-C to illustrate the approximation scheme in CalCS: (a) $\mathcal{A}$ is the search space (in grey) for the original non-convex problem including one RCC constraint; (b) $\mathcal{B}$ is search space when the RCC is dropped (over-approximation of $\mathcal{A}$); (c) $\mathcal{C}$ is the search space for the *reversed problem*, i.e. the problem obtained from the original one in (a) when the RCC is negated; the RP is therefore convex; (d) $\mathcal{D}$ is the under-approximation of $\mathcal{D}$ in (a) using a supporting hyperplane.

An SSF problem can now be formulated for $\mathcal{D}$ thus providing the satisfying assignment $x_d^* = (0.6, -0.33)$. The approximation procedure will stop here and return SAT.

Notice that, whenever x_b^* is on the boundary of $\bar{\mathcal{N}}$, a similar approximation as described above can be performed. In this case, x_b^* is the point through which the supporting hyperplane needs to be computed, and no orthogonal projection is necessary. The normal direction to the plane needs, however, to be numerically computed by approximating the gradient of $g(x)$ in x_b^*.

Case (ii): A second case occurs when $x_b^* \neq x_c^*$, but both x_b^* and x_c^* are inside $\bar{\mathcal{N}}$. In this case, starting from x_c^* we search the closest boundary point along the $(x_b^* - x_c^*)$ direction, and then compute the supporting hyperplane through this point as in the previous case. In fact, to find an under-approximation for the feasible region $\mathcal{A}$, we are looking for an *over-approximation* of the set $\bar{\mathcal{N}}$ in the form of a tangent hyperplane. Since the optimal point x_b^* moves to x_c^* after the addition of the negated RCC, $\bar{\mathcal{N}}$ will be more "centered" around x_c^* than around x_b^*. Therefore, a reasonable heuristic could be to pick the direction starting from x_c^* and looking outwards, namely $(x_b^* - x_c^*)$.

Case (iii): Assume now that $x_b^* = x_c^*$ (with both x_b^* and x_c^* inside $\bar{\mathcal{N}}$), but we have $\tilde{x}_b^* \neq \tilde{x}_c^*$, where $\tilde{x}_b^*$ and $\tilde{x}_c^*$ are the two optimal points, respectively, for the *SF* problem having just the convex constraints and for the the RP in the *SF* form, as computed in Step 2 (b) above. In this case, to operate the "cut", we cannot use the perturbation on x_b^* and x_c^*, as in Case (ii), but we can still exploit the information contained in the SF problems. This time, starting from $\tilde{x}_c^*$, we search the closest boundary point along the $(\tilde{x}_b^* - \tilde{x}_c^*)$ direction, and then compute the supporting hyperplane through this boundary point.

Case (iv): Finally, both $x_b^* = x_c^*$ and $\tilde{x}_b^* = \tilde{x}_c^*$ can also occur, as for the following formula:

$$(x_1^2 + x_2^2 - 1 \geq 0) \wedge (x_1^2 + x_2^2 - 4 \leq 0),$$

for which $\mathcal{A}$ would coincide with the white ring region in Fig. 1 b) (including the dashed boundary). In this case, no useful information can be extracted from perturbations in the optimal points. The feasible set appears "isotropic" to both x_b^* and $\tilde{x}_b^*$, meaning that any direction could potentially be chosen for the approximations. In our example, we infer from the *SF* problems that the inner circle is the active constraint and we need to replace the non-convex constraint corresponding to its exterior with a supporting hyperplane, e.g. $-x_1 + 1 \leq 0$, by simply picking it to be orthogonal to one of the symmetry axes of the feasible set. The resulting under-approximation is found SAT and we obtain a satisfying assignment consistent with this approximation.

This completes the description of the under-approximation procedure of Step 3. We note that we still have the possibility for the solver to return UNKNOWN. Depending on the target application, the user can interpret this as SAT (possibly leading to spurious counterexamples in BMC) or UNSAT (possibly missing counterexamples). For higher accuracies, the approximation scheme can also be iterated over a set of boundary points of the original constraint $f(x)$, to build a finer polytope bounding the non-convex set.

D. Overall Algorithm

Our theory solver is summarized in Fig. 3. This procedure generalizes that described in the preceding section by handling multiple reversed convex constraints (RCCs). In essence, if the conjunction of all convex constraints and any single RCC is found UNSAT, then we report UNSAT. In order to report SAT, on the other hand, we must consider all convex constraints and all affine under-approximations of non-convex constraints.

The details are as follows. For a given conjunction of CCs and RCCs, we first solve the *SSF* problem generated by the CCs alone (Section IV-A). If the problem is UNSAT, the algorithm returns the subset of constraints that provide the reason for inconsistency (infeasibility certificate) and stops. Otherwise, each RCC is processed sequentially. For each RCC, the initial convex problem is augmented and the RP is formulated and solved. If the RP is unfeasible then, as discussed in Section IV-C, the constraint is ignored since it is non-active for the current feasibility problem. On the contrary, if the RP is feasible we proceed by computing an approximation.

The `Approximate` method implements the under-approximation strategies outlined in Section IV-C and determines whether the constraint is non-active or can be dropped by solving additional SF problems (Section IV-B). If the negated RCC is fully included in the set generated by the CCs alone, (e.g. the megated RCC and the set generated by the CCs are both circles and the negated RCC is non-active for the RP) the problem is UNSAT, meaning that the RCC is incompatible with the whole set of CC (step 2(b) of Section IV-C). The full set, including both the CC and the

```
function [status, out] = Decision_Manager(CC, RCC)
% receive a set of convex (CC) and non-convex constraints (RCC)
% return SAT/UNSAT/UNKNOWN and MODEL/CERTIFICATE
%
% solve sum-of-slacks feasibility problems with CCs
[status, out] = SoS_solve(CC);
% OUT contains CERTIFICATE
if (status == UNSAT) return; end
AC = CC;% AC stores all constraints
for (k = 1, k <= length(RCC), k++)
    RP = reverse(CC, RCC(k));
    [status, out] = SoS_solve(RP);
    % strict separation: ignore RCC
    if (RP == UNSAT) continue; end
    % both CC and RP problems are SAT: approximation
    [approxCC, active, drop] = Approximate(RCC(k));
    % RCC incompatible (inclusion)
    if (~active)
        status = UNSAT;
        % certificate
        out = [CC, RCC(k)]; return;
        % over-approximation: ignore constraint
    elseif (drop) continue;
    else AC = AC ∪ approxCC;
        [status, out] = SoS_solve(AC);
        if (status == SAT)
          Check SAT assignment on original constraints;
          if (original constraints satisfied) status = SAT; return;
          end
        end
    end
end
status = UNKNOWN;
```

Fig. 3. Pseudo-code for the CalCS decision procedure.

current RCC is returned as an explanation for the conflict. If an over-approximation is required, then the constraint is ignored. If the constraint is compatible and cannot be dropped, the supporting hyperplane is computed and the new under-approximated problem is solved. The algorithm proceeds with visiting the other RCCs. Finally, when all non-convex constraints have been processed without returning UNSAT the algorithm is re-invoked on the set of convex constraints CC and the set of affine under-approximations of non-convex constraints RCC. If this invocation returns SAT, so does the overall algorithm; otherwise, it returns UNKNOWN. A SAT answer is accompanied by a satisfying valuation to variables.

V. Integrating Convex Solving and SAT Solving

Using the theory solver described in Section IV, we have implemented a proof-of-concept SMT solver, CalCS, that supports the convex sub-theory. As in [10], CalCS receives as input an SMT formula in a DIMACS-like CNF format, where atomic predicates can be both Boolean or convex constraints, according to the definitions in Section III. Following the lazy theorem proving paradigm, the SMT problem is first transformed into a SAT problem, by mapping the nonlinear constraints into auxiliary Boolean variables. This Boolean abstraction of the original formula is then passed to the SAT solver. If the outcome is UNSAT, the theory manager terminates and returns UNSAT. Conversely, the assigned auxiliary variables are mapped back to a conjunction of CC and RCC and are sent to the theory for consistency checking. If the theory solver returns SAT, a combined Boolean and real *model* (satisfying assignment) is also returned. Otherwise, whenever inconsistencies are found (UNSAT), the reason for the conflict (certificate) is encoded into the *learned clause* $(\neg l_1 \vee \ldots \vee \neg l_k)$, $l_1, \ldots, l_k$ being the auxiliary literals associated with the infeasible constraints. The SAT problem is then augmented and new SAT queries are performed until either the SAT solver terminates with UNSAT or the theory solver with SAT. To benefit from the most recent advances in SAT solving, MiniSAT2 [22] is adapted to our requirements by adding decision heuristics to prune our search space. To reduce the number of theory calls, we first assign values to the Boolean variables so as to satisfy as many clauses as possible. Subsequently, we start assigning values to some of the auxiliary variables, until all clauses are satisfied. Whenever we need to decide an assignment for an auxiliary variable, we affirm any CC and negate any RCC as a first choice, to maximize the number of CCs for each theory call, hence the chances of deciding without approximations. The following theorems state the properties of CalCS.

Theorem V.1. *Let ϕ be a convex SMT formula. Then, if CalCS reports* SAT *on ϕ, ϕ is satisfiable. Alternatively, if CalCS reports* UNSAT*, ϕ is unsatisfiable.* □

Note that the converse does not hold in general. If CalCS reports UNKNOWN, it is possible that the formula ϕ is either satisfiable or unsatisfiable. In the case of a monotone convex SMT formula, we have stronger guarantees.

Theorem V.2. *Let ϕ^+ be a monotone convex SMT formula. Then, CalCS reports* SAT *on ϕ^+ iff ϕ^+ is satisfiable and CalCS reports* UNSAT *iff ϕ^+ is unsatisfiable.* □

The above result follows straightforwardly from the fact that for monotone convex SMT formulas, all convex constraints are assigned true, so the theory solver never sees non-convex constraints.

VI. Experimental Results

In our prototype implementation, we use the Matlab-based convex programming package CVX [23] to solve the optimization problems, while theory solver and SAT solver interact via an external file I/O interface. We therefore allow for all functions and operations supported by disciplined convex programming [24]. We first validated our approach on a set of benchmarks [25], including geometric decision problems dealing with the intersection of n-dimensional geometric objects, and randomly generated formulae obtained from 3-SAT classical Boolean benchmarks [26], after replacing some of the Boolean variables with convex or RC constraints. Table I shows a summary of an experimental evaluation of our tool, also in comparison with iSAT. To evaluate the impact of generating a compact explanation of unsatisfiability (a certificate) we run CalCS in two modes: in the first mode (C in Table I), a subset of conflicting constraints is provided, as detailed in Section IV, while in the second mode (NC in Table I), the full set of constraints is returned as simply being inconsistent. All benchmarks were performed on a 3 GHz Intel Xeon machine with 2 GByte physical memory running Linux.

Results show that whenever problems are purely convex, they are solved without approximation and with full control of rounding errors and can provide results that are more accurate than the ones of iSAT, in comparable time, in spite of our

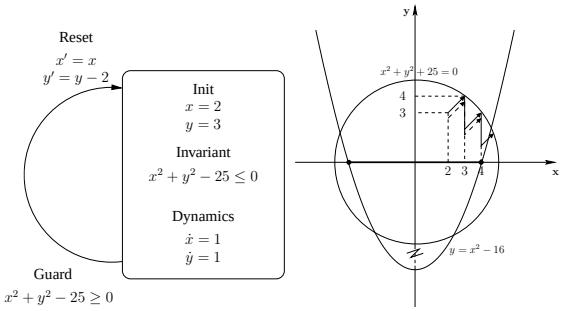


Fig. 4. Simple hybrid automata with convex guards and invariants (left) and representation of the error traces from CalCS (solid) and iSAT (dashed) in the (x, y) plane (right). The safety interval for x is $[-4, 4]$.

prototype implementation. In particular, the interval-based reasoning scheme can incur inaccuracies and large computation times when the satisfying sets are lower dimensional sets with respect to the full search space including all the real variables in the problems. As a simple example, for the formula:

$$(x_1^2 + x_2^2 - 1 \leq 0) \wedge (x_1^2 + x_2^2 - 6x_1 + 5 < 0), \qquad (8)$$

iSAT returns an interval that contains a spurious solution, while our convex sub-theory can rigorously deal with tight inequalities and correctly returns UNSAT (see (8) and Conj3 in Tab. I). Similarly, CalCS can provide the correct answer for the following formulae ((9) and (10) in Tab. I), mentioned as prone to unsound or spurious results in [12]:

$$(x+y < a) \wedge (x-y < b) \wedge (2x > a+b) \wedge (a = 1) \wedge (b = 0.1), \qquad (9)$$

$$(x \leq 10^9) \wedge (x + p > 10^9) \wedge (p = 10^{-8}). \qquad (10)$$

While for small problem instances (Bool1-2-3, Conj1) both the C and NC schemes show similar performances, the advantages of providing succinct certificates becomes evident for larger instances (Bool4-5-6-7, Conj2), where we rapidly reached a time-over (TO) limit (set to 200 queries to the theory solver) without certificates. A faster implementation would be possible by using commercial, or more optimized, convex optimization engines.

We have also tested CalCS on BMC problems, consisting in proving a property of a hybrid discrete-continuous dynamic system for a fixed unwinding depth k. We generated a set of hybrid automata (HA) including convex constraints in both their guards and invariants. For the simple HA in Fig. 4 we also report a pictorial view of the safety region for the x variable, and the error traces produced by CalCS (solid line) and iSAT (dashed line). The circle in Fig. 4 represents the HA invariant set, while the portion of the parabola underlying the x axis determines the set of points x satisfying the property we want to verify, i.e. $\{x \in \mathbb{R} : x^2 - 16 \leq 0\}$. Our safety region is therefore the closed interval $[-4, 4]$. The dynamics of the HA are represented by the solid and dash lines. As far as the invariant is satisfied, the continuous dynamics hold and the HA moves along the arrows on the (x, y) plane, starting from the point $(2, 3)$. When the trajectories intersect the circle's boundary, a jump occurs (e.g. from $(3, 4)$ to $(3, 2)$ and from $(4, 3)$ to $(4, 1)$) and the system is reset. Initially, both the solid and dashed trajectories are overlapped (they are drawn slightly apart for clarity). However, more accurately, we return unsafe

TABLE I
CALCS EXPERIMENTS: IN MODE C THE UNSAT CORE IS PROVIDED WHILE IN MODE NC THE FULL SET OF CONSTRAINTS IS RETURNED AS CONFLICTING; APPROX DENOTES THE NUMBER OF RCCS APPROXIMATED AS HYPERPLANES; S STANDS FOR SAT, U FOR UNSAT.

File	Res.	CalCS C/NC [s]	Approx C/NC	Queries C/NC	iSAT [s]
(8)	U	0.5 (U)	0	1	0.05 (S)
(9)	U	0.2 (U)	0	1	0 (S)
Conj3	U	22/23 (U)	5	3	0.05 (S)
(10)	S	0.2 (S)	0	1	0 (U)
Bool1	S	3.5 (S)	1	1	8 (S)
Bool2	S	16 (S)	3	1	0.91 (S)
Bool3	S	27/23 (S)	5/4	2	0.76 (S)
Conj1	U	8.7/9.5 (U)	3	2	0.3 (U)
Bool4	S	17.9/17.7 (S)	3	1	0.75 (S)
Conj2	U	17/23.3 (U)	4/5	4/7	0.4 (U)
Bool5	U	23.5/321.7 (U)	4/36	5/94	0.02 (U)
Bool6	U	29.8/TO (U)	5/−	6/−	0.4 (U)
Bool7	S	257.7/TO (S)	24/−	6/−	1.31 (S)

TABLE II
TCAS BMC CASE STUDY

Maneuver type	Crash state	#queries	run time [s]
UNSAFE	CRUISE	2	10.9
UNSAFE	LEFT	4	28
UNSAFE	STRAIGHT	6	50
SAFE	NONE	10	110

after 3 BMC steps ($k = 3$), while iSAT stops at the second step producing an error trace that is still in the safety region, albeit on the edge. As an additional case study, we considered aircraft conflict resolution [27] based on the Air Traffic Alert and Collision Avoidance System (TCAS) specifications (Tab. II). The hybrid automata in Fig. 5 models a standardized maneuver that two airplanes need to follow when they come close to each other during their flight. When the airplanes are closer than a distance d_{near}, they both turn left by $\Delta\phi$ degrees (which is kept fixed to a constant value in our maneuver) and fly for a distance d along the new direction. Then they turn right and fly until their distance exceeds a threshold d_{far}. At this point, the conflict is solved and the two airplanes can return on their original route. We verified that the two airplanes stay always apart, even without coordinating their maneuver with the help of a central unit.

Finally, we have applied CalCS to formulae generated in the context of static analysis of floating-point numerical code, and requiring an SMT solver that can handle non-linear arithmetic constraints over the reals. Tab. III summarizes the performance of CalCS on a set of benchmarks provided by Vo *et al.*, who are developing a static analyzer to detect floating-point exceptions (e.g., overflow and underflow) [28]. Early experience with CalCS on this set of benchmarks, mostly including conjunctions of linear and non-linear constraints, seems promising. After a fast pre-processing step, CalCS can deal with the formulae of interest providing an exact answer in reasonable computation time even when approximations are needed, which demonstrates that our solver can be general enough to be suitable for different application domains.

VII. CONCLUSIONS

We have proposed a procedure for satisfiability solving of a Boolean combination of non-linear constraints that are convex. Our prototype SMT solver, CalCS, combines fundamental

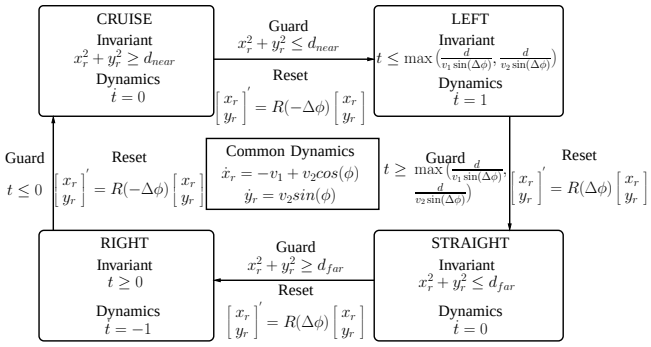


Fig. 5. Air Traffic Alert and Collision Avoidance System

TABLE III
BENCHMARKS FROM STATIC ANALYSIS OF NUMERICAL CODE: APPROX DENOTES THE NUMBER OF RCCS APPROXIMATED AS HYPERPLANES.

File	Result	Time [s]	Approx
Num1	SAT	1.08	0
Num2	SAT	4.35	2
Num3	UNSAT	0.55	0
Num4	UNSAT	0.55	0
Num5	SAT	4.27	2
Num6	SAT	0.49	0
Num7	SAT	2.82	1
Num8	UNSAT	2.64	2
Num9	UNSAT	2.10	0
Num10	UNSAT	0.53	0
Num11 − 13	UNSAT	0	0
Num14	UNSAT	1.91	0
Num15	UNSAT	1.94	0
Num16	UNSAT	0.53	0
Num17 − 18	UNSAT	0	0
Num19	UNSAT	0.49	0
Num20	UNSAT	0	0

results from convex programming with the efficiency of SAT solving. By restricting our domain to a subset of non-linear constraints, we can solve for conjunctions of constraints globally and accurately, by formulating a combination of convex optimization problems and exploiting information from their primal and dual optimal values. When the conjunction of theory predicates is infeasible, our formulation can generate certificates of unsatisfiability, thus enabling conflict-directed learning. Finally, whenever non-convex constraints originate from convex constraints due to Boolean negation, our procedure uses geometric properties of convex sets to generate conservative approximations of the original set of constraints. Experiments on several benchmarks, including examples of BMC for hybrid systems, show that CalCS can be more accurate than other state-of-the-art non-linear SMT solvers. In the future, we plan to further refine the proposed algorithms by devising more sophisticated learning and approximation schemes as well as more efficient implementations.

ACKNOWLEDGMENTS

The authors acknowledge the support of the Gigascale Systems Research Center and the Multiscale Systems Center, two of six research centers funded under the Focus Center Research Program, a Semiconductor Research Corporation program. The third author is also supported in part by the NSF grant CNS-0644436, and by an Alfred P. Sloan Research Fellowship. Comments of Susmit Jha on drafts of this paper are also acknowledged.

REFERENCES

[1] D. Walter, S. Little, and C. Myers, "Bounded model checking of analog and mixed-signal circuits using an SMT solver," in *Automated Technology for Verification and Analysis*, 2007, pp. 66–81.
[2] C. Tomlin, I. Mitchell, and R. Ghosh, "Safety verification of conflict resolution maneuvers," *IEEE Trans. Intell. Transp. Syst.*, vol. 2, no. 2, pp. 110–120, Jun. 2001.
[3] M. Franzle and C. Herde, "HySAT: An efficient proof engine for bounded model checking of hybrid systems," *Formal Methods in System Design*, pp. 179–198, 2007.
[4] S. Jha, B. Brady, and S. A. Seshia, "Symbolic reachability analysis of lazy linear hybrid automata," in *5th International Conference on Formal Modeling and Analysis of Timed Systems (FORMATS)*, October 2007, pp. 241–256.
[5] C. Barrett, R. Sebastiani, S. A. Seshia, and C. Tinelli, *Satisfiability Modulo Theories, Chapter in Handbook of Satisfiability*. IOS Press, 2009.
[6] S. Ratschan, "Efficient solving of quantified inequality constraints over the real numbers," *ACM Trans. Comput. Logic*, vol. 7, no. 4, pp. 723–748, 2006.
[7] 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*, vol. 53, no. 6, pp. 937–977, 2006.
[8] S. Boyd and L. Vandenberghe, *Convex Optimization*. Cambridge Univesity Press, 2004.
[9] M. Franzle, C. Herde, S. Ratschan, T. Schubert, and T. Teige, "Efficient solving of large non-linear arithmetic constraint systems with complex boolean structure," in *JSAT Special Issue on SAT/CP Integration*, 2007, pp. 209–236.
[10] A. Bauer, M. Pister, and M. Tautschnig, "Tool-support for the analysis of hybrid systems and models," in *Proc. of DATE*, 2007.
[11] "IPOPT," https://projects.coin-or.org/Ipopt.
[12] M. Ganai and F. Ivancic, "Efficient decision procedure for non-linear arithmetic constraints using CORDIC," in *Formal Methods in Computer-Aided Design, 2009. FMCAD 2009*, 15-18 2009, pp. 61–68.
[13] J. P. Fishburn and A. E. Dunlop, "TILOS: A posynomial programming approach to transistor sizing," in *IEEE International Conference on Computer- Aided Design: ICCAD-85*, 15-18 1985, pp. 326–328.
[14] S. S. Sapatnekar, V. B. Rao, P. M. Vaidya, and S.-M. Kang, "An exact solution to the transistor sizing problem for CMOS circuits using convex optimization," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 12, no. 11, pp. 1621–1634, 1993.
[15] M. del Mar Hershenson, S. P. Boyd, and T. H. Lee, "Optimal design of a CMOS op-amp via geometric programming," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 20, no. 1, pp. 1–21, 2001.
[16] X. Li, P. Gopalakrishnan, Y. Xu, and L. Pileggi, "Robust Analog/RF Design with Projection-Based Posynomial Modeling," in *Proc. IEEE/ACM International Conference on CAD*, 2004, pp. 855–862.
[17] Y. Xu, K. Hsiung, X. Li, I. Nuasieda, S. Boyd, and L. Pileggi, "OPERA: OPtimization with Ellispoidal uncertainty for Robust Analog IC design," 2005, pp. 632–637.
[18] A. Bemporad and N. Giorgetti, "A SAT-based hybrid solver for optimal control of hybrid systems," in *Hybrid Systems: Computation and Control*, 2004.
[19] ——, "Logic-based solution methods for optimal control of hybrid systems," *IEEE Transactions on Automatic Control*, vol. 51, no. 6, pp. 963–976, Jun. 2006.
[20] O. L. Mangasarian, "Set containment characterization," *J. of Global Optimization*, vol. 24, pp. 473–480, 2002.
[21] V. Jeyakumar, "Characterizing set containments involving infinite convex constraints and reverse-convex constraints," *SIAM J. Optimization*, vol. 13, pp. 947–959, 2003.
[22] "MiniSAT SAT solver," http://minisat.se.
[23] M. Grant and S. Boyd, "CVX: Matlab software for disciplined convex programming, version 1.21," http://cvxr.com/cvx, May 2010.
[24] M. Grant, S. Boyd, and Y. Ye, "Disciplined convex programming," in *Global Optimization: From Theory to Implementation*, 2006, pp. 155–210.
[25] "Benchmarks," http://www.eecs.berkeley.edu/∼nuzzo/Research.html.
[26] "3-sat," http://people.cs.ubc.ca/∼hoos/SATLIB/benchm.html.
[27] C. Tomlin, G. Pappas, and S. Sastry, "Conflict resolution for air traffic management: A study in multi-agent hybrid systems," *IEEE Transactions on Automatic Control*, vol. 43, no. 4, pp. 110–120, Apr. 1998.
[28] T. Vo, E. Barr, and Z. Su, personal communication.

Integrating ICP and LRA Solvers for Deciding Nonlinear Real Arithmetic Problems

Sicun Gao[1,2], Malay Ganai[1], Franjo Ivančić[1], Aarti Gupta[1], Sriram Sankaranarayanan[3], and Edmund M. Clarke[2]
1 NEC Labs America, NJ, USA 2 Carnegie Mellon University, PA, USA 3 University of Colorado Boulder, CO, USA

***Abstract*—We propose a novel integration of interval constraint propagation (ICP) with SMT solvers for linear real arithmetic (LRA) to decide nonlinear real arithmetic problems. We use ICP to search for interval solutions of the nonlinear constraints, and use the LRA solver to either validate the solutions or provide constraints to incrementally refine the search space for ICP. This serves the goal of separating the linear and nonlinear solving stages, and we show that the proposed methods preserve the correctness guarantees of ICP. Experimental results show that such separation is useful for enhancing efficiency.**

I. Introduction

Formal verification of embedded software and hybrid systems often requires deciding satisfiability of quantifier-free first-order formulas involving real number arithmetic. While highly efficient algorithms [10] exist for deciding linear real arithmetic (QFLRA problems, as named in SMT-LIB [5]), nonlinear formulas (QFNRA problems [5]) have been a major obstacle in the scalable verification of realistic systems. Existing complete algorithms have very high complexity for nonlinear formulas with polynomial functions (double-exponential lower bound [9]). Formulas containing transcendental functions are in general undecidable. It is thus important to find alternative practical solving techniques for which the completeness requirement may be relaxed to some extent ([11], [12], [20], [8]).

Interval Constraint Propagation (ICP) is an efficient numerical method for finding interval over-approximations of solution sets of nonlinear real equality and inequality systems ([15], [6]). For solving QFNRA formulas in a DPLL(T) framework, ICP can be used as the theory solver that provides decisions on conjunctions of theory atoms. What distinguishes ICP from other numerical solution-finding algorithms (such as Newton-Raphson or convex optimization) is that it guarantees the following reliability properties:

- ICP always terminates, returning either "unsatisfiable", or "satisfiable" with an interval overapproximation of a solution (or the solution set).
- When ICP returns an "unsatisfiable" decision, it is always correct.
- When ICP returns a "satisfiable" decision, the solution may be spurious; but its error is always within a given bound that can be set very small.

A detailed discussion of what these correctness guarantees of ICP imply for decision problems is given in Section IV. These properties ensure that an ICP solver only relaxes completeness *moderately* (see "δ-completeness", Section IV), while achieving efficiency. ICP algorithms have been applied to various nonlinear scientific computing problems involving thousands of variables and constraints (including transcendental functions) ([17], [18], [6]).

The HySAT/iSAT solver [11] is a state-of-the-art SMT solver for QFNRA problems. HySAT uses ICP for handling nonlinear real constraints. It carefully builds Boolean solving capacities into ICP by exploiting the similarity between SAT and interval constraint solving algorithms. HySAT successfully solved many challenging nonlinear benchmarks that arise from hybrid system verification problems [3].

However, a problem with HySAT is that it handles both linear and nonlinear constraints with ICP. It is known that ICP does not solve linear constraints efficiently enough. In fact, ICP can suffer from the "slow convergence" problem [7] on easy linear constraints such as "$x \geq y \wedge x \leq y$", where it needs a large number of iteration steps to return an answer. As there exist highly optimized algorithms for deciding linear arithmetic problems [10], solving all the constraints in ICP is suboptimal. Most practical formal verification problems contain a large number of linear and Boolean constraints, and only a small number of nonlinear ones. Ideally, we would like to solve linear and nonlinear constraints differently, and apply the efficient algorithms for linear constraints as much as possible.

Such separation of linear and nonlinear solving is not straightforward to design. In fact, it is suggested as an open question in the original HySAT paper [11]. There are several difficulties involved:

- The linear and nonlinear constraints share many variables in nontrivial problems. For the same variable, the linear solver returns point solutions while ICP returns interval solutions. It is not straightforward to check consistency between the different solutions.
- As both the linear solver and the nonlinear solver return only one solution (point or interval box) at a time, it is impossible to enumerate all the solutions in one solver and validate them in the other solver, since there are usually infinitely many solutions.
- Linear solvers use rational arithmetic and ICP uses floating point arithmetic. Efficient passing of values between the two solvers can compromise the guaranteed numerical error bounds in ICP. (See Example 2).

In this paper, we propose methods that tackle these problems. The main idea is to design an "abstraction refinement" loop between the linear and nonlinear solving stages: We use

the ICP solver to search for interval solutions of the nonlinear constraints, and use the LRA solver to validate the solutions and incrementally provide more constraints to the ICP solver for refining the search space. The difficulty lies in devising procedures that efficiently communicate solutions between the linear and nonlinear solving stages without compromising numerical correctness guarantees. Our main contributions are:

- We devise procedures that separate linear and nonlinear solving in a DPLL(T) framework to enhance efficiency in solving QFNRA problems.
- We give precise definitions of correctness guarantees of ICP procedures, named δ-completeness, as used in decision problems. We show that the devised separation between linear and nonlinear solving preserves such correctness guarantees.
- We describe how to exploit ICP in assertion and learning procedures in DPLL(T) to further enhance efficiency.

The paper is organized as follows: In Section II we briefly review ICP, DPLL(T), and LRA solvers; in Section III, we show the detailed design of the checking procedures; in Section IV, we discuss correctness guarantees of ICP in decision problems; in Section V, we further describe the design of the assertion and learning procedures. We show experimental results and conclusions in Section VI and VII.

II. Background

A. Interval Constraint Propagation

The method of ICP ([15], [6]) combines interval analysis and constraint solving techniques for solving systems of real equalities and inequalities. Given a set of real constraints and interval bounds on their variables, ICP successively refines an interval over-approximation of its solution set by narrowing down the possible value ranges for each variable. ICP either detects the unsatisfiability of a constraint set when the interval assignment on some variable is narrowed to the empty set, or returns interval assignments for the variables that tightly over-approximate the solution set, satisfying some preset precision requirement. (See Fig 1.) We will only be concerned with elementary real arithmetic in this paper. We first use a simple example to show how ICP works.

Example 1. *Consider the constraint set* $\{x = y, y = x^2\}$.

i) Suppose $I_0^x = [1, 4], I_0^y = [1, 5]$ *are the initial intervals for* x *and* y*. ICP approaches the solution to the constraint set in the following way:*

Step 1. Since the initial interval of y *is* $I_0^y = [1, 5]$*, to satisfy the constraint* $y = x^2$*, the value of* x *has to lie within the range of* $\pm\sqrt{I_0^y}$*, which is* $[-\sqrt{5}, -1] \cup [1, \sqrt{5}]$*. Taking the intersection of* $[-\sqrt{5}, -1] \cup [1, \sqrt{5}]$ *and the initial interval* $[1, 4]$ *on* x*, we can narrow down the interval of* x *to* $I_1^x = [1, \sqrt{5}]$*;*

Step 2. Given $I_1^x = [1, \sqrt{5}]$ *and the constraint* $x = y$*, the interval on* y *can not be wider than* $[1, \sqrt{5}]$*. That gives* $I_1^y = I_0^y \cap [1, \sqrt{5}] = [1, \sqrt{5}]$*;*

Step 3. Given I_1^y*, we can further narrow down the interval on* x*, by maintaining its consistency with* $x = \pm\sqrt{y}$*, and obtain* $I_2^x = I_0^x \cap \sqrt{I_1^y} = [1, \sqrt[4]{5}]$.

Iterating this process, we have two sequences of intervals that approach the exact solution $x = 1, y = 1$*:*

$I^x : [1, 4] \to [1, \sqrt{5}] \to [1, \sqrt[4]{5}] \to [1, \sqrt[8]{5}] \to \cdots \to [1, 1]$

$I^y : [1, 5] \to [1, \sqrt{5}] \to [1, \sqrt[4]{5}] \to [1, \sqrt[8]{5}] \to \cdots \to [1, 1]$

ii) On the other hand, ICP detects unsatisfiability of the constraint set over intervals $I_0^x = [1.5, 4]$ *and* $I_0^y = [1, 4]$ *easily:*

$I^x : [1.5, 4] \to [1.5, 4] \cap [1, \sqrt{4}] \to [1.5, 2] \to [1.5, 2] \cap [\sqrt{1.5}, \sqrt{2}] \to \emptyset$

$I^y : [1, 4] \to [1, 4] \cap [1.5, 2] \to [1.5, 2] \to [1.5, 2] \cap \emptyset \to \emptyset$

Note that ICP implements floating point arithmetic, therefore all the irrational boundaries are relaxed by decimal numbers in practice. □

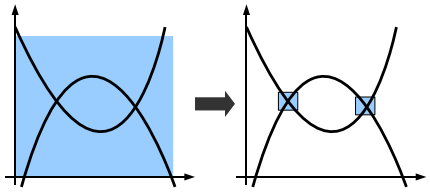


Fig. 1: Contraction of initial intervals to solution boxes

During the interval narrowing process, ICP can reach a fixed-point before the precision requirement is satisfied. In that case, ICP takes a *splitting step*, and recursively contracts the sub-intervals. This framework for solving nonlinear constraints is called the *branch-and-prune* approach [15].

We give the following formal definitions that will be referred to in the following sections. Let n be the number of variables and $\mathcal{I} = \{[a, b] : a, b \in \mathbb{R}\}$ the set of all intervals over $\mathbb{R}$. An n-ary constraint σ is a relation defined by equalities and inequalities over $\mathbb{R}$, i.e., $\sigma \subseteq \mathbb{R}^n$.

Definition 1. *Let* $\sigma \subseteq \mathbb{R}^n$ *be a constraint,* $\vec{I} \in \mathcal{I}^n$ *an interval vector whose* i*-th projection is written as* I_i*, i.e.,* $\vec{I} = \langle I_1, ..., I_n \rangle$*. We say* $\vec{I}$ ***over-approximates*** σ*, if for all* $(a_1, ..., a_n) \in \sigma$*,* $a_i \in I_i$*.*

Definition 2. *An* ***interval contractor*** $\natural : \mathcal{I}^n \to \mathcal{I}^n$ *is a function satisfying* $\natural\vec{I} \subseteq \vec{I}$*. The result of multiple applications of an interval contractor on* $\vec{I}$ *is written as* $\natural^*\vec{I}$*. A* ***contraction sequence*** *is a sequence of intervals* $S = (\vec{I_1}, ..., \vec{I_n})$ *where* $\vec{I}_{i+1} = \natural\vec{I_i}$*. A* ***contraction step*** *in S is defined as* $(\vec{I}, \natural\vec{I})$ *where* $\vec{I} = \langle I_1, ..., I_i, ..., I_n \rangle$*,* $\natural\vec{I} = \langle I_1, ..., \natural I_i, ..., I_n \rangle$ *and*

$$\natural I_i = I_i \cap F(I_1, ..., I_{i-1}, I_{i+1}, ..., I_n).$$

$F : \mathcal{I}^{n-1} \to \mathcal{I}$ *is an interval-arithmetic function whose graph over-approximates the constraint* σ*.*

Definition 3. *A* ***consistency condition*** $\mathcal{C} \subseteq \mathcal{I}^n \times \mathcal{I}^n$ *satisfies: for any constraint* $\sigma \subseteq \mathbb{R}^n$*, if* $\vec{I}$ *over-approximates* σ *and* $(\vec{I}, \natural\vec{I}) \in \mathcal{C}$*, then* $\natural\vec{I}$ *over-approximates* σ*.*

B. DPLL(T) and the Dutertre-de Moura algorithm

An SMT problem is a quantifier-free first-order formula φ with atomic formulas specified by some theory T. Most current SMT solvers use the DPLL(T) framework [19]. A DPLL(T)-based solver first uses a SAT solver on the Boolean abstraction φ^B of the formula φ. If φ^B is satisfiable, a theory solver (T-solver) is used to check whether the Boolean assignments correspond to a consistent set of *asserted theory atoms*. The T-solver should implement the following procedures:

Check() and Assert(): The Check() procedure provides the main utility of a T-solver. It takes a set of theory atoms and returns a "satisfiable"/"unsatisfiable" answer, depending on whether the set is consistent with respect to the theory T. The Assert() procedure provides a partial check for detecting early conflicts.

Learn() and Backtrack(): When the Check() or Assert() procedure detects inconsistency in a set of theory atoms, the T-solver provides *explanations* through the Learn() procedure, so that a clause can be learned for refining the search space. When inconsistency occurs, the T-solver performs efficient backtracking on the theory atoms in Backtrack().

LRA Solvers: The standard efficient algorithm for solving SMT problems with linear real arithmetic is proposed in [10], which we will refer to as the Dutertre-de Moura Algorithm. The algorithm optimizes the Simplex method for solving SMT problems by maintaining a fixed matrix for all the linear constraints so that all the operations can be conducted on simple bounds on variables. In what follows we assume that the LRA solver implements the Dutertre-de Moura algorithm.

C. Formula Preprocessing

We consider quantifier-free formulas over $\langle \mathbb{R}, \leq, +, \times \rangle$. The atomic formulas are of the form $p_i \sim c_i$, where $\sim \in \{<, \leq, >, \geq, =\}$, $c_i \in \mathbb{R}$ and p_i is a polynomial in $\mathbb{R}[\vec{x}]$.

Adopting similar preprocessing techniques as in [10], we preprocess input formulas so that a fixed set of constraints can be maintained such that the DPLL search can be done only on simple atoms of the form $x \sim c$. For any input formula, we introduce two sets of auxiliary variables: a set of nonlinear variables and a set of slack variables.

A nonlinear variable v_i is introduced when a nonlinear term t_i appears for the first time in the formula. We replace t_i by v_i and add an additional atomic formula $(t_i = v_i)$ to the original formula as a new clause.

Similarly, a slack variable s_i is introduced for each atomic formula $p_i \sim c_i$, where p_i is not a single variable. We replace p_i by s_i, and add $(p_i = s_i)$ to the original formula.

For instance, consider

$$\varphi \equiv_{df} ((x^2 + y \geq 10 \wedge x \cdot z < 5) \vee y + z > 0).$$

We introduce nonlinear and slack variables to get:

$$\underbrace{(x^2 = v_1 \wedge x \cdot z = v_2)}_{N_\varphi} \wedge \underbrace{(v_1 + y = s_1 \wedge y + z = s_2)}_{L_\varphi} \wedge \underbrace{((s_1 \geq 10 \wedge v_2 < 5) \vee s_2 > 0)}_{\varphi'}$$

The new formula is equi-satisfiable with the original formula. In general, after such preprocessing, any input formula φ is put into the following normal form:

$$\varphi \equiv \underbrace{\bigwedge_{i=1}^{n} \nu_i}_{N_\varphi} \wedge \underbrace{\bigwedge_{i=1}^{m} \mu_i}_{L_\varphi} \wedge \underbrace{\bigwedge_{j=1}^{p} (\bigvee_{i=1}^{q} l_{j_i})}_{\varphi'}.$$

The following notations will be used throughout the paper:

1. $V = \{x_1, ..., x_k\}$ denotes the set of all the variables appearing in φ. The set of variables appearing in any subformula ψ of φ is written as $V(\psi)$. In particular, write $V_N = \bigcup_i V(\nu_i)$ and $V_L = \bigcup_i V(\mu_i)$.

2. In N_φ, each atom ν_i is of the form $x_{i_0} = f_i(x_{i_1}, ..., x_{i_r})$ where $x_{i_j} \in V$. Note that x_{i_0} is the introduced nonlinear variable. f_i is a nonlinear function that does *not* contain addition/subtraction. We call N_φ the *nonlinear table* of φ.

3. In L_φ, each atom μ_i is of the form $\sum a_{i_j} x_{i_j} = 0$, where $a_{i_j} \in \mathbb{R}$ and $x_{i_j} \in V$. L_φ is called the *matrix* of the formula following [10].

4. In φ', each literal l_i is of the form $(x_j \sim c_i)$ or $\neg(x_j \sim c_i)$, where $x_j \in V$, $c_i \in \mathbb{R}$ and $\sim \in \{>, \geq, =\}$. The original Boolean structure in φ is now contained in φ'.

All the ν_is and μ_is are called *constraints* (nonlinear or linear, respectively), and $N_\varphi \wedge L_\varphi$ is called the *extended matrix* of the formula.

III. Interfacing Linear and Nonlinear Solving in the Check() Procedure

A. The Main Steps

As introduced in Section II-B, the Check() procedure provides the main utility of the theory solver in the DPLL(T) framework. It takes a set of asserted theory atoms and returns whether their conjunction is satisfiable in the theory T.

An intuitive way of separating linear and nonlinear solving is to have the following two stages:

1. The linear constraints are first checked for feasibility, so that linear conflicts can be detected early.

2. If no linear conflict arises, the nonlinear solver is invoked to check whether the nonlinear constraints are satisfiable *within the feasible region* defined by the linear constraints.

However, the difficulty lies in starting the second step. For checking linear feasibility, the LRA solver maintains only one point-solution throughout the solving process. That is, it stores and updates a rational number for each variable. To obtain the linear feasible region, extra computation is needed. A direct way is to use the optimization phase of the Simplex algorithm and collect optimal bounds of linear variables (their min/max values), which are used as the initial interval assignments for ICP. However, this is problematic for several reasons:

- Obtaining bounds on each variable requires solving two optimization problems involving *all* the linear constraints for *every* variable. This leads to heavy overhead.
- More importantly, the bounds on variables only constitute a box over-approximation of the linear feasible region.

After obtaining a nonlinear solution within this over-approximation, we still need to check whether this solution resides in the real feasible region. (See Fig. 2)

- Numerical errors in the optimization procedures are introduced in the decision procedure. They can compromise the correctness guarantees of ICP.

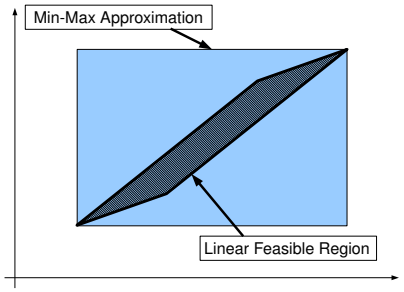


Fig. 2: Box approximations can be too coarse.

Consequently, we need more subtle interaction procedures between the linear and nonlinear solving stages.

We write the set of asserted theory atoms as Λ, i.e.,

$$\Lambda \subseteq \{x_i \sim c_i : x_i \sim c_i \text{ is a theory atom in } \varphi'\},$$

where φ' is as defined in the preprocessing step. Our Check() procedure (Fig. 3) consists of the following main steps:

Step 1. Check Linear Feasibility. (Line 2 in Fig. 3)

First, we use the LRA solver to check the satisfiability of the linear formula $L_\varphi \wedge \bigwedge \Lambda$. If the formula is unsatisfiable, there is a conflict in Λ with respect to the matrix L_φ, and Check() directly returns "*unsatisfiable*".

Step 2. Check Nonlinear Feasibility. (Line 4 in Fig. 3)

If the linear constraints are consistent, we start ICP directly on the set of nonlinear constraints; i.e., we check the satisfiability of the formula $N_\varphi \wedge \bigwedge \Lambda$. Note that after the linear solving phase in Step 1, the bounds on linear variables in Λ are already partially refined by the LRA solver [10] and we update Λ with the refined bounds. (Line 3 in Fig. 3)

If ICP determines that the nonlinear constraints are inconsistent over the initial intervals specified by Λ, the solver directly returns "*unsatisfiable*".

Step 3. Validate Interval Solutions. (L6 in Fig. 3; Fig. 4)

If ICP determines that the formula in Step 2 is satisfiable, it returns a vector $\vec{I}$ of interval assignments for all variables in V_N. Since we did not perform nonlinear checking within the linear feasible region, it is possible that the interval assignments for V_N are inconsistent with the matrix L_φ. Thus, we need to *validate* the interval solutions $\vec{I}$ with respect to the linear constraints L_φ. This validation step requires reasoning about the geometric properties of the interval solutions and linear feasibility region defined by $L_\varphi \wedge \bigwedge \Lambda$. We give detailed procedures for the validation step in Section III-B.

Step 4. Add Linear Constraints to ICP. (L7-10 in Fig. 3)

If in the previous step an interval solution $\vec{I}$ is not validated by the linear constraints, we obtain a set Σ of linear constraints (specified in Section III-B) that are violated by $\vec{I}$. Now we restart ICP and look for another solution that can in fact satisfy the linear constraints in Σ, by setting $N_\varphi := N_\varphi \wedge \bigwedge \Sigma$ and loop back to Step 2. This is further explained in Section III-C.

In this way, we incrementally add linear constraints into the set of constraints considered by ICP to refine the search space. The loop terminates when ICP returns unsatisfiable on N_φ because of the newly added linear constraints, or when the LRA solver successfully validates an interval solution.

Next, we give the detailed procedures for the validation steps.

```
1: Procedure Check(L_φ, N_φ, Λ)
2: if Linear_Feasible(L_φ ∧ ⋀Λ) then
3:    Λ ←Linear_Refine(L_φ ∧ ⋀Λ)
4:    while ICP_Feasible(N_φ ∧ ⋀Λ) do
5:       Ī ← ICP_Solution(N_φ ∧ ⋀Λ)
6:       Σ ← Validate(Ī, L_φ, Λ)
7:       if N_φ == N_φ ∪ Σ then
8:          return satisfiable
9:       else
10:         N_φ ← N_φ ∪ Σ
11:      end if
12:   end while
13: end if
14: return unsatisfiable
```

Fig. 3: Procedure Check()

```
1: Procedure Validate(Ī = ⟨l⃗, u⃗⟩, L_φ, Λ)
2: if Linear_Feasible(⋀(x_i = (l_i+u_i)/2) ∧ L_φ ∧ ⋀Λ) then
3:    y⃗ ← b⃗  /*LRA solver returns b⃗ as the solution of y⃗*/
4:    for μ : x⃗ ≤ e_j + d_j^T y⃗ ∈ L_φ do
5:       if c_j^T a⃗_j ≤ e_j + d_j^T b⃗ is false then
6:          /* See Proposition 1 for the definitions */
7:          Σ ← Σ ∪ μ
8:       end if
9:    end for
10: else
11:    Σ ← Linear_Learn(⋀(x_i = (l_i+u_i)/2) ∧ L_φ ∧ ⋀Λ)
12: end if
13: return Σ
```

Fig. 4: Procedure Validate()

B. The Validation Procedures

1) Relations between interval solutions and the linear feasible region: Geometrically, the interval solution returned by ICP forms a hyper-box whose dimension is the number of variables considered by ICP. The location of the hyper-box with respect to the linear feasible region determines whether the interval solution for the nonlinear constraints satisfies the linear constraints. There are three possible cases (see Fig. 5 for a two-dimensional illustration):

Case 1: (Box A in Fig. 5) The hyper-box does not intersect the linear feasible region. In this case, the interval solution returned by ICP does not satisfy the linear constraints.

Case 2: (Box B in Fig. 5) The hyper-box *partially* intersects the linear feasible region. In this case, the real solution of the nonlinear constraints contained in the solution box could either reside inside or outside the linear region.

Distinguishing this case is especially important when we take into account that the LRA solver uses precise rational arithmetic. The interval assignments returned by ICP satisfy certain precision requirements and usually have many decimal digits, which can only be represented as ratios of large integers in the LRA solver. Precise large number arithmetic is costly in the LRA solver. To efficiently validate the interval solutions, we need to truncate the decimal digits. This corresponds to a further overapproximation of the intervals. For example:

Example 2. *Consider* $(y = x^2) \wedge (y - x = s) \wedge (y \geq 2 \wedge x \geq 0 \wedge s \geq 0.6)$. *In Step 2, ICP solves the formula* $(y = x^2 \wedge y \geq 2 \wedge x \geq 0)$ *and returns a solution* $x \in [1.414213562373, 1.414213567742]$ *and* $y \in [2, 2.000000015186]$. *Its rational relaxation* $x \in [14/10, 15/10]$ *and* $y \in [2, 21/10]$ *is validated, since* $y - x \geq 0.6$ *is satisfied by* $x = 1.4, y = 2$. *But the original formula is unsatisfiable, which can in fact be detected if we use ICP on the nonlinear and linear constraints together.*

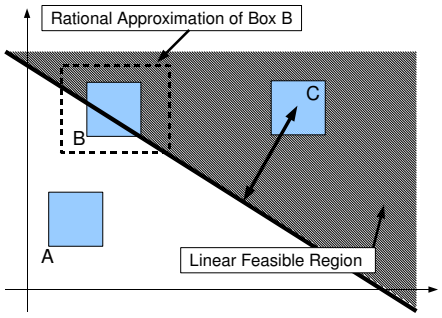


Fig. 5: Positions of hyper-boxes and the linear feasible region.

Case 3: (Box C in Fig. 5) The hyper-box completely resides in the linear feasible region. In this case, all the points in the interval solution returned by ICP satisfy the linear constraints and hence the formula should be *"satisfiable"*. To distinguish this case from Case 2, we propose the following consistency condition.

2) The Sufficient Consistency Check: We check whether *all* the points in $\vec{I}$ satisfy the linear constraints in L_φ. When that is the case, we say $\vec{I}$ is *consistent* with L_φ and accept the interval solution. This is a strong check that provides a *sufficient* condition for the existence of solutions. By enforcing it we may lose possible legitimate solutions (e.g., Box B may contain a solution that indeed resides in the linear region). This problem is handled in the refinement step (Section III-C).

We write variables contained in the nonlinear constraints as $V_N = \{x_1, ..., x_n\}$, and the variables that *only* occur in linear constraints as $V_L \setminus V_N = \{y_1, ..., y_m\}$.

Definition 4. *Let* $\vec{I} : \langle [l_1, u_1], \ldots, [l_n, u_n] \rangle$ *be an interval solution for variables in* V_N. *We write* $\vec{I} : [\vec{l}, \vec{u}]$, $\vec{x} = (x_1, ..., x_n)$, $\vec{y} = (y_1, ..., y_m)$. *We say* $\vec{I}$ *is* ***consistent*** *with the matrix* L_φ, *if*

$$\exists\, \vec{y}\, \forall\, \vec{x}\ \left[(\vec{x} \in \vec{I})\ \rightarrow\ (L_\varphi \wedge \bigwedge \Lambda)\right] \qquad \cdots(\star)$$

is true, where $\vec{x} \in \vec{I} =_{df} \bigwedge_{x_i \in V_N} (l_i \leq x_i \wedge x_i \leq u_i)$. *Note that* $L_\varphi \wedge \bigwedge \Lambda$ *is a formula in both* $\vec{x}$ *and* $\vec{y}$.

This condition states that, for an interval solution $\vec{I}$ to be consistent with the linear constraints, there must be a feasible point solution $\vec{b}$ for the remaining linear variables $\vec{y}$ such that for all the points $\vec{a} \in \vec{I}$, $(\vec{a}, \vec{b})$ satisfies the linear constraints. This is a direct formulation of Case 3.

3) The Validation Steps: We propose the following procedures for validating interval solution with the LRA solver (shown in Fig. 4).

Step 3.1. (Line 2-3, Fig. 4) First, we check whether the center of the hyper-box $\vec{I}$ resides in the linear feasible region (in fact it can be an arbitrary point of the box), by checking whether the linear formula:

$$\bigwedge_{x_i \in V_N} (x_i = \frac{l_i + u_i}{2}) \wedge L_\varphi \wedge \bigwedge \Lambda$$

is satisfiable. This can be done by the LRA solver.

If this formula is unsatisfiable, we know that Condition $(\star)$ is violated, since the center of $\vec{I}$ lies outside linear feasible region. We can obtain a set of violated linear constraints provided by the LRA solver (Line 11, Fig. 4). This is further explained in Section III-C.

If it is satisfiable, the Dutertre-de Moura Algorithm returns an exact point solution $\vec{b}$ for $\vec{y}$. (Line 3, Fig. 4)

Step 3.2. (Line 4-7, Fig. 4) Next we need to ensure that, after the remaining linear variables $\vec{y}$ are assigned $\vec{b}$, the interval box $\vec{I}$ resides "far away" from the boundaries of the linear feasible region.

Since $L_\varphi \wedge \bigwedge \Lambda$ only contains linear constraints, it can be written as the intersection of k half spaces:

$$L_\varphi \wedge \bigwedge \Lambda \equiv \bigwedge_{j=1}^{k} \vec{c}_j^{\mathsf{T}}\, \vec{x} \leq e_j + \vec{d}_j^{\mathsf{T}}\, \vec{y}.$$

where $\vec{c}_j = (c_{j1}, ..., c_{jn})$ and $\vec{d}_j = (d_{j1}, ..., d_{jm})$.

First, we make the observation that the maximum of each $\vec{c}_j^{\mathsf{T}}\, \vec{x}$ is obtained when the x variables take the min or max values in their intervals depending on their coefficients:

Lemma 1. *The solution to the linear program*

$$\mathsf{max}\ \vec{c}_j^{\mathsf{T}}\, \vec{x} \quad \textit{with respect to} \quad \vec{x} \in \vec{I} : [\vec{l}, \vec{u}]$$

is given by $\vec{x} = (a_1, ..., a_n)$, *where* $a_i = l_i$ *when* $c_{ji} \leq 0$ *and* $a_i = u_i$ *otherwise.*

Further, we know that the universal statement in the consistency condition is satisfied, if the max value of $\vec{c}^{\mathsf{T}}\, \vec{x}$ is bounded by the linear constraints $e_j + \vec{d}_j^{\mathsf{T}}\, \vec{y}$. That is:

Proposition 1. *The assertion*

$$\forall\, \vec{x}.\ ((\vec{x} \in \vec{I})\ \rightarrow\ \vec{c}_j^{\mathsf{T}}\, \vec{x} \leq e_j + \vec{d}_j^{\mathsf{T}}\, \vec{y})$$

holds for $\vec{y} = \vec{b}$ *iff*

$$\vec{c}_j^{\mathsf{T}}\, \vec{a}_j \leq e_j + \vec{d}_j^{\mathsf{T}}\, \vec{b},$$

wherein $\vec{a}_j = (a_{j1}, ..., a_{jn})$ *satisfying:* $a_{ji} = l_i$ *when* $c_{ji} \leq 0$*, and* $a_{ji} = u_i$ *otherwise.*

The condition in Proposition 1 can be verified by simple calculations: we only need to plug the values of $\vec{x} = \vec{a}_j$ and $\vec{y} = \vec{b}$ in each linear constraint, and check whether the constraint is satisfied.

To summarize, we use the LRA solver to search for a candidate solution $\vec{b}$ for the linear variables $\vec{y}$, and verify if the strong consistency condition $(\star)$ holds when $\vec{y} = \vec{b}$, using Proposition 1. If Condition $(\star)$ is verified, we return "satisfiable".

Again, Condition $(\star)$ and Proposition 1 provide a sufficient condition for the consistency of $\vec{I}$ and L_φ, which may refute legitimate solution boxes. This is compensated, because we use the strong condition to learn the violated linear constraints instead of directly refuting boxes. This is further explained in the next section.

C. Refinement of ICP Search Using Linear Constraints

In the validation steps, there are two places where we can detect that an interval solution has violated linear constraints:

- In Step 3.1, the linear formula is detected unsatisfiable by the LRA solver. In this case, we use the learning procedure in the LRA solver that returns a set of linear constraints.
- In Step 3.2, the condition in Proposition 1 can fail for a set of linear constraints. These are the constraints that the box solution does not completely satisfy.

In both cases, we have a set of linear constraints which we write as Σ. We then add Σ to N_φ and restart the ICP search on the updated N_φ. Now, the new interval solution obtained by the updated N_φ should not violate Σ modulo numerical errors in ICP, since it was obtained by ICP under the constraints in Σ.

Here, a tricky problem is that ICP allows numerical error (up to its precision bound). It is possible that even after Σ is added to ICP, the interval solution $\vec{I}$ that ICP returns may still violate Σ in terms of precise arithmetic. In such cases, the linear solver and the ICP solver disagree on the same set of constraints: Namely, ICP decides that $\vec{I}$ satisfies Σ up to its error bound, whereas the linear solver can decide that $\vec{I}$ is not consistent with Σ since it is not validated using precise arithmetic. When this happens, the same set Σ can be repeatedly violated and the refinement algorithm may loop forever without making progress. To avoid this problem, we pose the requirement that the added Σ should not be already contained in N_φ. Otherwise, we directly return "satisfiable" (Line 6 and 7 in Fig. 3). We will show in the next section that this decision preserves the correctness guarantees of ICP.

IV. Correctness Guarantees

Originally ICP is used in solving systems of nonlinear equalities/inequalities over real numbers. Thus, the notion of correctness of ICP is not directly formulated for the use in decision procedures. A well-known property [15] of ICP is that when a system S of real equalities and inequalities has a solution, ICP always returns an interval solution $\vec{I}$ of S. In deciding QFNRA problems, this property of ICP implies that when a system is satisfiable, ICP always returns "satisfiable". In other words, when ICP returns "unsatisfiable", the system must be unsatisfiable.

Conversely, if whenever ICP returns "satisfiable" the system is also satisfiable, we would have a sound and complete solver[1]. This can not be guaranteed by ICP because of its use of finite-precision arithmetic. In other words, the "satisfiable" answers from ICP can not always be trusted. In the design of HySAT [11], posterior validation procedures of the interval solutions are applied, and the solver can return "unknown" when a solution is not validated.

Similar validation procedures can be straightforwardly adopted in our solver. However, in what follows we aim to make clear the exact meanings of the answers returned by ICP algorithms in the context of decision problems. In fact, we will show that ICP does guarantee a moderately relaxed notion of soundness and completeness that can indeed prove useful for certain verification tasks.

Informally, when ICP returns "satisfiable" for a set S of theory atoms, it must be one of the following two cases:

- S is satisfiable.
- S is unsatisfiable, but if some constant terms in S are changed *slightly*, S will become satisfiable.

Contrapositively, if a system S remains unsatisfiable under small perturbations on its constant terms, ICP indeed returns that S is "unsatisfiable". This notion (as a special case of the formulation in [21]) is made precise in the following definition.

Definition 5 (δ-robustness)**.** *Let S be a system of equalities $\bigwedge_{i=1}^{k} f_i = 0$, where $f_i \in \mathbb{R}[\vec{x}]$, and $x_i \in I_i$ where $I_i \subseteq \mathbb{R}$ are intervals. Let $\delta \in \mathbb{R}^+$ be a positive real number.*

*S is called δ-**robustly unsatisfiable** if for any choice of $\vec{c} = (c_1, ..., c_k)$ where $|c_i| \leq \delta$, $\bigwedge_{i=1}^{k} f_i = c_i$ remains unsatisfiable. Each $\vec{c}$ is called a **perturbation** on S.*

We write the system perturbed by $\vec{c}$ as $S^{\vec{c}}$. Note that we only considered systems of equalities, because inequalities can be turned into equalities by introducing new bounded variables. The following example illustrates the definition.

Example 3. *Consider the system $S : y = x^2 \wedge y = -0.01$. S is unsatisfiable. If we set $\delta_1 = 0.1$, then there exists a perturbation $c = 0.01 < \delta_1$ such that $S^{(0,c)} : y = x^2 \wedge y = 0$ is satisfiable. However, if we set $\delta_2 = 0.001$, then there does not exist $\vec{c}$ that can make $S^{\vec{c}}$ satisfiable with $|c_i| \leq \delta_2$. Hence, we say S is δ_2-robustly unsatisfiable.* □

The bound δ of "undetectable perturbations" corresponds to the error bound of ICP. It can be made very small in practice (e.g., 10^{-6}). To be precise, we have the following theorem:

[1]The notion of soundness and completeness have quite different, although related, definitions in different communities. We will give clear definitions when a formal notion is needed (such as δ-completeness). Informally, we will only use "sound and complete" together to avoid mentioning their separate meanings that may cause confusion.

Theorem 1. *Let S be a system of real equalities and inequalities. Let δ be the preset error bound of ICP. If for any $\vec{c}$ satisfying $|c_i| \leq \delta$, $S^{\vec{c}}$ is unsatisfiable, then ICP returns "unsatisfiable" on S.*

Proof: First, note that we only need to consider systems of equalities. This is because by introducing a new variable, an inequality $f(\vec{x}) > c$ can be turned into an equality $f(\vec{x}) = y$ with the interval bound $y \in (c, +\infty)$.

Now, let $S : \bigwedge_{i=1}^{k} f_i(\vec{x}) = 0$ be a system of equalities, where the variables are bounded by the initial interval bounds $\vec{x} \in \vec{I}_0$, and $f_i \in \mathbb{R}[\vec{x}]$ are polynomials.

Suppose S is decided as satisfiable by ICP. ICP returns an interval solution $I_{\vec{x}}$ for $\vec{x}$. The δ error bound of ICP ensures that:

$$\exists \vec{x} \in I_{\vec{x}} \, [\bigwedge_{i=1}^{k} |f_i(\vec{x})| \leq \delta].$$

Let $\vec{a}$ be the witness to the above formula. We then have

$$(f_1(\vec{a}) = c_1 \wedge c_1 \leq \delta) \ \wedge ... \wedge \ (f_k(\vec{a}) = c_k \wedge c_k \leq \delta).$$

Consequently, $(c_1, ..., c_k)$ is indeed a perturbation vector that makes $S^{(c_1,...,c_k)} : \bigwedge_{i=1}^{k} f_i(\vec{x}) = c_i$ satisfiable with the solution $\vec{a}$. As a result, S is not δ-robustly unsatisfiable, which contradicts the assumption. ∎

This property ensures that ICP is not just a partial heuristic for nonlinear problems, but satisfies a "numerically relaxed" notion of completeness, which we call **δ-completeness**:

- *If S is satisfiable, then ICP returns "satisfiable".*
- *If S is δ-robustly unsatisfiable, then ICP returns "unsatisfiable".*

Consequently, the answer of ICP can only be wrong on systems that are unsatisfiable but not δ-robustly unsatisfiable, in which case ICP returns "satisfiable". We can say such systems are "fragilely unsatisfiable".

In practice, it can be advantageous to detect such fragile systems. In bounded model checking, an "unsatisfiable" answer of an SMT formula means that the represented system is "safe" (a target state can not be reached). Thus, fragilely unsatisfiable systems can become unsafe under small numerical perturbations. In the standard sense, a fragilely unsatisfiable formula should be decided as "unsatisfiable" by a complete solver, and such fragility will be left undetected. Instead, ICP categorizes such fragilely unsatisfiable systems as "satisfiable". Moreover, ICP returns a solution. Note that this solution is spurious for the unperturbed system, but is informative of the possible problem of the system under small perturbations. On the other hand, ICP returns "unsatisfiable" on a system *if and only if* the system is δ-robustly safe. The error bound δ of ICP can also be changed to allow different levels of perturbations in the system.

Our checking and validation procedures are devised to preserve such correctness guarantees of ICP. Formally, we have the following theorem.

Theorem 2 (δ-completeness of Check()). *Let φ be the preprocessed input formula for Check(), and δ the error bound of ICP. If φ is satisfiable, then the Check() procedure returns "satisfiable". If φ is δ-robustly unsatisfiable, then the Check() procedure returns "unsatisfiable".*

A detailed proof of the theorem is contained in our extended technical report [13].

As a technical detail, we need to mention that the preprocessing procedure may change the actual δ in the robustness claims. The reason is that when we preprocess a formula φ to φ', new variables are introduced for compound terms, and new constants are used. Perturbations allowed on the new constants may accumulate in φ'. For instance, $x^2 = 1 \wedge x = 0$ is robustly unsatisfiable for $\delta = 1/2$. But when it is preprocessed to $x^2 - h = 0 \wedge h = 1 \wedge x = 0$, the perturbations on the first two atoms can be added, and in effect the formula is no longer $1/2$-robustly unsatisfiable ($x^2 - h = -1/2 \wedge h = 1/2 \wedge x = 0$ is satisfiable). Note that the new formula is still $1/3$-robustly unsatisfiable. The change of δ is solely determined by the number of the new variables introduced in preprocessing. In practice, when the exact error bound is needed, a new δ' can be calculated for the robustness claims that we make for the original formula. As is usually the case, the error bound is small enough (e.g. 10^{-6}) such that δ' and δ are of the same order of magnitude.

V. Assertion and Learning Procedures

In a standard DPLL(T) framework, the theory solver provides additional methods that facilitate the main checking procedures to enhance efficiency. First, a partial check named Assert() is used to prune the search space before the complete Check() procedure. Second, when conflicts are detected by the checking procedures, the theory solver uses a Learn() procedure to provide explanations for the conflicts. Such explanations consist of theory atoms in the original formula, which are added to the original formula as "learned clauses". Third, when conflicts are detected, the theory solver should backtrack to a previous consistent set of theory atoms, using the Backtrack() procedure.

In this section, we briefly describe how these additional methods can be designed when the interval methods in ICP are used in the checking procedures. A complete description of the procedures requires references to more details of ICP, which can be found in our extended technical report [13].

A. Interval Contraction in Assert()

In the DPLL(T) framework, besides the Check() procedure, an Assert() procedure is used to provide a partial check of the asserted theory atoms [10]. We use interval contraction (Definition 2) to detect early conflicts in Assert() in the following way:

In each call to Assert(), a new atom $x \sim c$ is added to the set Λ of asserted theory atoms. First, the interval assignment on x is updated by the new atom $x \sim c$. Then, Assert() contracts the interval assignment $\vec{I}$ for all the variables with respect to the linear and nonlinear constraints. That is, it takes $\vec{I}$ as input, and outputs a new vector of intervals $\vec{I}'$, such that $(\vec{I}, \vec{I}')$ is a valid contraction step (Definition 2) preserving the consistency

conditions (Definition 3). If $\vec{I}'$ becomes empty, it represents an early conflict in Λ. Otherwise, Assert() returns the contracted intervals as the updated $\vec{I}$.

B. Generating Explanations and Backtracking

1) Generating Explanations: As described in Section II-A, ICP returns "unsatisfiable" when the interval assignment on some variable x is contracted to the empty set. When this happens, we need to recover the set of atoms that has contributed to the contraction of intervals on the variable x. This can be done by keeping track of the contraction steps.

Let x be a variable in φ, and l a theory atom of the form $y \sim c$ in φ. For convenience we can write l as $y \in I_c^y$. Suppose x has a contraction sequence $S_x = (I_1^x, ..., I_n^x)$. We define:

Definition 6. *The theory atom l is called a **contributing atom** for x, if there exists a contraction step (I_i^x, I_{i+1}^x) in S_x, satisfying that $I_{i+1}^x = I_i^x \cap F(\vec{I})$, where I_c^y appears in $\vec{I}$.*

The **contributing atom list** for a variable x is defined as $L_x = \bigwedge\{l : l \text{ is a contributing atom of } x\}$. We can prove that when the interval on x is contracted to the empty set, i.e., when $I_n^x = \emptyset$, it is sufficient to take the negation of L_x as the learned clause:

Proposition 2. *Let x be a variable in formula φ with a contraction sequence $(I_1^x, ..., I_n^x)$. Let L_x be the contributing atom list of x. Suppose $I_n^x = \emptyset$, then $N_\varphi \wedge L_\varphi \wedge L_x$ is unsatisfiable.*

A detailed proof is contained in [13].

2) Backtracking: When an inconsistent set Λ of atoms is detected by either Assert() or Check(), the solver calls the SAT solver to backtrack to a subset Λ' of Λ and assert new atoms. The theory solver assists backtracking by eliminating all the decisions based on atoms in $\Lambda \backslash \Lambda'$, and restores the solver state back to the decision level where Λ' is checked by Assert(). Since the Assert() procedure stores interval assignments during the contraction process, this is accomplished by restoring the interval assignment at that level.

VI. Experimental Results

We have implemented a prototype solver using the `realpaver` package for ICP [14] and the open-source SMT solver `opensmt` [4]. We accept benchmarks in the SMT-LIB [5] format, and have extended it to accept floating-point numbers. All experiments are conducted on a workstation with Intel(R) Xeon 2.4Ghz CPU and 6.0GB RAM running Linux.

A. Bounded Model Checking of Embedded Software

Our main target domain of application is bounded model checking of embedded software programs that contain nonlinear floating point arithmetic. The benchmarks (available online at [1]) in Table I are generated from unwinding a program that reads in an array of unknown values of bounded length, and tries to reach a target range by performing different arithmetic operations on the input values [16].

In Table I, We show the running time comparison between LRA+ICP and the HySAT/iSAT tool [3]. (`hysat-0.8.6` and its new version `isat` give roughly the same results on the benchmarks, we picked the best timings.)

D	#Vars	$\#L_\varphi$	$\#N_\varphi$	#l	Result	**LRA+ICP**	HySAT
Benchmark Set: AddArray							
6	10	3	0	1	`UNSAT`	0.06s	0.04s
8	36	10	0	1	`UNSAT`	0.09s	303.03s
31	1634	735	0	1	`UNSAT`	0.93s	mem-out
Benchmark Set: MultiArray-1							
5	10	3	20	1	`UNSAT`	0.23s	0.02s
7	30	8	28	1	`UNSAT`	0.04s	7.21s
8	121	40	32	1	`UNSAT`	0.12s	56.46s
16	817	320	64	1	`UNSAT`	0.32s	mem-out
26	1687	670	104	2	`SAT`	87.45s	mem-out
Benchmark Set: MultiArray-2							
9	208	75	36	1	`UNSAT`	0.73s	244.85s
10	295	110	40	1	`UNSAT`	0.11s	123.02s
11	382	145	44	1	`UNSAT`	0.12s	3.96s
20	1165	460	80	1	`UNSAT`	0.30s	mem-out
26	1687	670	104	2	`SAT`	65.72s	mem-out
Benchmark Set: MultiArrayFlags							
11	861	337	44	1	`UNSAT`	0.19s	mem-out
21	2131	847	84	1	`UNSAT`	0.93s	mem-out
31	3401	1357	124	1	`UNSAT`	0.65s	mem-out
51	5941	2377	204	1	`UNSAT`	26.17s	mem-out

TABLE I: LRA+ICP and HySAT on BMC Benchmarks

name	cvc3(s)	LRA+ICP	name	cvc3(s)	LRA+ICP
10u05	2.21	8.87	20revert	6.73	36.12
20u10	5.54	14.25	30u15	13.52	120.21
40f10	117.53	89.01	40f25	123.97	175.28
40f50	228.25	99.26	40f99	240.11	215.12
40m10	120.16	86.29	40m25	120.18	153.01
40m50	213.12	111.87	40m99	237.87	217.92
40s10	41.445	280.06	40s25	40.38	180.15
40s50	37.59	180.12	40s99	35.23	189.43
40u20	28.31	231.21	c40f	timeout	270.12
c40m	timeout	279.45	c40s	34.12	301.76
l40f	15.02	320.12	l40s	20.32	242.75
m40e	25.72	113.23	m40	226.21	182.12

TABLE II: LRA+ICP and CVC3 on QF_UFNRA Benchmarks

In the table, the first column ("D") is the unrolling depth of the original program. The number of variables (#Vars), linear constraints ($\#L_\varphi$), and nonlinear constraints ($\#N_\varphi$) are the ones that actually effective in the theory solver, after preprocessing is done. They can be much lower than the raw numbers appearing in the benchmark. The "#l" column is the number of iterations of the linear-nonlinear checking loop (Step 2-4 in Section III-A) that are used in obtaining the answer. "mem-out" indicates that HySAT aborted for the reason that no more memory can be allocated.

For the "`UNSAT`" instances, the linear solver detects conflicts in the linear constraints early on, and avoids solving the nonlinear constraints directly as in HySAT. For the "`SAT`" instances, the two iterations of the linear-nonlinear checking loop proceed as follows: Initially, no linear conflicts were detected, and ICP is invoked to solve the nonlinear constraints and return an interval solution. The linear solver then detects that the interval solutions violate a set of linear constraints, which are added to the constraints considered by ICP (Line 6 in Fig. 3). This concludes the first iteration. In the second iteration, ICP solves the expanded set of constraints and return

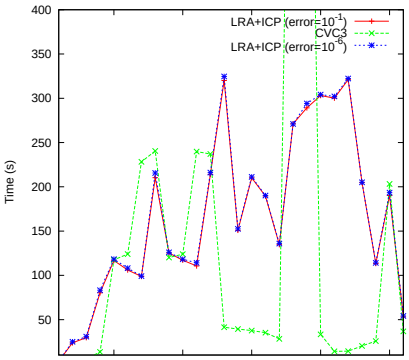


Fig. 6: LRA+ICP and CVC3 on QF_UFNRA Benchmarks

a new interval solution. Then the linear solver detects no further conflict and return "SAT" as the final answer. This concludes the second iteration.

We see that by separating the linear and nonlinear solving stages, we can exploit the highly efficient linear solver for solving the linear constraints and invoke the nonlinear solving capacity of ICP only when needed. The proposed refinement loop ensures that the correctness guarantees of ICP are preserved under such separation of linear and nonlinear solving.

B. QFNRA Problems from SMT-LIB

We have obtained results on QF_UFNRA benchmarks on SMT-LIB [5]. So far the only solver that solves the same set of benchmarks is CVC3 [2]. (The HySAT solver uses a special format. CVC3 does not accept floating point numbers in the previous set of benchmarks.)

In Table II we compare the LRA+ICP solver with CVC3. The data are plotted in Fig 6. (The timeout limit is 1000s.) The timing result is mixed. Note that our solver ensures δ-completeness and does not have specific heuristics. Consequently, our solver performs rather uniformly on all the benchmarks, whereas CVC3 can be much faster or slower on some of them. (We are not aware of the solving strategy in CVC3.) To evaluate the influence of the error bound δ on the speed of the solver, we have set it to different values $\delta_1 = 10^{-1}$ and $\delta_2 = 10^{-6}$. However, the difference is not significant on this set of benchmarks. The reason for this may be that the nonlinear constraints in the benchmarks are all of the simple form "$x = yz$" with few shared variables.

VII. Conclusion

We have proposed a novel integration of interval constraint propagation with SMT solvers for linear real arithmetic to decide nonlinear real arithmetic problems. It separates linear and nonlinear solving stages, and we showed that the proposed methods preserve the correctness guarantees of ICP. Experimental results show that such separation is useful for enhancing efficiency. We envision that the use of numerical methods with correctness guarantees such as ICP can lead to more practical ways of handling nonlinear decision problems. Further directions involve developing heuristics for different systems with specific types of nonlinear constraints and extend the current results to transcendental functions.

Acknowledgment

We thank Professor Martin Fränzle and the HySAT project group for their help and important suggestions (Professor Fränzle pointed out the problem that the robustness parameters can be changed due to preprocessing), and the anonymous referees for their valuable comments. We thank Professor Clark Barrett for his patient help on CVC3.

References

[1] http://www.nec-labs.com/research/system/systems_SAV-website/benchmarks.php, Item 7.
[2] CVC3. http://cs.nyu.edu/acsys/cvc3/index.html.
[3] Hysat. http://hysat.informatik.uni-oldenburg.de.
[4] Opensmt. http://code.google.com/p/opensmt/.
[5] C. Barrett, S. Ranise, A. Stump, and C. Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). www.SMT-LIB.org, 2008.
[6] F. Benhamou and L. Granvilliers. Continuous and interval constraints. In F. Rossi, P. van Beek, and T. Walsh, editors, *Handbook of Constraint Programming*, chapter 16. Elsevier, 2006.
[7] L. Bordeaux, Y. Hamadi, and M. Y. Vardi. An analysis of slow convergence in interval propagation. In *CP*, pages 790–797, 2007.
[8] C. Borralleras, S. Lucas, R. Navarro-Marset, E. Rodríguez-Carbonell, and A. Rubio. Solving non-linear polynomial arithmetic via sat modulo linear arithmetic. In *CADE*, pages 294–305, 2009.
[9] C. W. Brown and J. H. Davenport. The complexity of quantifier elimination and cylindrical algebraic decomposition. In *ISSAC-2007*.
[10] B. Dutertre and L. M. de Moura. A fast linear-arithmetic solver for DPLL(T). In *CAV-2006*.
[11] M. Fränzle, C. Herde, T. Teige, S. Ratschan, and T. Schubert. Efficient solving of large non-linear arithmetic constraint systems with complex boolean structure. *JSAT*, 1(3-4):209–236, 2007.
[12] M. K. Ganai and F. Ivančić. Efficient decision procedure for non-linear arithmetic constraints using cordic. In *Formal Methods in Computer Aided Design (FMCAD)*, 2009.
[13] S. Gao, M. Ganai, F. Ivančić, A. Gupta, S. Sankaranarayanan, and E. M. Clarke. Integrating icp and lra solvers for deciding nonlinear real arithmetic problems. Technical report, "http://www.cs.cmu.edu/~sicung/papers/fmcad10.html".
[14] L. Granvilliers and F. Benhamou. Algorithm 852: Realpaver: an interval solver using constraint satisfaction techniques. *ACM Trans. Math. Softw.*, 32(1):138–156, 2006.
[15] P. V. Hentenryck, D. McAllester, and D. Kapur. Solving polynomial systems using a branch and prune approach. *SIAM Journal on Numerical Analysis*, 34(2):797–827, 1997.
[16] F. Ivančić, M. K. Ganai, S. Sankaranarayanan, and A. Gupta. Numerical stability analysis of floating-point computations using software model checking. In *MEMOCODE 2010*.
[17] L. Jaulin. Localization of an underwater robot using interval constraint propagation. In *CP*, pages 244–255, 2006.
[18] L. Jaulin, M. Kieffer, O. Didrit, and È. Walter. *Applied Interval Analysis with Examples in Parameter and State Estimation, Robust Control and Robotics*. Springer-Verlag, London, 2001.
[19] D. Kroening and O. Strichman. *Decision Procedures: An Algorithmic Point of View*. Springer, 2008.
[20] A. Platzer, J.-D. Quesel, and P. Rümmer. Real world verification. In *CADE*, pages 485–501, 2009.
[21] S. Ratschan. Quantified constraints under perturbation. *J. Symb. Comput.*, 33(4):493–505, 2002.

A Halting Algorithm to Determine the Existence of Decoder

ShengYu Shen, Ying Qin, JianMin Zhang, and SiKun Li
School of Computer Science, National University of Defense Technology
410073,ChangSha,China
Email: {syshen,qy123,jmzhang,skli}@nudt.edu.cn

***Abstract*—Complementary synthesis automatically synthesizes the decoder circuit E^{-1} of an encoder E. It determines the existence of E^{-1} by checking the parameterized complementary condition (PC). However, this algorithm will not halt if E^{-1} does not exist. To solve this problem, we propose a novel halting algorithm to check PC in two steps.**

First, we over-approximate PC with the linear path unique condition (LP), and then falsify LP by searching for a loop-like path. If such a loop is found, then E^{-1} does not exist; otherwise, LP can eventually be proved within E's recurrence diameter.

Second, with LP proved above, we construct a list of approximations that forms an onion-ring between PC and LP. The existence of E^{-1} can be proved by showing that E belongs to all these rings.

To illustrate its usefulness, we have run our algorithm on several complex encoder circuits, including PCIE and 10G Ethernet. Experimental results show that our new algorithm always distinguishes correct Es from incorrect ones and halts properly.

***Index Terms*—Halting Algorithm, Complementary Synthesis**

I. Introduction

Complementary synthesis has been proposed by us [1] to automatically synthesize an encoder circuit E's decoder E^{-1} in two steps. **First**, it determines the existence of E^{-1} by checking the parameterized complementary condition(***PC***), i.e., whether E's input can be uniquely determined by its output on a bounded unfolding of E's transition function. **Second**, it builds E^{-1} by characterizing its Boolean function with an all-solution SAT solver.

However, the bounded nature of the first step makes it an incomplete algorithm that will not halt if E^{-1} does not exist.

To solve this problem, as shown in Figure 1, we propose a novel halting algorithm to check PC in two steps:

1) First, we over-approximate PC with the linear path unique condition (***LP***), i.e., every linear path of E longer than a particular parameter p always reaches the unique state set S^U, in which the input letter can be uniquely determined by the output letter, the current state and the next state. We then define the negative condition of LP,i.e., the loop-like non-unique condition(***LL***). We can falsify LP and prove LL by searching for a loop-like path that does not reach S^U within E's recurrence diameter rd. If we find such a loop-like path, then LL is proved and E^{-1} does not exist; otherwise, a parameter p can eventually be found to prove LP. In this case, we need the second step below to further check PC.
2) Second, with p found in the first step that proves LP, we construct a list of approximations that forms an onion-ring between PC and its over-approximation LP. If E is found in a certain ring but not in the next inner ring, then PC is falsified and E^{-1} does not exist; otherwise, the existence of E^{-1} is proved.

We have implemented our algorithm with the OCaml language, and solved the generated SAT instances with Zchaff SAT solver [2]. The benchmark set includes several complex encoders from industrial projects (e.g., PCIE and Ethernet), and their slightly modified variants without corresponding decoders. Experimental results show that our new algorithm always distinguishes correct encoders from their incorrect variants and halts properly. All experimental results and programs can be downloaded from http://www.ssypub.org.

This paper's contribution is: We propose the first halting algorithm to determines the existence of an encoder's decoder.

The remainder of this paper is organized as follows. Section II presents background materials. Section IV introduces how to over-approximate PC with LP, and how to falsify LP by searching for loop-like paths. Section V discusses how to construct the onion-ring, and how to determine whether E belongs to a certain ring. Section VI describes how to remove redundant output letters to minimize circuit area, while Section VII and VIII present experimental results and related works. Finally, Section IX concludes with a note on future work.

II. Preliminaries

A. Basic Notation of Propositional Satisfiability Problem

For a Boolean formula F over a variable set V, the **Propositional Satisfiability Problem**(abbreviated as **SAT**) is to find a satisfying assignment $A : V \rightarrow \{0, 1\}$, so that F can

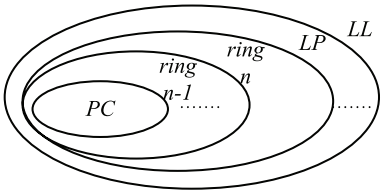


Fig. 1. Relationship between PC,LP and LL

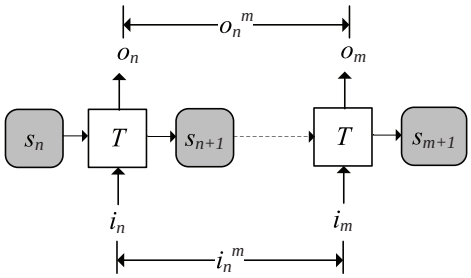


Fig. 2. Mealy finite state machine

be evaluated to 1. If such a satisfying assignment exists, then F is **satisfiable**; otherwise, it is **unsatisfiable**.

A computer program that decides the existence of such a satisfying assignment is called a **SAT solver**, such as Zchaff [2], Grasp [3], Berkmin [4], and MiniSAT [5]. A formula to be solved by a SAT solver is also called a **SAT instance**.

B. Recurrence Diameter

A circuit can be modeled by **Kripke structure** $M = (S, I, T, A, L)$, with a finite state set S, the initial state set $I \subseteq S$, the transition relation $T \subseteq S \times S$, and the labeling of the states $L : S \rightarrow 2^A$ with atomic proposition set A.

Kroening et al. [6] defined the **state variables recurrence diameter** with respect to M, denoted by $rrd(M)$, as the longest loop-free path in M starting from an initial state.

$$rrd(M) \stackrel{def}{=} \max\{i | \exists s_0 \ldots s_i : I(s_0) \wedge \bigwedge_{j=0}^{i-1} T(s_j, s_{j+1}) \wedge \bigwedge_{j=0}^{i-1} \bigwedge_{k=j+1}^{i} s_j \neq s_k\} \quad (1)$$

In this paper, we define a similar concept: the **uninitialized state variables recurrence diameter** with respect to M, denoted by $uirrd(M)$, is the longest loop-free path in M.

$$uirrd(M) \stackrel{def}{=} \max\{i | \exists s_0 \ldots s_i : \bigwedge_{j=0}^{i-1} T(s_j, s_{j+1}) \wedge \bigwedge_{j=0}^{i-1} \bigwedge_{k=j+1}^{i} s_j \neq s_k\} \quad (2)$$

The only difference between these two definitions is that our $uirrd$ does not consider the initial state.

These definitions are only used in proving our theorems below. Our algorithm does not need to compute these diameters.

C. The Original Algorithm to Determine the Existence of Decoder

The complementary synthesis algorithm [1] includes two steps: determining the existence of decoder and characterizing its Boolean function. We will only introduce the first step here.

The encoder E can be modeled by a Mealy finite state machine [7].

Definition 1: **Mealy finite state machine** is a 5-tuple $M = (S, s_0, I, O, T)$, consisting of a finite state set S, an initial state $s_0 \in S$, a finite set of input letters I, a finite set of output letters O, a transition function $T : S \times I \rightarrow S \times O$ that computes the next state and output letter from the current state and input letter.

As shown in Figure 2, as well as in the remainder of this paper, the state is represented as a gray round corner box, and the transition function T is represented by a white rectangle.

We denote the state, input letter and output letter at the n-th cycle respectively as s_n, i_n and o_n. We further denote the sequence of state, input letter and output letter from the n-th to the m-th cycle respectively as s_n^m, i_n^m and o_n^m.

A sufficient condition for the existence of E^{-1} is the **unique condition**, i.e., there exist two parameters d and l, so that i_n of E can be uniquely determined by the output sequence o_{n+d-l}^{n+d-1}. As shown in Figure 3, d is the relative delay between o_{n+d-l}^{n+d-1} and the input letter i_n, while l is the length of o_{n+d-l}^{n+d-1}.

However, the unique condition is unnecessarily restrictive, because it may not hold when s_n is not reachable, even if E is a correct encoder whose input can be uniquely determined by its output in its reachable state set. So we need to rule out unreachable states before checking the unique condition.

The continuous running character of communication circuits provides us an opportunity to rule out unreachable states easily without paying the expensive cost of computing the reachable state set. That is to say, we only need to check the unique condition on the state set RS^∞ that can be reached infinitely often from S.

$$RS^q \stackrel{def}{=} \{s_q | \bigwedge_{m=0}^{q-1} \{(s_{m+1}, o_m) \equiv T(s_m, i_m)\}\} \quad (3)$$

$$RS^{>p} \stackrel{def}{=} \bigcup_{q>p} RS^q \quad (4)$$

$$RS^\infty \stackrel{def}{=} \lim_{p \rightarrow \infty} RS^{>p} \quad (5)$$

Here, RS^q is the set of states that can be reached from S with exact q steps.

According to Equation (5) and Figure 3, RS^∞ can be easily over-approximated by prepending a state transition sequence of length p to s_n, which forces s_n to be in the state set $RS^{>p} = \bigcup_{q>p} RS^q$. Obviously, RS^∞ and all $RS^{>p}$ form a total order shown below, which means a tighter over-approximation of RS^∞ can be obtained by increasing the length p of prepended state transition sequence.

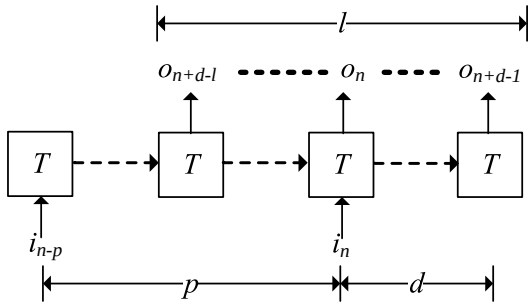


Fig. 3. The parameterized complementary condition

$RS^{\infty} \subseteq \cdots \subseteq RS^{>p2} \subseteq \cdots \subseteq RS^{>p1} \subseteq \ldots$ where $p2 > p1$

Thus, as shown in Figure 3, the parameterized complementary condition(PC) [1] can be defined as:

***Definition** 2:* **Parameterized Complementary Condition ($\boldsymbol{PC}$) :** For encoder E, $E \vDash PC(p,d,l)$ holds if i_n can be uniquely determined by o_{n+d-l}^{n+d-1} on s_{n-p}^{n+d-1}. This equals the unsatisfiability of $F_{PC}(p,d,l)$ in Equation (6). We further define $E \vDash PC$ as $\exists p,d,l : E \vDash PC(p,d,l)$.

$$F_{PC}(p,d,l) \stackrel{def}{=} \left\{ \begin{array}{c} \bigwedge_{m=n-p}^{n+d-1}\{(s_{m+1},o_m) \equiv T(s_m,i_m)\} \\ \wedge \quad \bigwedge_{m=n-p}^{n+d-1}\{(s'_{m+1},o'_m) \equiv T(s'_m,i'_m)\} \\ \wedge \quad \bigwedge_{m=n+d-l}^{n+d-1} o_m \equiv o'_m \\ \wedge \quad i_n \neq i'_n \end{array} \right\} \quad (6)$$

The 2nd and 3rd lines of Equation (6) correspond respectively to two unfolded instances of E's transition function. The only difference between them is that a prime is appended to every variable in the 3rd line. The 4th line forces the output sequences of these two unfolded instances to be the same, while the 5th line forces their input letters to be different.

III. Case Studies

To facilitate the understanding of our idea, we use some small examples shown in Figure 4,5 and 6.

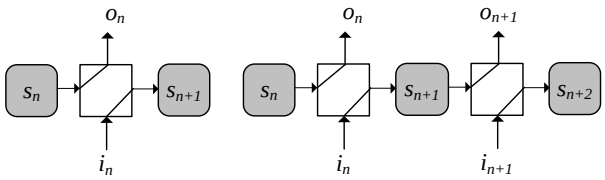
A. Case study 1

The circuit in Figure 4a) stores its input port i_n in register s_{n+1}, and then outputs it to output port o_{n+1}. The unfolding of its transition function is shown in Figure 4b).

Obviously, i_n is same as, and therefore can be uniquely determined by s_{n+1}. So i_n can be uniquely determined by s_n, o_n and s_{n+1}. So LP is satisfied by this circuit. Here, the tuple $< s_n, o_n, s_{n+1} >$ can be seen as a ring that surrounds i_n.

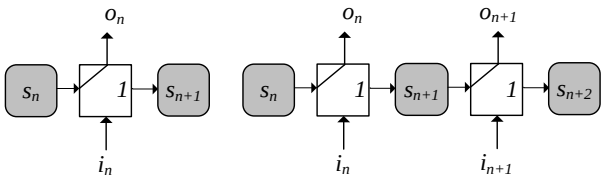
Next, we expand the ring $< s_n, o_n, s_{n+1} >$ to another ring $< s_n, o_n, o_{n+1}, s_{n+2} >$, and perform the following 3 checks:

1) Whether i_n can be uniquely determined by the ring $< s_n, o_n, o_{n+1}, s_{n+2} >$? Obviously the answer is yes.
2) Whether i_n can be uniquely determined by $< o_n, o_{n+1}, s_{n+2} >$, the ring with s_n removed? Obviously the answer is yes.
3) Whether i_n can be uniquely determined by $< o_n, o_{n+1} >$, the ring with s_n and s_{n+2} both removed? Obviously the answer is yes. In this case, we find that i_n can be uniquely determined by the output sequence $< o_n, o_{n+1} >$. Thus, a decoder exists for this circuit.



a) Mealy machine b) Unfolding

Fig. 4. Case study 1



a) Mealy machine b) Unfolding

Fig. 5. Case study 2

B. Case study 2

The circuit in Figure 5a) connects a constant 1, instead of input port i to register s. So i_n can never be determined by s_n, o_n and s_{n+1} in all states. Thus, a loop-like path with length 1 will reach such a state, which satisfies LL and falsifies LP. So no decoder exists for this circuit.

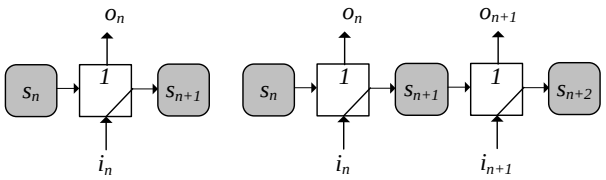
C. Case study 3

For the circuit in Figure 6a), the unfolding of its transition function is shown in Figure 6b). It's output is driven by constant 1, instead of register s. Obviously, this circuit can satisfy LP, which means i_n can be uniquely determined by s_n,o_n and s_{n+1}.

Next, we expand the ring $< s_n, o_n, s_{n+1} >$ to another ring $< s_n, o_n, o_{n+1}, s_{n+2} >$, and perform the following 3 checks:

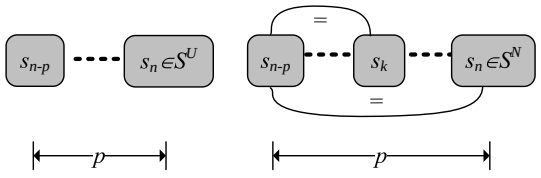
1) Whether i_n can be uniquely determined by the ring $< s_n, o_n, o_{n+1}, s_{n+2} >$? The answer is no, because i_n never goto o_n, o_{n+1} and s_{n+2}.
2) Whether i_n can be uniquely determined by $< o_n, o_{n+1}, s_{n+2} >$, the ring with s_n removed? The answer is still no with the same reason.
3) Whether i_n can be uniquely determined by $< o_n, o_{n+1} >$, the ring with s_n and s_{n+2} both removed? The answer is still no with the same reason.

So in this case, no more expansion is needed, no decoder exists for this circuit.



a) Mealy machine b) Unfolding

Fig. 6. Case study 3



a) The linear path unique condition

b) The loop-like non-unique condition

Fig. 7. Two new uniqueness conditions

IV. Over-approximating PC with LP and Falsifying LP by Searching for Loop-like Path

A. Definition of Over-approximation

We first present some related definitions before defining the over-approximation of PC.

***Definition** 3:* **Unique State Set S^U and Non-unique State Set S^N:** For a circuit E, its unique state set S^U is the set of states s_n that makes i_n to be uniquely determined by s_n,o_n and s_{n+1}, i.e., makes F^U in Equation (7) unsatisfiable. The non-unique state set S^N is the complementary set of S^U, i.e., $S^N \stackrel{def}{=} S - S^U$.

$$F^U \stackrel{def}{=} \left\{ \begin{array}{c} (s_{n+1}, o_n) \equiv T(s_n, i_n) \\ \wedge (s'_{n+1}, o'_n) \equiv T(s'_n, i'_n) \\ \wedge o_n \equiv o'_n \wedge s_n \equiv s'_n \wedge s_{n+1} \equiv s'_{n+1} \\ \wedge i_n \neq i'_n \end{array} \right\} \quad (7)$$

To obtain a halting algorithm, we need to develop a negative condition for PC, which can recognize all those $E \vDash \neg PC$.

Unfortunately, it is very difficult, if not impossible, to develop such a condition. So we choose to first over-approximate PC with the linear path unique condition(LP), and then develop a negative condition for LP, i.e., the loop-like non-unique condition(LL). The definitions of LP and LL are given below, and presented intuitively in Figure 7a) and 7b).

***Definition** 4:* **Linear Path Unique Condition (LP) :** For encoder E, $E \vDash LP(p)$ holds if every linear path of length p always reaches the unique state set S^U. This equals the unsatisfiability of $F_{LP}(p)$ in Equation (8). We further define $E \vDash LP$ as $\exists p : E \vDash LP(p)$.

$$F_{LP}(p) \stackrel{def}{=} F^U \wedge \bigwedge_{m=n-p}^{n} \{(s_{m+1}, o_m) \equiv T(s_m, i_m)\} \quad (8)$$

***Definition** 5:* **Loop-like Non-unique Condition (LL) :** For encoder E, $E \vDash LL(p)$ holds if there exists a loop-like path of length p that reaches the non-unique state set S^N. This equals the satisfiability of $F_{LL}(p)$ in Equation (9). We further define $E \vDash LL$ as $\exists p : E \vDash LL(p)$.

$$F_{LL}(p) \stackrel{def}{=} F_{LP}(p) \wedge \bigvee_{m=n-p+1}^{n} \{s_m \equiv s_{n-p}\} \quad (9)$$

Equation (9) is very similar to Equation (8), except that $\bigvee_{m=n-p+1}^{n}\{s_m \equiv s_{n-p}\}$ is inserted to find a loop-like path.

The intuition behind LP and LL is to check whether i_n can be uniquely determined by s_n,s_{n+1} and o_n with prepended s_{n-p}^{n-1}. Here, parameters d and l are removed, which makes it easier to find the value of p .

B. Relationships between PC, LP and LL

The relationships between PC, LP and LL are :

1. LP over-approximates PC,i.e.,$E \vDash PC \rightarrow E \vDash LP$.

2. Between LP and LL, there is always one and only one that holds,i.e.,$E \vDash LP \leftrightarrow E \vDash \neg LL$.

These relationships are presented intuitively in Figure 1, and their proofs are presented below. Those impatient readers can skip the remainder of this subsection.

Before proving these theorems, we need a lemma that defines a new formula for LP.

***Lemma** 1:* With $\overline{F}_{LP}(p, d, l)$ defined below:

$$\overline{F}_{LP}(p, d, l) \stackrel{def}{=} \left\{ \begin{array}{c} \bigwedge_{m=n-p}^{n+d-1}\{(s_{m+1}, o_m) \equiv T(s_m, i_m)\} \\ \wedge \quad \bigwedge_{m=n-p}^{n+d-1}\{(s'_{m+1}, o'_m) \equiv T(s'_m, i'_m)\} \\ \wedge o_n \equiv o'_n \wedge s_n \equiv s'_n \wedge s_{n+1} \equiv s'_{n+1} \\ \wedge i_n \neq i'_n \end{array} \right\} \quad (10)$$

we have: $\overline{F}_{LP}(p, d, l) \leftrightarrow F_{LP}(p)$

Proof: **First, for the $\rightarrow$ direction.** It is obvious that the clause set of $\overline{F}_{LP}(p, d, l)$ is a supper set of $F_{LP}(p)$, so the $\rightarrow$ direction is proved.

Second, to prove the $\leftarrow$ direction, we list below all additional sub-formulas that have been added into Equation (8) to obtain (10), and also our methods to satisfy them with a particular satisfying assignment A of $F_{LP}(p)$.

1. $\bigwedge_{m=n-p}^{n}\{(s'_{m+1}, o'_m) \equiv T(s'_m, i'_m)\}$: This formula can be satisfied by assigning $A(s_m)$, $A(i_m)$ and $A(o_m)$ to s'_m, i'_m and o'_m respectively.

2. $\bigwedge_{m=n+1}^{n+d-1}\{(s_{m+1}, o_m) \equiv T(s_m, i_m)\}$: this formula represents a state transition sequence starting from s_{n+1}, which is satisfiable.

3. $\bigwedge_{m=n+1}^{n+d-1}\{(s'_{m+1}, o'_m) \equiv T(s'_m, i'_m)\}$: this formula represents a state transition sequence starting from s'_{n+1}, which is satisfiable with the same assignment defined in 2.

So, every satisfying assignment A of $F_{LP}(p)$ can make $\overline{F}_{LP}(p, d, l)$ satisfiable. So the $\leftarrow$ direction is proved.

Thus, this theorem is proved. ■

In the remainder of this paper, we will use $F_{LP}(p)$ and $\overline{F}_{LP}(p, d, l)$ interchangeably.

***Theorem** 1:* $E \vDash PC(p, d, l) \rightarrow E \vDash LP(p)$

Proof: Let's prove it by contradiction. Assume that $A : V \rightarrow \{0, 1\}$ is a satisfying assignment of $\overline{F}_{LP}(p, d, l)$.

We define a new satisfying assignment A' as:

$$A'(v) \stackrel{def}{=} \left\{ \begin{array}{llll} A(o_m) & v \equiv o'_m & m \neq n & \\ A(i_m) & v \equiv i'_m & m \neq n & \\ A(s_m) & v \equiv s'_m & m \neq n & and \quad m \neq n+1 \\ A(v) & otherwise & & \end{array} \right. \quad (11)$$

Thus, A' is also a satisfying assignment of $\overline{F}_{LP}(p, d, l)$.

By comparing Equation (6) with (10), it is obvious that A' is a satisfying assignment of the unsatisfiable formula $F_{PC}(p, d, l)$.

This contradiction concludes the proof. ■

***Theorem** 2:* $E \vDash LP \leftrightarrow E \vDash \neg LL$

Proof: **For the $\rightarrow$ direction**, let's prove it by contradiction. Assume that $E \vDash LL$. This means there exists a loop-like path that reaches state $s_n \in S^N$.

Assume the length of this loop is q, and the parameter of $E \vDash LP$ is p. Then we can unfold this loop $[p/q]+1$ times, to get a path that is longer than p and reaches a state $s_n \in S^N$. This will lead to $E \vDash \neg LP(p)$.

This contradiction concludes the proof of the $\rightarrow$ direction.

For the $\leftarrow$ direction, assume that $E \vDash \neg LP$ and $E \vDash \neg LL$, then for all p, $F_{LP}(p)$ is satisfiable.

Assume the uninitialized state variables recurrence diameter of E is $uirrd$, and let $p = uirrd + 1$. Then $F_{LP}(p)$ is satisfiable, which means there is a path of length p that reaches a state $s_n \in S^N$. Because p is larger than $uirrd$, this path must contain a loop in it, which also makes F_{LL} satisfiable.

So $E \vDash LL$ holds, which contradicts with $E \vDash \neg LL$.

This contradiction concludes the proof of the $\leftarrow$ direction. ■

C. Algorithm to Check $E \vDash LP$ and $E \vDash LL$

Based on the relationships discussed in Subsection IV-B, we develop Algorithm 1(as shown below) to check $E \vDash LP$ and $E \vDash LL$. This algorithm also discovers the value of parameter p if $E \vDash LP$ holds.

Algorithm 1 checkLPLL

```
1: for p = 0 → ∞ do
2:   if F_LP(p) is unsatisfiable then
3:     print "E ⊨ LP(p)"
4:     halt;
5:   else if F_LL(p) is satisfiable then
6:     print "no E^-1 due to E ⊨ LL(p)"
7:     halt;
8:   end if
9: end for
```

According to Theorem 2, Algorithm 1 will eventually halt at line 3 or 6 before p reaches E's uninitialized state variables recurrence diameter $uirrd$. Thus, we have the following theorem.

***Theorem** 3:* Algorithm 1 is a halting algorithm.

With Algorithm 1, we can determine whether E is an improperly designed encoder that leads to $E \vDash LL$. **But if $E \vDash LP$, how to determine whether E is a correct encoder that leads to $E \vDash PC$?** We will discuss this problem in the next section.

V. Checking $E \vDash PC$ by Constructing Onion-Ring

A. Intuitive Description

To make it easier to follow our presentation, we present our idea intuitively here with an example.

As shown in Figure 8, we add two new parameters b and f to replace d and l. The backward parameter b refers to the distance between state s_{n-b} and s_n. The forward parameter f refers to the distance between state s_{n+1} and s_{n+1+f}. The relations between $< b, f >$ and $< d, l >$ are:

$$\begin{aligned} d &= f \\ l &= b + f + 1 \end{aligned} \tag{12}$$

Because Algorithm 1 already recognizes all Es that lead to $E \vDash LL$, we only need to deal with those Es that lead to $E \vDash LP(p)$ here. This will result in the following proposition:

***Proposition** 1:* i_n is uniquely determined by s_n, o_n and s_{n+1}.

As shown in Figure 8, we can further generalize Proposition 1 by:

1) Replacing o_n with o_{n-b}^{n+f},
2) Replacing s_n with s_{n-b},
3) Replacing s_{n+1} with s_{n+f+1},

and thus obtain:

***Proposition** 2:* i_n is uniquely determined by s_{n-b}, o_{n-b}^{n+f} and s_{n+f+1}.

It is obvious that Proposition 1 is a special case of Proposition 2, with $b \equiv 0$ and $f \equiv 0$.

With this generalization, our algorithm will be intuitively described in the following five steps:

1) First, we ignore both s_{n-b} and s_{n+f+1}, and test whether i_n can be uniquely determined by o_{n-b}^{n+f}. **If yes, our algorithm halts with $E \vDash PC$.**
2) Otherwise, we ignore s_{n-b}, and test whether i_n can be uniquely determined by o_{n-b}^{n+f} and s_{n+f+1}. If yes, then i_n definitely does **NOT** depend on any o_k with $k < n-b$, but it may still depend on some o_k with $k > n+f$. So we need to increase f by 1 and goto step 1.
3) Otherwise, we ignore s_{n+f+1}, and test whether i_n can be uniquely determined by s_{n-b} and o_{n-b}^{n+f}. If yes, then i_n definitely does **NOT** depend on any o_k with $k > n+f$, but it may still depend on some o_k with $k < n-b$. So we need to increase b by 1 and goto step 1.
4) Otherwise, we test whether i_n can be uniquely determined by s_{n-b}, o_{n-b}^{n+f} and s_{n+f+1}. If yes, then i_n may depend on some o_k with both $k < n-b$ and $k > n+f$, so we need to increase b and f by 1, and goto step 1.
5) If the algorithm reaches here, then i_n had been uniquely determined by $s_{n-b'}$, $o_{n-b'}^{n+f'}$ and $s_{n+f'+1}$ previously, but **NOT** by s_{n-b}, o_{n-b}^{n+f} and s_{n+f+1} now, where $b' \leq b$ and $f' \leq f$. This means that adding more o_k into o_{n-b}^{n+f} by

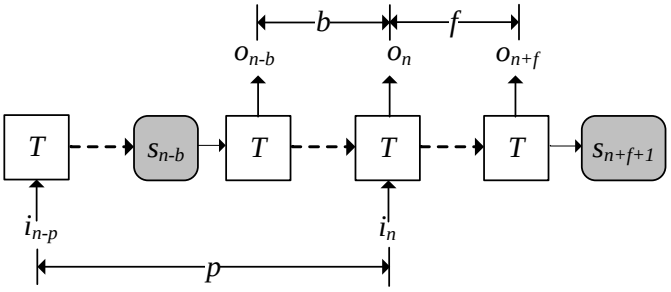


Fig. 8. Forward and backward constraints

increasing b and f, will never make PC holds. **Our algorithm halts with** $E \vDash \neg PC$.

In this algorithm, every step with a pair of b and f corresponds to an onion-ring defined in subsection V-B. If it halts at step 5, then E belongs to the ring corresponding to b' and f', but does not belong to the next inner ring corresponding to b and f, which means E^{-1} does not exist.

Formal presentation and proof will be given in the next two subsections.

B. Constructing Onion-Ring between PC and LP

According to Figure 8, we define the following formulas:

F_{unfold} defines two unfolded instances of transition function, and constrains that their output sequence are equivalent, whereas their input letters are inequivalent:

$$F_{unfold}(p,b,f) \stackrel{def}{=} \left\{ \begin{array}{c} \bigwedge_{m=n-p}^{n+f}\{(s_{m+1},o_m) \equiv T(s_m,i_m)\} \\ \wedge \quad \bigwedge_{m=n-p}^{n+f}\{(s'_{m+1},o'_m) \equiv T(s'_m,i'_m)\} \\ \bigwedge_{m=n-b}^{n+f}\{o_m \equiv o'_m\} \\ \wedge \quad i_n \neq i'_n \end{array} \right\} \tag{13}$$

$F_{backward}$ constrains that s_{n-b} equals s'_{n-b}:

$$F_{backward}(p,b,f) \stackrel{def}{=} \{s_{n-b} \equiv s'_{n-b}\} \tag{14}$$

$F_{forward}$ constrains that s_{n+1+f} equals s'_{n+1+f}:

$$F_{forward}(p,b,f) \stackrel{def}{=} \{s_{n+1+f} \equiv s'_{n+1+f}\} \tag{15}$$

With these formulas, we define 4 new unique conditions between PC and LP.

$\boldsymbol{LP\_nobf(p,b,f)}$: i_n can be uniquely determined by o_{n-b}^{n+f}. This equals the unsatisfiability of F_{LP_nobf} in Equation (16).

$$F_{LP\_nobf}(p,b,f) \stackrel{def}{=} F_{unfold} \tag{16}$$

$\boldsymbol{LP\_f(p,b,f)}$: i_n can be uniquely determined by o_{n-b}^{n+f} and s_{n+1+f}. This equals the unsatisfiability F_{LP_f} in Equation (17).

$$F_{LP\_f}(p,b,f) \stackrel{def}{=} F_{unfold} \wedge F_{forward} \tag{17}$$

$\boldsymbol{LP\_b(p,b,f)}$: i_n can be uniquely determined by s_{n-b} and o_{n-b}^{n+f}. This equals the unsatisfiability F_{LP_b} in Equation (18).

$$F_{LP\_b}(p,b,f) \stackrel{def}{=} F_{unfold} \wedge F_{backward} \tag{18}$$

$\boldsymbol{LP\_bf(p,b,f)}$: i_n can be uniquely determined by s_{n-b}, o_{n-b}^{n+f} and s_{n+1+f}. This equals the unsatisfiability F_{LP_bf} in Equation (19).

$$F_{LP\_bf}(p,b,f) \stackrel{def}{=} F_{unfold} \wedge F_{backward} \wedge F_{forward} \tag{19}$$

These new unique conditions, when coupled with parameters b and f, will act as onion-rings between PC and its over-approximation LP.

It is obvious that, LP_bf is very similar to LP, while LP_nobf is very similar to PC. Such similarities will be employed to prove the correctness of our approach.

Some lemmas that will bed used to prove the correctness of the onion-ring approach are given below:

***Lemma** 2:* $E \vDash LP\_bf(p,b,f) \leftarrow E \vDash LP\_bf(p,b,f+1)$

Proof: Let's prove it by contradiction. Assume that $E \vDash \neg LP\_bf(p,b,f)$ and $E \vDash LP\_bf(p,b,f+1)$, which means that $F_{LP\_bf}(p,b,f)$ is satisfiable while $F_{LP\_bf}(p,b,f+1)$ is unsatisfiable.

We can append a state transition to $F_{LP\_bf}(p,b,f)$ after s_{n+f+1}, and get a new formula $F'_{LP\_bf}(p,b,f)$.

$$\begin{array}{l} F'_{LP\_bf}(p,b,f) \stackrel{def}{=} F_{LP\_bf}(p,b,f) \\ \wedge (s_{n+f+2},o_{n+f+1}) = T(s_{n+f+1},i_{n+f+1}) \end{array} \tag{20}$$

This newly appended state transition is satisfiable, which makes F'_{LP_bf} a satisfiable formula.

Assume A is a satisfying assignment of $F'_{LP\_bf}(p,b,f)$. We define another satisfying assignment A' as

$$A'(v) \stackrel{def}{=} \left\{ \begin{array}{ll} A(o_{n+f+1}) & v \equiv o'_{n+f+1} \\ A(i_{n+f+1}) & v \equiv i'_{n+f+1} \\ A(s_{n+f+2}) & v \equiv s'_{n+f+2} \\ A(v) & otherwise \end{array} \right. \tag{21}$$

Obviously, A' is a satisfying assignment of unsatisfiable formula $F_{LP\_bf}(p,b,f+1)$.

This contradiction concludes the proof. ■

***Lemma** 3:* $E \vDash LP\_b(p,b,f) \leftarrow E \vDash LP\_b(p+1,b+1,f)$

Its proof is similar to that of Lemma 2.

***Lemma** 4:* If $E \vDash PC(p,d,l)$, then the following Equation holds.

$$\begin{array}{rlll} & E \vDash & LP\_bf & (p+d-l+1,0,d) \\ \leftrightarrow & E \vDash & LP\_b & (p+d-l+1,0,d) \end{array} \tag{22}$$

Proof: **For the $\leftarrow$ direction**, according to Equation (18) and (19), it is obvious that the clause set of F_{LP_bf} is a super set of F_{LP_b}, which means the unsatisfiability of the latter one implies the unsatisfiability of the former one. So the $\leftarrow$ direction is proved.

For the $\rightarrow$ direction, let's prove it by contradiction. Assume that $F_{LP\_bf}(p+d-l+1,0,d)$ is unsatisfiable, and A is a satisfying assignment of $F_{LP\_b}(p+d-l+1,0,d)$.

We define another assignment A' as :

$$A'(v) \stackrel{def}{=} \left\{ \begin{array}{llll} A(o_k) & v \equiv o'_k & where & k<n \\ A(i_k) & v \equiv i'_k & where & k<n \\ A(s_k) & v \equiv s'_k & where & k<n \\ A(v) & otherwise & & \end{array} \right. \tag{23}$$

Obviously, A' is a satisfying assignment of Equation (6), which means $E \nvDash PC(p,d,l)$. This contradiction concludes the proof of the $\rightarrow$ direction.

■

***Lemma** 5:* If $E \vDash PC(p,d,l)$, then the following Equation holds.

$$\begin{array}{rlll} & E \vDash & LP\_b & (p,l-d-1,d) \\ \leftrightarrow & E \vDash & LP\_nobf & (p,l-d-1,d) \end{array} \tag{24}$$

Its proof is similar to that of Lemma 4.

An important theorem that defines the onion-ring is presented and proved below:

Theorem 4: If $E \vDash PC(p,d,l)$, then there exists a list of unique conditions with their relationship shown below:

$$\begin{array}{lll} & E \vDash LP & (p+d-l+1) \\ \leftrightarrow & E \vDash LP\_bf & (p+d-l+1,0,0) \\ \leftarrow & E \vDash LP\_bf & (p+d-l+1,0,1) \\ & \ldots & \\ \leftarrow & E \vDash LP\_bf & (p+d-l+1,0,d-1) \\ \leftarrow & E \vDash LP\_bf & (p+d-l+1,0,d) \\ \leftrightarrow & E \vDash LP\_b & (p+d-l+1,0,d) \\ \leftarrow & E \vDash LP\_b & (p+d-l+2,1,d) \\ & \ldots & \\ \leftarrow & E \vDash LP\_b & (p,l-d-1,d) \\ \leftrightarrow & E \vDash LP\_nobf & (p,l-d-1,d) \\ \leftrightarrow & E \vDash PC & (p,d,l) \end{array} \quad (25)$$

Proof: According to Equation (10) and (19), the $\leftrightarrow$ relation between the 1st and 2nd line of Equation (25) holds.

According to Lemma 2, the $\leftarrow$ relations between the 2nd and 6th line of Equation (25) holds.

According to Lemma 4, the $\leftrightarrow$ relation between the 6th and 7th line of Equation (25) holds.

According to Lemma 3, the $\leftarrow$ relations between the 7th and 10th line of Equation (25) holds.

According to Lemma 5, the $\leftrightarrow$ relation between the 10th and 11th line of Equation (25) holds.

According to Equation (6) and (16), the $\leftrightarrow$ relations between the last two lines of Equation (25) holds. ■

In Equation (25), all $\leftarrow$ symbols form a total order, which makes all unique conditions on the right-hand side of $\vDash$s to form an onion-ring(as shown in Figure 1).

C. Algorithm Implementation

With those theorems presented in Subsection V-B, we use the following Algorithm 2 to check $E \vDash PC$.

Algorithm 2 $checkPCLP(p,b,f)$

```
1: if F_{LP_nobf}(p,b,f) is unsatisfiable then
2:    print "E ⊨ PC(p,f,b+f+1)"
3:    halt;
4: else if F_{LP_f}(p,b,f) is unsatisfiable then
5:    checkPCLP(p,b,f+1)
6: else if F_{LP_b}(p,b,f) is unsatisfiable then
7:    checkPCLP(p+1,b+1,f)
8: else if F_{LP_bf}(p,b,f) is unsatisfiable then
9:    checkPCLP(p+1,b+1,f+1)
10: else
11:    print "no E^{-1} due to E ⊨ ¬PC"
12:    halt;
13: end if
```

Algorithm 2 is invoked with the form $checkPCLP(p,0,0)$, with p computed by Algorithm 1.

Algorithm 2 just follows the onion-ring defined by Equation (25), from the first line to the last line. If $E \vDash PC$ holds, it will eventually reach line 2, and the existence of E^{-1} is proved; otherwise, it will eventually reach line 11, which means that E does not belong to the current ring, and PC is falsified. So E^{-1} does not exist.

Thus, with Theorem 4, we have the following theorem.

Theorem 5: Algorithm 2 is a halting algorithm.

VI. Removing Redundant Output Letters

Although Algorithm 1 and 2 together are sufficient to determine the existence of E^{-1}, the parameters found by line 2 of Algorithm 2 contain some redundancy, which will cause unnecessary large overhead of circuit area.

For example, as shown in Figure 9, assume that l is the smallest parameter value that leads to $E \vDash PC(p,d,l)$, and $l < d$, which means that i_n is uniquely determined by some output letters o_k with $k > n$.

We further assume that line 2 of Algorithm 2 find out $E \vDash PC(p,d,l')$. It is obvious that $l' > d$, which make i_n to depend on some redundant o_k with $k \leq n$.

So $o_{n+d-l'}^{n+d-l-1}$ is the sequence of redundant output letters, which should be removed to prevent them from being instantiated as latches in circuit E^{-1}.

Algorithm 3 that removes these redundant output letters is presented below:

Algorithm 3 $RemoveRedundancy(p,d,l')$

```
1: for l = 0 → l' do
2:    if F_PC(p,d,l) is unsatisfiable then
3:       print "E ⊨ PC(p,d,l)"
4:       halt;
5:    end if
6: end for
```

VII. Experimental Results

We have implemented our algorithm in Zchaff [2], and run it on a PC with a 2.4GHz Intel Core 2 Q6600 processor, 8GB memory and CentOS 5.2 linux. All experimental results and programs can be downloaded from http://www.ssypub.org.

A. Benchmarks

Table I shows information of the following benchmarks.

1) A XGXS encoder compliant to clause 48 of IEEE-802.3ae 2002 standard [8].

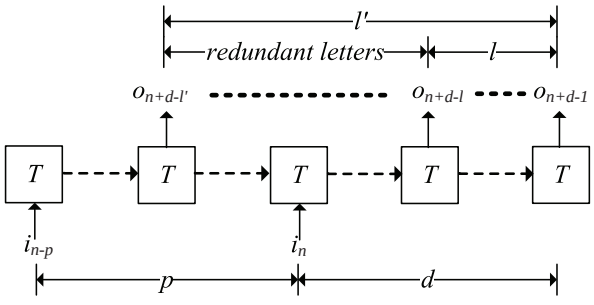


Fig. 9. Redundant Output Letters

TABLE I
INFORMATION OF BENCHMARKS

	XGXS	XFI	scrambler	PCIE	T2 ethernet
Line number of verilog source code	214	466	24	1139	1073
#regs	15	135	58	22	48
Data path width	8	64	66	10	10

2) A XFI encoder compliant to clause 49 of the same IEEE standard.
3) A 66-bit scrambler used to ensure that a data sequence has sufficiently many 0-1 transitions , so that it can run through high-speed noisy serial transmission channel.
4) A PCIE physical coding module.
5) The Ethernet module of Sun's OpenSparc T2 processor.

B. Experimental Results on Properly Designed Encoders

The 2nd and 6th rows of Table II compares the run time of checking $E \vDash PC$ between [1] and our approach. The run time of our approach are much larger than [1]. This is caused by checking the unique and non-unique conditions defined in Section IV and V.

The 3rd and 7th rows compare the discovered parameter values, and some minor differences are found on parameter p. This is caused by the different orders in checking various parameter combinations.

According to [9], p is used to constrain the reachable states, while d and l will affect the run time of building E^{-1} and its circuit area. To prove this, we compared the run time of building E^{-1} with all-solution SAT solver in the 4th and 8th rows of Table II, and also compared the area of E^{-1} in the 5th and 9th rows. These E^{-1}s were synthesized with DesignCompiler and LSI10 target library.

The results indicate that the differences in parameter p do not cause significant overhead in the run time of all-solution SAT solver and circuit area.

C. Experimental Results on Improperly Designed Encoders

To further show the usefulness of our algorithm, we need some improperly designed encoders without corresponding decoders.

TABLE II
EXPERIMENTAL RESULTS ON PROPERLY DESIGNED ENCODERS

		XGXS	XFI	scrambler	PCIE	T2 ethernet
[1]	time chk PC(sec)	0.49	59.19	2.52	1.46	35.17
	d, p, l	1,0,1	0,3,2	0,1,2	2,1,1	4,0,1
	run time allsat(sec)	1.16	1047.19	2.00	0.96	29.51
	area	765	19443	1455	398	648
Ours	time chk PC(sec)	1.32	88.68	7.23	2.73	84.47
	d, p, l	1,1,1	0,3,2	0,2,2	2,1,1	4,1,1
	run time allsat(sec)	1.38	1055.64	3.23	1.18	29.42
	area	773	19481	1455	400	535

TABLE III
EXPERIMENTAL RESULTS ON IMPROPERLY DESIGNED ENCODERS

	XGXS	XFI	scrambler	PCIE	T2 ethernet
Alg 1 result	$LP(1)$	$LL(2)$	$LL(2)$	$LP(1)$	$LP(1)$
Alg 2 result	$\neg PC$	NA	NA	$\neg PC$	$\neg PC$
time(sec)	1.23	44.58	3.26	1.67	21.49

We obtained these improperly designed encoders by modifying each benchmark's output statements, such that they can explicitly output the same letter for two different input letters. In this way, input letter i_n can never be uniquely determined by E's output sequence.

The 2nd row of Table III shows the result of Algorithm 1, while the 3rd row shows the result of Algorithm 2. The total run time is shown in the 4th row.

For XFI and scrambler, the result of Algorithm 1 is LL, which falsifies PC directly. So the result of Algorithm 2 is **NA**.

The results indicate that our algorithm always terminated, and recognized these modified incorrect encoders.

VIII. RELATED WORKS

A. Complementary Synthesis

The concept of complementary synthesis was first proposed by us [1] in ICCAD 2009. Its major shortcomings are that it is incomplete, and its run-time overhead of building complementary circuit is too large.

The incomplete problem has been addressed by this paper, while we [9] addresses the second shortcoming by simplifying the SAT instance with unsatisfiable core extraction before building complementary circuits.

B. The Completeness of Bounded Model Checking

Bounded model checking(BMC) is a model checking technology that considers only those paths of limited length. Many researchers try to find out complete approaches for BMC.

One line of research [6], [10] tries to find out a bound b, which can guarantee the correctness of a specification on all paths, if the specification is correct on all paths shorter than b.

The other line of research [11] tries to find out a pattern for induction, such that the correctness of a specification within any bound b implies the correctness on bound $b+1$.

Our approach achieves completeness without following these two approaches. Instead, we define two complement uniqueness conditions, LP and LL, and find out proper algorithms to check them.

C. Temporal Logic Synthesis

The temporal logic synthesis was first addressed by Clarke et.al [12] and Manna et.al [13]. But Pnueli et.al [14] pointed out that the complexity of LTL synthesis is double exponent.

One line of research [15]–[17] focuses on the so-called generalized reactive formulas of the form: $(\Box\Diamond p_1 \wedge \cdots \Box\Diamond p_m) \rightarrow (\Box\Diamond q_1 \wedge \cdots \Box\Diamond q_n)$. Complexity of solving synthesis problem for such formula is $O(N^3)$.

The other line of research focuses on finding efficient algorithm [18] for expensive safra determination algorithm [19] on an useful formula subset, or just avoiding it [20].

Based on these research works, some tools [21] that can handle small temporal formulas have been developed.

All these works assume a hostile environment, which seems too restrictive for many applications. So Fisman et.al [22], Chatterjee et.al [23] and Ummels et.al [24] proposed rational synthesis algorithm, which assumes that each agents act to achieve their own goals instead of failing each other.

D. Protocol Converter Synthesis

The protocol converter synthesis was first proposed by Avnit et.al [25] to automatically generate a translator between two different communication protocols. Avnit et.al [26] improved it with a more efficient design space exploration algorithm. The implementation of this tool is introduced in [27].

IX. Conclusions and Future Works

This paper proposes the first halting algorithm that checks whether a particular encoder E has corresponding decoder. Theoretical analysis and experimental results show that our approach always distinguishes correct encoders from their incorrect variants and halts properly.

One future work is to develop a debugging method to find out why E^{-1} does not exist. For the failure caused by loop-like path, we plan to develop a debugging mechanism based on our previous work on loop-like counterexample minimization [28].

Acknowledgment

The authors would like to thank the editors and anonymous reviewers for their hard work.

This work was fund by project 60603088 supported by National Natural Science Foundation of China.

This work was also supported by the Program for Changjiang Scholars and Innovative Research Team in University No IRT0614.

References

[1] ShengYu Shen, JianMin Zhang, Ying Qin, SiKun Li. Synthesizing Complementary Circuits Automatically. in ICCAD'09, pp 381-388, 2009.

[2] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, S. Malik. Chaff: Engineering an Efficient SAT Solver. In DAC'01, pp 530-535, 2001.

[3] João P. Marques Silva, Karem A. Sakallah. GRASP - a new search algorithm for satisfiability. in ICCAD'96, pp 220-227, 1996.

[4] E. Goldberg, Y Novikov. BerkMin: A Fast and Robust Sat-Solver. in DATE'02, pp 142-149, 2002.

[5] N. Eén, N. Sörensson. Extensible SAT-solver. in SAT'03, pp 502-518, 2003.

[6] D. Kroening, Ofer Strichman. Efficient Computation of Recurrence Diameters. in VMCAI'03, pp 298-309, 2003.

[7] Mealy, George H. A Method for Synthesizing Sequential Circuits. Bell Systems Technical Journal v 34, pp 1045-1079, 1955.

[8] *IEEE Standard for Information technology Telecommunications and information exchange between systems Local and metropolitan area networks Specific requirements Part 3: Carrier Sense Multiple Access with Collision Detection (CSMA/CD) Access Method and Physical Layer Specifications Amendment: Media Access Control (MAC) Parameters, Physical Layers, and Management Parameters for 10 Gb/s Operation*, IEEE Std. 802.3, 2002.

[9] ShengYu Shen, Ying Qin, KeFei Wang, LiQuan Xiao, JianMin Zhang, SiKun Li. Synthesizing Complementary Circuits Automatically. IEEE transaction on CAD of Integrated Circuits and Systems, 29(8):1191-1202, 2010.

[10] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, Yunshan Zhu. Symbolic Model Checking without BDDs. In TACAS'99, pp 193-207, 1999.

[11] Mary Sheeran, Satnam Singh, Gunnar Stalmarck. Checking Safety Properties Using Induction and a SAT-Solver. In FMCAD'00, pp 108-125, 2000.

[12] E.M. Clarke, E.A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In IBM Workshop on Logics of Programs,LNCS 131, pp 52-71, 1981.

[13] Z. Manna, P. Wolper. Synthesis of communicating processes from temporal logic specifications. ACM Trans. Prog. Lang. Sys., 6:68-93, 1984.

[14] A. Pnueli, R. Rosner. On the synthesis of a reactive module. In Proc. 16th ACM Symp. Princ. of Prog. Lang.,pages 179-190, 1989.

[15] E. Asarin, O. Maler, A. Pnueli, J. Sifakis. Controller synthesis for timed automata. In IFAC Symposium on System Structure and Control, pages 469-474. Elsevier, 1998.

[16] R. Alur, S. La Torre. Deterministic generators and games for LTL fragments. ACM Trans. Comput. Log., 5(1):1-25,2004.

[17] N. Piterman, A. Pnueli, Y. Saar, Synthesis of Reactive(1) Designs, in VMCAI'06, pp 364-380, 2006.

[18] O. Maler, D. Nickovic, A. Pnueli. On Synthesizing Controllers from Bounded-Response Properties. In CAV'07, pp 95-107, 2007.

[19] S. Safra. Complexity of Automata on Infinite Objects. PhD thesis, The Weizmann Institute of Science, Rehovot, Israel, March 1989.

[20] A. Harding, M. Ryan, P. Schobbens. A New Algorithm for Strategy Synthesis in LTL Games. in TACAS'05, pp 477-492, 2005.

[21] B. Jobstmann, S. Galler, M. Weiglhofer, R. Bloem. Anzu: A Tool for Property Synthesis. in CAV'07, pp 258-262, 2007.

[22] D Fisman, O Kupferman, Yoad Lustig. Rational Synthesis. in TACAS'10, pp 190-204, 2010.

[23] Chatterjee, K., Henzinger, T.A. Assume-guarantee synthesis. In TACAS'07, pp 261-275, 2007.

[24] Ummels, M. Rational behaviour and strategy construction in infinite multiplayer games. In FSTTCS'06, pp 212-223, 2006.

[25] K. Avnit, V. D'Silva, A. Sowmya, S. Ramesh, S. Parameswaran. A Formal Approach To The Protocol Converter Problem. in DATE'08, pp 294-299, 2008.

[26] K. Avnit, A. Sowmya. A formal approach to design space exploration of protocol converters. in DATE'09, pp 129-134, 2009.

[27] K. Avnit, A. Sowmya, J. Peddersen. ACS: Automatic Converter Synthesis for SoC Bus Protocols. in TACAS'10, pp 343-348, 2010.

[28] ShengYu Shen, Ying Qin, Sikun Li. Minimizing Counterexample of ACTL Property. in CHARME'05, pp 393-397, 2005.

Synthesis for Regular Specifications over Unbounded Domains

Jad Hamza*, Barbara Jobstmann†, Viktor Kuncak‡
*ENS Cachan, France †CNRS/Verimag, France, ‡EPFL, Switzerland

***Abstract*—Synthesis from declarative specifications is an ambitious automated method for obtaining systems that are correct by construction. Previous work includes synthesis of reactive finite-state systems from linear temporal logic and its fragments. Further recent work focuses on a different application area by doing functional synthesis over unbounded domains, using a modified Presburger arithmetic quantifier elimination algorithm. We present new algorithms for functional synthesis over unbounded domains based on automata-theoretic methods, with advantages in the expressive power and in the efficiency of synthesized code.**

Our approach synthesizes functions that meet given regular specifications defined over unbounded sequences of input and output bits. Thanks to the translation from weak monadic second-order logic to automata, this approach supports full Presburger arithmetic as well as bitwise operations on arbitrary length integers. The presence of quantifiers enables finding solutions that optimize a given criterion. Unlike synthesis of reactive systems, our notion of realizability allows functions that require examining the entire input to compute the output. Regardless of the complexity of the specification, our algorithm synthesizes linear-time functions that read the input and directly produce the output. We also describe a technique to synthesize functions with bounded lookahead when possible, which is appropriate for streaming implementations. We implemented our synthesis algorithm and show that it synthesizes efficient functions on a number of examples.

I. Introduction

Automated synthesis of systems from specifications is a promising method to increase development productivity. Automata-based methods have been the core technique for reactive synthesis of finite-state systems [1], [2], [3]. In this paper, we show that automata-based techniques can also be used to perform functional synthesis over unbounded data domains. In functional synthesis, we are interested in synthesizing functions that accept a tuple of input values (ranging over possibly unbounded domains), and generate a tuple of output values that satisfy a given specification. Our efforts are inspired in part by advances in software synthesis for bit-manipulating programs [4]. Our goal is to develop and analyze complete algorithms that require only a declarative specification as input. Recently, researchers have proposed [5] a technique for functional synthesis based on quantifier elimination of Presburger arithmetic.

In the previous approach, the functions generated by quantifier elimination can be inefficient if the input contains inequalities, possibly performing search over a very large space of integer tuples. Furthermore, this approach handles disjunctions by a transformation into disjunctive normal form. Finally, the specification language accepts integer arithmetic but not bitwise constructs on integers.

In this paper we present a synthesis procedure that is guaranteed to produce an efficient function that computes a solution of a given constraint on unbounded integers in time linear in the combined length of input and the shortest output, represented in binary. Moreover, our specification language supports not only Presburger arithmetic operations, but also bitwise operations and quantifiers. We achieve this expressive power by representing integers as sets in weak monadic second-order logic of one successor (WS1S) which is known to be more expressive than pure Presburger arithmetic [6], [7]. We use an off-the-shelf procedure, MONA [8], to obtain a deterministic automaton that represents a given WS1S specification.

As our central result, we show how to convert an arbitrary automaton recognizing the input/output relation into a function that reads the input sequence and produces an output sequence that satisfies the input/output relation. Consequently, we obtain functions that are guaranteed to run in linear-time on arbitrarily large integers represented as bit sequences. Assuming constant-time lookup of automaton transition, the running time of the synthesized functions is independent of the automaton size. These properties are a consequence of our algorithm, and we have also experimentally verified them on a number of examples. Our result solves the problem of synthesis of general WS1S specifications that are not necessarily causal. Our basic algorithm generates implementations that have $O(N)$ time and space complexity, where N is the number of bits of input and output. We show how to reduce space consumption to $O(\log N)$ if the time is increased to $O(N \log N)$.

We also examine synthesis for sub-classes of WS1S specifications that can be implemented using bounded memory. We introduce a class of implementations based on a finite union of asynchronous transducers, and show that they can be used to implement k-causal specifications as well as specifications in Presburger arithmetic without bitwise operations.

II. Examples

A. Parity Bit Computation

The goal of our first example is to illustrate the form of the functions produced by our synthesizer. For a non-negative integer x, let $x[k]$ denote the k-th least significant bit in the

This research was facilitated by the COST Action IC0901 *Rich Model Toolkit—An Infrastructure for Reliable Computer Systems* and the Dagstuhl Seminar on Software Synthesis, December 2009. The author list has been sorted according to the alphabetical order.

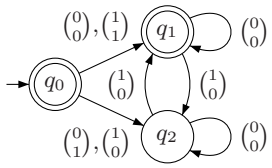
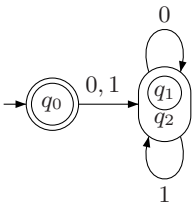


Fig. 1. Automaton A for parity specification between x and y

```
x: 0 1 1 0 1
y: 1 0 0 0 0
```

Fig. 2. Input x and output y satisfying parity specification



Transition	State	τ
$\{q_0\} \xrightarrow{0} \{q_1, q_2\}$	q_1	$(q_0, 0)$
$\{q_0\} \xrightarrow{0} \{q_1, q_2\}$	q_2	$(q_0, 1)$
$\{q_0\} \xrightarrow{1} \{q_1, q_2\}$	q_1	$(q_0, 1)$
$\{q_0\} \xrightarrow{1} \{q_1, q_2\}$	q_2	$(q_0, 0)$
$\{q_1, q_2\} \xrightarrow{0} \{q_1, q_2\}$	q_1	$(q_1, 0)$
$\{q_1, q_2\} \xrightarrow{0} \{q_1, q_2\}$	q_2	$(q_2, 0)$
$\{q_1, q_2\} \xrightarrow{1} \{q_1, q_2\}$	q_1	$(q_2, 0)$
$\{q_1, q_2\} \xrightarrow{1} \{q_1, q_2\}$	q_2	$(q_1, 0)$

Fig. 3. Automaton A' for computing parity y of input x

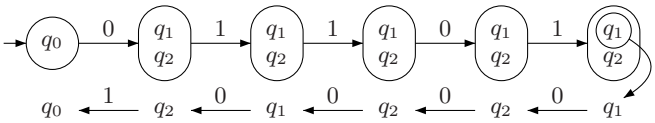


Fig. 4. Running synthesized function on input shown in Fig. 2

binary representation of x. (We write the binary digits starting with the least significant one on the left, so $\overline{11001}_2$ is a binary representation of 19.) Our first specification states that the first output bit, $y[0]$ indicates the parity of the number of one-bits in the input (Figure 2): $y[0] = |\{k \mid x[k] = 1\}|\%2$.

Consequently, the synthesized function must examine the entire input before emitting the first bit of the output.

One way to specify this computation is as follows. Let n_{max} have the property $\forall k > n_{max}.\, x[k] = 0$. To specify y, introduce first an auxiliary sequence of bits z such that

$$z[n] = |\{k \leq n \mid x[k] = 1\}|\%2$$

for all $n \leq n_{max}$, by defining $z[k+1]$ as xor of $z[k]$ and $x[k+1]$. Then define $y[0]$ to be $z[n_{max}]$.

Figure 1 shows the generated automaton A for this specification, accepting the words $\binom{x[0]}{y[0]}\binom{x[1]}{y[1]}\ldots\binom{x[n]}{y[n]}$ which satisfy the given relation between x and y. After applying our construction to compute a function from x to y, we obtain the input-determinstic automaton A' shown on the left of Figure 3, augmented with two labeling functions τ and ϕ. The automaton is the result of first projecting out the part of A' labels corresponding to the output, then applying the subset construction. Therefore, the labels in A' correspond to input bits, and the states are sets of states of the automaton A. Function τ tells us how to move backwards within a run of A' to construct an accepting run of the underlying automaton A; it thus recovers information lost in applying the projection to A. Finally, function ϕ tells us for every accepting state in A' at which state of A to start the backward reconstruction. The table on the right of Figure 3 shows τ for A': it maps every transition $S \xrightarrow{\sigma_i} S'$ of A' and every state $q' \in S'$ into a predecessor state $q \in S$, and a matching output value σ_o, such that $(q, (\sigma_i \cup \sigma_o), q')$ is a transition in the automaton A. We indicate function ϕ in A' by additional circles around individual states, e.g., $\phi(\{q_1, q_2\}) = q_1$. Figure 4 shows the run of A' on the input 01101. The synthesized function first runs the deterministic automaton A' (the upper part of Figure 4, ending in state $\{q_1, q_2\}$). The synthesized function then picks a state q according to ϕ (the state q_1 in case of our example), and runs backwards according to τ while computing the output bits. The lower part of Figure 4 shows the backward computation following τ defined in Figure 3; the backward run generates the bits 10000 of the output.

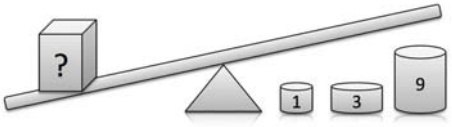


Fig. 5. Beam balance with three weights

B. Synthesizing Specialized Constraint Solvers

Our next examples illustrate a range of problems to which our synthesis technique applies. Consider first the beam balance (scale) depicted in Figure 5. We are interested in a function that tells us, for any object on the left-side of the beam, how to arrange the weights to balance the beam. We are given three weights, with 1, 3, and 9kg, respectively. We use the variable w for the weight of the unknown object. For each available weight i, we use two variables l_i to indicate whether the weight is placed on the left side and r_i to indicate it is placed on the right side of the beam. We obtain the constraint:

$$w + l_1 + 3l_3 + 9l_9 = r_1 + 3r_3 + 9r_9. \quad (1)$$

Because each weight can only be use at most once, we require that the solution also respects the following three constraints

$$l_1 + r_1 \leq 1,\ l_3 + r_3 \leq 1,\ l_9 + r_9 \leq 1. \quad (2)$$

When we give these four constraints to our tool, it compiles them into a function. The function accepts arbitrary input values and returns corresponding output values, performing computation in time linear in the number of bits in the input. E.g., if the object weights 11kg, then the program tells us that we should use Weight 1 on the left and Weight 3 and 9 on the right side to balance the beam. It is easy to verify that this response is correct by insertion into Equation 1 leading to $11 + 1 \cdot 1 = 3 \cdot 1 + 9 \cdot 1$. When asked for $w = 15$, the program correctly responds with "There is no output for your input."

C. Modifying Example to Minimize Output

Next, we consider a modified version of the balance example to show that neither inputs nor outputs need to be bounded. It also shows how to specify a function that minimizes the output. In the previous example, we could only balance objects up to 13kg because only one copy of each weight was available. Assume we want to balance arbitrary heavy objects with the minimal number of balance weights of 1, 3, and 9kg. We keep the constraint from Eqn. (1) and replace the

constraints in Eqn. (2) by a constraint that asks for a minimal solution:

$$\forall l'_1, l'_3, l'_9, r'_1, r'_3, r'_9.\quad \text{balance}(w, l'_1, l'_3, l'_9, r'_1, r'_3, r'_9) \rightarrow \\ \text{sum}(l_1, l_3, l_9, r_1, r_3, r_9) \leq \text{sum}(l'_1, l'_3, l'_9, r'_1, r'_3, r'_9)$$

where balance$(w, l'_1, l'_3, l'_9, r'_1, r'_3, r'_9)$ is the constraint obtained from Eqn. 1 by replacing l_i and r_i by l'_i and r'_i, respectively, and sum refers to the sum of the listed variables. This constraint requires that every other solution that would also balance the scale for the given object has to use more weights than the solution returned.

The newly synthesized program gives correct answers for arbitrary large natural numbers. E.g., let us assume the object weighs 123451234512345123451234567 89kg, then the program tells us to take 137168038347050137168038408 8 times Weight 9 on the right side and once Weight 3 on the left side.

D. Finding Approximate Solutions

Consider the constraint $6x+9y = z$, where z is the input and x, y are inputs. The solution exists only when z is a multiple of 3, so we may wish to find x, y that minimizes $|6x + 9y - z|$, using a similar encoding with quantifiers as in the previous example. The support for disjunctions allows us to encode the absolute value operator that is useful for finding approximate solutions. The tool synthesizes a function that given a value of z, computes x, y to be as close to z as possible. For example, given the input 104, the tool outputs $x = 13$ and $y = 3$.

E. Folding and Inverting Computations

Consider the Syracuse algorithm function, whose one step is given by $f(x) =$ if $(2 \mid x)$ then $x/2$ else $3x+1$. Consider a relation on integers corresponding to iterating f six times: $r(x, y) \leftrightarrow f^6(x) = y$. (We could use such function to speed-up experimental verification of the famous $3n + 1$ conjecture that states $\forall x > 0.\exists n.f^n(x) = 1$.) When we use $r(x, y)$ as the specification and indicate x as input and y as output, our synthesizer generates a function that accepts a sequence of bits of x and outputs in linear time a sequence of bits of y that is given by 6-fold iteration of f. Note that, if the synthesis from a specification (e.g. $y = f^n(x)$) succeeds, the runtime of the computation is independent of n and is linear in the number of bits of x. Therefore, our approach can effectively fold n iterations of f into one linear-time function on the binary representations of inputs and outputs.

F. Processing Sequences of Bits

We next illustrate the use of specification of unbounded numbers in simple signal processing task. Suppose we have an input signal X with discrete values in the range $\{0, 1, 2, \ldots, 15\}$ and we wish to compute a smoothed output signal Y by averaging signal values with its neighbors, using the formula $Y_i = (X_{i-1} + 2X_i + X_{i+1})$ div 4. We specify this function in WS1S as a relation between unbounded integers x and y, where we reserve 4 bits for value of the signal at each time point (see Figure 6). For constants a, b, let $x[k+a, k+b]$ denote the number represented by the subrange of digits of x between $k + a$ and $k + b$:

$$x[k+a, k+b] = x[k+a] + 2x[k+a+1] + \ldots + 2^{b-a}x[k+b]$$

We define the smoothing relation between numbers x and y by:

$$\forall i.\ \ (4|i) \rightarrow y[i{+}4..i{+}7] = \\ (x[i..i{+}3] + 2x[i{+}4..i{+}7] + x[i{+}8..i{+}11])\ \text{div}\ 4$$

Our synthesizer generates a function that, given the sequence of bits x, produces a sequence of bits y. Figure 7 shows an input signal (dotted line) and the resulting smoothed signal (full line) that results after we applied the linear-time function synthesized by our tool to the input.

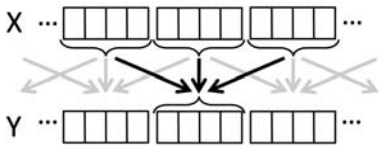


Fig. 6. Averaging signal values

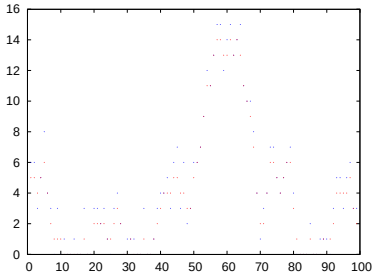


Fig. 7. The result of applying the synthesized function that computes a smoothed version of a signal. The function on an arbitrarily long signal was specified in WS1S.

III. Preliminaries

A. Words and Automata

Given a finite set of variables V, we use Σ_V to denote the alphabet $\Sigma_V = 2^V$. We omit V in Σ_V if it is clear from the context. When used as a letter, we denote $\emptyset \in \Sigma_V$ by $\mathbf{0}$. Given a finite word $w \in \Sigma_V^*$, we use $|w|$ to denote the length of w, and w_i to denote the letter on the i-th position of w. By ε we denote the empty word, of length zero. Given a partitioning of V into the sets I and O and a letter $\sigma \in \Sigma_V$, we use $\sigma|_I$ to denote the projection of σ to I, i.e., $\sigma|_I = \sigma \cap I$. We extend projection in usual sense to words and languages.

A *finite automaton* A over a finite set of variables V is a tuple $(\Sigma, Q, \text{init}, F, T)$, where $\Sigma = 2^V$ is the *alphabet*, Q is a finite set of *states*, init $\in Q$ is the *initial state*, $T \subseteq Q \times \Sigma \times Q$ is the *transition relation*, and $F \subseteq Q$ is a set of *final states*. Automaton A is *deterministic*, if for all transitions $(q_1, \sigma_1, q'_1), (q_2, \sigma_2, q'_2) \in T$, $q_1 = q_2$ and $\sigma_1 = \sigma_2$ implies $q'_1 = q'_2$ holds. A is *complete*, if for all states $q \in Q$ and letters $\sigma \in \Sigma$, there exists a state $q' \in Q$ such that $(q, \sigma, q') \in T$. Note that if A is deterministic and complete T describes a total function from Q and Σ to Q.

$$
\begin{array}{rcl}
F & ::= & F \wedge F \mid F \vee F \mid \neg F \mid t_N < t_N \mid t_N = t_N \\
& \mid & t_N[t_P] \mid t_P < t_P \mid t_P = t_P \mid (C \,|\, t_N) \mid t_N \underset{t_P}{\sim} t_N \\
& \mid & \forall_{\mathsf{pos}} k.F \mid \exists_{\mathsf{pos}} k.F \mid \forall x.F \mid \exists x.F \\
t_N & ::= & x \mid C \mid t_N + t_N \mid C \cdot t_N \mid t_N \,\mathsf{div}\, C \mid t_N \,\%\, C \\
& \mid & (t_N \veebar t_N) \mid (t_N \barwedge t_N) \mid t_N \ll C \mid t_N \gg C \\
& \mid & 2^{t_P} \mid t_N[t_P..^{+}C] \mid t_N[0..t_P] \\
t_P & ::= & k \mid C \mid k + C \mid k \dot{-} C \mid \mathsf{maxBit}(t_N) \\
C & ::= & \text{non-negative integer constant}
\end{array}
$$

Fig. 8. Syntax of WS1S where sets denote natural numbers (T_N) and elements denote positions (T_P) in binary representations of numbers

Given an automaton $A = (\Sigma, Q, \text{init}, F, T)$ and a state $q \in Q$, we use A_q to refer to the automaton (Σ, Q, q, F, T) that has the same structure as A but starts at q.

A *run* ρ of A on a word $w \in \Sigma^*$ is a sequence of states $q_1 \dots q_{|w|+1}$ such that (i) $q_1 = \text{init}$ and (ii) for all $1 \leq i \leq |w| : (q_i, w_i, q_{i+1}) \in T$. A run is *accepting* if $q_{|w|+1} \in F$. We say w *is accepted by* A if there exists a run of A on w that is accepting. We denote by $\mathcal{L}(A) \subseteq \Sigma^*$ the set of words accepted by A.

The *exhaustive run* ρ of A on a word $w \in \Sigma^*$ is a sequence of sets of states $S_1 \dots S_{|w|+1}$ such that (i) $S_1 = \{\text{init}\}$ and (ii) for all $1 \leq i \leq |w|$, $S_{i+1} = \{q' \in Q \mid \exists\, q \in S_i, (q, w_i, q') \in T\}$. An exhaustive run is *accepting* if $S_{|w|+1} \cap F \neq \emptyset$. Note that if A is deterministic, then the run of A on a word w is unique and the elements in the exhaustive run of A on w are singletons.

Lemma 1: For an automaton A with a set of states Q, computing an exhaustive run of A for a word $w \in \Sigma^*$ can be done in time $O(|Q| \cdot |w|)$ for a non-deterministic A, and can be done in time $O(|w|)$ for a deterministic A.

Given an automaton $A = (\Sigma_V, Q, \text{init}, F, T)$ over variables V and a set $I \subset V$, the *projection of A to I*, denoted by $A|_I$, is the automaton $(\Sigma_I, Q, \text{init}, F, T_I)$ with $T_I = \{(q, \sigma_I, q') \in Q \times \Sigma_I \times Q \mid \exists \sigma \in \Sigma_V, (q, \sigma, q') \in T \wedge \sigma|_I = \sigma_I\}$. In the remainder, we fix I to be the set of input and O to be the set of output variables.

B. WS1S as extension of Presburger Arithmetic

Figure 8 shows the syntax of weak monadic second-order logic of one successor, which we use as our specification language for unbounded non-negative integers. The logic contains all integer linear arithmetic operations and quantifiers, thus subsuming Presburger arithmetic. Furthermore, it contains the expression $x[k]$ to extract the k-th least significant bit of the number x. It is also possible to find a c-successor of position k, with notation $k + c$, as well as the c-predecessor, with notation $k \dot{-} c$, denoting the position $\max(k - c, 0)$. Together with quantification over positions, this allows the specification of arbitrary uniform bitwise relations on integer variables. To illustrate the expressive power of WS1S, we introduce shorthands for some of the constraints that can be defined in this way: bitwise operations ($\barwedge$, $\veebar$), left and right shifting ($\ll$, $\gg$), a sub-word of length c at position k of a given integer x (denoted $x[k..^{+}c]$), congruence modulo 2^p (denoted $x \sim_p y$), the initial prefix of an integer $x[0..k]$, the integer 2^p for a position p, and the smallest p such that $x < 2^p$, denoted $\mathsf{maxBit}(x)$.

C. Amortized Cost of Synthesis

We describe the cost of synthesis and synthesized program in a unified framework, by considering the entire amortized cost of applying a given specification a on a series of inputs $b_1, \dots, b_n$. Let f be a function with two arguments, so that $f(a, b) = c$ if the input-output pair (b, c) satisfies the specification a. We implement function f using a function of the form $g(a, b, s) = (f(a, b), s')$ that computes f and updates its local state from s to s'. We assume a fixed initial state s_0. The presence of local state can make the computation more efficient on a series of inputs. This framework accounts for simple cases such as memoization and caching, as well as the more general case of on-the-fly specialization.

Given the specification a and the inputs $b_1, \dots, b_n$ we define $s_i = g(a, b_i, s_{i-1})$ for $i \in \{1, \dots, n\}$. Let $g'(a, b, s)$ denote the time to compute $g(a, b, s)$. Let $|x|$ denote the length of value x. We define the amortized cost of g on inputs $a; b_1, \dots, b_n$ by $\frac{1}{n}\sum_{i=1}^{n} g'(a, b_i, s_{i-1})$. Our main complexity measure is then $c(s_a, s_b, n)$, which we define as the maximum amortized cost over all $a; b_1, \dots, b_n$ for which $|a| \leq s_a$ and $|b_i| \leq s_b$ for all i.

Observe that $c(s_a, s_b, 1)$ is simply the complexity of running function f once on inputs of size s_a and s_b, respectively. Another useful measure, of particular interest in synthesis, is $c_\infty(s_a, s_b) = \lim_{n \to \infty} c(s_a, s_b, n)$, which amortizes any pre-computation that happens in finitely many steps. We next present several examples to illustrate the cost function $c_\infty(s_a, s_b)$ for implementations of several problems.

Example 1 (Finding an enclosing interval): Consider the problem of computing the smallest interval enclosing a given number. More precisely, the goal is to compute $f([x_1, \dots, x_m], y) = (L, U)$ where $L = \max\{x_i \mid x_i \leq y\}$ and $U = \min\{x_j \mid y \leq x_j\}$ given an unordered list of numbers $x_1, \dots, x_m$ (with the result arbitrary if the $\max$ or $\min$ expressions above are not defined). In this example, we assume that each number takes constant space to represent, so $|[x_1, \dots, x_m]| = m$ and $|y| = 1$. An algorithm for one invocation can simply make a single pass through the list, computing the current $\max$ of lower bounds of y and the current $\min$ of the upper bounds up to a given position in the list. This gives the worst-case complexity m of the algorithm. If we use this algorithm as the implementation g (without making use of state), we obtain $c_\infty(m, 1)$ of $O(m)$.

Consider next an alternative implementation, given by $g'([x_1, \dots, x_m], y, s)$, which behaves as follows: on the first invocation, $g([x_1, \dots, x_m], y, s_0)$, builds a balanced binary search tree storing the set of numbers $x_1, \dots, x_m$ in time $O(m \log m)$, and returns this tree in the resulting state s'. On subsequent invocations, g uses this tree to find the enclosing interval (L, U), which can be done in time $O(\log m)$ by doing

a lookup in the tree. Therefore, we obtain that n invocations require $O(m \log m + n \log m)$, which gives $c(m,1,n) \in O(\frac{1}{n}(m \log m) + \log m)$ and $c_\infty(m,1) = O(\log m)$. Thus, we have seen that precomputation improves the amortized time $c_\infty(m,1)$ from $O(m)$ to $O(\log m)$. □

IV. Synthesis Algorithm

A. Constructing Specification Automaton

The input to our algorithm is a WS1S formula G whose free variables $z_1, \ldots, z_r$ denote unbounded integers. We assume a partitioning of the index set $\{1, \ldots, r\}$ into inputs I and the outputs O. In the first step, our algorithm constructs a deterministic specification automaton A accepting words in the alphabet $\Sigma_{I \cup O}$. We use a standard automaton construction [9] and obtain an automaton A characterizing the satisfying assignments of G, i.e. whose language $\mathcal{L}(A)$ contains precisely the words $\sigma_0 \sigma_1 \ldots \sigma_n \in \Sigma^*_{I \cup O}$ for which G holds in the variable assignment $(z_1, \ldots, z_r)$ in which the k-th least significant bit of z_i is one iff $0 \leq k \leq n$ and $i \in \sigma_k$. We use $\mathcal{L}(G)$ to denote the language over $\Sigma_{I \cup O}$ characterizing the satisfying assignments of G. From this correctness property it follows that $w \in \mathcal{L}(A)$ implies $w\mathbf{0}^p \in \mathcal{L}(A)$ for every $p \geq 0$.

B. Overview

All subsequent steps of our algorithm work with the specification automaton A and do not depend on how this automaton was obtained. Given A, our goal is to construct a function that computes, for a given sequence of inputs bits a corresponding sequence of output bits such that the combined word is accepted by the deterministic automaton.

Note that we seek an implementation that works uniformly for *arbitrarily long sequences of bits*, which means that it is not possible to pre-compute all possible input/output pairs.

We show our construction in several steps. First, we assume that we are only interested in outputs whose length does not exceed the length of inputs. For this case we start by describing a less time-efficient implementation (Subsection IV-C) that depends on the size of A, then describe an efficient version, showing that we can avoid the dependence on the size of A (Subsection IV-D). Finally, we show how to lift the assumption that the outputs are no longer than the inputs (Subsection IV-E).

C. Input-Bounded Synthesis of Unspecialized Implementations

In the first version of our solution we assume that, given an input bit sequence, we seek an output sequence of the *same length* such that the input and output pair are accepted by the specification automaton A.

Our unspecialized implementation P_{unspec} simulates the given automaton $A = (\Sigma_{I \cup O}, Q, \text{init}, F, T)$ on the input word $w \in \Sigma^*_I$ and tries to find an accepting run. P_{unspec} first constructs the exhaustive run $\rho = S_1 \ldots S_{|w|+1}$ of the projected automaton $A|_I$ on w (see preliminaries for the definition of automaton projection and exhaustive run). If ρ is not accepting, then there is no matching output word and P_{unspec} terminates. Otherwise, P_{unspec} picks a state $q_{|w|+1}$ in $S_{|w|+1} \cap F$ and constructs an accepting run $q_1 \ldots q_{|w|+1}$ of A and the output word v by proceeding backwards over i, from $i = |w|$ to $i = 1$, as follows: it picks $v_i \in \Sigma_O$ and $q_i \in S_i$ such that $(q_i, w_i \cup v_i, q_{i+1}) \in T$. When it reaches one of the initial states in S_1, the result is an accepting run of the automaton A; the desired output is the sequence $v_1 \ldots v_{|w|}$ of the output components of the labels in the reconstructed run.

The P_{unspec} implementation repeats the above construction for each input word w. From Lemma 1 we obtain the amortized cost of P_{unspec}.

Lemma 2: If s_A denotes the size of the input automaton A and s_w denotes the size of the input word, then the unspecialized implementation P_{unspec} solves the synthesis for input-bounded specifications in amortized time $c(s_A, s_w, n)$ of $O(s_A \cdot s_w)$ (consequently, $c_\infty(s_A, s_w)$ is also $O(s_A \cdot s_w)$).

D. Input-Bounded Synthesis of Specialized Implementations

We next present our main construction (illustrated in the Example II-A), which avoids the dependence of the running time of computation of on the (potentially large) number of states of the automaton A. To obtain an implementation with optimal runtime, we transform the given automaton A into an input-deterministic automaton A' using the subset construction on the projection $A|_I$. The challenge is to extend the subset construction with the additional labeling functions that allow us to efficiently reconstruct an accepting run of A from an accepting run of A'. Given such additional information, our specialized implementation P_{spec} runs A' on the input w and uses the labeling to construct the output v.

Our construction introduces two labeling functions, ϕ and τ. The function ϕ maps each accepting state S of A' into one state $q \in S$ that is accepting in A. The τ function indicates how to move backwards through the accepting run; it maps each transition (S, σ_i, S') of A' and a state $q' \in S'$ into a pair $(q, \sigma_o) \in S \times \Sigma_o$ of new a state and an output letter, such that $(q, \sigma_i \cup \sigma_o, q')$ is a transition of the original automaton A.

Definition of synthesized data structure A', ϕ, τ. Given an automaton $A = (\Sigma_{I \cup O}, Q, \text{init}, F, T)$, we construct an automaton $A' = (\Sigma_I, Q', \text{init}', F', T')$ and two labeling functions $\phi : F' \to Q$ and $\tau : (T' \times Q) \to (Q \times \Sigma_O)$ such that (i) A' is deterministic, (ii) $\mathcal{L}(A)|_I = \mathcal{L}(A')$, and (iii) for every word $u \in \mathcal{L}(A')$ with an accepting run $S_1 \ldots S_{n+1}$ of A', there exists a word $w \in \mathcal{L}(A)$ with $w|_I = u$ and an accepting run $q_1 \ldots q_{n+1}$ of A such that $\phi(S_{n+1}) = q_{n+1}$ and for all $1 \leq i \leq n$, $(q_i, w_i|_O) \in \tau((S_i, u_i, S_{i+1}), q_{i+1})$. We define A' as follows:

$$
\begin{array}{lcl}
Q' & = & 2^Q \\
\text{init}' & = & \{\text{init}\} \\
F' & = & \{S \in Q' \mid S \cap F \neq \emptyset\} \\
T' & = & \{(S, i, S') \in Q' \times \Sigma_I \times Q' \mid \\
 & & \quad S' = \{q' \mid \exists q, \sigma . (q, \sigma, q') \in T \wedge q \in S \wedge \sigma|_I = i\}\}
\end{array}
$$

We define $\phi : F' \to Q$ such that if $S \in F'$ then $\phi(S) \in S \cap F$; such value exists by definition of F'.

We define $\tau : (T' \times Q) \to (Q \times \Sigma_O)$ for $(S, i, S') \in T'$ and $q' \in S'$ as follows. By definition of T', there exists a transition

$(q, \sigma, q') \in T$ of the original automaton such that $\sigma|_I = i$. We pick an arbitrary such transition and define $\tau((S, i, S'), q') = (q, \sigma|_O)$.

Computing A' and τ through automata transformations. In our implementation, we represent both A' and τ in one automaton, which we compute using the following sequence of automata transformations. Because τ refers to sets of transitions, we first turn each transition of A into a state, i.e, given $A = (\Sigma_{I\cup O}, Q, \text{init}, F, T)$, we construct an automaton $B = (\Sigma_{I\cup O}, Q_B, \text{init}_B, F_B, T_B)$ such that

$$\begin{array}{lcl} \text{init}_B & = & (q, \sigma, \text{init}_A) \quad \text{for arbitrarily chosen } q, \sigma \\ Q_B & = & \{\text{init}_B\} \cup T \\ F_B & = & \{(q, \sigma, q') \in Q_B \mid q' \in F\} \\ T_B & = & \{(t, \sigma, t') \in Q_B \times \Sigma_{I\cup O} \times Q_B \mid \\ & & \quad \exists q, q', q'' \in Q.\ \exists \sigma' \in \Sigma_{I\cup O}. \\ & & \quad t = (q, \sigma', q') \text{ and } t' = (q, \sigma, q'')\}. \end{array}$$

Next, we project B to I, i.e., we replace every transition (q, σ, q') in B by (q, σ_I, q'). Finally, we obtain automaton C by determinizing $B|_I$ using the classical subset construction. Now, every reachable state in C (other than init_B) corresponds to a transition in A'. Assume we are given a state q_c in C, then q_c has the form $\{(q_1, \sigma_1, q_1'), \ldots, (q_k, \sigma_k, q_k')\}$ with $\forall i, j, \sigma_i|_I = \sigma_j|_I = \sigma_I$ and corresponds to the transition (t, σ_I, t') in A', where $t = \{q_i \mid 1 \le i \le k\}$ and $t' = \{q_i' \mid 1 \le i \le k\}$. So, every state q_c in C defines a labeling function for the corresponding transition (t, σ_I, t') that maps every state $q_i' \in t'$ to a set of available pairs $(q_i, \sigma_i|_O)$. Our final labeling function τ picks for each state q_i one of the available pairs.

Specialized implementation and its complexity. The specialized implementation P_{spec} runs A' on the input word w and constructs a run $\rho = S_1 \ldots S_{|w|+1}$. If ρ is not accepting, then there is no matching output word and the function terminates. Otherwise, it computes an accepting run $q_1 \ldots q_{|w|+1}$ of A and the output word v as follows: $\phi(S_{|w|+1}) = q_{|w|+1}$ and, for all $1 \le i \le |w|$, $(q_i, v_i) = \tau((S_i, w_i, S_{i+1}), q_{i+1})$.

The following theorem states the correctness of P_{spec} and follows by construction.

Theorem 1: Consider an automaton A and an input $w_1 \ldots w_n$. Then if there exists an output $v_1 \ldots v_n$ such that $(w_1 \cup v_1) \ldots (w_n \cup v_n)$ is accepted by A, then P_{spec} computes one such output $v_1 \ldots v_n$. If there is no corresponding output then P_{spec} indicates that there is no output.

The following theorem states that our construction achieves the desired linear-time behavior and independence from the size of the initial automaton. The construction of A', ϕ, τ takes time singly exponential in the size of the automaton, but is done only once, so it is amortized for each invocation of the automaton. Extracting the output for a given input takes time independent of the number of states in A' because A' and τ have deterministic transitions.

Theorem 2: If s_A denotes the size of the specification automaton A and s_w denotes the size of the input word, then P_{spec} solves the synthesis for input-bounded specifications in amortized time $c(s_A, s_w, n)$ of $O(\frac{1}{n}2^{s_A} + s_w)$. Consequently, the amortized time $c_\infty(s_A, s_w)$ as the number of queries approaches infinity is $O(s_w)$.

E. Extending Synthesis to Arbitrary Regular Specifications

In this section we extend the result of the previous section to allow computing an output that satisfies the specification even if the output has a larger number of bits than the input. Consider the simple specification $x < y$, where x is the input and y is the output. Given the input $\overline{111}_2$ of length three (representing the number 7), every value of output satisfying the specification has the length at least four.

To adapt the solution in the previous section to the full synthesis problem we generalize the notion of acceptance to take into account any number of zeros that could be appended to the input without changing the meaning of the input. Therefore, if the automaton A' finishes reading the input word and none of the states reached in the last step are accepting, it checks whether one of the states can reach an accepting state while reading only the input letter $\mathbf{0}$. The closure with the input $\mathbf{0}$ can be computed in polynomial time by computing the states that are backward-reachable from an accepting state using only edges with input label $\mathbf{0}$.

To be able to emit the appropriate segment of the output, the backward-reachability computation keeps, for every state, an output word that leads to an accepting state. We use the function $\psi : Q \to \Sigma_O^* \cup \{\bot\}$ to store these words, where Q are the states of the specification automaton A. We write $\psi(q) = \bot$ to denote that there is no input word $w \in \mathbf{0}^*$ that is accepted starting from q. Formally, given the automaton $A = (\Sigma_{I\cup O}, Q, \text{init}, F, T)$, we set $\psi = \psi^{|Q|}$ and define ψ^i inductively: for all $q \in Q$:

(i) $\psi^0(q) = \begin{cases} \varepsilon & \text{if } q \in F \\ \bot & \text{otherwise} \end{cases}$

(ii) let R^i be the set of states q for which $\psi^i(q) \neq \bot$,

$$\psi^{i+1}(q) = \begin{cases} \psi^i(q) & \text{if } q \in R^i \\ \sigma|_O \psi^i(q') & \text{elsif } \exists (q, \sigma, q') \in T : \sigma|_I = \mathbf{0} \wedge q' \in R^i, \\ \bot & \text{otherwise.} \end{cases}$$

Observe that if $\psi(q) \neq \bot$ then $\psi(q)$ is a word of length bounded by the number of states of the specification automaton A. Therefore, the maximal amount by which the output is longer than the input is bounded by the size of the specification automaton.

To recognize leading zeros, we adapt the final states F' of A' (computed as for P_{spec} in the previous section) and extend the labeling function ϕ as follows. Let $\mathsf{fin}(S) = \{q \in S \mid \psi(q) \neq \bot\}$ be the states in S that can reach input on zeros.

$$\begin{array}{l} F' \quad = \{S \in Q' \mid \mathsf{fin}(S) \neq \emptyset\} \\ \phi(S) = q \in \mathsf{fin}(S) \text{ s.t. } |\psi(q)| = \min\{|\psi(q') \mid q' \in \mathsf{fin}(S)|\} \end{array}$$

Note that the function $\phi(S)$ chooses one of the states that lead to an accepting state with an output word of minimal length.

The implementation and its time complexity. Given an input word $w_1 \ldots w_n$, the implementation P_{gspec} generates, as P_{spec} in the previous Subsection (IV-D), a run $S_1, \ldots, S_{n+1}$.

a lookup in the tree. Therefore, we obtain that n invocations require $O(m \log m + n \log m)$, which gives $c(m,1,n) \in O(\frac{1}{n}(m \log m) + \log m)$ and $c_\infty(m,1) = O(\log m)$. Thus, we have seen that precomputation improves the amortized time $c_\infty(m,1)$ from $O(m)$ to $O(\log m)$. □

IV. Synthesis Algorithm

A. Constructing Specification Automaton

The input to our algorithm is a WS1S formula G whose free variables $z_1, \ldots, z_r$ denote unbounded integers. We assume a partitioning of the index set $\{1, \ldots, r\}$ into inputs I and the outputs O. In the first step, our algorithm constructs a deterministic specification automaton A accepting words in the alphabet $\Sigma_{I \cup O}$. We use a standard automaton construction [9] and obtain an automaton A characterizing the satisfying assignments of G, i.e. whose language $\mathcal{L}(A)$ contains precisely the words $\sigma_0 \sigma_1 \ldots \sigma_n \in \Sigma^*_{I \cup O}$ for which G holds in the variable assignment $(z_1, \ldots, z_r)$ in which the k-th least significant bit of z_i is one iff $0 \leq k \leq n$ and $i \in \sigma_k$. We use $\mathcal{L}(G)$ to denote the language over $\Sigma_{I \cup O}$ characterizing the satisfying assignments of G. From this correctness property it follows that $w \in \mathcal{L}(A)$ implies $w\mathbf{0}^p \in \mathcal{L}(A)$ for every $p \geq 0$.

B. Overview

All subsequent steps of our algorithm work with the specification automaton A and do not depend on how this automaton was obtained. Given A, our goal is to construct a function that computes, for a given sequence of inputs bits a corresponding sequence of output bits such that the combined word is accepted by the deterministic automaton.

Note that we seek an implementation that works uniformly for *arbitrarily long sequences of bits*, which means that it is not possible to pre-compute all possible input/output pairs.

We show our construction in several steps. First, we assume that we are only interested in outputs whose length does not exceed the length of inputs. For this case we start by describing a less time-efficient implementation (Subsection IV-C) that depends on the size of A, then describe an efficient version, showing that we can avoid the dependence on the size of A (Subsection IV-D). Finally, we show how to lift the assumption that the outputs are no longer than the inputs (Subsection IV-E).

C. Input-Bounded Synthesis of Unspecialized Implementations

In the first version of our solution we assume that, given an input bit sequence, we seek an output sequence of the *same length* such that the input and output pair are accepted by the specification automaton A.

Our unspecialized implementation P_{unspec} simulates the given automaton $A = (\Sigma_{I \cup O}, Q, \text{init}, F, T)$ on the input word $w \in \Sigma^*_I$ and tries to find an accepting run. P_{unspec} first constructs the exhaustive run $\rho = S_1 \ldots S_{|w|+1}$ of the projected automaton $A|_I$ on w (see preliminaries for the definition of automaton projection and exhaustive run). If ρ is not accepting, then there is no matching output word and P_{unspec} terminates. Otherwise, P_{unspec} picks a state $q_{|w|+1}$ in $S_{|w|+1} \cap F$ and constructs an accepting run $q_1 \ldots q_{|w|+1}$ of A and the output word v by proceeding backwards over i, from $i = |w|$ to $i = 1$, as follows: it picks $v_i \in \Sigma_O$ and $q_i \in S_i$ such that $(q_i, w_i \cup v_i, q_{i+1}) \in T$. When it reaches one of the initial states in S_1, the result is an accepting run of the automaton A; the desired output is the sequence $v_1 \ldots v_{|w|}$ of the output components of the labels in the reconstructed run.

The P_{unspec} implementation repeats the above construction for each input word w. From Lemma 1 we obtain the amortized cost of P_{unspec}.

Lemma 2: If s_A denotes the size of the input automaton A and s_w denotes the size of the input word, then the unspecialized implementation P_{unspec} solves the synthesis for input-bounded specifications in amortized time $c(s_A, s_w, n)$ of $O(s_A \cdot s_w)$ (consequently, $c_\infty(s_A, s_w)$ is also $O(s_A \cdot s_w)$).

D. Input-Bounded Synthesis of Specialized Implementations

We next present our main construction (illustrated in the Example II-A), which avoids the dependence of the running time of computation of on the (potentially large) number of states of the automaton A. To obtain an implementation with optimal runtime, we transform the given automaton A into an input-deterministic automaton A' using the subset construction on the projection $A|_I$. The challenge is to extend the subset construction with the additional labeling functions that allow us to efficiently reconstruct an accepting run of A from an accepting run of A'. Given such additional information, our specialized implementation P_{spec} runs A' on the input w and uses the labeling to construct the output v.

Our construction introduces two labeling functions, ϕ and τ. The function ϕ maps each accepting state S of A' into one state $q \in S$ that is accepting in A. The τ function indicates how to move backwards through the accepting run; it maps each transition (S, σ_i, S') of A' and a state $q' \in S'$ into a pair $(q, \sigma_o) \in S \times \Sigma_o$ of new a state and an output letter, such that $(q, \sigma_i \cup \sigma_o, q')$ is a transition of the original automaton A.

Definition of synthesized data structure A', ϕ, τ. Given an automaton $A = (\Sigma_{I \cup O}, Q, \text{init}, F, T)$, we construct an automaton $A' = (\Sigma_I, Q', \text{init}', F', T')$ and two labeling functions $\phi : F' \to Q$ and $\tau : (T' \times Q) \to (Q \times \Sigma_O)$ such that (i) A' is deterministic, (ii) $\mathcal{L}(A)|_I = \mathcal{L}(A')$, and (iii) for every word $u \in \mathcal{L}(A')$ with an accepting run $S_1 \ldots S_{n+1}$ of A', there exists a word $w \in \mathcal{L}(A)$ with $w|_I = u$ and an accepting run $q_1 \ldots q_{n+1}$ of A such that $\phi(S_{n+1}) = q_{n+1}$ and for all $1 \leq i \leq n$, $(q_i, w_i|_O) \in \tau((S_i, u_i, S_{i+1}), q_{i+1})$. We define A' as follows:

$$
\begin{array}{lcl}
Q' & = & 2^Q \\
\text{init}' & = & \{\text{init}\} \\
F' & = & \{S \in Q' \mid S \cap F \neq \emptyset\} \\
T' & = & \{(S, i, S') \in Q' \times \Sigma_I \times Q' \mid \\
 & & \quad S' = \{q' \mid \exists q, \sigma . (q, \sigma, q') \in T \wedge q \in S \wedge \sigma|_I = i\}\}
\end{array}
$$

We define $\phi : F' \to Q$ such that if $S \in F'$ then $\phi(S) \in S \cap F$; such value exists by definition of F'.

We define $\tau : (T' \times Q) \to (Q \times \Sigma_O)$ for $(S, i, S') \in T'$ and $q' \in S'$ as follows. By definition of T', there exists a transition

idea then is to avoid storing all states $q_0, \ldots, q_N$ of the forward run, and instead compute them on demand, storing only a sparse subsequence $q_{i_0}, q_{i_1}, \ldots, q_{i_m}$ where $m = \lceil \log N \rceil$. Let p denote the current position in the backward run of the synthesized function. The synthesized function maintains the invariant $0 = i_0 \leq i_1 \leq \ldots \leq i_m \leq p$. Initially it sets $i_k \approx N(1 - 2^{-k})$. To move back from p to $p-1$, it re-runs the forward automaton from the largest i_k for which $i_k < p$, and redistributes $i_{k+1}, \ldots, i_m$, similarly as for the initial run, maintaining the ordering and the decreasing geometric progression of distances $i_{k+j+1} - i_{k+j}$. Because each position pointer i_j requires $O(\log N)$ space and there are $\log N$ of them, this implementation needs $O(\log^2 N)$ space. A run that updates pointers i_{k+i} for $i \geq 0$ re-reads 2^{-k} fraction of the input and is called 2^k times, so the total time is $O(N \log N)$.

Unions of asynchronous transducers. An *(asynchronous) transducer* $M = (A, \lambda, \varphi)$ over input variables I and output variables O consists of (1) a deterministic automaton $A = (\Sigma_I, Q, \text{init}, F, T)$ and (2) two labeling functions $\lambda : T \to \Sigma_O^*$ and $\varphi : F \to \Sigma_O^*$. A (more conventional) *synchronous transducer* is a special case of an asynchronous transducer where $|\lambda(t)| = 1$ for all $t \in T$ and $|\varphi(q)| = \varepsilon$ for all $q \in F$.

The *outcome* of $M = (A, \lambda, \varphi)$ on a valid input word $w \in \mathcal{L}(A)$, denoted by $\text{out}_M(w)$, is the concatenation of output words $u_1, \ldots, u_n$ produced by M while reading w concatenated with the final word produced by φ, i.e., if $\rho = q_1 q_2 \ldots q_{n+1}$ is the accepting run of A on $w \in \mathcal{L}(A)$, then $\text{out}_M(w) = u_1 \ldots u_n u_{n+1}$, where $u_i = \lambda(q_i, w_i, q_{i+1})$ for all $1 \leq i < n$ and $u_{n+1} = \varphi(q_{n+1})$. Note that the outcome of M is only defined for valid input words. The language of M, denoted $\mathcal{L}(M)$ is the is the union of valid input/output pairs padded with trailing zeros to have equal length: $\mathcal{L}(M) = \{w \in \Sigma_{I \cup O}^* \mid \exists j, k.\ w|_I \in \mathcal{L}(A)\mathbf{0}^j \wedge w|_O = \text{out}_M(w|_I)\mathbf{0}^k\}$.

An asynchronous transducer can express even certain specifications that are not WS1S expressible. For example, consider a transducer that emits ε when reading 0 and emits 1 when reading 1. Such transducer outputs a contiguous sequence of output bits whose length is the number of bits in the input.

Given a finite set of transducers $M_1, \ldots, M_k$ with $M_i = (A_i, \lambda_i)$ and a language L over the variables $I \cup O$, we say that $M_1, \ldots, M_k$ *jointly implement* L, written $M_1, \ldots, M_k \models L$ iff (1) each transducers M_i produces outputs satisfying the specification, i.e., $\mathcal{L}(M_i) \subseteq \mathcal{L}(G)$ and (2) the union of M_i's covers the valid inputs, i.e., $\mathcal{L}(G)|_I \subseteq \bigcup_i \mathcal{L}(A_i)$. We say $M_1, \ldots, M_k$ implements a WS1S formula G, denoted $M_1, \ldots, M_k \models G$, iff $M_1, \ldots, M_k \models \mathcal{L}(G)$.

Note that if $M_1, \ldots, M_k \models G$, then there exists a finite-memory implementation for G that performs only two passes over the input (regardless of k). In the first pass, the implementation generates no output, but simply determines which of the transducers accept. In the second pass, the implementation generates the output for one of the transducers that accept.

Transducers for Presburger specifications. The following lemma can be shown by analyzing the output of the quantifier-elimination based synthesis algorithm for Presburger arithmetic specifications [5]. They key observation is that functions implementing Presburger specifications have the form $\bigvee_i (P_i(x) \wedge y = t_i(x))$ for input x and output y.

Lemma 3: For every WS1S specification G that encodes a formula in Presburger arithmetic, there exists a finite set of transducers $M_1, \ldots, M_k$ such that $M_1, \ldots, M_k \models G$.
The key observation is that witness terms $t(x)$ are computable using asynchronous transducers.

Note that Presburger specifications are not computable using only one asynchronous transducer due to presence of disjunctions in specifications. They are also not computable using a finite union of *synchronous* transducers because of the division by constants.

Limitations of asynchronous transducers.

Lemma 4: There exists WS1S specifications cannot be implemented using a finite union of asynchronous transducers.

The proof is based on consider the following WS1S specification G over input I and output O. We give G as regular expression over the binary presentation over I and O:

$$G = \begin{matrix} I: \\ O: \end{matrix} \left(\begin{pmatrix} 1 \\ 0 \end{pmatrix}^+ \begin{pmatrix} 0 \\ 0 \end{pmatrix} \right)^* \begin{pmatrix} 1 \\ 1 \end{pmatrix}^+ \begin{pmatrix} 0 \\ 1 \end{pmatrix} \begin{pmatrix} 0 \\ 0 \end{pmatrix}^*.$$

Observation 1: Every transducer $M = (A, \lambda, \varphi)$ with $\mathcal{L}(M) \subseteq \mathcal{L}(G)$ and less than n states that accepts an input word $(1^n 0)^k$ must output a non-empty word within every n steps while reading this input.

Observation 2: Every transducer $M = (A, \lambda, \varphi)$ with $\mathcal{L}(M) \subseteq \mathcal{L}(G)$ and less than n states that accepts the input word $(1^n 0)^k$ for some $k > 0$, rejects all input words $(1^n 0)^l$ for $l > k$.

Using the above observations and given k asynchronous transducers $M_1, \ldots, M_k$ with $M_i = (A_i, \lambda_i, \varphi_i)$ such that $\mathcal{L}(M_i) \subseteq \mathcal{L}(G)$ it suffices to consider words $(1^n 0)^i$ for $i = 1, \ldots, k+1$ to conclude that it cannot be the case that $\mathcal{L}(G)|_I = \bigcup_{i=1,\ldots,k} \mathcal{L}(A_i)$.

Note that G can be implemented by a finite set of transducers if the input is read from right to left. However, we can concatenate specifications such as G with their reversed versions to obtain specifications that cannot be realized by transducers making both forward and backward passes.

VII. Lookahead-Causal Specifications

An interesting class of specifications that can be implemented using a single asynchronous transducer are lookahead-causal specifications discussed in this section.

The algorithms presented so far first read the entire input and then generate a corresponding output. In some cases (e.g., in streaming applications), one might prefer an implementation that starts outputting before reading the entire input. Specifications such as the signal processing example require reading a bounded number of bits ahead (three, in this case) to compute an output bit.

For notational simplicity we consider specifications $\mathsf{spec}(x, y)$ containing a single input-output pair x and y. Furthermore, we assume that the specifications are total, that is, $\forall x. \exists y. \mathsf{spec}(x, y)$. If a specification is not total, we can

transform it into a total specification $\mathsf{spec}'(x, y, e)$ given by $(\mathsf{spec}(x, y) \wedge e = 0) \vee ((\neg\exists y.\mathsf{spec}(x, y)) \wedge y = 0 \wedge e = 1)$.

Definition of k-causality. We next define lookahead-k-causality, or k-causality for short. We say that an input output pair x, y is k-causal for spec, written $\mathsf{causal}_k(x, y)$ iff $\forall p.\ \forall x' \sim_{p+k} x.\ \exists y' \sim_p y.\ \mathsf{spec}(x', y')$. where $z' \sim_p z$ means that z' and z have identical the initial p bits. spec is *k-causal* iff it implies $\mathsf{causal}_k(x, y)$ for all x, y.

Observe that a k-causal specification can be implemented by an asynchronous transducer, but there are specifications (such as the sign function) implementable by asynchronous transducers that are not k-causal. If spec is not k-causal but some inputs have multiple possible outputs, a general strategy to turn spec it into a causal specification is to simply conjoin it with $\mathsf{causal}_k(x, y)$ and check whether the resulting specification is still total, that is, whether $\forall x.\exists y.\mathsf{spec}(x, y) \wedge \mathsf{causal}_k(x, y)$.

Synthesized system for a k-causal specifications. Let $\mathsf{spec}(x, y)$ be a k-causal and total specification. We show how to construct an implementations that emits the input after reading k steps of the output. Construct first the specification automaton A and apply the construction described in Section IV-E. We obtain the automaton A' and the labeling functions τ, ϕ, and ψ. We extend A', τ, ϕ, and ψ so that they include, for all states q of A, the determinized version A'_q of A_q, where A_q is the automaton that differs from A only in that its initial state is changed to q. The synthesized program P_{caus} for k-causal specification has a fill parameter $\mu > 0$. It uses a buffer of length at least $(1 + \mu)k$ and alternates operations Read and Flush. The Read operation reads one more input bit into the buffer and advances the state S of A' accordingly, as for P_{gspec}. The Flush operation is invoked when the input buffer contains at least j input bits for $j \geq \lceil(1 + \mu)k\rceil$. It runs backwards k steps from the current state $q = \phi(S)$ following τ and reaches state q'. It then treats q' as a final state of the entire input, emits $j - k$ outputs going backwards and reaching state q''. It then empties the corresponding buffer elements and moves forward from q' using $A_{q'}$ until the current position of the input. This gives a streaming implementation that traverses the input bits only $1/\mu$ more times compared to P_{gspec}, regardless of k.

VIII. Related Work

Like synthesis of combinational circuits from relations (e.g.,[11]) our work synthesizes a function implementing the given relation. However, our implementation works for arbitrarily long input sequences. Techniques [1], [2], [12] to synthesize reactive systems that implement a given S1S specification can handle arbitrarily long input sequences. They assume that the specification can be implemented by a (usually finite-state) system that produces the output immediately while reading the input, i.e., the system cannot look ahead. These techniques usually take specifications in a fragment of temporal logic [13] and have resulted in tools that can synthesize useful hardware components [14], [3]. Recent work [15] establishes theoretical results (without implementation) regarding the problem of deciding when an S1S specification can be implemented using a system with lookahead. The (k-bounded) causality checks in our problem could be performed using this decision procedure based on infinite game theory. Our specification language uses finite instead of infinite words, which allows us to eliminate the non-causal behaviors and thus simplify the synthesis process. Moreover, our technique is not restricted to k-bounded specifications.

The work on graph types [16] proposes to synthesize fields given by definitions in monadic second-order logic and also uses the MONA tool [8]. However, it focuses on computing assignments to update fields of linked data structures as opposed to numerical and bit constraints.

IX. Conclusion

We presented an algorithm to synthesize linear-time functions from general WS1S specifications. Our software implementation works on a number of interesting examples. We have also identified interesting classes of specifications that can be implemented using finite unions of asynchronous transducers, and provided examples of specifications for which such finite-memory implementations do not suffice. Our results therefore contribute to the understanding and to the algorithm toolbox of automated synthesis approaches for software and hardware.

Acknowledgements. We thank Nir Piterman for discussion of the synthesis algorithm for general specifications, and Roderick Bloem for inspiring questions about the necessity of unbounded memory.

References

[1] J. R. Büchi and L. H. Landweber, "Solving sequential conditions by finite-state strategies," *Trans. of the American Math. Society*, 1969.
[2] M. O. Rabin, *Automata on Infinite Objects and Church's Problem*, ser. Regional Conference Series in Mathematics, 1972.
[3] B. Jobstmann and R. Bloem, "Optimizations for LTL synthesis," in *FMCAD*, 2006.
[4] A. Solar-Lezama, R. Rabbah, R. Bodík, and K. Ebcioğlu, "Programming by sketching for bit-streaming programs," in *ACM PLDI*, 2005.
[5] V. Kuncak, M. Mayer, R. Piskac, and P. Suter, "Complete functional synthesis," in *ACM PLDI*, 2010.
[6] W. Thomas, "Languages, automata, and logic," in *Handbook of Formal Languages Vol.3: Beyond Words*. Springer-Verlag, 1997.
[7] T. Schüle and K. Schneider, "Verification of data paths using unbounded integers: Automata strike back," in *Haifa Verification Conference*, 2006.
[8] N. Klarlund, A. Møller, and M. I. Schwartzbach, "MONA implementation secrets," in *CIAA*. LNCS, 2000.
[9] J. R. Büchi, "Weak second-order arithmetic and finite automata," *Z. Math. Logik Grundl. Math.*, vol. 6, pp. 66–92, 1960.
[10] M. Odersky, L. Spoon, and B. Venners, *Programming in Scala: a comprehensive step-by-step guide*. Artima Press, 2008.
[11] J. H. Kukula and T. R. Shiple, "Building circuits from relations," in *CAV*, 2000, pp. 113–123.
[12] A. Pnueli and R. Rosner, "On the synthesis of a reactive module," in *POPL '89*. New York, NY, USA: ACM, 1989, pp. 179–190.
[13] N. Piterman, A. Pnueli, and Y. Sa'ar, "Synthesis of reactive(1) designs," in *VMCAI*, 2006.
[14] B. Jobstmann, S. Galler, M. Weiglhofer, and R. Bloem, "Anzu: A tool for property synthesis," in *CAV*, 2007.
[15] M. Holtmann, L. Kaiser, and W. Thomas, "Degrees of lookahead in regular infinite games," in *FOSSACS*, 2010, pp. 252–266.
[16] N. Klarlund and M. I. Schwartzbach, "Graph types," in *ACM POPL*, 1993.

Automatic Inference of Memory Fences

Michael Kuperstein
Technion

Martin Vechev
IBM Research

Eran Yahav
IBM Research and Technion

***Abstract*—This paper addresses the problem of placing memory fences in a concurrent program running on a relaxed memory model. Modern architectures implement relaxed memory models which may reorder memory operations or execute them non-atomically. Special instructions called *memory fences* are provided to the programmer, allowing control of this behavior. To ensure correctness of many algorithms, in particular of non-blocking ones, a programmer is often required to explicitly insert memory fences into her program. However, she must use as few fences as possible, or the benefits of the relaxed architecture may be lost. Placing memory fences is challenging and very error prone, as it requires subtle reasoning about the underlying memory model.**

We present a framework for automatic inference of memory fences in concurrent programs, assisting the programmer in this complex task. Given a finite-state program, a safety specification and a description of the memory model, our framework computes a set of ordering constraints that guarantee the correctness of the program under the memory model. The computed constraints are maximally permissive: removing any constraint from the solution would permit an execution violating the specification. Our framework then realizes the computed constraints as additional fences in the input program.

We implemented our approach in a tool called FENDER and used it to infer correct and efficient placements of fences for several non-trivial algorithms, including practical concurrent data structures.

I. Introduction

> *On the one hand, memory barriers are expensive (100s of cycles, maybe more), and should be used only when necessary. On the other, synchronization bugs can be very difficult to track down, so memory barriers should be used liberally, rather than relying on complex platform-specific guarantees about limits to memory instruction reordering.* – Herlihy and Shavit, The Art of Multiprocessor Programming [1].

Modern architectures use relaxed memory models in which memory operations may be reordered and executed non-atomically [2]. These models enable improved hardware performance with respect to the standard sequentially consistent model [3]. However, they pose a burden on the programmer, forcing her to reason about non-sequentially consistent program executions. To allow programmer control over those executions, processors provide special *memory fence* instructions.

As multicore processors become increasingly dominant, highly-concurrent algorithms emerge as critical components of many systems [4]. Highly-concurrent algorithms are notoriously hard to get right [5] and often rely on subtle ordering of events, an ordering that may be violated under relaxed memory models (cf. [1, Ch.7]).

Finding a *correct and efficient* placement of memory fences for a concurrent program is a challenging task. Using too many fences (over-fencing) hinders performance, while using too few fences (under-fencing) permits executions that violate correctness. Manually balancing between over- and under-fencing is very difficult, time-consuming and error-prone as it requires reasoning about non sequentially consistent executions (cf. [1], [6], [7]). Furthermore, the process of finding fences has to be repeated whenever the algorithm changes, and whenever it is ported to a different architecture.

Our Approach In this paper, we present a tool that automatically infers *correct and efficient* fence placements. Our inference algorithm is defined in a way that makes the dependencies on the underlying memory model explicit. This makes it possible to use our algorithm with various memory models. To demonstrate the applicability of our approach, we implement a relaxed memory model that supports key features of modern relaxed memory models. We use our tool to automatically infer fences for several state of the art concurrent algorithms, including popular lock-free data structures.

Main Contributions The main contributions of this paper are:

- A novel algorithm that automatically infers a correct and efficient placement of memory fences in concurrent programs.
- A prototype implementation of the algorithm in a tool capable of inferring fences under several memory models.
- An evaluation of our tool on several highly concurrent practical algorithms such as: concurrent sets, work-stealing queues and lock-free queues.

II. Existing Approaches

We are aware of two existing tools designed to assist programmers with the problem of finding a correct and efficient placement of memory fences. However, both of these suffer from significant drawbacks.

CheckFence In [7], Burckhardt et al. present "CheckFence", a tool that checks whether a specific fence placement is correct for a given program under a relaxed memory model. In terms of checking, "CheckFence" can only consider finite executions of a linear program and therefore requires loop unrolling. Code that utilizes spin loops requires custom manual reductions. This makes the tool unsuitable for checking fence placements in algorithms that have unbounded spinning (e.g. mutual exclusion and synchronization barriers). To use "CheckFence" for inference, the programmer uses an iterative process: she starts with an initial fence placement and if the placement is

incorrect, she has to examine the (non-trivial) counterexample from the tool, understand the cause of error and attempt to fix it by placing a memory fence at some program location. It is also possible to use the tool by starting with a very conservative placement and choose fences to remove until a counterexample is encountered. This process, while simple, may easily lead to a "local minimum" and an inefficient placement.

mmchecker presented in [8] focuses on model-checking with relaxed memory models, and also proposes a naive approach for fence inference. Huynh et. al formulate the fence inference problem as a minimum cut on the reachability graph. While the result produced by solving for a minimum cut is sound, it is often suboptimal. The key problem stems from the lack of one-to-one correspondence between fences and removed edges. First, the insertion of a single fence has the potential effect of removing many edges from the graph. So it is possible that a cut produced by a single fence will be much larger in terms of edges than that produced by multiple fences. [8] attempts to compensate for this by using a weighing scheme, however this weighing does not provide the desired result. Worse yet, the algorithm assumes that there exists a *single* fence that can be used to remove any given edge. This assumption may cause a linear number of fences to be generated, when a single fence is sufficient.

III. Overview

In this section, we use a practically motivated scenario to illustrate why manual fence placement is inherently difficult. Then we informally explain our inference algorithm.

A. Motivating Example

Consider the problem of implementing the Chase-Lev work-stealing queue [9] on a relaxed memory model. Work stealing is a popular mechanism for efficient load-balancing used in runtime libraries for languages such as Java, Cilk and X10. Fig. 1 shows an implementation of this algorithm in C-like pseudo-code. For now we ignore the fences shown in the code.

The data structure maintains an expandable array of items called wsq and two indices top and $bottom$ that can wrap around the array. The queue has a single owner thread that can only invoke the operations `push()` and `take()` which operate on one end of the queue, while other threads call `steal()` to take items out from the opposite end. For simplicity, we assume that items in the array are integers and that memory is collected by a garbage collector (manual memory management presents orthogonal challenges [10]).

We would like to guarantee that there are no out of bounds array accesses, no lost items overwritten before being read, and no phantom items that are read after being removed. All these properties hold for the data structure under a sequentially consistent memory model. However, they may be violated when the algorithm executes on a relaxed model.

Under the SPARC RMO [11] memory model, some operations may be executed out of order. Tab. I shows possible reorderings under that model (when no fences are used) that lead to violation of the specification. The column *locations* lists the two lines in a given method which contain memory operations that might get reordered and lead to a violation. The next column gives an example of an undesired effect when the operations at the two labels are reordered. There could be other possible effects (e.g., program crashes), but we list only one. The last column shows the type of fence that can be used to prevent the undesirable reordering. Informally, the type describes what kinds of operations have to complete before other type of operations. For example, a store-load fence executed by a processor forces all stores issued by that processor to complete before any new loads by the same processor start.

```
1  typedef struct {
2    long size;
3    int  *ap;
4  } item_t;
5
6  long top, bottom;
7  item_t *wsq;
```

```
1  void push(int task) {
2   long b = bottom;
3   long t = top;
4   item_t* q = wsq;
5   if (b-t ≥ q→size-1){
6     q = expand();
7   }
8   q→ap[b % q→size]=task;
    fence("store-store");
9   bottom = b + 1;
10 }
```

```
1  int take() {
2   long b = bottom - 1;
3   item_t* q = wsq;
4   bottom = b;
    fence("store-load");
5   long t = top;
6   if (b < t) {
7     bottom = t;
8     return EMPTY;
9   }
10  task = q→ap[b % q→size];
11  if (b > t)
12    return task;
13  if (!CAS(&top, t, t+1))
14    return EMPTY;
15  bottom = t + 1;
16  return task;
17 }
```

```
1  int steal() {
2   long t = top;
    fence("load-load");
3   long b = bottom;
    fence("load-load");
4   item_t* q = wsq;
5   if (t ≥ b)
6     return EMPTY;
7   task=q→ap[t % q→size];
    fence("load-store");
8   if (!CAS(&top, t, t+1))
9     return ABORT;
10  return task;
11 }
```

```
1  item_t* expand() {
2    int  newsize = wsq→size * 2;
3    int* newitems = (int *) malloc(newsize*sizeof(int));
4    item_t *newq =  (item_t *)malloc(sizeof(item_t));
5    for (long i = top; i < bottom; i++) {
6      newitems[i % newsize] = wsq→ap[i % wsq→size];
7    }
8    newq→size = newsize;
9    newq→ap = newitems;
     fence("store-store");
10   wsq = newq;
11   return newq;
12 }
```

Fig. 1. Pseudo-code of the Chase-Lev work stealing queue [9].

#	Locations	Effect of Reorder	Needed Fence
1	push:8:9	`steal()` returns phantom item	store-store
2	take:4:5	lost items	store-load
3	steal:2:3	lost items	load-load
4	steal:3:4	array access out of bounds	load-load
5	steal:7:8	lost items	load-store
6	expand:9:10	`steal()` returns phantom item	store-store

TABLE I
POTENTIAL REORDERINGS OF OPERATIONS IN THE CHASE-LEV ALGORITHM OF FIG. 1 RUNNING ON THE RMO MEMORY MODEL.

Avoiding Failures with Manual Insertion of Fences To guarantee correctness under the RMO model, the programmer can try to manually insert fences that avoid undesirable reorderings. As an alternative to placing fences based on her intuition, the programmer can use an existing tool such as CheckFence [7] as described in Section II. Repeatedly adding fences to avoid each counterexample can easily lead to over-fencing: a fence used to fix a counterexample may be made redundant by another fence inferred for a later counterexample. In practice, localizing a failure to a single reordering is challenging and time consuming as a failure trace might include multiple reorderings. Furthermore, a single reordering can exhibit multiple failures, and it is sometimes hard to identify the cause underlying an observed failure. Even under the assumption that each failure has been localized to a single reordering (as in Tab. I), inserting fences still requires considering each of these 6 cases.

In a nutshell, the programmer is required to manually produce Tab. I: summarize and understand all counterexamples from a checking tool, localize the cause of failure to a single reordering, and propose a fix that eliminates the counterexample. Further, this process might have to be repeated manually every time the algorithm is modified or ported to a new memory model. For example, the fences shown in Fig. 1 are required for the RMO model, but on the SPARC TSO model the algorithm only requires the single fence in `take()`. Keeping all of the fences required for RMO may be inefficient for a stronger model, but finding which fences can be dropped might require a complete re-examination.

Automatic Inference of Fences It is easy to see that the process of manual inference does not scale. In this paper, we present an algorithm and a tool that automates this process. The results of applying our tool on a variety of concurrent algorithms, including the one in this section, are discussed in detail in Section V.

B. Description of the Inference Algorithm

Our inference algorithm works by taking as input a finite-state program, a safety specification and a description of the memory model, and computing a constraint formula that guarantees the correctness of the program under the memory model. The computed constraint formula is maximally permissive: removing any constraint from the solution would permit an execution violating the specification.

Applicability of the Inference Algorithm Our approach is applicable to any operational memory model on which we can define the notion of an *avoidable transition* that can be prevented by a *local* (per-processor) fence. Given a state, this requires the ability to identify: (i) that an event happens out of order; (ii) what alternative events could have been forced to happen instead by using a local fence. Requirement (i) is fairly standard and is available in common operational memory model semantics. Requirement (ii) states that a fence only affects the order in which instructions execute for the given processor but not the execution order of other processors. This holds for most common models, but not for PowerPC, where the SYNC instruction has a cumulative effect [12].

```
R1 = R2 = X = Y = 0;

     A:                      B:
A1: STORE 1, X   ||   B1: LOAD Y, R1
A2: STORE 1, Y        B2: LOAD X, R2
```

Fig. 2. A simple program illustrating relaxed memory model behavior

State Given a memory model and a program, we can build the transition system of the program, i.e. explore all reachable states of the program running on that memory model. A state in such a transition system will typically contain two kinds of information: (i) assignments of values to local and global variables; (ii) per-process execution buffer containing events that will eventually occur (for instance memory events or instructions waiting to be executed), where the order in which they will occur has not yet been determined.

Computing Avoid Formulae Given a transition system and a specification, the goal of the inference algorithm is to infer fences that prevent execution of all traces leading to states that violate the specification (error states). One naive approach is to enumerate all (acyclic) traces leading to error states, and try to prevent each by adding appropriate fences. However, such enumeration does not scale to any practical program, as the number of traces can be exponential in the size of the transition system which is itself potentially exponential in the program length. Instead, our algorithm works on individual states and computes for each state an *avoid formula* that captures all the ways to prevent execution from reaching the state. Using the concept of an *avoidable transition* mentioned earlier, we can define the condition under which a state is avoidable. The avoid formula for a state σ considers all the ways to avoid all incoming transitions to σ by either: (i) avoiding the transition itself; or (ii) avoiding the source state of the transition. Since the transition system may contain cycles, the computation of avoid formulae for states in the transition system needs to be iterated to a fixed point.

Consider the simple program of Fig. 2. For this program, we would like to guarantee that $R1 \geq R2$ in its final state. For illustrative purposes, we consider a simple memory model where the stores to global memory are atomic and the only allowed relaxation is reordering data independent instructions. Fig. 3 shows part of the transition system built for the program running on this specific memory model. We only show states that can lead to an error state. In the figure, each state contains: (i) assignments to local variables of each process ($L1$ and $L2$), and the global variables G; (ii) the execution buffer of each process ($E1$ and $E2$); (iii) an avoid formula which we explain below.

The initial state (state 1) has $R1 = R2 = X = Y = 0$. There is a single error state where $R1 = 0$ and $R2 = 1$ (state 9). The avoid formula for each state is computed as mentioned earlier. For example, the avoid formula for state 2 is computed by taking the disjunction of avoiding the transition

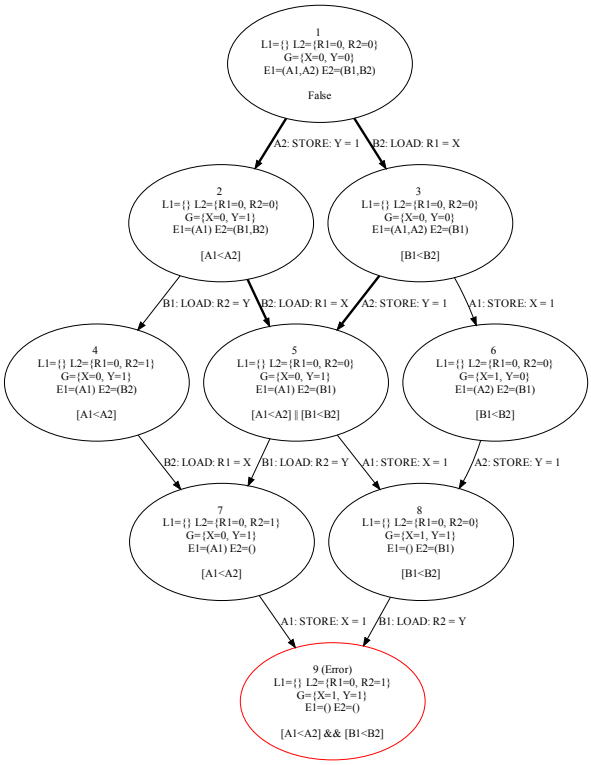


Fig. 3. A partial transition system of the program in Fig. 2. Avoidable transitions are drawn with thicker lines.

A_2 and avoiding the source state of the transition (state 1). To check whether A_2 is an avoidable transition from state 1, we check whether A_2 is executed out of order, and what are the alternative instructions that could have been executed by A instead. We examine the execution buffer $E1$ of state 1 and find all instructions that precede A_2. We find that A_2 is executed out of order, and that A_1 could have been executed to avoid this transition. So, we generate the constraint $[A_1 < A_2]$ as a way to avoid the transition A_2. The meaning of the constraint is that this transition can be avoided if A_1 is executed before A_2. Since the source state (state 1) cannot be avoided, the avoid formula for state 2 is just $[A_1 < A_2]$. The constraint $[B_1 < B_2]$ for state 3 is obtained similarly.

For state 5, there are two incoming transitions: B_2 and A_2. Here, B_2 is taken out of order from state 2 and hence we generate the constraint $[B_1 < B_2]$. The constraint for the parent state 2 is $[A_1 < A_2]$, so the overall constraint becomes $[B_1 < B_2] \vee [A_1 < A_2]$. Similarly, we perform the computation for transition A_2 from state 3 which generates an identical constraint. The final avoid formula for state 5 is thus the conjunction of $[B_1 < B_2] \vee [A_1 < A_2]$ with itself. In other words, it is this exact formula. The transition from state 2 to state 4 is taken in order. Therefore, the transition itself cannot be avoided and the only way to avoid reaching 4 is through the avoid formula of its predecessor, state 2. For the error state 9, the two incoming transitions do not generate constraints as they are executed in-order. The overall constraint is thus generated as conjunction of the constraints of the predecessor states 7 and 8, and it is $[B_1 < B_2] \wedge [A_1 < A_2]$.

Because our example graph is acyclic, a single pass over the graph is sufficient. It is easy to check the formulas that appear in Fig. 3 indeed correspond to a fixed point. Since there is only one error state, the resulting overall constraint is the avoid constraint of that error state: $[A_1 < A_2] \wedge [B_1 < B_2]$.

Finally, this constraint can be implemented by introducing a store-store fence between A_1 and A_2 and a load-load fence between B_1 and B_2.

C. Memory Models

To demonstrate our fence inference algorithm on realistic relaxed memory models, we define and implement the model RLX that contains key features of modern memory models. According to the categorization of [2], summarized in Fig. 4, there are five such key features. The leftmost three columns in the table represent order relaxations. For instance, $W \rightarrow R$ means the model may reorder a write with a subsequent read from a different variable. The rightmost columns represent store atomicity relaxations - that is, whether a store can be seen by a process before it is globally performed. Our memory model supports four of these features, but precludes "reading other's writes early" and speculative execution of load instructions.

The memory model is defined operationally, in a design based on [13] and [14]. We represent instruction reordering by using an execution buffer, similar to the "reordering box" of [15] and the "local instr. buffer" of [14]. To support non-atomic stores we, like [13], split store operations into a "store; flush" sequence, and allow local load operations to read values that have not yet been flushed. This allows us to talk about the model purely in terms of reordering, without paying any additional attention to the question of store atomicity.

Barring speculative execution of loads, RLX corresponds to Sun SPARC v9 RMO and is weaker than the SPARC v9 TSO and PSO models. RLX is strictly weaker than the IBM 370. Since RLX is weaker than these models, any fences that we infer for correctness under RLX are going to guarantee correctness under these models.

Our framework allows to instantiate models stronger than RLX, by disabling some of the relaxations in RLX. In fact, the framework supports any memory model that can be expressed using a bypass table (similar to [14] and the "instruction reordering table" of [13]). This enables us to experiment with fence inference while varying the relaxations in the underlying memory model. In Section V, we show how different models lead to different fence placements in practical concurrent algorithms, demonstrating the importance of automatic inference.

IV. Inference Algorithm

In this section, we describe our fence inference algorithm. Due to space restrictions, the description is mostly informal. The full technical details can be found in [16].

Relaxation	W → R Order	W → W Order	R → RW Order	R Others' W Early	R Own W Early
SC					✓
IBM 370	✓				
TSO	✓				✓
PSO	✓	✓			✓
Alpha	✓	✓	✓		✓
RMO	✓	✓	✓		✓
PowerPC	✓	✓	✓	✓	✓

Fig. 4. Categorization of relaxed memory models, from [2].

A. Preliminaries

We define a program P in the standard way, as a tuple containing an initial state $Init$, the program code $Prog_i$ for each processor, and an initial statement $Start_i$. The program code is expressed in a simple assembly-like programming language, which includes load/store memory operations, arbitrary branches and compare-and-swap operations. We assume that all statements are uniquely labeled, and thus a label uniquely identifies a statement in the program code, and denote the set of all program labels by *Labs*.

Transition Systems A transition system for a program P is a tuple $\langle \Sigma_P, T_P \rangle$, where Σ_P is a set of states, T_P is a set of labeled transitions $\sigma \xrightarrow{l} \sigma'$. A transition is in T_P if $\sigma, \sigma' \in \Sigma_P$ and $l \in$ *Labs*, such that executing the statement at l results in state σ'. The map $enabled\colon \Sigma_P \to \mathcal{P}(Labs)$ is tied to the memory model and specifies which transitions may take place under that model.

Dynamic Program Order Much of the literature on memory models (e.g. [11], [12], [17]) bases the model's semantics on the concept of *program order*, which is known a priori. This is indeed the case for loop-free or statically unrolled programs. For programs that contain loops, Shen et. al show in [13] that such an order is not well defined, unless a memory model is also provided. Furthermore, for some memory models the program order may depend on the specific execution.

To accommodate programs with loops, we define a *dynamic program order*. This order captures the program order at any point in the execution. For a given state σ and a process p, we write $l_1 <_{\sigma,p} l_2$ when l_1 precedes l_2 in the dynamic program order. The intended meaning is that in-order execution from state σ would execute the statement at l_1 before executing the statement at l_2.

B. An Algorithm for Inferring Ordering Constraints

Given a finite-state program P and a safety specification S, the goal of the algorithm is to infer a set of ordering constraints that prevent all program executions violating S and can be implemented by fences.

Avoidable Transitions and Ordering Constraints The ordering constraints we compute are based on the concept of an *avoidable transition* — a transition taken by the program that could have been *prohibited* by some fence. This captures the intuition of a transition that was taken out of order. To identify such transitions we use the dynamic program order: a transition $t = \sigma \xrightarrow{l_t} \sigma'$ is avoidable if there exists some l_1 such that $l_1 <_{\sigma,p} l_t$.

With every pair of labels $l_1, l_2 \in$ *Labs* we associate a proposition $[l_1 \prec l_2]$. We call such a proposition an *ordering constraint*. We define a *constraint formula* as a propositional formula over ordering constraints. For each transition $t = \sigma \xrightarrow{l_t} \sigma'$ we then define the formula $prevent(t) = \bigvee\{[l_1 \prec l_t] \mid l_1 <_{\sigma,p} l_t\}$. Intuitively, $prevent(t)$ is the formula that captures all possible ordering constraints that would prohibit the execution of t by the program. Note that if t is not avoidable, this is an empty disjunction and $prevent(t) = false$.

Algorithm 1: Fence Inference

```
   Input: Program P, Specification S
   Output: Program P' satisfying S
1  compute ⟨Σ_P, T_P⟩
2  avoid(Init) ← false
3  foreach state σ ∈ Σ_P \ {Init} do
4      avoid(σ) ← true
5  workset ← Σ_P \ {Init}
6  while workset is not empty do
7      σ ← select and remove state from workset
8      φ ← avoid(σ)
9      foreach transition t = (μ ⟶ σ) ∈ T_P do
10         φ ← φ ∧ (avoid(μ) ∨ prevent(t))
11     if avoid(σ) ≢ φ then
12         avoid(σ) ← φ
13         add all successors of σ in Σ_P to workset
14 ψ ← ⋀{avoid(σ) | σ ⊭ S}
15 return implement(P, ψ)
```

Inference The algorithm operates directly on program states. For every state σ in the program's transition system, the algorithm computes a constraint formula $avoid(\sigma)$ such that satisfying it prevents execution from reaching σ. The computed formula $avoid(\sigma)$ captures all possible ways to prevent execution from reaching σ by forbidding avoidable transitions.

The algorithm computes a fixed point of avoid constraints for all states in the program's transition system. First, we build the transition system $\langle \Sigma_P, T_P \rangle$ of the program. For $\sigma = Init$, we initialize $avoid(\sigma)$ to *false*. For all other states, we initialize it to *true*. We then add all states to the workset. The algorithm proceeds by picking a state from the workset, and computing the new avoid constraint for the state. A state can only be avoided by avoiding all incoming transitions (a conjunction). To avoid the transition, we must (i) consider all possible ways to avoid the transition from the predecessor state (by using $prevent(t)$); *or* (ii) avoid the predecessor state, by using its own avoid constraint. (see line 10 of the algorithm).

As shown in line 11 every such computation step requires comparing two boolean formulas for equality. While in general NP-hard, this is not a problem in practice due to the structure of our formulas and their relatively modest size.

When a fixed point is reached, the algorithm computes the overall constraint ψ by taking the conjunction of avoid constraints for all error states. Any implementation satisfying ψ is guaranteed to avoid all error states, and thus satisfy

the specification. Finally, the algorithm calls the procedure `implement(P,`ψ`)` which returns a program that satisfies ψ.

Ensuring Termination In cases where the transition system is an acyclic graph (e.g. transition systems for spinloop-free programs), we can avoid performing the fixed point computation altogether. If the states are topologically sorted, the computation can be completed with a single linear pass over the transition system. In the general case, we can show the set of mappings between states and constraints forms a finite lattice and our function is monotonic and continuous. Thus convergence is assured.

Safety and Maximal Permissiveness Given a program P and a specification S, the avoid formula φ computed by Algorithm 1 is the *maximally permissive* avoid formula such that all traces of P satisfying φ are guaranteed to satisfy S. More formally, we say a constraint formula admits a transition $t = \sigma \xrightarrow{l_t} \sigma'$ if there exists an assignment $\alpha \vDash \varphi$ so that every proposition of the form $v = [l_1 \prec l_t]$ where $l_1 <_{\sigma,p} l_t$ we have $[\![v]\!]_\alpha = false$. Here $[\![v]\!]_\alpha$ is the value of proposition v in the assignment α. We can lift this definition of *admits* from transitions to program traces. Then if $\varphi \neq false$ it only admits traces that satisfy S, but for any $\psi \neq \varphi$ such that $\varphi \Rightarrow \psi$, there exists a trace π of P that reaches σ such that ψ admits π, but $\sigma \nvDash S$.

C. Fence Inference

Our algorithm computes a maximally permissive constraint formula ψ. We can then use a standard SAT-solver to get assignments for ψ, where each assignment represents a set of constraints that enforces correctness. Since for a set of constraints C, a superset C' cannot be more efficiently implemented, we need only consider minimal (in the containment sense) sets.

An orthogonal problem is to define criteria that would allow us to select optimal fences that enforce one of those sets. In our work, we focus on a simple natural definition using set containment: a fence placement is a set of program labels where fences are placed and we say that a placement P_1 is better than P_2 when $P_1 \subseteq P_2$.

Given a minimal assignment C for the formula ψ, for each satisfied proposition $[l_1 \prec l_2]$, we can insert a fence either right after l_1 or right before l_2, thus getting a correct placement of fences. We can try this for all minimal assignments of ψ, and select only the minimal fence placements. This procedure can be improved by defining a formula ξ s.t. every proposition in ψ is replaced with $after(l_1) \vee before(l_2)$. Here, $after(l)$ and $before(l)$ map labels to a new set of propositions, so that if l_2 appears immediately after l_1 in the program, then $after(l_1) = before(l_2)$. Then, our fence placements will be the minimal assignments to ξ. This allows us to directly apply a SAT-solver and consider fewer fence placements.

Of course this local approach will not guarantee a minimal placement of fences because there can be many ways to implement a constraint $[l_1 \prec l_2]$ aside from inserting a fence immediately after l_1 or before l_2. For instance, if $l_1, ...l_4$ appear in this order in the program, and $\psi = [l_1 \prec l_4] \wedge [l_2 \prec l_3]$ then we can implement ψ by a single fence between l_2 and l_3. More precise and elaborate implementation strategies are possible if the program's control flow graph is taken into account. However, in our experiments we found the simple local fence placement strategy to produce optimal results.

V. Experiments

We have implemented our algorithm in a tool called FENDER. Our tool takes as input a description of a memory model, a program and a safety specification. The tool then automatically infers the necessary memory fences.

A. Methodology

We experiment with FENDER by varying the following:

(i) Input Algorithm - we experiment with five concurrent data structures and one mutual exclusion algorithm.
(ii) Client Program - we experiment with clients of varying size and complexity.
(iii) Memory Model - we experiment with 3 relaxed models and the sequentially consistent model as a baseline.
(iv) Specification - in some benchmarks, there is more than one reasonable specification.
(v) Bound on the execution buffer, when required.

Algorithms We applied our tool to various challenging state-of-the-art concurrent algorithms:

- *MSN*: Michael&Scott's lock-free queue [18].
- *LIFO WSQ*: LIFO idempotent work-stealing queue [19].
- *Chase-Lev WSQ*: Chase&Lev's work-stealing queue [9].
- *Dekker*: Dekker's mutual exclusion [20].
- *Treiber*: Treiber's lock-free stack [21].
- *VYSet*: Vechev&Yahav's concurrent list-based set [22].

Clients For each algorithm, we ran FENDER with several clients. Our tool permits exhaustive exploration of bounded clients that consist of a (bounded) sequence of initialization operations followed by (bounded) sequences of operations performed in parallel. A client typically consists of 2 or 3 processes, where each process invokes several data structure operations. Below, we use the term "program" to refer to the combination of an algorithm and a client.

Memory Models As noted earlier, our RLX model is equivalent to SPARC RMO without support for speculation. Our framework can instantiate stronger models, and in our experiments, we infer fences under four memory models: RMO, PSO, TSO, and as a reference, SC, the sequentially consistent model. The models RMO, PSO and TSO implement three different sets of relaxations as described in [2]. All three implement the "read own writes early" relaxation. RMO implements the $W \rightarrow R$, $W \rightarrow W$ and $R \rightarrow RW$ relaxations. PSO removes the $R \rightarrow RW$ relaxation and TSO additionally removes the $W \rightarrow W$ relaxation.

Specification We consider safety specifications realized as state invariants on the program's final state. To write an invariant, for most algorithms, we observed the results a specification of sequential consistency would produce and then write invariants that are implied by this specification. In

	Initial State	Client	$\|E\|$ Bnd	Time (sec.)	States	Edges	#C
MSN	empty	e|d	∞	0.83	1219	2671	2
	empty	e|e	∞	1.78	4934	12670	1
	empty	ee|dd	∞	5.21	24194	61514	3
	empty	ed|ed	∞	13.05	86574	242822	2
	empty	ed|de	∞	9.26	59119	167067	4
	empty	e|e|d	∞	31.43	233414	653094	3
ChaseLev WSQ	empty	pppt(tpt|sss)	∞	97.22	386283	1030857	-
	empty	tttt(ptt|sss)	∞	255.5	1048498	2819355	-
	empty	pppt(ttp|sss)	∞	90.28	281314	878880	-
	empty	tttt(tpp|sss)	∞	355.95	1325858	4150650	-
	empty	tttp(tptp|ss)	∞	37.98	280396	698398	-
"LIFO" WSQ	2/2	tp|ss	∞	0.69	2151	3190	2
	2/2	tpt|ss	∞	1.94	9721	16668	2
	2/2	ptp|ss	∞	11.41	89884	195246	3
	2/2	ptt|ss	∞	11.31	85104	198353	4
	1/1	ptt|ss	∞	4.07	23913	48997	4
Dekker	-	-	1	0.64	1388	2702	2
	-	-	10	2.13	7504	14477	2
	-	-	20	2.71	13879	26422	2
	-	-	50	5.99	33004	62257	2
Treiber	empty	p|t	∞	1	71	93	2
	empty	pt|tp	∞	1.02	3054	6190	2
	empty	pp|tt	∞	0.6	1276	2250	2
VYSet	empty	ar|ra	10	1.98	4079	6247	2
	empty	aa|rr	10	4.56	20034	31623	2
	empty	ar|ar	10	2.19	6093	9905	2
	empty	aaa|rrr	10	7.98	41520	66533	2

TABLE II
EXPERIMENTAL RESULTS FOR THE RMO MODEL

this context, sequential consistency refers not to the memory model, but to the high level specification that an algorithm should satisfy. In some experiments we also used additional, weaker specifications.

Bound on the Execution Buffer As recently shown in [23], the reachability problem for weak memory models is, depending on the model, either undecidable or non-primitive recursive even for finite-state programs. To avoid this problem we add a stronger condition and require the execution buffers to be bounded. In four of our benchmarks this was the natural behavior, and in the other two we've had to enforce a bound.

Experimental Setup Experiments were performed on an IBM xSeries 336 with 4 Intel Xeon 3.8Ghz processors, 5GB memory, running a 64-bit Red Hat Enterprise Linux. Tab. II contains performance metrics for RMO, the most relaxed memory model that we considered.

B. Results

A summary of our experimental results is shown in Tab. II. For each data structure, several parallel clients were used. For each client, the "Initial" and "Client" columns represent the initial state of the data structure and the operations performed by the client respectively. "e" represents an *enqueue* operation, "d" a *dequeue*, "p" *put*, "s" *steal*, "a" *add* and "r" *remove*. The "|E|" column represents the bound on the length of execution buffers, and "#C" the number of constraints in a minimal solution to the avoid formula for that client. Since for *Chase-Lev* the constraint formula was solved only for the conjunction of all clients, individual "#C" values are not given. The "Time" column shows the total analysis time. This includes the state exploration time, the constraint inference time and the SAT-solving time. Note that in all cases the solving component was negligible.

In Tab. III we show a comparison of the performance of FENDER for different memory models it supports. On average the number of states for PSO was ≈ 4.5 times smaller and for TSO ≈ 40 times smaller than for RMO.

Chase-Lev Work Stealing Queue For this data structure, we ran an exhaustive set of clients with two bounds: (i) all clients were of the form of 4 initializer operations, followed by a parallel section with $5 > X > 3$ invocations by the owner, and $6 - X$ steal invocations by another process. (ii) If a particular client's state space exceeded 2.5 million states, it was terminated and discarded. In Tab. II we show representative clients that produced useful constraints. In those experiments, FENDER inferred a set of 9 constraints which can be implemented using the 6 fences of Fig. 1. In particular, the fence between lines 9 and 10 in `expand()` also prevents the reordering of the store on line 10 with the stores on lines 8 an 6. Under PSO, we are left with 6 constraints and 3 fences—all of the fences in `steal()` are no longer needed. Even under TSO, one fence still remains necessary—it is the store-load fence between lines 4 and 5 in the `take()` operation.

Michael-Scott Queue For MSN FENDER inferred all 3 required fences under RMO. The placement for this algorithm in [7] contained 7 fences, however, 2 of these are the result of [7] allowing extra speculation, and 2 are not required in our model due to conservative memory allocation. Under PSO a single fence was inferred, and under TSO no fences are required.

Idempotent Work-Stealing The reference placement in [19] is phrased only in terms of constraints, and requires 5 constraints. Under RMO, FENDER produced 4 constraints which are a subset of those 5. The one constraint not inferred is, again, only required because of possible speculation.

Dekker's Algorithm It is well known that Dekker's algorithm requires a fence in the entry section and a fence at the end of the section (to preserve semantics of critical section). In our experiments, FENDER successfully inferred the required fences. Under RMO and PSO both fences were inferred, and under TSO, the tool inferred only the entry section fence. This is consistent with the reference placement appearing in Appendix J of [11].

C. Discussion

In our experiments, we observe that the fences inferred by FENDER are quite tricky to get manually. For some of the algorithms, there are known correct fence assignments, and for these we show that FENDER derives all necessary fences for our memory models with a small number of clients driving the algorithm. For most of our benchmarks, a bound on the execution buffer was not required. In the two cases where it was required, all fences were obtained with a small bound.

A recurring theme in our results was that several different maximally permissive constraint sets could be

	Initial	Client	$\|E\|$	RMO			PSO			TSO			SC	
			Bound	States	Edges	#C	States	Edges	#C	States	Edges	#C	States	Edges
MSN	empty	e|d	∞	1219	2671	2	455	743	1	228	316	0	146	180
	empty	e|e	∞	4934	12670	1	2678	6354	1	586	994	0	252	328
	empty	ee|dd	∞	24194	61514	3	7025	13689	2	1724	2512	0	1029	1325
	empty	ed|ed	∞	86574	242822	2	15450	35362	2	2476	3972	0	1538	2126
	empty	ed|de	∞	59119	167067	4	11023	24362	2	2570	4010	0	1541	2073
	empty	e|e|d	∞	233414	653094	3	51990	119050	2	9638	16822	0	4928	7632
Chase-Lev WSQ	empty	pppt(tpt|sss)	∞	386283	1030857	-	74533	256613	-	12348	20004	-	4961	6740
	empty	tttt(ptt|sss)	∞	1048498	2819355	-	124455	255390	-	6418	9380	-	3101	4069
	empty	pppt(ttp|sss)	∞	281314	878880	-	66960	241814	-	10564	16317	-	4199	5700
	empty	tttt(tpp|sss)	∞	1325858	4150650	-	361855	1080835	-	9878	13956	-	3473	4537
	empty	tttp(tptp|ss)	∞	280396	698398	-	29573	54696	-	9197	14499	-	4760	6455
"LIFO" WSQ	2/2	tp|ss	∞	2151	3190	2	882	1171	1	676	852	0	570	694
	2/2	tpt|ss	∞	9721	16668	2	3908	5811	1	2256	3116	0	1410	1786
	2/2	ptp|ss	∞	89884	195246	3	31289	64133	3	4045	5688	0	2317	3007
	2/2	ptt|ss	∞	85104	198353	4	29920	62020	3	4130	5987	0	2198	2866
	1/1	ptt|ss	∞	23913	48997	4	9976	18002	3	2353	3269	0	1314	1654
Dekker	-	-	1	1388	2702	2	1388	2702	2	489	674	1	388	490
	-	-	10	7504	14477	2	7504	14477	2	2560	3750	1	388	490
	-	-	20	13879	26422	2	13879	26422	2	4845	7115	1	388	490
	-	-	50	33004	62257	2	33004	62257	2	11770	17210	1	388	490
Treiber	empty	p|t	∞	71	93	2	71	93	2	43	48	0	36	38
	empty	pt|tp	∞	3054	6190	2	3041	6167	2	407	609	0	392	482
	empty	pp|tt	∞	1276	2250	2	1276	2250	2	325	407	0	270	323
VYSet	empty	ar|ra	10	4079	6247	2	4079	6247	2	1088	1308	0	1088	1308
	empty	aa|rr	10	20034	31623	2	20034	31623	2	1168	1411	0	1168	1411
	empty	ar|ar	10	6093	9905	2	6093	9905	2	1671	1968	0	1671	1968
	empty	aaa|rrr	10	41520	66533	2	41520	66533	2	3311	4072	0	3311	4072

TABLE III
EXPERIMENTAL RESULTS FOR DIFFERENT MEMORY MODELS

derived from the constraint formula. However, in all cases, all of those sets represented one "natural" solution. The reason for the appearance of those apparently different solutions involves data dependencies. Consider the simple example program shown on the right. Assume that the constraint $[l_1 \prec l_3]$ must be enforced in any execution. However, if $[l_1 \prec l_2]$ is enforced, then it is impossible to reorder l_3 with l_1. Due to a data dependency, l_2 must come before l_3, and we get the order $\sigma_1 \xrightarrow{l_2} \sigma_2 \xrightarrow{l_3} \sigma_3 \xrightarrow{l_1} \sigma_4$ in which the first transition violates $[l_1 \prec l_2]$. Thus, our constraint formula will necessarily contain the disjunction $[l_1 \prec l_2] \vee [l_1 \prec l_3]$. It is an interesting question whether there exists an input algorithm which permits several *substantially* different constraint sets.

```
1  STORE Z = 1
2  LOAD R = X
3  STORE Y = R
```

As expected, when we ran the tool with more restricted memory models, the number of required fences decreases. For example, the move from PSO to TSO disables reordering of independent stores and hence all constraints between stores to different locations are not required.

VI. RELATED WORK

Earlier we discussed work directly related to fence inference, that is [7], [8]. Additional related work includes:

Explicit-State Model Checking The works closest to ours in the way they explore the state space for a weak memory model are [15] and [24]. Both describe explicit-state model checking under the Sparc RMO model, but neither uses it for inference.

Delay Set Analysis A large body of work relies on the concepts of *delay set* and *conflict graph* of [25] for reasoning about relaxed memory models. In particular, the Pensieve project [26], [27], [28] implements fence synthesis based on delay set analysis. This kind of analysis is, however, necessarily more conservative than ours since it prevents any *potential* specification violations due to non-SC execution, and is not appropriate for highly concurrent algorithms.

Verification Approaches In [29] and [30] algorithms are presented that can find violations of sequential consistency under the TSO and PSO memory models. Those algorithms find violations based purely on sequentially consistent executions, thus making them very efficient. However, just like delay set analysis, this is often needlessly conservative. Another approach to verification is to try to establish a property which ensures the program remains correct under relaxed models. The most common such property is *data-race freedom*, as for data-race free programs the "fundamental property of memory models" [31] ensures that there can be no sequentially inconsistent executions. In our work we deal with programs that do not satisfy such properties. Further, none of those works supports fence inference for programs that are found to violate SC.

Inference of Synchronization In [32], [22], a semi-automated approach is used to explore a space of concurrent garbage collectors and linearizable data-structures. These works do not support weak memory models. In [33] a framework similar to ours is used to infer minimal synchronization. However the technique used there enumerates traces explicitly, which does not scale in our setting and thus cannot be applied as-is.

Effect Mitigation Several works have been published on mitigating the effect of memory fences [34], [35] and making synchronization decisions during runtime [36]. Those architectural improvements are complementary to our approach.

VII. Summary and Future Work

We presented a novel fence inference algorithm and demonstrated its practical effectiveness by evaluating it on various challenging state-of-the-art concurrent algorithms. In future work, we intend to improve the tool's scalability and add support for more memory models. Another direction we intend to pursue is memory model abstraction and fence inference under abstraction. This will allow us to avoid bounding the execution buffer and make our algorithm more suitable for more general input programs.

VIII. Acknowledgements

We would like to thank Maged Michael for his valuable comments and advice during the preparation of this paper.

References

[1] M. Herlihy and N. Shavit, *The Art of Multiprocessor Programming*. Morgan Kauffman, Feb. 2008.

[2] S. V. Adve and K. Gharachorloo, "Shared memory consistency models: A tutorial," *IEEE Computer*, vol. 29, pp. 66–76, 1995.

[3] L. Lamport, "How to make a multiprocessor computer that correctly executes multiprocess program," *IEEE Trans. Comput.*, vol. 28, no. 9, pp. 690–691, 1979.

[4] H. Sutter and J. Larus, "Software and the concurrency revolution," *Queue*, vol. 3, no. 7, pp. 54–62, 2005.

[5] M. M. Michael and M. L. Scott, "Correction of a memory management method for lock-free data structures," Tech. Rep., 1995.

[6] S. Burckhardt, R. Alur, and M. M. K. Martin, "Bounded model checking of concurrent data types on relaxed memory models: A case study," in *CAV*, 2006, pp. 489–502.

[7] S. Burckhardt, R. Alur, and M. M. K. Martin, "CheckFence: checking consistency of concurrent data types on relaxed memory models," in *PLDI*, 2007, pp. 12–21.

[8] T. Q. Huynh and A. Roychoudhury, "Memory model sensitive bytecode verification," *Form. Methods Syst. Des.*, vol. 31, no. 3, 2007.

[9] D. Chase and Y. Lev, "Dynamic circular work-stealing deque," in *SPAA*, 2005, pp. 21–28.

[10] M. M. Michael, "Safe memory reclamation for dynamic lock-free objects using atomic reads and writes," in *PODC*, 2002, pp. 21–30.

[11] I. SPARC International, *The SPARC architecture manual (version 9)*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1994.

[12] A. Adir, H. Attiya, and G. Shurek, "Information-flow models for shared memory with an application to the powerpc architecture," *IEEE Trans. Parallel Distrib. Syst.*, vol. 14, no. 5, pp. 502–515, 2003.

[13] X. Shen, Arvind, and L. Rudolph, "Commit-reconcile & fences (crf): a new memory model for architects and compiler writers," *SIGARCH Comput. Archit. News*, vol. 27, no. 2, pp. 150–161, 1999.

[14] Y. Yang, G. Gopalakrishnan, and G. Lindstrom, "Umm: an operational memory model specification framework with integrated model checking capability," *Concurr. Comput. : Pract. Exper.*, vol. 17, no. 5-6, pp. 465–487, 2005.

[15] S. Park and D. L. Dill, "An executable specification and verifier for relaxed memory order," *IEEE Transactions on Computers*, vol. 48, 1999.

[16] M. Kuperstein, M. Vechev, and E. Yahav, "Automatic inference of memory fences: Technical report," Technion, TR, 2010. [Online]. Available: http://www.cs.technion.ac.il/~mkuper/autoinf.pdf

[17] S. Sarkar, P. Sewell, F. Z. Nardelli, S. Owens, T. Ridge, T. Braibant, M. O. Myreen, and J. Alglave, "The semantics of x86-cc multiprocessor machine code," in *POPL*, 2009, pp. 379–391.

[18] M. M. Michael and M. L. Scott, "Simple, fast, and practical non-blocking and blocking concurrent queue algorithms," in *PODC*, 1996, pp. 267–275.

[19] M. M. Michael, M. T. Vechev, and V. A. Saraswat, "Idempotent work stealing," in *PPoPP*, 2009, pp. 45–54.

[20] E. Dijkstra, "Cooperating sequential processes, TR EWD-123," Technological University, Eindhoven, Tech. Rep., 1965.

[21] R. Treiber, "Systems programming: Coping with parallelism," IBM Almaden Research Center, Tech. Rep. RJ 5118, Apr. 1986.

[22] M. Vechev and E. Yahav, "Deriving linearizable fine-grained concurrent objects," in *PLDI*, 2008, pp. 125–135.

[23] M. F. Atig, A. Bouajjani, S. Burckhardt, and M. Musuvathi, "On the verification problem for weak memory models," in *POPL*, 2010, pp. 7–18.

[24] B. Jonsson, "State-space exploration for concurrent algorithms under weak memory orderings: (preliminary version)," *SIGARCH Comput. Archit. News*, vol. 36, no. 5, pp. 65–71, 2008.

[25] D. Shasha and M. Snir, "Efficient and correct execution of parallel programs that share memory," *ACM Trans. Program. Lang. Syst.*, vol. 10, no. 2, pp. 282–312, 1988.

[26] J. Lee and D. A. Padua, "Hiding relaxed memory consistency with a compiler," *IEEE Trans. Comput.*, vol. 50, no. 8, pp. 824–833, 2001.

[27] X. Fang, J. Lee, and S. P. Midkiff, "Automatic fence insertion for shared memory multiprocessing," in *ICS*, 2003, pp. 285–294.

[28] Z. Sura, C. Wong, X. Fang, J. Lee, S. Midkiff, and D. Padua, "Automatic implementation of programming language consistency models," *LNCS*, vol. 2481, p. 172, 2005.

[29] S. Burckhardt and M. Musuvathi, "Effective program verification for relaxed memory models," in *CAV*, 2008, pp. 107–120.

[30] J. Burnim, K. Sen, and C. Stergiou, "Sound and complete monitoring of sequential consistency in relaxed memory models," Tech. Rep. UCB/EECS-2010-31. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/2010/EECS-2010-31.html

[31] V. A. Saraswat, R. Jagadeesan, M. Michael, and C. von Praun, "A theory of memory models," in *PPoPP*. ACM, 2007, pp. 161–172.

[32] M. T. Vechev, E. Yahav, D. F. Bacon, and N. Rinetzky, "Cgcexplorer: a semi-automated search procedure for provably correct concurrent collectors," in *PLDI*, 2007, pp. 456–467.

[33] M. Vechev, E. Yahav, and G. Yorsh, "Abstraction-guided synthesis of synchronization," in *POPL '10*, 2010.

[34] O. Trachsel, C. von Praun, and T. Gross, "On the effectiveness of speculative and selective memory fences," *IPDPS*, p. 15, 2006.

[35] C. Blundell, M. M. Martin, and T. F. Wenisch, "Invisifence: performance-transparent memory ordering in conventional multiprocessors," in *ISCA*, 2009, pp. 233–244.

[36] C. von Praun, H. W. Cain, J.-D. Choi, and K. D. Ryu, "Conditional memory ordering," in *ISCA*, 2006, pp. 41–52.

Applying SMT in Symbolic Execution of Microcode

Anders Franzén
anders@disi.unitn.eu
FBK-irst and DISI-Univ.Trento
Trento, Italy

Alessandro Cimatti
cimatti@fbk.eu
FBK-irst
Trento, Italy

Alexander Nadel
alexander.nadel@intel.com
Intel Corp.
Israel

Roberto Sebastiani
rseba@disi.unitn.it
DISI-Univ.Trento
Italy

Jonathan Shalev
jonathan.shalev@intel.com
Intel Corp.
Israel

***Abstract*—Microcode is a critical component in modern microprocessors, and substantial effort has been devoted in the past to verify its correctness. A prominent approach, based on symbolic execution, traditionally relies on the use of boolean SAT solvers as a backend engine. In this paper, we investigate the application of Satisfiability Modulo Theories (SMT) to the problem of microcode verification. We integrate MathSAT, an SMT solver for the theory of Bit Vectors, within the flow of microcode verification, and experimentally evaluate the effectiveness of some optimizations. The results demonstrate the potential of SMT technologies over pure boolean SAT.**

I. Introduction

A modern Intel CPU may have over 700 instructions in the Instruction Set Architecture (ISA), some of them for backward compatibility with the very first x86 processors. Although the processor itself is a Complex Instruction Set Computer (CISC), the *microarchitecture* (basically the implementation of the ISA) is what can be likened to a Reduced Instruction Set Computer (RISC). The instructions in the ISA are translated into a smaller set of simpler instructions called *microinstructions* or *micro-operations*. Most instructions in Intel processors correspond to a single microinstruction, while larger programs are stored in a microcode program memory called the Microcode ROM. Some of these programs may be surprisingly large, such as string move in the Pentium 4 which was reported in [15] to use thousands of microinstructions.

Verification of these programs is a critical, but time-consuming process. To aid in the verification effort, a tool suite called MicroFormal has been developed at Intel starting in 2003 and under intensive research (in collaboration with academic partners) and development since. This system is used for several purposes:

- Generation of execution paths. These execution paths are used in traditional testing to ensure full path coverage, and to generate test cases which execute these paths, described in [2], [3].
- Assertion-based verification. Microcode developers annotate their programs with assertions, and these can be verified to hold using MicroFormal.
- Verification of backwards compatibility, described in [1]. When new generation CPUs are developed, they should be backwards compatible with older generations, although they may include more features.

At the heart of this set of tools is a system for *symbolic execution* (often called also *symbolic simulation*) of microcode, which is the part of the tool suite on which we will concentrate.

The symbolic execution engine explores the paths of the microcode, generating proof obligations, that have to be solved by a satisfiability engine. Such proof obligations can be thought of as constraints over bit-vectors. Traditionally, they are transformed into boolean satisfiability problems, and analyzed by means of boolean SAT solvers [7]. Although SAT technology is very efficient and has been highly specialized to the context of application, the time spent in the satisfiability engine is a very significant fraction of the total time devoted to symbolic execution.

In this work, we tackle this problem by presenting an alternative approach, based on the use of Satisfiability Modulo Theory (SMT) techniques [6] to replace boolean SAT. Modern approaches to SMT can be thought of leveraging the structure of the problem, by reasoning at a higher level of abstraction than SAT: efficient SAT reasoning is used to deal with the boolean component, and it is complemented by specialized rewriting and constraint solving to deal with more complex information at the level of bit-vectors.

The work presented in this paper (and described in greater detail in [12]) is based on the MathSAT SMT solver [9], that was the winner of the 2009 SMT competition on the bit-vector (BV) category, and was still unbeaten in 2010 edition. MathSAT was first integrated within the MicroFormal platform, and then customized to deal with the specific proof obligations arising from symbolic simulation of microcode. In particular, tailored solutions were adopted to deal with the satisfiability of sequences of formulae, and of sets of formulae. The approach was evaluated on a selected set of realistic microcode programs. MathSAT was able to provide substantial leverage over in-house SAT techniques on single problems; combined with the solutions described in this paper, we were able to significantly reduce the total execution time. As a consequence, a modified version of MathSAT was put in the production version of MicroFormal. Substantial speed-ups are reported on a wide class of real-world problems.

The rest of this paper is structured as follows. In § II we present an overview of the MicroFormal framework. In § III we describe the nature of the proof obligations resulting from MicroFormal, and in § III-A and III-B we discuss tailored techniques to deal with them. In § IV we present the experimental evaluation. In § V we discuss related work. In § VI we draw some conclusions and outline directions for future work.

II. Background

A. Intermediate Representation Language

To simplify the process, the symbolic execution engine does not work directly with microcode. Instead it works with an intermediate representation called Intermediate Representation Language, or IRL. This is a simple formal language with all features necessary to model microcode programs. Microcode programs are translated into IRL by a set of IRL templates, which define the translation from microcode instructions into a corresponding sequence of IRL instructions. This makes adapting the tool suite to a new microarchitecture simpler, since all that needs to be written is a new set of templates describing how instructions are translated into IRL. Another benefit of using IRL is that it is possible to handle other types of low-level software. Although the precise details of the language used in MicroFormal are not public, its main features have been presented in [1].

The correctness of the translation from actual microcode programs into IRL is crucial, but outside the scope of this high-level description of MicroFormal. We will also make many simplifications and skip over details that are not immediately relevant for the work presented.

B. Symbolic execution of microcode

The MicroFormal symbolic execution engine is used to compute a set of *paths* through a program, where a path is a sequence of locations that the program can follow from start to finish. A path through the program for which there exists an assignment to input registers such that the execution follows that path is called *feasible*. A *partial path* is a path from the start to some non-exit location within the program. The problem solved by the symbolic execution engine is to find all paths from the starting location to one of the exit locations. Symbolic execution [18] is a form of execution where all input (or initial values of variables) are symbolic. Consider the following simple example, which swaps values in two bit-vector variables

```
x, y : BitVector[64];
1: x := x + y;
2: y := x - y;
3: x := x - y;
4: exit;
```

To execute this program symbolically, we start by giving the symbolic values x_0, y_0 to the variables `x` and `y`. For the first assignment `x := x + y` we create a new symbolic value x_1 and compute how it relates to the symbolic values of the variables in the right hand side of the assignment $x_1 \hat{=} x_0 + y_0$ and so on for all instructions in the program, accumulating the equations that define the symbolic values we have created.

`1: x := x + y`	$x_1 \hat{=} x_0 + y_0$
`2: y := x - y`	$x_1 \hat{=} x_0 + y_0,\ y_1 \hat{=} x_1 - y_0$
`3: x := x - y`	$x_1 \hat{=} x_0 + y_0,\ y_1 \hat{=} x_1 - y_0,\ x_2 \hat{=} x_1 - y_1$
`4: exit`	$x_1 \hat{=} x_0 + y_0,\ y_1 \hat{=} x_1 - y_0,\ x_2 \hat{=} x_1 - y_1$

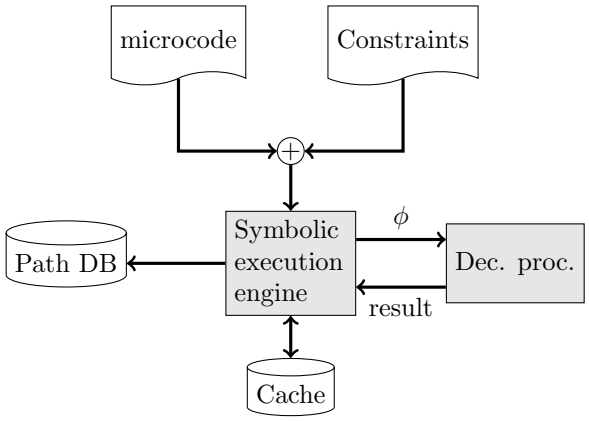


Fig. 1. Overview of the MicroFormal symbolic execution engine

By expanding the final definitionswe can see that the final values of the variables (x', y') depend on the initial given by the equations $x' = (x_0 + y_0) - x_0$ and $y' = (x_0 + y_0) - y_0$ which can be simplified to $x' = y_0$ and $y' = x_0$ respectively.

Apart from the current symbolic values for all variables in the program, during symbolic execution we also keep track of a *path condition* and the program location. The path condition is the conjunction of the conditions on the conditional branches along the current execution path, expressed in terms of the initial symbolic values. A more detailed description of how this may be performed is presented in [17].

An execution starts by executing the basic block (a non-branching sequence of instructions) starting at the beginning of the program to the first branch instruction. This partial path is marked as *open*. Then as long as there exists an open partial path π, all feasible branch targets continuing this path are computed by generating a sequence of *path feasibility conditions* which are sent to a decision procedure. A path feasibility condition is the path condition which would result when branching into a given branch target. If this path condition is satisfiable, the target is feasible in the sense that there exists some input that would execute down the current path and branch to that target. For every feasible branch target, MicroFormal extends π with the basic block starting at that location into a new path π'. If π' reaches a terminating instruction, this path is stored in the path database. Otherwise it is marked as an open path and the execution continues. An overview of the symbolic execution engine in MicroFormal can be seen in figure 1.

A path feasibility condition for a partial path π is a formula which describes the possible branch targets symbolically in terms of the input variables combined with some query on the target, and which is used to determine the possible values for the branch target. The details on the formulation of path feasibility conditions are outside the scope of this paper, here we will focus on the decision procedure used to solve these and other decision problems generated by MicroFormal.

From the point of view of the decision procedure, the symbolic execution engine feeds it a sequence of formulae,

and the result returned for one formula affects the future paths taken by the symbolic execution engine, and therefore also which formulae it receives in the future.

C. Improvements to the basic symbolic execution algorithm

To improve performance of the symbolic execution, several techniques are used as described in [3]; here we will briefly present three of them. One problem is the sheer size of the formulae sent to the decision procedure. In order to reduce the size of formulae, MicroFormal merges sets of partial paths ending up in the same location into a single path by introducing extra variables and conditional assignments. The details are explained in [3], but for our purposes the relevant effect is that it removes open partial paths which have so far been generated, and replaces them with a new merged path which is equivalent to but syntactically different from the previous paths. Two other techniques that are used are based on *caching* and *SSAT*, briefly described below.

Caching of solver results: The result of each solver call is stored in a cache shown in figure 1. This cache stores for every formula solved whether it is satisfiable or not, as well as the model for satisfiable formulae. If a formula α has been shown previously to be satisfiable, then any future formula $\alpha \vee \beta$ can be determined to be satisfiable without calling a solver. In the same way, if α has been shown to be unsatisfiable, any future occurrence of it as a subformula in future formulae can be replaced with $\perp$ as a simplification step. In case this fails, it is possible to take a model stored in the cache and evaluate the current formula with it. In case it evaluates to true, there is no need to call the solver. It may also happen that the evaluation results in a new smaller formula due to some variable occurring in the formula which did not occur in the model. In this case it is possible to send this simplified formula to the solver: if it is satisfiable, then it is possible to extend the old model into a model for the current formula. The motivations for caching models is that if a path feasibility check for some partial path shows it to be feasible, then there exists an extension to this path. Therefore the model for this path feasibility check should be useful in the future.

SSAT: In most cases, the symbolic execution engine generates a single formula which must be solved before execution can continue, because the satisfiability of this formula determines how the execution should proceed. But in some cases, it is possible to generate more than one formula, which it can predict must be solved regardless of their satisfiability. One technique used to improve performance of solving in these cases is to apply *Simultaneous SAT* (SSAT) introduced by Khasidashvili et al. [16]. This technique is a modification of the standard DPLL algorithm which allows the user to solve multiple *proof objectives* for a single formula in CNF. The solver will solve all proof objectives and for each of them return their satisfiability and a model in cases of satisfiable proof objectives. The motivation behind this technique is twofold; First a single model may satisfy more than one proof objective, and second information learnt while solving one proof objective may be helpful in solving the others. Both of these assume that the proof objectives are closely related to each other, which is the case in this application.

III. SMT(BV) for Symbolic Execution

The primary objective of this wok is the reduction of time spent in satisfiability checking of the proof obligations generated during symbolic execution. The problem has been tackled along two directions: (i) improve execution time for each call to the decision procedure, and (ii) identify a more efficient use of the decision procedure. (In the following, it suffices to see MicroFormal as a generator of bit-vector formulae to be solved.)

Direction (i) was pursued by replacing the backend engine used in MicroFormal, called Prover, with the MathSAT SMT solver. Prover is composed of an encoder from bit-vector formulae to boolean formulae (through a process of bit-blasting), pipelined to a customized (and extremely efficient) SAT solver working on a boolean formulae in CNF. MathSAT, on the other hand, can be seen as working at a higher level of abstraction, and leveraging structural information at the level of bit vectors to perform simplifications and rewritings. For example, reasoning at BV level allows simplification based on the theory of equality. This step, though conceptually simple, allows exploiting recent progress made in dealing with the theory of bit vectors in the field of SMT [8], [10]. We refer the reader to [12] for a detailed description of how MathSAT deals with BV. Notice that MathSAT won the 2009 SMT competition on the BV category, see http://www.smtcomp.org/2009/.

In order to identify more effective ways to use the decision procedure (ii), we consider that MicroFormal presents to the solver a sequence $\Phi_1, \Phi_2, \ldots, \Phi_N$, where each Φ_i is a nonempty set of formulae. The sequence of formulae is not known a priori, meaning that the set Φ_{i+1} is not known until all formulae in the set Φ_i have been solved. Since all formulae in the sequence derive from the symbolic execution of the same microcode program, they will share the same set of variables.

The sets of formulae in the sequence have typically a very distinct nature: the vast majority are *singleton sets*, containing a single formula; the remaining few, *non-singleton sets*, however, can contain large numbers of formulae, in some cases even thousands. Thus, we concentrated on two specific way to use the decision procedure, i.e. how to efficiently solve

- sequences of single formulae,
- large sets of formulae.

A. Solving sequences of single formulae

In MicroFormal, most sets in a sequence contain a single formula, and we need to solve this one formula to advance the search. Each formula is usually very similar to the previous one. This can be seen by measuring similarity for a number of medium to large sequences. Seeing each formula as a Directed Acyclic Graph (DAG) using perfect sharing, we can compare the similarity of a pair of formulae by measuring the number of nodes in the DAG for one which do not occur in the DAG for the other. Formally, given two formulae ϕ and ψ, we compute the ratio of terms occurring in ϕ which do not occur in ψ to

Algorithm 1: Solve reusing information

Input: $\phi_1, \phi_2, \ldots, \phi_N$
Input: Reset interval k
$\phi \leftarrow \top$;
for $i \in [1, N]$ **do**
  if $i \mod k = 0$ **then**
    $\phi \leftarrow \top$;
  end
  $p_i \leftarrow$ fresh proposition;
  $\phi \leftarrow \phi \wedge (p_i \equiv \phi_i)$;
  solve ϕ under the assumptions $\{p_i\}$;
end

Algorithm 2: MSPSAT

Input: $\phi_1, \phi_2, \ldots, \phi_N$
$P \leftarrow \{p_1, ..., p_N\}$; // p_i fresh predicates
$\phi \leftarrow \bigwedge_{i=1}^{N} (p_i \equiv \phi_i)$;
Sat $\leftarrow \emptyset$; Unsat $\leftarrow \emptyset$;
while $P \neq \emptyset$ **do**
  $p_i \leftarrow$ some element in P;
  if ϕ *under the ass.* $\{p_i\}$ *satisfiable with model* μ **then**
    $\mathrm{Sat} \leftarrow \mathrm{Sat} \cup \{\phi_j \mid \mu \models p_j\}$;
  else
    $\mathrm{Unsat} \leftarrow \mathrm{Unsat} \cup \phi_i$;
  end
  $\mathrm{P} \leftarrow P \setminus \{p_j \mid \phi_j \in (\mathrm{Sat} \cup \mathrm{Unsat})\}$;
end
return $\mathrm{Sat}, \mathrm{Unsat}$

the total number of terms in ϕ, and vice versa. The minimal of the two ratios denotes the similarity.

Consecutive formulae appear to be highly similar, with a median similarity of 78%, 95% and 99%, respectively in the sequences of three typical programs (see § IV-A), and this is something we would wish to take advantage of. The cases with very small similarity between formulae are almost always combined with at least one of the two formulae being very small. The approach we have taken is to reuse learnt information from the solving of one formula to help solving the next.

Modern SAT solvers are often quite good at handling irrelevant information, since the heuristics they use often manage to focus on the relevant parts of a formula, ignoring the rest. MathSAT inherits these features from its underlying SAT solver. We will take advantage of this fact by retaining all information stored in the solver from one formula to the next. We will also take advantage of the fact that MathSAT implements incremental solving under assumptions [11]. The basic approach is shown in algorithm 1. When solving a sequence of individual formulae $\phi_1, \phi_2, \ldots$, the basic algorithm is to first create one fresh predicate p_1, add the formula $p_1 \equiv \phi_1$ and solve under the assumption of p_1 to discover if ϕ_1 is satisfiable; then, we create another fresh predicate p_2 and add $p_2 \equiv \phi_2$ to the solver and solve under the assumption of p_2. In the second iteration, the complete formula in the solver will be $(p_1 \equiv \phi_1) \wedge (p_2 \equiv \phi_2)$ and all learnt information from the solving of ϕ_1 is still available when solving ϕ_2.

Although the solver might be good at ignoring irrelevant information, eventually as the amount of irrelevant clauses grow these will have a negative impact on performance, and of course also on memory usage. Therefore it is important to at some point remove this information. The simplest possible approach would be to just throw away *all* information irrelevant or not, and then solve the next formula as if it is the first one encountered. The advantages of this is that it is very easy to implement and to use. The disadvantage are that we also throw away potentially useful information.

The main question with this approach of dealing with the accumulation of irrelevant information is, when to reset the solver? Several solutions suggest themselves:

- Use fixed reset frequency. Reset every k formulae.
- Reset based on subformula reuse. Measure how much the next formula is already known to the solver, how much of it is not previously known, and how much of the solver information is irrelevant.
- Use an adaptive strategy. Measure solver performance, and try to predict when degradation starts to occur. Reset before it becomes detrimental.
- Delete only irrelevant information from the solver, and keep the rest. This sounds like the best solution, but computing which information is irrelevant is not a simple problem. Just because it is not relevant for the current formula does not mean it will not become relevant again in the future.

Even in the cases where no learnt information is explicitly removed, the underlying solver is free to remove learnt clauses, as any standard SAT solver does. This can be more or less aggressive, and works regardless of how the solver is used. However, these techniques will not work on the original clauses generated from encoding of the formulae given to the solver, only the learnt clauses. In this application an aggressive heuristic for clause removal may be interesting, such as suggested in [4] and used in the glucose SAT solver.

B. Solving sets of formulae

In the cases where the current set of formulas contain more than one formula, we should try to take advantage of this in order to improve performance. For three medium-sized to large microcode programs (see § IV-A) the simulator generates sets of formulae with 93 non-singleton sets with between 100 and 1000 instances, and 11 sets with over 1000 instances.

To take advantage of this fact, we would like to make the solver aware of all formulae beforehand. In this way we may be able to satisfy more than one formula at a time, and also reuse learnt information to discover that several formulae in the set are unsatisfiable. One way of achieving this is shown in a simple algorithm 2 we will call Multiple Similar Properties SAT (MSPSAT). Here we create one fresh predicate (boolean

variable) p_i for each formula ϕ_i and give the solver the formula

$$\bigwedge_i p_i \equiv \phi_i$$

To solve ϕ_i, we solve under the assumption p_i. Should it be satisfiable under this assumption, we can easily check which of the other formulae are also satisfied by the same model by checking the truth assignment for the other fresh variables. The algorithm iteratively picks one unsolved formula as a goal and solves under the assumption of the corresponding fresh variable. If it is satisfiable we check if any other unsolved formulae are satisfied by the same model, and discharge all satisfiable formulae.

IV. Experimental evaluation

A. Benefits of incremental and simultaneous solving

We now turn to an experimental evaluation of the techniques proposed in this paper. Except where explicitly noted, all experiments were carried out on a machine with dual Intel Xeon E5430 CPUs running at 2.66 GHz using 32 GB of RAM running Linux.

The initial experiments are run on instances coming from three nontrivial microcode programs. For these three, MicroFormal was instrumented to dump all instances to files in SMT-LIB format, and produce a log describing how these instances were created. In this paper the programs will be called "program 1", "program 2" and "program 3". Table I gives the number of formulae generated in each of these three MicroFormal runs. A test bench has been created which can replay the solver calls in these three runs of MicroFormal, which makes it easy to experiment with different strategies and instrument the system to extract interesting information. In order to emulate the behaviour of MicroFormal, when solving a formula it is first loaded into memory in a separate data structure to avoid measuring the time taking for parsing formulae. From this data structure the MathSAT API is called creating and solving formulae simulating the in-memory usage in MicroFormal as closely as possible without actually running MicroFormal.

Apart from the techniques described in this paper, these experiments were performed with MathSAT set up to simply bit-blast and solve the formula using a SAT solver. Since the vast majority of formulas generated by MicroFormal are trivial, this seems to deliver good performance, and this setup should also mean that the techniques described here will also translate to SAT solvers. For the instances taking the most execution time, more aggressive preprocessing techniques can be effective, but the total execution time is dominated by a large number of trivial instances, and the preprocessing normally used in MathSAT seems to be too expensive to be used here.

1) Solving sequences of single formulae: We start by investigating the effect of fixed reset strategies on singleton sets. For these experiments, we solve only singleton sets, skipping over the other calls completely. The result on the three programs are summarized in figure 2. It shows the relative improvement of

TABLE I
MicroFormal test sets

Program	Instances	Satisfiable	Unsatisfiable
Program 1	52933	44359	8574
Program 2	5468	4341	1127
Program 3	28962	13757	15205

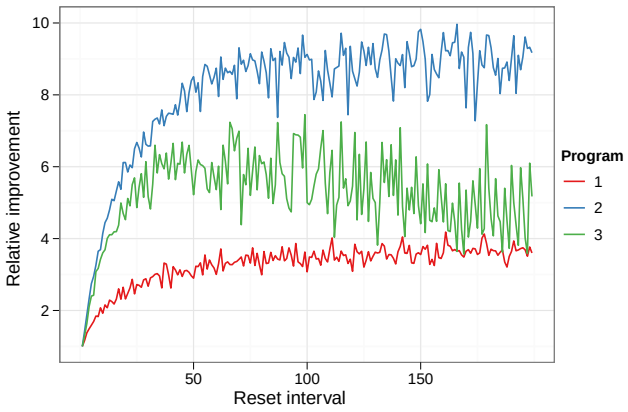


Fig. 2. Effect of reset interval on singleton calls

reusing solver information compared to solving each formula in isolation. The horizontal axis shows the reset interval, that is how frequently all learnt information is thrown away. A reset interval of 1 corresponds to solving each formula in isolation. From the figure, it is clear that there is a positive effect of reusing solver information. For program 1 the best improvement is a factor of 4 (at a reset interval of 161), and for program 2 the best improvement is a factor of almost 10 (at reset a interval of 169). Lastly for program 3 the best improvement is a factor of 7.4 (at a reset interval of 99).

We can also see that the exact reset frequency is not critical. For program 1 and program 2, there is only a minor difference between different reset intervals above 50. For program 3, the trend is similar but the data appears to be more noisy. This is due to some outliers among the instances to be solved, which are both large and significantly different from any of the others. These cause significant overhead when these instances are retained in the solver and we attempt to solve fresh instances. Performance depends on being able to divest the solver of this irrelevant information as soon as possible, but with a fixed reset interval how quickly this happens is largely due to chance. To avoid this, we will choose a reset interval of 25 for future experiments, which although shorter than what is indicated as the optimal, should on the other hand handle such outliers better. With this reset interval, the improvement for these three programs is a factor of 2.7, 6.7 and 4.9 respectively.

To check if reuse of solver information is usable outside of MicroFormal, the technique has also been applied to the instances coming from the SAGE tool [13] (available in SMT-LIB under QF_BV/sage). Out of 12 sets of instances, a

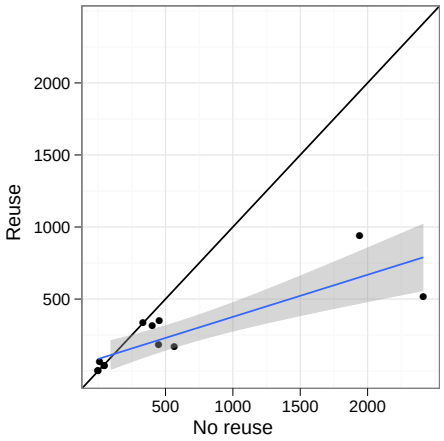


Fig. 3. Effect of reusing solver information on SAGE instances. Execution times in seconds

TABLE II
PERFORMANCE OF THE MSPSAT ALGORITHMS

Method	Program 1	Program 2	Program 3
No reuse	104459.86	1722.31	55539.64
Reset (25)	9104.31	217.13	5434.52
Reset in-between	4485.51	243.91	2694.61
MSPSAT	6064.98	278.00	2826.98

TABLE III
AMPLE PERFORMANCE SUMMARY (EXECUTION TIMES IN SECS)

Solver	Type	Median	Mean	Standard Dev.
Prover	Singleton	1072.14	2887.13	5973.29
	Non-singleton	389.01	2264.52	4432.13
	Ample	2412.00	6282.90	10316.34
MathSAT	Singleton	98.48	289.05	704.25
	Non-singleton	233.25	975.24	1751.98
	Ample	997.00	2183.03	2842.62

fixed reset strategy of resetting every 25 instances helped in all but two sets. In one of the two, execution time was comparable (332 versus 334 seconds). In the other reusing solver information used 65 seconds versus 11 seconds for solving each instance individually. The added time is taken up in two instances which take considerably more time than the rest. Full results for these sets of instances can be found in figure 3, where total execution time (in seconds) for each set of instances is reported. Although the improvement is not as large as for the three microcode programs seen earlier, there is still a fairly clear improvement, and, indeed this improvement is statistically significant ($p = 0.016$).

2) Solving sets of formulae: For the cases where MicroFormal generates multiple formulae to solve there are several choices, we will look at a few of them as listed below:

1) Solve them in the same way as single formulae. There might not after all be any need to treat these instances any different from any other.
2) Solve them as with single formulae, but with an infinite reset interval. The motivation is that similarity can be expected to be better within each set than between singleton instances since all instances in a set have been generated at a specific point in symbolic execution.
3) Solve them with MSPSAT.

As a baseline, let's look at the performance when treating each instance as a singleton, disregarding that more than one instance is known a priori. The results are shown in the first row in table II. We can see a significant improvement using MSPSAT over solving each formula individually. For comparison, we also include the execution time when solving all instances reusing solver information using a reset interval of 25, and also when resetting only in between sets of instances. We can see that using a reset interval of 25 gives worse performance than using the MSPSAT algorithm, so there seems to be some value in treating these sets in a special way. For these three programs at least there does however not seem to be an advantage with MSPSAT when compared to using a separate solver instance for non-singleton sets which is reset in-between every set. Indeed, the latter technique has a small, but statistically insignificant, advantage over the others.

B. Overall impact of MathSAT within Ample

As a final experiment the impact of the usage of MathSAT on the *Ample* tool is evaluated. Ample (Automatic Microcode Path Logic Extraction) is a tool in MicroFormal used for generation of execution paths for dynamic testing, and this will be used for experimental evaluation in this section. For this evaluation 32 different microcode programs have been selected to be representative of small, medium, and large programs. For each, Ample is run with its standard backend engine, the in-house SAT solver Prover, and with MathSAT. In MathSAT, reusing of solver information was used with a fixed reset frequency of 25, and for non-singleton sets MSPSAT was used. For Prover, singleton sets were solved individually, and non-singleton sets were solved using the SSAT algorithm. The tool was run on machines with Intel Xeon 5160 CPUs running at 3 GHz and 32GB RAM running Linux, and the execution times of solver calls, other processing, total execution time and memory usage was measured. In these experiments, in no case was memory usage an issue.

The results are summarized in table III, which presents the median, mean, and standard deviation values for the total execution time on, respectively, singleton sets of formulae, non-singleton sets (using MSPSAT for MathSAT, SSAT for Prover) and for the total execution time for Ample. The corresponding values, with one point for each of the 32 programs, are plotted in Figure 4.

For every program, the performance of MathSAT is better than that of Prover, and for total execution time the improvement is at worst a factor of 1.17, at best a factor of 4.43, and overall the improvement is a factor of 2.88. Not surprisingly, the improvement is statistically significant ($p = 9 \cdot 10^{-9}$). As the experiments on non-singleton sets showed, simply reusing solver information resetting the solver in-between each set may

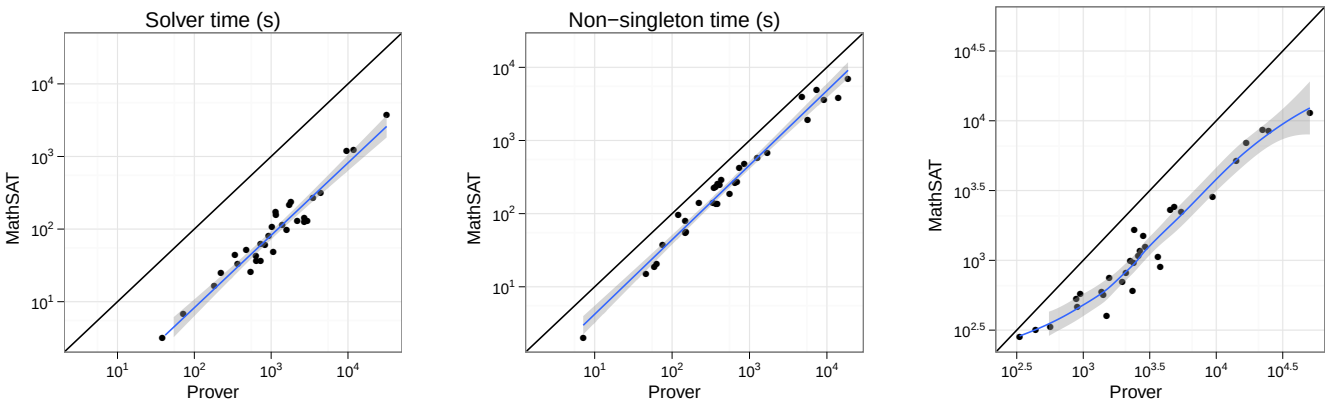


Fig. 4. Results for each of the 32 programs: (left) total solving times on the singleton sets; (center) total solving times on the non-singleton sets; (right) Ample total execution times.

improve performance further. At the time of writing, this has not been tried on these 32 microcode programs.

It should be noted that the difference on non-singleton sets are not necessarily due to the different algorithms (MSPSAT versus SSAT) being used, since two completely different solvers are used for the comparison.

V. Related work

Whittemore et al. [21] describes reusing of learnt clauses in the SATIRE SAT solver. This is an incremental SAT solver which allows the user to retract clauses and add new ones before searching again. To implement this the solver keeps track of the dependencies between learnt clauses and original clauses. If a clause is retracted, all clauses which have been learnt using this clause are also removed. Silva and Sakallah [20] proposed a technique for reusing clauses from one formula to the next in automatic test pattern generation (ATPG) for circuits. In this application a SAT solver is used to try to generate stimuli that expose a particular fault. They notice that some learnt clauses are independent of the current target fault instead depending only on the circuit being studied, and could be reused from one SAT problem to the next. This happens if a learnt clause is derived solely from clauses originating in the circuit. Strichman [19] noticed that in the context of Bounded Model Checking (BMC), certain clauses could be reused from one unrolling to the next.

Eén and Sörensson showed in [11] how learnt clauses could be reused when doing k-induction. This relies on the idea that in this application we are monotonically adding non-unit clauses to the solver, and all unit-clauses can be used as assumptions rather than adding them permanently to the solver.

In [14] Große and Drechsler propose to reuse clauses learnt while solving one formula when solving another iff they can be derived from the intersection of the clauses in the two formulae.

Babić and Hu proposed some simple heuristics to decide if a fact is reusable of not in [5], which allow for reuse of learnt unit clauses.

The only work which considers the idea of reusing all information is the work by Eén and Sörensson, which is targeted for the case of k-induction where all non-unit clauses in one formula will occur also in the next. For general solving of similar formulae which are not extensions of one another, all previous work concentrate on techniques to compute the relevant parts of the learnt clauses and reuse only those.

A. Simultaneous SAT

Khasidashvili et al. [16] introduced a technique for solving a set of related formulae using an algorithm they call Simultaneous SAT (SSAT). Given a formula in CNF and a set of *proof objectives* being literals in this formula, their algorithm is a modification of a normal DPLL-like algorithm. They always keep a particular proof objective as the current goal to satisfy, the *currently watched proof objective*. At any decision this literal is chosen unless it has already been given a truth value. When the solver finds a model, it checks all other proof objectives and records all that have been satisfied by the model. Then a new currently watched proof objective is chosen among those which has not yet been solved. This is repeated until all proof objectives have been solved. The SSAT algorithm can be seen as a special case of reusing learnt information when all formulae to be solved are known in advance.

In contrast to the SSAT algorithm, the MSPSAT algorithm presented in this work doesn't require any modifications of the underlying solver. Indeed it would be possible to implement using the MathSAT API rather than modifying any part of the solver.

VI. Conclusions

In this industrial case study, we have seen how the introduction of SMT technology can result in increased performance over pure boolean SAT. The experience also demonstrates that a tailored integration within a given verification flow can have a big impact on performance. In particular, reusing

learnt information from solving previous formulae can be very useful, and that in some cases it is possible to achieve good performance without resorting to more complex techniques for reusing information that have been proposed in the past. Simply retaining *all* information, relevant or not, can provide a significant performance boost with a very low implementation cost and with no added solver complexity.

The activity described in this paper has had substantial impact. MathSAT has now been successfully integrated into MicroFormal, and it delivers significantly improved performance over the SAT-based solver previously used. A version of the tool set with MathSAT integrated has been made available to users within Intel with MathSAT available as a command-line option. Using MathSAT, this version has been successfully used for verification of a next generation microarchitecture. In the future, MathSAT will be made the default decision procedure in MicroFormal.

Although some improvements have been made to MicroFormal in this case study, the time taken to solve formulae is still considerable compared to the rest of the work of the symbolic execution engine, on average over half the execution time is spent in solving formulae. Therefore, it would be interesting to look for ways of further reducing the time taken to solve instances as well as reducing the number of instances that need to be solved. Listed below are a few possibilities which may be interesting to investigate.

Better models: Since MicroFormal is currently capable of storing models for previous formulae, and then use them in a model caching scheme to either avoid future solver calls, or significantly reduce the complexity of future calls, it makes sense to attempt to adapt the models returned from the solver to maximize the utility of this feature. A "good" model in this case is one which models (or can be extended to model) as many future formulae as possible, therefore minimal (or near minimal) models may be interesting.

Heuristics for resets: The reset strategy used in this work is a simple strategy with a fixed reset frequency. Although it has been shown to deliver a significant performance improvement, it is still vulnerable to outliers in the sequence of instances. It would be interesting to discover heuristics capable of detecting when irrelevant information stored in the solver is likely to negatively affect performance, and build an adaptive reset strategy around such a heuristic. This should allow for longer reset intervals in the cases where no outliers exists, and further improve performance.

A hybrid concrete/symbolic execution engine: One technique which can quickly discover sets of paths in a program is fuzz testing. It might be possible to combine fuzzing with symbolic execution by starting with generating a number of paths with fuzzing, and then extending this set using symbolic execution. The two methods can be interleaved by a technique similar to [13]. Judicial use of fuzzing and concrete execution may in the best case be able to significantly reduce the number of formulae that need to be solved, and taking a closer look at this possibility may be a fruitful avenue of research.

Other possibilities: There are many other possibilities for future improvement. Among them are the following:

- Support for uninterpreted functions. MicroFormal abstracts some parts with uninterpreted functions, but currently those are eliminated using Ackermann's expansion by MicroFormal itself. Passing the original formula on to the solver may improve performance.
- Parallelism. There are opportunities for parallelism in MicroFormal. One example would be performing the symbolic execution in parallel exploring several paths simultaneously.

ACKNOWLEDGEMENTS

This work is supported in part by SRC/GRC under Custom Research Project 2009-TJ-1880 WOLFLING, and by MIUR under PRIN project 20079E5KM8 002.

REFERENCES

[1] T. Arons, E. Elster, L. Fix, S. Mador-Haim, M. Mishaeli, J. Shalev, E. Singerman, A. Tiemeyer, M. Y. Vardi, and L. D. Zuck. Formal verification of backward compatibility of microcode. In *CAV*. 2005.

[2] T. Arons, E. Elster, T. Murphy, and E. Singerman. Embedded Software Validation: Applying Formal Techniques for Coverage and Test Generation. *Int. Workshop on Microprocessor Test and Verification*, 2006.

[3] T. Arons, E. Elster, S. Ozer, J. Shalev, and E. Singerman. Efficient symbolic simulation of low level software. In *DATE*. ACM, 2008.

[4] G. Audemard and L. Simon. Predicting learnt clauses quality in modern SAT solvers. In *IJCAI*. Morgan Kaufmann, 2009.

[5] D. Babić and A. Hu. Approximating the safely reusable set of learned facts. *STTT*, 11(4), Oct. 2009.

[6] C. W. Barrett, R. Sebastiani, S. A. Seshia, and C. Tinelli. Satisfiability Modulo Theories. In Biere et al. [7]. Part II, Chapter 26.

[7] A. Biere, M. Heule, H. van Maaren, and T. Walsh, editors. *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2009.

[8] 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 *CAV*. 2007.

[9] R. Bruttomesso, A. Cimatti, A. Franzén, A. Griggio, and R. Sebastiani. The MathSAT 4 SMT solver. In *CAV*. 2008.

[10] R. Bruttomesso and N. Sharygina. A Scalable Decision Procedure for Fixed-Width Bit-Vectors. In ICCAD 2009, 2009.

[11] N. Eén and N. Sörensson. Temporal induction by incremental SAT solving. *Electronic Notes in Theoretical Computer Science*, 89(4), 2003.

[12] A. Franzén. *Efficient Solving of the Satisfiability Modulo Bit-Vectors Problem and Some Extensions to SMT*. PhD thesis, DISI – University of Trento, 2010.

[13] P. Godefroid, M. Y. Levin, and D. Molnar. Automated Whitebox Fuzz Testing. Technical Report MSR-TR-2007-58, Microsoft Research Redmond, Redmond, WA, May 2007.

[14] D. Große and R. Drechsler. Acceleration of SAT-Based iterative property checking. In *Correct Hardware Design and Verification Methods*. 2005.

[15] G. Hinton, D. Sager, M. Upton, D. Boggs, D. P. Group, and I. Corp. The Microarchitecture of the Pentium®4 Processor. *Intel Technology Journal*, 1, 2001.

[16] Z. Khasidashvili, A. Nadel, A. Palti, and Z. Hanna. Simultaneous SAT-Based model checking of safety properties. In *Hardware and Software, Verification and Testing*, volume 3875 of *LNCS*. Springer-Verlag, 2006.

[17] S. Khurshid, C. P. asăreanu, and W. Visser. Generalized symbolic execution for model checking and testing. In *TACAS*. 2003.

[18] J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7), 1976.

[19] O. Shtrichman. Pruning techniques for the SAT-Based bounded model checking problem. In *CHARME*. 2001.

[20] J. P. M. Silva and K. A. Sakallah. Robust search algorithms for test pattern generation. In *FTCS*. IEEE Computer Society, 1997.

[21] J. Whittemore, J. Kim, and K. Sakallah. SATIRE: a new incremental satisfiability engine. In *DAC*. ACM, 2001.

Automated Formal Verification of Processors Based on Architectural Models

Ulrich Kühne*
LSV ENS de Cachan
Cachan, France

Sven Beyer†
OneSpin Solutions GmbH
Munich, Germany

Jörg Bormann†
Abstract RT Solutions GmbH
Munich, Germany

John Barstow
Infineon Technologies Ltd
Bristol, UK

Abstract—
To keep up with the growing complexity of digital systems, high level models are used in the design process. In today's processor design, a comprehensive tool chain can be built automatically from architectural or transaction level models, but disregarding formal verification. We present an approach to automatically generate a complete property suite from an architecture description, that can be used to formally verify a register transfer level (RTL) implementation of a processor. The property suite is complete by construction, i.e. an exhaustive verification of all the functionality of the processor is ensured by the method. It allows for the efficient verification of single pipeline processors, including several advanced processor features like multicycle instructions. At the same time, the structured approach reduces the effort for verification significantly compared to a manual complete formal verification. The presented techniques have been implemented in the tool FISACO, which is demonstrated on an industrial processor.

I. Introduction

The complexity of digital hardware systems has shown an exponential growth over the last decades and it is growing still. To keep track of large systems during the design process, high level models are used increasingly. Especially for the design of processors, architecture or transaction level models form the core of an elaborate tool chain that enables the automatic generation of simulators, assemblers or compilers, like *Facile* [27] or *LISA* [9]. However, formal verification of the functionality of the design is still not part of this tool chain.

There exist several techniques for the verification of hardware designs. In simulation based verification, the outputs of the implementation are compared to a golden reference model, that is usually based on a transaction level description. But, simulation is not well suited to cover the whole functionality of a pipelined processor because achieving a sufficient design quality for such a processor requires a huge simulation-based verification effort and there is no guarantee that all possible bugs have been considered. In contrast, formal techniques offer the highest quality of verification [15].

One successful technique is *Interval Property Checking* (IPC) [23], a technique similar to Bounded Model Checking [3]. IPC is used to check if a system satisfies a set of properties about the operations of a design like the processing of a request in a bus bridge, the execution of an instruction in a processor pipeline, or an arbitration cycle in an arbiter. IPC has been extended with further proofs which ensure that a set of properties verifies all input/output behavior of a circuit [5]. This methodology has already been used in industrial context for the verification of a wide variety of designs [12] including small or medium size processors [6].

However, these projects also illustrate that the integration of a thorough formal examination into industrial verification practice requires larger changes to the education and opinions of verification engineers. Compared to simulation based approaches, formal verification requires a deep knowledge of the internals of the design under verification (DUV) in order to write assertions. An important motivation of the work summarized here and presented in [20] is therefore, that the automation of the formal verification of some well defined class of circuits eases the migration from simulation to formal verification and hence helps to introduce this technology. We chose smaller single pipeline processors as this class.

For processors, a structured manual verification flow is available today [2]. But, automation of the verification is quite low, the more comprehensive the verification is. On the other hand, existing approaches for the automatic verification of processors (see related work in Sect. II) often require a background of deep and general insight into verification goals and correctness criteria.

In this paper we present a technique for the automatic generation of a complete property suite for processors. The starting point of the approach is an architecture description of the processor. By defining a number of mapping functions the user captures how the abstract concepts are mapped to the register transfer level (RTL) implementation. These mapping functions refer to pipeline stages, stall and cancel signals, and similar objects that design and verification engineers are familiar with. Following this approach, the specification is captured in a concise and readable form, while the underlying general processor model enables the verification of several advanced processor features like multicycle instructions, out-of-order termination as well as exceptions. The generated property suite is complete by construction in the sense of [5]. As a driving verification engine, the OneSpin 360 MV tool [24] is used, offering the performance and capacity to formally verify whole processor designs.

The main contribution of this work is a well structured yet pragmatic approach to tackle the formal verification of processors. It offers an exhaustive verification for a certain class

This work was carried out jointly by the group for computer architecture at University of Bremen*, Germany and OneSpin Solutions GmbH†. It was supported in part by the German Federal Ministry of Education and Research (BMBF) within the project VerisoftXT under contract no. 01 IS 07008 C.

of designs, while the automatic generation of the properties increases the verification productivity significantly compared to manual coding of properties. As the input for the approach is an abstract architecture description, the method can easily be integrated with existing tool chains for processor design. The automatic generation of the properties is implemented in the tool FISACO. The approach is demonstrated with an industrial control processor used in embedded automotive systems, a domain with particularly high quality requirements.

The paper is structured as follows. Related work is discussed in Sect. II. In Sect. III, our formal verification techniques are reviewed. The automatic property generation is described in Sect. IV. Sect. V shows the application of the approach to an industrial processor design. Sect. VI concludes the work.

II. Related Work

An approach for the automatic equivalence verification of general transaction level models (TLM) with timed implementations is presented in [4], where the different levels of abstraction are related by events. However, the lowest level of abstraction in [4] is behavioral RTL and it is not clear how the concept of events relates to optimized pipeline designs. In other words, an automated equivalence check between a sequential processor architecture and a pipelined RTL implementation is not feasible for optimized industrial designs.

There has been work on the formalization of pipelined designs. Part of the approaches in the literature use formal models for the automatic design of correct pipelines [19], or to accompany the design process [10], [16]. In [10], starting from a simple model, the design is incrementally refined until a pipelined implementation is obtained. A CTL specification is transformed along with the design to prove the correctness of the refinement steps. A similar approach is presented in [22]. It decomposes the correctness proof for a complex pipelined machine with branch prediction into several steps, the first of which proves the compliance of a simple version of the processor with its ISA. The drawback of these approaches is that they cannot handle industrial designs containing legacy code and manual optimizations that are needed to match hard power and timing constraints.

There are various techniques for the verification of existing processors [1], [13], [14], [30]. In [1], a formal pipeline model is introduced that is based on *parcels* (instructions) processing through the pipeline. By instantiating several predicates describing the pipeline, the correctness of the design can be proved formally. However, the model is rather abstract and the predicates seem difficult to derive. In contrast, we provide a clear distinction between the architecture layer and the mapping to the implementation. Furthermore, our mapping functions have a more intuitive counterpart in the designer's intent of implementing a pipeline.

Further approaches for processor verification rely on interactive theorem proving [18], [26], [29]. This generally offers a high level view on the design. Theorem proving however requires a significant level of expertise that is usually not available to designers or verification engineers in practice.

Approaches for the automatic generation of properties are given in [17], [25]; they are based on learning dependencies or properties from simulation traces. However, they are only suited for an initial design understanding rather than for a verification against a specification. In contrast, our approach starts with a specification that is then related to the implementation in a well structured way.

III. Formal Verification Setting

Within the last two decades, there has been a lot of research in formal verification techniques. Methods based on Boolean satisfiability (SAT) have proven to be a robust solution. One prominent technique is SAT based *Bounded Model Checking* (BMC), that has first been described in [3]. Successive improvements in performance have made BMC a suitable method for the formal verification of larger scale designs. For the work at hand, we use the techniques described in [23], referred to as *interval property checking* (IPC). In the following, this verification methodology will be briefly outlined.

In contrast to BMC, only safety properties are verified using IPC. As digital circuits always have a finite response time, this is not a serious restriction in practice. It is rather natural to capture the specification of a design in terms of safety properties. Furthermore, using IPC, these properties can be verified with bounded proofs, which can be checked efficiently using a SAT solver.

The main idea of IPC is to use an arbitrary starting state instead of the initial state used in BMC. Any property that holds starting from an arbitrary state then also holds from any reachable state and thus, it is exhaustively verified. Conversely, false negatives can occur in IPC, i.e. counterexamples for properties starting in unreachable states may be produced. These false negatives need to be removed by adding invariants in order to restrict the starting state. For more details on the idea of IPC and the following formalization, refer to [23].

A synchronous circuit is modeled as a finite state machine (FSM) $M = (I, S, S_0, \Delta, \Lambda, O)$ with input alphabet $I \subseteq \mathbb{B}^n$, output alphabet $O \subseteq \mathbb{B}^w$, a finite set of states $S \subseteq \mathbb{B}^m$, output function Λ and next state function Δ. The set $S_0 \subseteq S$ denotes the initial states. With next state function $\Delta : \mathbb{B}^n \times \mathbb{B}^m \to \mathbb{B}^m$, the transition relation of the circuit is given by

$$T(s, s') \;=\; \exists x \in \mathbb{B}^n : s' \equiv \Delta(x, s). \tag{1}$$

A safety property $f = \mathsf{AG}(\varphi)$ is translated to a Boolean function $[[f]]_t$, checking the validity of formula φ at time-point t. Here, the translation is done such that a satisfying assignment of $[[f]]_t$ corresponds to a counterexample of φ. The resulting function depends on the inputs, outputs and states within a bounded time interval $[0, c]$. IPC searches for counterexamples by solving the SAT instance

$$\bigwedge_{i=0}^{c} T(s^{t+i}, s^{t+i+1}) \wedge [[f]]_t. \tag{2}$$

The transition relation is unrolled within the time interval $[0, c]$ and it is connected to the single instantiation of $[[f]]_t$. In order to avoid unreachable counterexamples, invariants are added. In many cases, such invariants can even be generated automatically [31]. In the context of the described methodology, the needed invariants are usually less complex than

Fig. 1. Verification Flow with Property Generation

the main properties; they can thus be verified using inductive proof techniques like *k-induction* [28]. For more details on the method, refer to [23].

IPC is a powerful verification technique, enabling the formalization of a specification in terms of safety properties and its verification against the implementation. However, to be sure that no bugs have been missed, the verification engineer needs to reason about the *completeness* of the written property suite. A technique to formally check whether a set of properties forms a complete specification is described in [5], [8]. These techniques have been successfully applied to industrial processor designs [6].

Completeness analysis determines whether every possible input scenario—corresponding to a transaction sequence of the design—can be covered by a chain of properties that predicts the value of states and outputs at every point in time. In other words, any two designs fulfilling all the properties of a complete property suite are formally equivalent. The completeness analysis basically boils down to check in the end state of each property whether (1) there is always a successor property with matching assumptions, (2) the successor property is uniquely determined and (3) each property describes the outputs and states of the design uniquely. For more details on the methodology please refer to [5], [8].

For the formal verification of the generated property suite against the RTL, we use OneSpin 360MV [24]. This commercial solution covers the required spectrum of formal verification—from the verification of SystemVerilog assertions all the way to the automatic completeness analysis described above. Among various other proof engines, 360MV also offers IPC and k-induction with sufficient capacity and performance to handle the complete verification of processors [6].

IV. Verification Using Generated Properties

Technically, the basis of the approach presented here is to provide a general formal processor model that can be customized by the user to match his specific implementation. The general processor model can be thought of as a tool box with several design features to be picked out. The customization is done by setting the architecture design parameters, like the number of pipeline stages and the possible interface transactions. Furthermore the mapping from the architecture description to the RTL has to be established by defining a number of mapping functions. The basic flow is shown in Fig. 1. The general processor model consists of three parts:

1) The **pipeline model** describes the movement of the instructions through the stages
2) The **datapath model** describes register access and data forwarding
3) The **interface model** describes memory and bus accesses

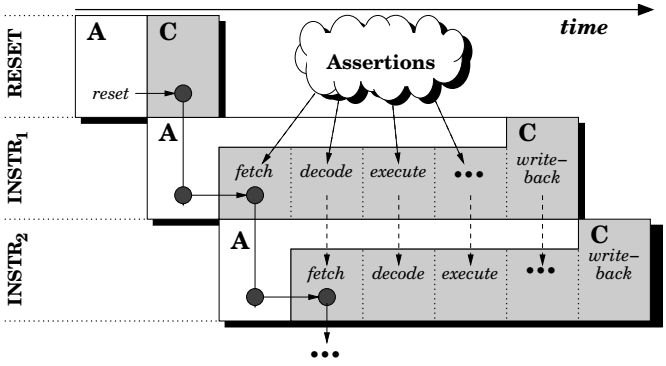


Fig. 2. Interaction of generated properties

After identifying the visible registers and interfaces, the instruction set and the exception behavior of the processor are described on the architecture level. The generated property suite consists of *instruction properties* and a set of *consistency assertions*. While each instruction property describes the processing of a single instruction until it leaves the pipeline, the consistency assertions ensure the correct interaction of multiple instructions and the consistent pipeline behavior, if no instruction is present in a dedicated stage. The latter includes e.g. checking that empty stages will not update any state elements. These assertions also help the user in finding an appropriate mapping by giving him a feedback for debugging.

Basically, the equivalence of the property suite and the DUV is established by chaining the generated properties, as shown in Figure 2. Each property is depicted as a rectangular box, consisting of an assume part (*assumption A*) and prove part (*consequent C*, shaded gray in the figure). The properties are hooked up at the time point when the processor is ready to execute the next instruction, represented by the big black dots in the picture. Thus, starting from reset, the first property proves that the *new instruction state* (NIS) will be reached. Then, the following properties assume the NIS and prove that after fetching the dedicated instruction, NIS will be reached again, enabling the connection to the next instruction property.

The basic approach has been described in detail in [20]; it is based on a patent application [7].

A. General Processor Model

The approach presented here is limited to a class of processors that is common in industrial designs. The class is characterized by the following features:

- Single pipeline
- In-order-execution, out-of-order termination
- Register files with multiple prioritized write channels
- Exceptions and interrupts
- Delayed branch instructions
- Branch prediction
- Multicycle instructions
- Multiple interfaces, including pipelined protocols

Note that a typical CPU also contains complex data memory and prefetch logic. With our approach, the core of such a CPU can be verified with generated properties, providing exact interface descriptions to the data memory and prefetch.

These modules can in turn be verified manually, thus ensuring correctness of the overall CPU. In such manual extensions, the already established mappings and models can easily be reused.

The components that are included in the architecture view are described in the following. A processor receives its instructions via an instruction memory interface, that is addressed by a program counter PC. The currently processed instruction word is denoted IW. There can be an arbitrary number of architecture registers and flags. There can be interfaces to data memories or buses, each associated with a set of transactions, at least containing the idle transaction.

The instructions are described on the architecture level. In order to verify them against the RTL implementation, a mapping has to be established by the user. For the components PC and IW, the corresponding mapping function is usually pointing to a dedicated signal in the RTL showing the value of the program counter and an instruction register, respectively. However, the mapping of an architecture register file requires a model of the pipelining due to forwarding mechanisms that are part of every pipelined processor design. We first introduce the architecture description, followed by a discussion of the models for the pipeline, the data path, and the interfaces. Finally, the generation of the property suite is described, and the completeness of the model is discussed.

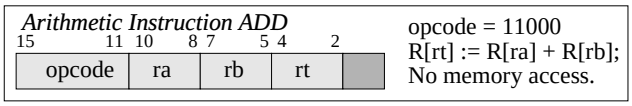
B. Architecture Description

In our approach, there is a clear distinction between the architecture description and the mapping functions. In this way, a readable and proven correct description of the ISA is obtained. The mapping functions relate the ISA to the RTL.

In the first section of the architecture description, the components of the processor are listed, comprising all architecture registers and flags. Furthermore the interfaces to memories and buses are given, as well as the respective transaction types on these interfaces. The main section of the architecture description consists of the ISA description, where all instructions of the processor are defined. In the ISA description, the registers are referred to by their specification name.

For each instruction, first the execution condition is given (`TRIGGER`). Then, the updates of the program counter and the architecture registers and flags need to be defined, followed by the definition of one transaction per interface. The updates are defined by the read registers (`VREGISTER`), the target register (`UREGISTER`) and the value that will be written by the instruction (`UPDATE`).

As an example, consider Fig. 3(b) with a simple processor description including an `ADD` instruction. The triggers for the instruction are divided into two statements, one of which only depends on the architecture state (`TRIGGER_STATE`), while the second one depends on the instruction word (`TRIGGER_IW`). Besides the update of the program counter in line 8, there is one update of the register `R`, where two registers are read addressed by parts of the instruction word (lines 9 and 10). The target register is given in line 11 and the sum of the two source registers is defined in line 12. Finally, there is no transaction on the data memory interface, indicated by the statement `DMEM_IDLE` in line 13.



(a) Specification

```
1  registers  := R;
2  interfaces := DMEM;
3  transactions_DMEM := IDLE, READ, WRITE;
4
5  simple_instruction ADD {
6    TRIGGER_STATE := true;
7    TRIGGER_IW := IW[15:11] == ADD_op;
8    UPDATE_PC := (PC + 2)[7:0];
9    VREGISTER_1 := R(IW[10:8]);
10   VREGISTER_2 := R(IW[7:5]);
11   UREGISTER_1 := R(IW[4:2]);
12   UPDATE_1 := (VREGISTER_1 + VREGISTER_2)[15:0];
13   DMEM_IDLE; }
```

(b) Architecture description

Fig. 3. Informal specification and architecture description example

C. Pipeline Model

In a pipeline, the processing of instructions is overlapped in order to speed up computations. Thus, a new instruction starts before the preceding one has terminated. For example, a typical simple pipeline would partition an instruction into fetching the instruction word from the memory, decoding it, executing logical and arithmetic operations and writing the result back into the register file. Note that this section only introduces basic pipeline modeling for the control path in order to keep track of the different instructions in the pipeline. The handling of forwarding is part of the data path of a pipeline and discussed in the following Sect. IV-D.

The major challenge in designing a correct pipeline are hazards, i.e. conflicts between instructions that are processed at the same time in different stages. If an instruction needs data that is currently being computed by a preceding instruction, a read-after-write conflict occurs and the succeeding instruction needs to wait for the data. Thus, a mechanism to *stall* a stage is needed. Another hazard is related to branching instructions. When a jump is taken, this is typically detected at a time when subsequent instructions from the sequential program flow already have been fetched. Therefore, the pipeline must possibly be cleaned from wrongly fetched instructions, requiring a *cancel* mechanism. As this may lead to stages that are not processing any instructions, it is desirable to distinguish between empty and *full* stages to prevent spurious register updates or similar faults. Based on these requirements, we now define our pipeline model.

Given the number of pipeline stages n, we define the set $\mathcal{S} = \{1, 2, \ldots n\}$ of pipeline stages. The pipeline architecture is further classified by defining some constants that refer to certain stages like the decode stage $dec \in \mathcal{S}$ and the stages $ia, iv \in \mathcal{S}$ that denote the stage when the instruction memory is accessed and when the instruction word is valid, respectively. The processing of instructions by the pipeline is defined by the mapping functions[1] $full, stall, cancel : \mathcal{S} \to \mathbb{B}$.

The value of $full(s)$ reflects if the pipeline stage s currently

[1]The state of the design is an *implicit* parameter of all mapping functions.

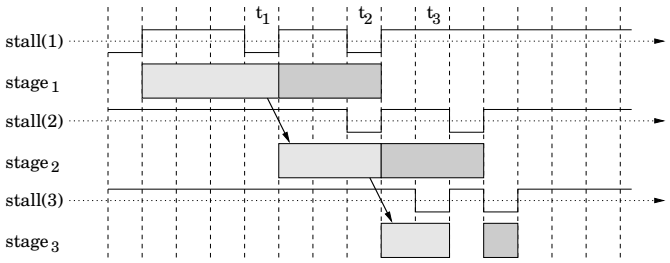


Fig. 4. Normal processing of instructions by the pipeline

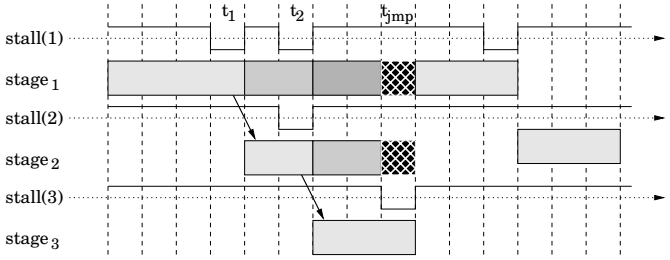


Fig. 5. Pipeline for a taken jump instruction

holds an instruction. If $stall(s)$ is true, the instruction in stage s will not proceed to the succeeding stage $s+1$. $cancel(s)$ indicates that the instruction currently in stage s will be removed from the pipeline and will have no more effects in later stages. The normal processing of two consecutive instructions is shown in Fig. 4, where time progresses from left to right. The time-points when the first instruction is allowed to proceed to the next stage are denoted by t_1 to t_3, i.e., $stall(s)$ evaluates to false at t_s. The boxes indicate the time-points when the respective stage is processing an instruction, i.e. $full(s) = 1$. The pipeline for a taken jump or mispredicted branch instruction is shown in Fig. 5. At timepoint t_{jmp}, the canceling of two succeeding instructions is indicated by the dark boxes. After the taken jump, the target instruction is fetched from the instruction memory.

The mapping functions have to be defined by the user. This means for example, that the user needs to identify how the implementation encodes the fact that a stage is full. Since the functions are used in the properties, the verification fails as long as the model is not completed properly.

In addition to the basic model, further pipeline operations can be supported. It is common for instructions to leave the pipeline before the last stage, if no more actions will be taken in later stages, in order to prevent conflicts. Thus, a *last stage* can be defined for each instruction. Exceptions are a crucial feature for practical applications. By nature, they interrupt the normal instruction processing. The most general exception model, that is still suited to conform with our approach, is an *injection* of a new instruction into the pipeline after an exception has been acknowledged. Finally, for more complex arithmetic operations or interactions with protocol driven interfaces, multicycle instructions are frequently used in processor designs. Typically, an FSM in an early stage is responsible for *dispatching* partial instructions in the pipeline.

For these refinements of the simple model, additional mapping functions for of out-of-order termination, exceptions, and multicycle instructions need to be defined.

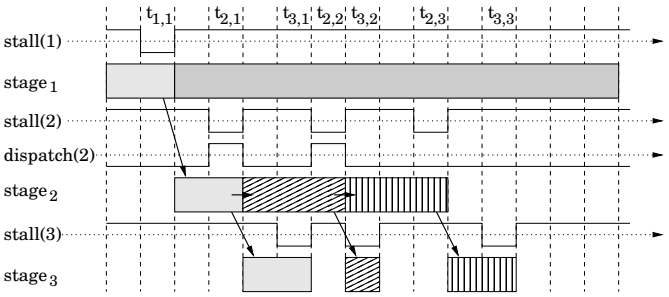


Fig. 6. Pipeline for a multicycle instruction

$$last\_stage, inject, dispatch : \mathcal{S} \to \mathbb{B}, \tag{3}$$

Here, $last\_stage(s)$ indicates that the instruction in stage s will leave the pipeline, $inject(s)$ states that an instruction will be injected into stage s in the next cycle due to an exception, and $dispatch(s)$ describes that a multicycle instruction is started in stage s. The pipeline of a multicycle instruction according to our model is shown in Fig. 6. There, the partial instructions are dispatched in stage 2.

D. Data Path Model

Based on the above control path model of the pipeline, we can now define the data path model, describing the way how data is read, forwarded and stored in the registers.

For a register file R, a mapping function $current_R : \mathcal{I}_R \to \mathcal{D}_R$ is defined that returns the current implementation state of the register, where $\mathcal{I}_R$ is the index set and $\mathcal{D}_R$ is the data domain of register R. For the data path of the register the following mapping functions have to be defined:

$$write_R, valid_R : \mathcal{S} \to \mathbb{B} \tag{4a}$$

$$dest_R : \mathcal{S} \to \mathcal{I}_R \tag{4b}$$

$$data_R : \mathcal{S} \to \mathcal{D}_R, \tag{4c}$$

where $write_R(s)$ indicates if the instruction in stage s is going to update register R, while $dest_R(s)$ and $data_R(s)$ specify the target register and the data to be written, respectively. By $valid_R(s)$, it is stated if stage s already produces a valid result. With these building blocks, the forwarding in the pipeline to some forwarding target stage $s \in \mathcal{S}$ can easily be captured: the value of a register R with index $i \in \mathcal{I}_R$ in the forwarding target stage s is recursively given by checking whether succeeding stages write to register i; this corresponds to the forwarding logic in the pipeline.

$$Data_R(s,i) = \begin{cases} current_R(i), & \text{if } s \geq writeback_R; \\ data_R(s+1), & \text{if } write_R(s+1) \wedge \\ & dest_R(s+1) = i; \\ Data_R(s+1,i), & \text{otherwise.} \end{cases}$$

Note that this automatically generated function $Data_R$ actually captures the complex mapping of the visible register R to the implementation, i.e., the architecture value of R for an instruction in the pipeline is the value of $Data_R$ in the

forwarding target stage of that instruction. Since the value of $Data_R$ may be invalid because the result of some instruction is not available yet, we introduce an additional mapping function capturing whether the forwarding data is indeed valid:

$$Valid_R(s,i) = \begin{cases} false, & \text{if } s < dec; \\ true, & \text{if } s \geq writeback_R; \\ valid_R(s+1), & \text{if } write_R(s+1) \wedge \\ & dest_R(s+1) = i; \\ Valid_R(s+1,i), & \text{otherwise.} \end{cases}$$

E. Interface Transactions

In order to verify the interfaces of the processor, the following model is used. For each interface IF the constants $da_{IF}, dv_{IF} \in \mathcal{S}$ denote the stage when a data access is issued and when valid data is returned, respectively. For each interface, a set of transactions TA_{IF} is defined by the user with at least $IDLE \in TA_{IF}$. For each transaction $ta \in TA_{IF}$ a function $ta_{IF} : \mathbb{N} \times \mathbb{N} \to \mathbb{B}$ is defined, where $ta_{IF}(addr, wdata)$ captures that the specified transaction takes place in the design, optionally involving the address $addr$ and (for writing transactions) the write data $wdata$. As for the example in Fig. 3(b), the three functions $IDLE_{DMEM}, READ_{DMEM}$ and $WRITE_{DMEM}$ need to be defined, capturing for given address and data, if the respective transaction is issued on the data memory interface.

Besides the transactions, for each interface a mapping function $rdata_{IF}$ points to the implementation port, where data is read in to the processor. Finally, there is a static interface to the instruction memory, given by the predicate $ibus\_read : \mathbb{N} \to \mathbb{B}$, which checks if the instruction memory is currently being accessed for a given value of the program counter.

F. Consistency Assertions

While the above models describe the processing of instructions by the successive pipeline stages, additional assertions are needed for the overall correctness of the processor. This includes the behavior of empty pipeline stages as well as the interaction of succeeding instructions. For this purpose, a set of *consistency assertions* are automatically generated.

Note that the overall verification is fail safe, i.e. it cannot succeed if the design is not correct. But, even for a correct design, finding the appropriate mapping functions can be difficult. The consistency assertions provide useful information on the status of the modeling. Failing assertions can point the user to certain mapping functions that need to be revised to complete the verification, thereby guiding the debugging process.

We show the following assertion as an example. For a more detailed description of the consistency assertions, see [20].

$$\begin{array}{l} \forall s, 2 \leq s \leq n : \\ (\ (\neg full^t(s-1) \vee stall^t(s-1)) \wedge \\ (\neg full^t(s) \vee \neg stall^t(s))\ ) \Rightarrow \neg full^{t+1}(s) \end{array} \quad (5)$$

This assertion states that it is illegal to create full stages in the middle of the pipeline: when the stage before s is empty or stalled, and s is empty or it will proceed to the next stage, then s must be empty in the next cycle. Here, f^t denotes the value of f at timepoint t. Other assertions ensure, for example, that instructions do not overwrite each other and that empty pipeline stages do not have an effect on the visible registers or issue interface transactions.

TABLE I
USER INPUT FOR PROPERTY GENERATION

(a) Constants

Name	Domain	Description
n	$\mathbb{N}$	number of stages
dec	$\mathcal{S} = \{1, \ldots, n\}$	decode stage
ia	$\mathcal{S}$	instruction memory access stage
iv	$\mathcal{S}$	stage in which instr. word is valid
int	$\mathcal{S}$	highest stage for interrupt injection
da_{IF}	$\mathcal{S}$	access stage for interface IF
dv_{IF}	$\mathcal{S}$	data valid stage for interface IF
$writeback_R$	$\mathcal{S}$	writeback stage for register R

(b) Mapping functions

Arch.	Function	Signature	Description
	Basic components		
PC	pc	$\mathbb{N}$	program counter
IW	iw	$\mathbb{N}$	instruction word
	Pipeline Model		
	$full$	$\mathcal{S} \to \mathbb{B}$	stage active
	$stall$	$\mathcal{S} \to \mathbb{B}$	stage stalled
	$cancel$	$\mathcal{S} \to \mathbb{B}$	stage is canceled
	$inject$	$\mathcal{S} \to \mathbb{B}$	inject launch instr.
	$dispatch$	$\mathcal{S} \to \mathbb{B}$	dispatch micro instr.
	$last\_stage$	$\mathcal{S} \to \mathbb{B}$	instr. leaves pipeline
	Datapath Model		
R	$current_R$	$\mathcal{D}_R$	implementation register
R	$write_R$	$\mathcal{S} \to \mathbb{B}$	stage will write
R	$dest_R$	$\mathcal{S} \to \mathcal{I}_R$	write destination
R	$data_R$	$\mathcal{S} \to \mathcal{D}_R$	write data
R	$valid_R$	$\mathcal{S} \to \mathbb{B}$	data is valid
	Interfaces		
	$ibus\_read$	$\mathbb{N} \to \mathbb{B}$	instruction fetch
IF_TA	ta_{IF}	$\mathbb{N} \times \mathbb{N} \to \mathbb{B}$	transaction
IF_RDATA	$rdata_{IF}$	$\mathbb{N}$	read data

G. Generating The Property Suite

In order to adapt the general processor model to the actual DUV, the user needs to specify the mapping functions described in Sections IV-C to IV-E. The user input is summarized in Table I. Besides the basic data on the pipeline, given by a set of constants, the table shows the mapping functions corresponding to the architectural components of the general processor model.

During the generation of the property suite, an architecture register $R(i)$ is replaced by an instantiation of the function $Data_R(s_{fwd}, i)$, where s_{fwd} is the forwarding target stage, which is usually the decode stage.

The generated properties prove the correctness of the instructions on the implementation level. For this, we define t_0 to be the timepoint when the respective instruction enters the pipeline and $t_i > t_{i-1}, 1 \leq i \leq n$ to be the timepoints when the instruction is allowed to proceed from stage i (see also Fig. 4–6). The properties have an implication structure

$A \Rightarrow C$. Whenever the assumptions A evaluate to true, the prove part C must hold as well. In the following, we give an overview of the templates for instruction execution without exceptions. There are similar templates for exceptions; for more details, see [7]. Note that there are two templates for conditional branches depending on whether the branch is taken or not. In this way, branch prediction can easily be modeled. The assume part for an instruction m basically consists of the following assumptions:

- The instruction enters the pipeline at t_0.
- In decode, instruction m is triggered.
- The instruction proceeds from stage i at t_i, $i \leq 1 \leq n$.
- The instruction is not canceled by preceding instructions and not replaced by an exception call.

For each instruction, mainly the following will be proved:

- The instruction is fetched from the instruction memory
- The program counter is updated correctly
- The full stages are correctly tagged by the $full$ function
- No $cancel$ is generated (except for jump instructions)
- All read registers are valid
- The registers will be updated (or remain stable) corresponding to the ISA; this includes the verification of correct forwarding
- The correct transactions will take place on the interfaces

H. Completeness

The pipeline model is built such that the final property suite in combination with the consistency assertions is complete by construction, if some rules are respected for the definition of the mapping functions. For a proof for the basic pipeline model, see [7].

However, the completeness of a concrete generated property suite additionally depends on the proper definition of some of the functions. If, for example, the user defined the function for a read transaction by simply returning true, it is obvious that the interface signals are not checked at all for read transactions and there is a gap in the verification. In summary, the generation ensures that all possible scenarios are covered with properties, but not that all transactions verify all outputs. However, the automatic gap detection of OneSpin 360 can be used to close these gaps as well.

V. Application

The above method has been implemented as a front-end for OneSpin 360 MV; we call it FISACO (Formal Instruction Set Architecture Compiler). It takes an architecture description and automatically generates the instruction properties and the consistency assertions in a form readable for 360MV. The mapping information needs to be supplied in a temporal logic format. The processor model formed by both the architecture description and the mapping information can then be verified and debugged using 360MV.

In the following we will describe the application of the proposed method on an industrial processor design. We successfully verified a control processor that is used in automotive applications, the *Peripheral Control Processor* (PCP) by Infineon Technologies. First, the basic data of the PCP will be given, followed by a presentation of the verification results. Besides, during its development, the method has been applied for the complete verification of smaller processor designs from the opencores site (`www.opencores.org`). Details cannot be given here due to page limitation.

A. PCP Processor

The PCP processor is a control processor that is part of automotive systems. Its main purpose is the monitoring of peripheral components in order to release the central CPU [11]. Therefore, a great share of the instruction set is dedicated to data transfer and bit operations, which are frequently used in typical control applications. The PCP is connected to a data memory and a pipelined FPI bus (*Flexible Peripheral Interface*). As the bus operations require complex protocol transactions, 35% of the instruction set are multicycle instructions. In total, the PCP has 66 instructions, divided into arithmetic/logic instructions, jump and control, memory instructions, bus instructions and complex math instructions.

The processor is implemented as a four stage pipeline. The register file contains 8 registers of 32 bit, where one register is a special purpose register containing various status flags and the program counter. The whole RTL implementation adds up to about 17.000 lines of VHDL code, accompanied by a detailed informal specification. Regarding the complexity of the design and the quality of the source code and documentation, the time for the formal verification was estimated with 8 to 10 person months, needed to manually write a complete property suite using OneSpin 360 MV.

B. Results

The PCP has been verified using the presented approach. The informal specification was ported to an architecture description. Most of the manual effort was spent for the definition and refinement of the pipeline and datapath model, given by the mapping functions explained in Sect. IV-C to IV-E. Using our approach, the instruction set of the PCP could be successfully verified except for two highly complex bus instructions involving nested loops and excluding three complex math instructions. For these instructions, the control mechanisms of the PCP did not match our general pipeline model. It does not seem useful to extend the model for these cases, as they are very specific to the PCP implementation. Instead, having found a good representation of most of the functionality based on our processor model, the defined functions can be reused for further manual verification. This has also been done for some additional functionality beyond the ISA, like loading and storing full register contexts. The overall verification of the PCP with our methodology took about 5 person months. Thus, we could achieve an estimated productivity gain of 100%.

The verification has been carried out on 2.2 GHz workstation with 16 GB memory. Details on the proof times and the used memory can be found in Table II. As can be seen, the generated consistency assertions could be proved quickly. Most of the time was spent for the verification of arithmetic and bus instructions. The latter ones are mostly multicycle instructions and thus the design needs to be unrolled for up to

TABLE II
VERIFICATION RESULTS

Category	Properties	Total time	Avg. time	Memory
assertions	34	600 s	17.6 s	937 MB
arithmetic	15	18788 s	1252.5 s	2500 MB
logic/bit	18	3368 s	187.1 s	2500 MB
jump	7	220 s	31.4 s	2500 MB
memory	16	2135 s	133.4 s	2500 MB
bus	10	3214 s	321.4 s	2500 MB
other	3	147 s	49.1 s	1763 MB
total	103	7:54 h		

26 cycles. Note that the two most difficult instructions make up 3 hours and 48 minutes or 48% of the total runtime.

VI. CONCLUSIONS

We have presented an approach for the automatic generation of a complete property suite from an architecture description of a processor. There is a clear distinction between the architecture model and the mapping information connecting architecture to RTL implementation. The architecture model can be easily derived from an informal specification.

The mapping from the specification to the implementation is based on a general pipeline model that reflects the designer's intent in implementing a correct pipelined processor. A set of consistency assertions is automatically generated to check the correctness of the model and helps the user in finding a suitable mapping. When the mapping and the architecture description are finished, the generated property suite forms a model of the design, i.e. the verification is exhaustive.

The practicability of the approach has been demonstrated on an industrial processor design, a control processor from the automotive domain. With the presented methodology, the estimated verification productivity could be doubled.

In the future, we want to integrate this approach with our automatic generation of efficient instructions set simulators (ISS) [21]. This allows to generate both a complete property suite and an efficient ISS from a common architecture description, ensuring that the generated ISS complies to the verified RTL code. A complementary extension would be the use of existing ADL like *LISA*, facilitating the integration of formal methods into the tool chain for processor design.

REFERENCES

[1] M. Aagaard. A hazards-based correctness statement for pipelined circuits. In *CHARME*, pages 66–80, 2003.

[2] S. Beyer, C. Jacobi, D. Kroening, D. Leinenbach, and W. Paul. Putting it all together—formal verification of the VAMP. *STTT Journal*, 8(4–5):411–430, Aug. 2006.

[3] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 1579 of *LNCS*, pages 193–207. Springer Verlag, 1999.

[4] N. Bombieri, F. Fummi, G. Pravadelli, and J. Marques-Silva. Towards equivalence checking between TLM and RTL models. In *Int'l Conf. on Formal Methods and Models for Codesign*, pages 113–122, 2007.

[5] J. Bormann. *Vollstaendige funktionale Verifikation*. PhD thesis, University of Kaiserslautern, 2009.

[6] J. Bormann, S. Beyer, A. Maggiore, M. Siegel, S. Skalberg, T. Blackmore, and F. Bruno. Complete formal verification of TriCore2 and other processors. In *Design and Verification Conference (DVCon)*, 2007.

[7] J. Bormann, S. Beyer, and S. Skalberg. Equivalence verification between transaction level models and RTL at the example of processors, 2008. European Patent Application, publication number EP1933245.

[8] J. Bormann and H. Busch. Method for determining the quality of a set of properties, applicable for the verification and specification of circuits, 2007. European Patent Application, publication number EP1764715.

[9] G. Braun, A. Nohl, A. Hoffmann, O. Schliebusch, R. Leupers, and H. Meyr. A universal technique for fast and flexible instruction-set architecture simulation. *IEEE Transactions on CAD*, 23(12):1625–1639, 2004.

[10] C. Braunstein and E. Encrenaz. Formalizing the incremental design and verification process of a pipelined protocol converter. In *IEEE Int'l Workshop on Rapid System Prototyping (RSP)*, pages 103–109, 2006.

[11] S. Brewerton. Dual core processor solutions for IEC61508 SIL3 vehicle safety systems. In *Embedded World Conference*, 2007.

[12] R. Brinkmann, P. Johansson, and K. Winkelmann. *Advanced Formal Verification*, chapter Application of Property Checking and Underlying Techniques, pages 125–166. Kluwer Academic Publishers, 2004.

[13] R. Bryant, S. German, and M. Velev. Processor verification using efficient reductions of the logic of uninterpreted functions to propositional logic. *ACM Transactions on Computational Logic*, 2(1):93–134, 2001.

[14] J. Burch and D. Dill. Automatic verification of pipelined microprocessor control. In *Proc. of the Int'l Conf. on Computer Aided Verification*, pages 68–80, 1994.

[15] W. Büttner. Complex hardware modules can now be made free of functional errors without sacrificing productivity. In *Proc. of Int'l Conf. on Abstract State Machines, B and Z*, LNCS, pages 1–3. Springer, 2008.

[16] A. Chattopadhyay, A. Sinha, D. Zhang, R. Leupers, G. Ascheid, and H. Meyr. Integrated verification approach during ADL-driven processor design. In *IEEE Int'l Workshop on Rapid System Prototyping (RSP)*, pages 110–118, 2006.

[17] G. Fey and R. Drechsler. Design understanding by automatic property generation. In *Workshop on Synthesis And System Integration of Mixed Information technologies (SASIMI)*, pages 274–281, 2004.

[18] R. Hosabettu, G. Gopalakrishnan, and M. Srivas. Verifying advanced microarchitectures that support speculation and exceptions. In *Proc. of the Int'l Conf. on Computer Aided Verification*, pages 521–537, 2000.

[19] D. Kroening and W. J. Paul. Automated pipeline design. In *Proc. of the 38th Conference on Design Automation*, pages 810–815. ACM, 2001.

[20] U. Kühne. *Advanced automation in formal verification of processors*. PhD thesis, University of Bremen, 2009.

[21] U. Kühne, S. Beyer, and C. Pichler. Generating an efficient instruction set simulator from a complete property suite. In *Proc. of the IEEE Int'l Symposium on Rapid System Prototyping*, 2009.

[22] P. Manolios and S. Srinivasan. A complete compositional reasoning framework for the efficient verification of pipelined machines. In *Proc. of the Int'l Conf. on Computer Aided Design*, pages 863–870, 2005.

[23] M. Nguyen, M. Thalmaier, M. Wedler, J. Bormann, D. Stoffel, and W. Kunz. Unbounded protocol compliance verification using interval property checking with invariants. *IEEE Transactions on CAD*, 27(11):2068–2082, Nov. 2008.

[24] OneSpin Solutions GmbH, Munich, Germany. *OneSpin Verification Solutions*. http://www.onespin-solutions.com, 2009.

[25] F. Rogin, T. Klotz, G. Fey, R. Drechsler, and S. Rülke. Automatic generation of complex properties for hardware designs. In *Design, Automation and Test in Europe*, 2008.

[26] J. Sawada and W. Hunt. Processor verification with precise exceptions and speculative execution. In *Proc. of the Int'l Conf. on Computer Aided Verification*, pages 135–146, 1998.

[27] E. Schnarr, M. Hill, and J. Larus. Facile: a language and compiler for high-performance processor simulators. In *Proc. of the Conf. on Programming language design and implementation*, pages 321–331, 2001.

[28] M. Sheeran, S. Singh, and G. Stålmarck. Checking safety properties using induction and a SAT-solver. In *FMCAD*, pages 127–144. Springer, 2000.

[29] J. Skakkebæk, R. Jones, and D. Dill. Formal verification of out-of-order execution using incremental flushing. In *Proc. of the Int'l Conf. on Computer Aided Verification*, pages 98–109, 1998.

[30] M. Velev. Formal verification of pipelined microprocessors with delayed branches. In *Int'l Symp. on Quality Electronic Design*, page 4pp., 2006.

[31] M. Wedler, D. Stoffel, and W. Kunz. Exploiting state encoding for invariant generation in induction-based property checking. In *Asia and South Pacific Design Automation Conference*, pages 424–429, 2004.

Encoding Industrial Hardware Verification Problems into Effectively Propositional Logic

Moshe Emmer, Zurab Khasidashvili
Intel Israel Design Center
Haifa 31015, Israel
{memmer,zurabk}@iil.intel.com

Konstantin Korovin, Andrei Voronkov
School of Computer Science,
University of Manchester, UK
korovin@cs.man.ac.uk, andrei@voronkov.com

***Abstract*—Word-level bounded model checking and equivalence checking problems are naturally encoded in the theory of bit-vectors and arrays. The standard practice of deciding formulas of such theories in the hardware industry is either SAT- (using bit-blasting) or SMT-based methods. These methods perform reasoning on a low level but perform it very efficiently. To find alternative potentially promising model checking and equivalence checking methods, a natural idea is to lift reasoning from the bit and bit-vector levels to higher levels. In such an attempt, in [14] we proposed translating memory designs into the Effectively PRopositional (EPR) fragment of first-order logic.**

The first experiments with using such a translation have been encouraging but raised some questions. Since the high-level encoding we used was incomplete (yet avoiding bit-blasting) some equivalences could not be proved. Another problem was that there was no natural correspondence between models of EPR formulas and bit-vector based models that would demonstrate non-equivalence and hence design errors.

This paper addresses these problems by providing more refined translations of equivalence checking problems arising from hardware verification into EPR formulas. We provide three such translations and formulate their properties. All three translations are designed in such a way that models of EPR problems can be translated into bit-vector models demonstrating non-equivalence.

We also evaluate the best EPR solvers on industrial equivalence checking problems and compare them with SMT solvers designed and tuned for such formulas specifically. We present empirical evidence demonstrating that EPR-based methods and solvers are competitive.

I. Introduction

Use of theorem proving in hardware and software verification is not new. A first classification of the use of theorem proving in formal verification would be to divide it into Higher-Order Logic (HOL) and First-Order Logic (FOL) theorem proving. Because HOL theorem proving is highly interactive and requires from the user both an expertise in theorem proving and a good familiarity of the design (or program) under verification, the use of higher-order theorem proving in hardware verification is limited to particular styles of design for which no good fully-automatic verification methods exist.[1] Unlike HOL, there are highly efficient fully automatic FOL theorem provers, so the potential of FOL for a wider use in formal verification is significantly higher.

In this paper we are interested in equivalence checking and model checking problems in hardware verification involving decision procedures for bit-vectors and arrays. Such problems can be solved efficiently by Satisfiability Modulo Theories (SMT) [16] solvers [5], [6], [21]. More precisely, we are interested in problems in the theory of fixed-size bit-vectors and extensional arrays, known as the theory QF_AUFBV. It has also been shown [14] that such problems can be encoded into the Effectively Propositional (EPR) fragment of FOL, which is decidable and for which efficient FOL solvers exist [20], [15], [3]. The EPR fragment consists of first-order formulas which in clausal normal form contain no function symbols other than constants.

The current understanding (on which many experts in the field agree) is that FOL solvers are good at "pure first-order problems" involving formulas with (interleaving and nested) universal and existential quantifiers, while SMT solvers are best at quantifier-free theories.[2] In this paper we set out to investigate the scalability of EPR solvers with different proof-calculi to real-life problems involving reasoning with bit-vectors and arrays, and comparing their performance with the best SMT solvers for the theory QF_AUFBV. We propose several sound and complete encodings of problems in this theory into EPR, and discuss and experimentally evaluate the advantages and disadvantages of different encodings. We also discuss and experimentally evaluate advantages and disadvantages of different proof calculi for FOL with respect to solving the EPR problems arising from industrial scale hardware verification.

To the best of our knowledge, no similar analysis was reported before. We find this analysis interesting and important especially because the significance of the EPR fragment in software and hardware verification has been realized only recently [17], [18], by showing that many interesting verification problems can be encoded in this fragment and can often be solved efficiently. We hope that the theoretical and experiential analysis reported in this work will help in cross-learning between the calculi and algorithms employed in SMT and FOL approaches, for the class of problems with bit-vectors and arrays.

This work is partially supported by EPSRC and the Royal Society.

[1] This by no means diminishes the importance of the use of HOL theorem proving in verification – in certain areas of verification it is indispensable.

[2] Few SMT solvers, like Z3, do support limited quantified theories; see [22] for further references.

In the next section, we recall a sound but incomplete encoding of problems with bit-vectors and arrays into EPR, as described in [14]. As a consequence of incompleteness, the powerful abstraction of the size of bit-vectors and arrays on which the encoding to EPR is based, is often the source of false counter-examples in verification. Debugging of verification failures is the main source of inefficiency in hardware design projects, and false failures (also called false negatives, caused by the nature of the verification tool or methodology rather than an actual bug in the design) are simply unacceptable. In Section III, we therefore propose several approaches to achieving the completeness of encoding to EPR, thereby eliminating the possibility of false negatives.

In Section III, and further in Section IV, we analyze the advantages and disadvantages of the proposed sound and complete encodings to EPR, and relate these to the strengths and weaknesses, relative to the problems we are interested in, of several important proof calculi employed in the best EPR solvers (the winners of recent theorem proving competitions in the EPR and other categories). Extensive experimental results comparing the performance of the best solvers for EPR problems with the performance of winning SMT solvers in the category of bit-vectors and arrays on hardware verification problems arising from real-life Intel micro-processor design are reported in Section V. The benchmarks were selected and organized carefully so to expose the strengths and weaknesses of different decision procedures, and their sensitivity to the nature of the benchmarks (such as the presence of extensive bit-level reasoning as opposed to really bit-vector level reasoning, the design style, the writing style of RTL, the nature of compilation of RTL and schematic descriptions into model-checking instances). Conclusions appear in Section VI.

II. The relational encoding

In this paper we consider the theory of fixed-size bit-vectors and extensional arrays. We assume that bit-vector arithmetic operators are synthesized (or bit-blasted) in the verification front-end, and the solver engines do not receive arithmetic operations in the expressions to solve.

For arrays, we assume the standard operations: read (or select), write (or store), and equality (if the array dimensions are the same), and the standard consistency and extensionality axioms [19], [7], [16]:

$$mem\{i \leftarrow e\}(i) = e;$$
$$mem\{i \leftarrow e\}(j) = mem(j), \quad \text{if } j \neq i;$$
$$(\forall j : mem_1(j) = mem_2(j)) \rightarrow mem_1 = mem_2.$$

The encodings of the theory of bit-vectors and arrays that we consider here are all refinements of an encoding proposed in [14], called the *relational encoding* (as opposed to the *algebraic encoding* that was also considered there and was shown not to scale on even small verification problems). To explain the relational encoding and to describe our contribution clearly, we choose to use as the running example slightly modified toy specification and implementation designs used as the running example in [14].

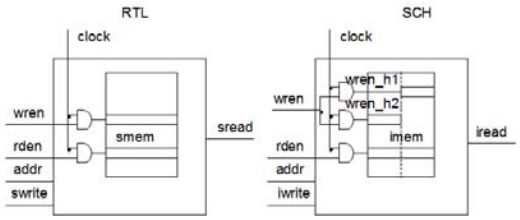


Fig. 1. Specification and Implementation memories

The toy designs are depicted in Figure 1. The specification design corresponds to the register-transfer level description (RTL), while the implementation design corresponds to its schematic implementation (SCH). The specification model contains a memory `smem` with 64 rows and 71 columns, its address bit-vector `addr` is of width 12, and it is used to pass the memory address for both write and read operations: bits `addr`$[5:0]$ are used for the write operation and bits `addr`$[11:6]$ are used for the read operation. Further, `swrite` and `sread` denote the write and read data vectors, respectively, of width 71, and `wren` and `rden` are the control bits enabling write and read operations, respectively. The bits `wren` and `rden`, as well as the clock `clock` and address `addr`, are shared between the specification and implementation designs. The implementation design has memory `imem` with the same dimensions as `smem`. [3] The write operation in the implementation model is different: bit-vector data `iwrite` is split into two parts during the write operation – `iwrite`$[70:36]$ and `iwrite`$[35:0]$. Each of the parts is written to the corresponding part of `imem`, so the implementation memory is shown split into two parts. Beyond the boundaries of the implementation memory unit the data is bitwise negated before being written to the implementation memory and after being read from it, so the write data `iwrite` (respectively, read data `iread`) in the implementation memory is the negation of the write (respectively, read) data in the specification memory.

With the relational encoding [14] of memory equivalence checking problems into EPR, any bit-vector b is considered as a relation on integers. Thus, for every integer k, it holds that $b(k)$ is true if and only if the kth bit of b is 1. If b does not have the kth bit, the relational encoding in [14] assumes that $b(k)$ is either true or false. Such a representation of bit-vectors is a powerful abstraction, since, instead of considering a bit-vector a mapping from a finite range of integers to booleans, as is done in the SMT theory QF_AUFBV, a bit-vector is now a mapping from *all* integers to booleans. The width of bit-vectors is thus abstracted away. In the relational encoding, a memory becomes a binary relation: the first argument denotes an address and the second a bit. For example, `imem`(a, k) denotes the value of the k-th bit of the element at the address

[3] Intel's logic extraction tool can identify memories and address decoders in the schematic models [14].

a in $\texttt{imem}$.

Let us now recall the relational encoding (as in [14]) of our running example. First, define the correspondence between the specification and the implementation designs as the conjunction of correspondence of the memories and of the read data.

$$\forall A \forall B(\texttt{imem}(A,B) \leftrightarrow \neg \texttt{smem}(A,B)). \tag{1}$$

$$\forall B(\texttt{iread}(B) \leftrightarrow \neg \texttt{sread}(B)). \tag{2}$$

The input write data correspondence is specified as follows:

$$\forall B(\texttt{iwrite}(B) \leftrightarrow \neg \texttt{swrite}(B)). \tag{3}$$

To be able to use the relational approach, one should identify bit-vectors in the design used as addresses and add equations enabling to decide when any pair of addresses is equal. For example, in our running example, we might need formulas describing when the term $\texttt{writeAddr}$ corresponding to $\texttt{addr}[5:0]$ is equal to the term $\texttt{readAddr}$ corresponding to $\texttt{addr}[11:6]$. Since the bit indexes involved in these two bit-vectors are different (in particular, are shifted), the corresponding bit-index constants $\texttt{bitInd}_0, \ldots, \texttt{bitInd}_{11}$ are introduced, and the following axiom is added:

$$\begin{array}{l}(\texttt{writeAddr} = \texttt{readAddr}) \leftrightarrow \\ \quad ((\texttt{addr}(\texttt{bitInd}_0) \leftrightarrow \texttt{addr}(\texttt{bitInd}_6)) \wedge \ldots \wedge \\ \quad (\texttt{addr}(\texttt{bitInd}_5) \leftrightarrow \texttt{addr}(\texttt{bitInd}_{11}))).\end{array} \tag{4}$$

The transition relation for the specification memory is as follows, where the prime symbol $'$ is used to denote next-state variables:

$$\begin{array}{l}\forall A(\texttt{clock} \wedge \texttt{wren} \wedge A = \texttt{writeAddr} \rightarrow \\ \quad \forall B(\texttt{smem}'(A,B) \leftrightarrow \texttt{swrite}(B))); \\ \forall A(\neg(\texttt{clock} \wedge \texttt{wren} \wedge A = \texttt{writeAddr}) \rightarrow \\ \quad \forall B(\texttt{smem}'(A,B) \leftrightarrow \texttt{smem}(A,B))).\end{array} \tag{5}$$

Splitting bit-vectors into parts is done by introducing predicates true on bits belonging to the LSB part. For the running example, predicate $\texttt{less}_{36}$ is introduced, intended to hold (only) on bits with numbers strictly less than 36. We also introduce propositional variables $\texttt{wren}_{h1}$ and $\texttt{wren}_{h2}$ for enabling writing into the two parts of the memory.

$$\begin{array}{l}\texttt{wren}_{h1} \leftrightarrow \texttt{wren} \wedge \texttt{clock}; \\ \texttt{wren}_{h2} \leftrightarrow \texttt{wren} \wedge \texttt{clock}.\end{array} \tag{6}$$

The transition relation for the implementation memory is then given as follows:

$$\begin{array}{l}\forall A(\texttt{wren}_{h1} \wedge A = \texttt{writeAddr} \rightarrow \\ \quad \forall B(\texttt{less}_{36}(B) \rightarrow (\texttt{imem}'(A,B) \leftrightarrow \texttt{iwrite}(B)))); \\ \forall A(\neg(\texttt{wren}_{h1} \wedge A = \texttt{writeAddr}) \rightarrow \\ \quad \forall B(\texttt{less}_{36}(B) \rightarrow (\texttt{imem}'(A,B) \leftrightarrow \texttt{imem}(A,B)))); \\ \forall A(\texttt{wren}_{h2} \wedge A = \texttt{writeAddr} \rightarrow \\ \quad \forall B(\neg\texttt{less}_{36}(B) \rightarrow (\texttt{imem}'(A,B) \leftrightarrow \texttt{iwrite}(B)))); \\ \forall A(\neg(\texttt{wren}_{h2} \wedge A = \texttt{writeAddr}) \rightarrow \\ \quad \forall B(\neg\texttt{less}_{36}(B) \rightarrow (\texttt{imem}'(A,B) \leftrightarrow \texttt{imem}(A,B)))).\end{array} \tag{7}$$

The definitions of the read operations for the specification memory are as follows.

$$\begin{array}{l}\texttt{clock} \wedge \texttt{rden} \rightarrow \forall B(\texttt{sread}'(B) \leftrightarrow \texttt{smem}(\texttt{readAddr}, B)); \\ \neg(\texttt{clock} \wedge \texttt{rden}) \rightarrow \forall B(\texttt{sread}'(B) \leftrightarrow \texttt{sread}(B)).\end{array}$$

The definitions of the read operations for the implementation memory are similar; one should only replace $\texttt{sread}$ and $\texttt{smem}$ by $\texttt{iread}$ and $\texttt{imem}$.

III. Relational encodings eliminating spurious models

Unfortunately, as pointed out in [14], the powerful abstraction resulting from considering bit-vectors as functions on *all* integers comes in the expense of loosing the completeness of the encoding – false negatives (i.e, counter-examples that are not real) are possible. For example, if a, b, c represent bit-vectors of length 1, the formula

$$a = b \vee a = c \vee b = c \tag{8}$$

is valid, but it is not valid in the abstraction since its negation is satisfiable.

In order to avoid the possibility of false negatives (i.e., spurious models), we would like the relational encoding to become aware of the ranges of bit-vectors and arrays involved in circuit operation, and we would like to record this information in the translation. The main idea of the refined encoding – let us call it *range-aware relational encoding* to EPR – is that for every formula that we generate during the encoding, the range of bits in the involved bit-vectors or arrays is explicitly encoded in the formula using the less-predicates and bit-index constants (such as $\texttt{less}_{36}$ or $\texttt{bitInd}_5$). For this to work, we need to relate the less predicates introduced during the encoding with the bit-index constants introduced during the encoding. Note that there is no need to relate a bit-index $\texttt{bitInd}_k$ with a less predicate $\texttt{less}_n$ for many pairs (k, n): it might be irrelevant to capture the fact that

$$\begin{array}{ll}\texttt{less}_n(\texttt{bitInd}_k) & \text{if } k < n; \\ \neg\texttt{less}_n(\texttt{bitInd}_k) & \text{otherwise.}\end{array} \tag{9}$$

Next we discuss several ways to eliminate false negatives, and discuss the advantages and disadvantages of each approach.

A. Encoding 1: precise ranges

Let us first define range-predicates: For a pair of non-negative integers $n \leq m$, let us define

$$\texttt{range}_{[m,n]}(B) \leftrightarrow \texttt{less}_{m+1}(B) \wedge \neg\texttt{less}_n(B).$$

When equality between arrays is introduced, it should be guaranteed that there will be no bits beyond the range of the data on which the array equality will fail. For example, we write the invariant formula (1) for memories as

$$\forall A \forall B(\texttt{range}_{[70,0]}(B) \rightarrow (\texttt{imem}(A,B) \leftrightarrow \neg\texttt{smem}(A,B))).$$

When equality between bit-vectors of the same range is introduced, we explicitly restrict the corresponding equivalence of bits to the relevant bit-range. For example, we now write the formula (7) as

$$\forall A(\mathtt{wren}_{h1} \wedge A = \mathtt{writeAddr} \rightarrow \forall B(\mathtt{range}_{[35,0]}(B) \rightarrow (\mathtt{imem}'(A,B) \leftrightarrow \mathtt{iwrite}(B))));$$
$$\forall A(\neg(\mathtt{wren}_{h1} \wedge A = \mathtt{writeAddr}) \rightarrow \forall B(\mathtt{range}_{[35,0]}(B) \rightarrow (\mathtt{imem}'(A,B) \leftrightarrow \mathtt{imem}(A,B))));$$
$$\forall A(\mathtt{wren}_{h2} \wedge A = \mathtt{writeAddr} \rightarrow \forall B(\mathtt{range}_{[70,36]}(B) \rightarrow (\mathtt{imem}'(A,B) \leftrightarrow \mathtt{iwrite}(B))));$$
$$\forall A(\neg(\mathtt{wren}_{h2} \wedge A = \mathtt{writeAddr}) \rightarrow \forall B(\mathtt{range}_{[70,36]}(B) \rightarrow (\mathtt{imem}'(A,B) \leftrightarrow \mathtt{imem}(A,B)))).$$

Similarly, instead of (2) we now write

$$\forall B(\mathtt{range}_{[70,0]}(B) \rightarrow (\mathtt{iread}(B) \leftrightarrow \neg\mathtt{sread}(B))).$$

We add axioms stating that bit-index terms corresponding to different indexes are not equal. For a less-predicate like $\mathtt{less}_{36}$, that has been introduced, we add the axiom

$$\forall B(\mathtt{less}_{36}(B) \leftrightarrow (B = \mathtt{bitInd}_0) \vee \ldots \vee (B = \mathtt{bitInd}_{35})). \quad (10)$$

How can the range-aware encoding solve the incompleteness of the relational encoding of [14]? With the relational encoding, the equation (8) is represented with the following formula, which is false already for 2-bit bit-vectors, say $a = 10, b = 00, c = 11$.

$$(\forall B(a(B) \leftrightarrow b(B))) \vee (\forall B(a(B) \leftrightarrow c(B))) \vee (\forall B(b(B) \leftrightarrow c(B))).$$

With the range-aware relational encoding, the same formula is represented by

$$(\forall B(\mathtt{range}_{[0,0]}(B) \rightarrow a(B) \leftrightarrow b(B))) \vee (\forall B(\mathtt{range}_{[0,0]}(B) \rightarrow a(B) \leftrightarrow c(B))) \vee (\forall B(\mathtt{range}_{[0,0]}(B) \rightarrow b(B) \leftrightarrow c(B))).$$

From the axiomatization (10) of the `less` and `range` predicates, we conclude that the above formula is equivalent to the one below, which is clearly true.

$$(a(\mathtt{bitInd}_0) \leftrightarrow b(\mathtt{bitInd}_0)) \vee (a(\mathtt{bitInd}_0) \leftrightarrow c(\mathtt{bitInd}_0)) \vee (b(\mathtt{bitInd}_0) \leftrightarrow c(\mathtt{bitInd}_0)).$$

Note that with the precise-ranges relational encoding the widths of bit-vectors and the dimensions of arrays are still abstracted away. This is different from the information specified to SMT solvers in the theory of fixed-size bit-vectors and extensional arrays. However, since our modeling of every bit-vector or array operations explicitly encodes the relevant ranges, the bit-vector width and array size information becomes redundant.

Theorem 1: The precise ranges encoding is sound and complete: an EPR formula obtained by the precise ranges encoding is satisfiable if and only if it is satisfiable over bit-vectors of the specified size.

B. Encoding 2: bit-index pre-instantiation

In the precise-ranges encoding, the axioms like (10) introduce many equalities between terms describing bit-indexes. Dealing with many such equalities may significantly slow down the EPR solvers. This is explained in the next section. The most straightforward way to avoid these equalities between the bit-index terms (and still retain the completeness) is to pre-instantiate all quantifiers ranging over bit-indexes with concrete index values. This is a meaningful alternative in the case where there is a lot of bit-wise reasoning, like in schematic models, and most of the bit-indexes introduced during pre-instantiation would have been introduced anyway with the precise-ranges approach.

This approach is sensitive to the amount of bit-index constants that will be introduced during pre-instantiation. In 64-bit based real-life micro-processor designs, there normally are no bit-vectors longer than around 71 bits (which consist of 64 bits of data and several other encryption bits and flags). However, because of the writing style of RTL and Schematic, and the way many RTL compilers work, often long vectors are created from nested structures. For example, in our toy example, if two different bit-vectors of width 6 were used for the write and read addresses instead of using a 12-bit vector `addr`, there will be no need to introduce bits $\mathtt{bitInd}_6, \ldots, \mathtt{bitInd}_{11}$ to the instance. Similarly, if the write and read data bit-vectors `swrite` and `sread` were defined as the LSB and MSB halves of a data bit-vector $\mathtt{sdata}[141:0]$, or as a structure with two fields $[70:0]$ `swrite` and $[70:0]$ `iwrite`, the functionality of the design would not change but the encoding with index pre-instantiation would force us to introduce extra bit-indexes $\mathtt{bitInd}_{71}, \ldots, \mathtt{bitInd}_{141}$. In our experiments below, we will see how introduction of bit-indexes caused by the RTL and SCH writing style and compilation of RTL and SCH into the model-checking instance can affect the solvers performance.

C. Encoding 3: Skolem predicates

We now introduce a smarter way to avoid introduction of equalities between bit-index terms as in the less predicate axioms like (10). Our approach can be seen as reasoning modulo a fixed domain of indexes and is inspired by approaches used in state-of-the-art finite model finders [1], [8].

First, note that the Skolemization of the invariant formulas (1) and (2) introduces Skolem constants. For example, let boolean variable `readeq` denote the truth value of the equality (2):

$$\mathtt{readeq} \leftrightarrow \forall B(\mathtt{range}_{[70,0]}(B) \rightarrow (\mathtt{iread}(B) \leftrightarrow \neg\mathtt{sread}(B))).$$

This formula is translated into a collection of following clauses, where $sk0$ is a fresh Skolem constant:

$$\begin{array}{l}\mathtt{readeq} \vee \neg\mathtt{iread}(sk0) \vee \neg\mathtt{sread}(sk0);\\ \mathtt{readeq} \vee \mathtt{iread}(sk0) \vee \mathtt{sread}(sk0);\\ \mathtt{readeq} \vee \neg\mathtt{less}_0(sk0);\\ \mathtt{readeq} \vee \mathtt{less}_{71}(sk0);\\ \mathtt{sread}(B) \vee \neg\mathtt{iread}(B) \vee \mathtt{less}_0(B) \vee \neg\mathtt{less}_{71}(B) \vee \mathtt{readeq};\\ \mathtt{iread}(B) \vee \neg\mathtt{sread}(B) \vee \mathtt{less}_0(B) \vee \neg\mathtt{less}_{71}(B) \vee \mathtt{readeq}.\end{array} \quad (11)$$

On the clauses containing occurrences of Skolem constants, we perform the following transformation: For each Skolem constant $\mathtt{sk}_i$ we introduce a new unary predicate $\mathtt{skP}_i$ and the following axiom, where $k, \ldots, m$ is the index range corresponding to $\mathtt{sk}_i$.

$$\mathtt{skP}_i(\mathtt{bitInd}_k) \vee \ldots \vee \mathtt{skP}_i(\mathtt{bitInd}_m). \tag{12}$$

Informally if $\mathtt{skP}_i(\mathtt{bitInd}_j)$ is true then we can assign $\mathtt{sk}_i$ to be equal $\mathtt{bitInd}_j$. Further, wherever $\mathtt{sk}_i$ occurs in a clause $C(\mathtt{sk}_i, X_1, \ldots X_n)$ then we replace this clause with $\neg\mathtt{skP}_i(Y) \vee C(Y, X_1, \ldots X_n)$. After this transformation, the first four formulas in (11) will have the following form:

$$\begin{array}{l}\mathtt{readeq} \vee \neg\mathtt{iread}(B) \vee \neg\mathtt{sread}(B) \vee \neg\mathtt{skP}_0(B);\\ \mathtt{readeq} \vee \mathtt{iread}(B) \vee \mathtt{sread}(B) \vee \neg\mathtt{skP}_0(B);\\ \mathtt{readeq} \vee \neg\mathtt{less}_0(B) \vee \neg\mathtt{skP}_0(B);\\ \mathtt{readeq} \vee \mathtt{less}_{71}(B) \vee \neg\mathtt{skP}_0(B).\end{array}$$

Then we can define predicates $\mathtt{less}_i$ for i as follows: e.g. for $\mathtt{less}_3$ we have unit clauses:

$$\begin{array}{l}\mathtt{less}_3(\mathtt{bitInd}_0), \mathtt{less}_3(\mathtt{bitInd}_1), \mathtt{less}_3(\mathtt{bitInd}_2),\\ \neg\mathtt{less}_3(\mathtt{bitInd}_3), \ldots, \neg\mathtt{less}_3(\mathtt{bitInd}_n).\end{array} \tag{13}$$

Note now we do not need axioms like (10) (introducing equalities between bit-index terms) any more.

Theorem 2: The Skolem predicates encoding is sound and complete: a formula obtained by the precise ranges encoding is satisfiable if and only if the corresponding formula obtained by the Skolem predicates encoding is satisfiable.

Proof: An adaptation of results from [1], [8]. ■

IV. Analysis of proof calculi for EPR

We compare general purpose first-order reasoners with dedicated SMT solvers on the benchmarks generated from industrial memory designs. Since our encodings are falling into the EPR fragment we focus on instantiation-based first-order reasoners which are especially efficient in this fragment, as witnessed by recent CASC competitions [4]. Instantiation-based methods are general purpose reasoning methods for first-order logic which are based on combining efficient propositional, or more generally ground, reasoning techniques with instantiation of first-order formulas. Instantiation-based methods are therefore well-suited for reasoning with fragments closely related to propositional logic such as the EPR fragment and in particular decide the EPR fragment. We consider two sate-of-the-art instantiation-based reasoners: the Darwin system [3], based on the *Model Evolution calculus* [2] and the iProver system [15], based on the *Inst-Gen calculus* [10].

The Model Evolution calculus can be seen as a lifting of efficient propositional DPLL calculus into first-order logic together with a number of DPLL-style techniques such as (dynamic) backtracking and lemma learning. The Model Evolution calculus is space efficient since only the candidate model is growing during the proof search (and optionally, the set of lemmas if lemma learning is applied). On the other hand, such lifting to the first-order logic requires to compute expensive context unifiers and considerably complicates dynamic backtracking and lemma learning, generally rendering them not as effective as in the propositional case.

The Inst-Gen calculus is based on a modular combination of propositional reasoning with refined instantiations of first-order formulas. One of the distinctive features of the Inst-Gen approach is that it allows one to employ off-the-shelf efficient propositional solvers (currently iProver integrates MiniSAT [9]) for reasoning with propositional abstractions of first-order clauses, guiding the instantiation inferences and simplification of clauses. We believe that such a modular integration of industrial-strength propositional solvers gives a considerable advantage when solving large real-life problems. Another important requirement from a solver used in a verification environment is to produce models for satisfiable problems. Such models correspond to bugs in the design and it is crucial to have a model representation amendable to efficient analysis. As a byproduct of this work, iProver has been extended with a representation of models such that the value of each bit in a bit-vector can be retrieved efficiently; this considerably simplified model analysis.

Our experimental results show that already non-tuned general purpose instantiation-based systems are close in performance and in some examples outperform highly optimized dedicated SMT solvers. These initial results are very encouraging and we believe that instantiation-based reasoners can be tuned further by exploiting the problem structure and by optimizing inference selection.

Let us now discuss different effects of our encodings on the EPR reasoners. First we note that the size of bit-vectors is directly related to the size of the search space. Therefore reducing the size of bit-vectors in the encodings is a promising research direction. Moreover, large ranges of bit-indexes produce clauses with large numbers of equational literals like (10). In general, instantiation-based methods are more tolerant to clauses with many literals than resolution-based methods since the number of literals in clauses does not increase during the instantiation process. Nevertheless equational axioms as (10) can produce numerous redundant inferences by substituting the variable B with different indexes during equational reasoning. All this instantiations are redundant and can be avoided as shown in Section III-C.

Let us compare our approach of encoding bit-vector and array reasoning into the EPR fragment with approaches used in SMT solvers. Reasoning in SMT solvers is done at the ground level and frequently results in full bit-blasting. Using first-order logic we can use higher levels of abstraction which can result in memory/bit-vector size independent reasoning. We believe this can lead to better scalability of our approach to large memories and bit-vectors. On the other hand, SMT solvers have advanced built-in bit-vector functions which are needed in many memory designs. Although it is possible to bit-blast such functions in our approach, a better approach

[4] http://www.cs.miami.edu/~tptp/CASC/

would be to devise encodings of these functions into the EPR fragment.

A further research direction is to strengthen our encodings by introducing higher level abstractions and by more sophisticated encoding of bit-vector reasoning. Such an encoding can also pave the way for using powerful resolution-based first-order reasoners such as Vampire [23], as the current encoding tends to produce very long clauses which are known to be hard for resolution-based reasoners.

V. Experimental evaluation

In this section, we evaluate the three above-discussed sound and complete encodings of problems with bit-vectors and arrays into EPR on two fastest EPR solvers, iProver [15] and Darwin [3]. We further compare their performance to that of the fastest SMT solvers for the theory QF_AUFBV – Boolector [5] and MathSAT [6]. We used a standard, and straightforward, encoding of RTL descriptions of hardware to the theory QF_AUFBV (for example, we haven't used the abstraction technique in [11] to reduce the number of involved bits during the encoding). With the incomplete encoding of [14], iProver returned spurious models on all problems which are UNSAT with the complete encodings; therefore we do not report here the EPR solver results with this encoding.

A. Description of benchmarks

In our experiments, we use five equivalence checking problem instances originating from a recent micro-processor design at Intel. Each problem instance corresponds to an equivalence checking problem between an RTL functional block (FUB) and the corresponding FUB in the schematic model.

The first group of experiments reported in Table 2 correspond to the original RTL and schematic FUBs. The schematic model contains lots of bit-level reasoning, and as a result the resulting EPR instances contain lots of bit-level equations (using the bit-indexes). On such instances, the abstraction techniques are less efficient, and the solvers that do not really employ the bit-vector level reasoning can perform almost as efficiently as on the problems with lots of reasoning at higher, bit-vector level reasoning.

Recall that in EPR encodings often there is a need to write axioms at bit level, say in equations like (4). As explained above, one expects that existence of a large amount of such index constants will negatively affect the performance of EPR solvers. To evaluate this point experimentally, for each equivalence checking benchmark we tried to produce an equivalent instance involving significantly fewer bit-indexes. This transformation was performed by manually editing the RTL and SCH descriptions and changing compilation switches when generating the model-checking instances (e.g., the next-state functions) from hardware descriptions. In brief, because of the way how the compiler works, linearization (or flattening) of (nested) structures or modules, causes creation of long bit-vectors containing the original bit-vector fields of structures and bit-vectors of modules as sub-vectors. This phenomenon is an artifact of compilation and does not change the meaning of the design. Undoing or preventing this linearization (even if it was not done as aggressively as possible), allowed us to significantly reduce the amount of long bit-vectors and the amount of bit-indexes involved in the generated EPR instances for FUBs 1 and 2. For the other FUBs the maximal width of bit-vectors remained unchanged. Benchmark results on these modified FUBs are reported in Table 3.

It is well understood that solvers might perform particularly well or badly on SAT vs UNSAT problems, and we aim to evaluate the selected solvers and encoding methods from this angle as well. To generate SAT instances, we manually introduced several common types of bugs into the designs or verification instances (such bugs include mismatches between the corresponding read or write enables, mixture in the order of data bits, incorrect or missing constraints (3) connecting the corresponding write data of the compared slices of specification and implementation designs, etc.). Tables 4 and 5 report runtime results on 5 FUBs obtained by these manipulations from the equivalence checking problems evaluated in Tables 2 and 3, respectively.

The formulas checked for SAT/UNSAT correspond to the induction step formulas [24] at depths smaller or equal to 3 – the depths needed to prove the induction invariant stating the equality of memories (1) and the read data (2). For the sake of performance efficiency, checking these formulas there split into two independent runs of the solvers; in one run, the initial value of the main clock was set to true, while in the second check it was set to false.

B. Performance results

One of the most important observations based on our experimental results is that already at this initial stage, non-tuned general purpose instantiation-based methods can solve industrial-size hardware verification problems within a reasonable time limit. Moreover, there are a number of problems where instantiation-based solvers outperform highly optimised SMT solvers, see Tables 2–5. In particular, instantiation-based methods perform well on the problems with long bit-vectors such as problems FUB 4 and FUB 5 (Tables 2–3), with maximal bit-vector sizes 994 and 1047 respectively. We believe this is one of the promising aspects of the instantiation-based approach which is achieved due to a higher level reasoning.

Let us note that SMT solvers and an instantiation-based solver iProver are all using SAT solvers as the back-end. In the case of Boolector it is PrecoSAT and in the case of MathSAT and iProver it is MiniSAT. Recently developed PrecoSAT is a highly optimized propositional solver which won the latest SAT competition. Thus, comparing MathSAT and iProver better highlights the differences between SMT and instantiation approaches since the same SAT solver is employed. We can see that iProver outperforms MathSAT on many problems both in SAT and UNSAT categories.

These experimental results indicate that instantiation-based methods and SMT technology complement each other and both are useful alternatives for industrial-size hardware verification. There are still a number of problems were SMT

solvers perform better than instantiation-based methods, especially on satisfiable problems. Therefore we are planning to explore applicability of recent advances in bit-vector reasoning developed in the SMT framework [5] into instantiation-based reasoning.

Let us compare our different encodings. Tables 2–5 indicate that there is no clear winner among our encodings. There is a trade-off between concise, higher-level encodings such as precise ranges and Skolem predicates encodings; and more explicit bit-index pre-instantiation encoding. These tables show that explicit encodings are better for unsatisfiable problems whereas concise encodings are better for satisfiable problems. The reason for this can be that in many cases low level reasoning is unavoidable for unsatisfiable problems whereas for satisfiable problems it is sufficient to consider concise representations.

Tables 2–5 show experiments with longer/shorter bit-vector encodings. The reduction of bit-vector sizes was not always successful, only in two first FUBs there was a noticeable reduction in the maximal bit-vector size: in FUB 1 from 286 to 185 and in FUB 2 from 640 to 203, in other three cases the instances have changed but the max bit-vector size was unchanged. We can see that in some cases shorter bit-vectors lead to performance improvement and therefore in our future work we will study how to reduce bit-vector sizes in our encodings.

Finally, we run Darwin on several groups of benchmarks (including the simplest FUBs 1-3) with time limit of 500 seconds, but unfortunately it could not solve any single problem. The reason can be the large number of clauses in the resulting problems which ranges from 30 thousands to over 100 thousands of clauses. We believe that a modular integration of propositional reasoning as it is done in iProver is advantageous on such problems. The problem of reducing the size of the encodings is also needed to be addressed.

VI. Conclusions and future work

The aim of this work was to explore the scalability and potential of several approaches to first-order logic theorem proving in solving industrial-sized verification problems involving reasoning with bit-vectors and arrays, and to compare them with SMT-based techniques. Taking into account that the EPR solvers are currently less optimized on industrial sized problems compared to more mature SMT solvers, the reported experiential results and theoretical analysis indicate that several first-order proof calculi do have a great potential in this domain. Furthermore, we believe that smarter encodings into EPR of the problems with bit-vectors and arrays can be developed by exploring abstraction and refinement techniques similar to those proposed for accelerating SMT solving and this can make the EPR-based approaches even more efficient.

Another big promise of using EPR solvers in model checking is that bounded model checking problems have a succinct encoding into EPR, such that the size of the BMC formulas is not affected by the unrolling bound [17]: unlike the SAT-based BMC [4], it is not needed to replicate copies of the temporal assertion and the transition relation for every unrolling depth. One of our major next goals in the EPR related model-checking research is to combine the ability of solving bit-vector and array reasoning instances in EPR at the word level with the EPR-based BMC proposed in [17] for bit-blasted model-checking instances. Furthermore, we believe that EPR solvers can be optimized on model checking instances resulted from this combined encodings.

This reported and future work is part of an ongoing research collaboration between Intel's formal technology group developing efficient model-checking and equivalence checking solutions for Intel's chip design project and between the University of Manchester. The developed word-level equivalence checking method will replace the more traditional sequential equivalence checking solution implemented in Intel's sequential equivalence checking tool, Seqver [12], [13], [14].

Acknowledgements. We would like to thank the developers of Boolector and MathSAT for their collaboration on this work.

References

[1] Baumgartner, P., A. Fuchs, H. de Nivelle, C. Tinelli. Computing finite models by reduction to function-free clause logic. J. of Applied Logic, 2007.
[2] Baumgartner P., C. Tinelli. The model evolution calculus as a first-order DPLL method. Artif. Intell. 172(4-5): 591-632, 2008.
[3] Baumgartner P., A. Fuchs, C. Tinelli. Implementing the Model Evolution Calculus. Inter. J. on Artificial Intelligence Tools 15(1): 21-52, 2006.
[4] Biere A., A. Cimatti, E. Clarke, Y. Zhu. Symbolic model checking without BDDs, TACAS 1999.
[5] Brummayer R., A. Biere. Boolector: An efficient SMT solver for bit-vectors and arrays, TACAS 2009.
[6] Bruttomesso R., A. Cimatti, A. Franzén, A. Griggio, R. Sebastiani. The MathSAT 4 SMT solver, CAV 2008.
[7] Bradley, A.R., Manna Z., Sipma H.B. What's decidable about arrays? VMCAI 2006.
[8] Claessen K., N. Sörensson. New techniques that improve MACE-style model finding. Workshop on Model Computation (MODEL), 2003.
[9] Eén N., N. Sörensson. An extensible SAT-solver. SAT 2003: 502-518.
[10] Ganzinger H., Korovin K. New directions in instantiation-based theorem proving, LICS 2003.
[11] Johannsen P. Reducing bitvector satisfiability problems to scale down design sizes for RTL property checking, HLDVT 2001.
[12] Kaiss, D., S. Goldenberg, Z. Hanna, Z. Khasidashvili. Seqver: a sequential equivalence verifier for hardware designs, ICCD 2006.
[13] Khasidashvili, Z., D. Kaiss, D. Bustan. A compositional theory for post-reboot observational equivalence checking of hardware, FMCAD 2009.
[14] Khasidashvili Z., Kinanah M., Voronkov A. Verifying equivalence of memories using a first order logic theorem prover. FMCAD 2009.
[15] Korovin, K. iProver–an instantiation-based theorem prover for first-order logic (system description), IJCAR 2008.
[16] Kroening D., Strichman O. *Decision Procedures*, Springer, 2008.
[17] Navarro Pérez J. A., A. Voronkov. Encodings of Bounded LTL Model Checking in Effectively Propositional Logic. CADE 2007.
[18] Navarro-Pérez J.A., A. Voronkov. Proof systems for effectively propositional logic, IJCAR 2008.
[19] McCarthy, J., J. Painter. Correctness of a compiler for arithmetic expressions. Symposium in Applied Mathematics, Vol. 19, Mathematical Aspects of Computer Science, American Mathematical Society, 1967.
[20] de Moura, L.M., N. Bjørner. Deciding effectively propositional logic using DPLL and substitution sets, IJCAR 2008.
[21] de Moura, L.M., N. Bjørner. Z3: An efficient SMT solver, TACAS 2008.
[22] Ge Y., de Moura L.M. Complete instantiation for quantified SMT formulas, CAV 2009.
[23] Riazanov, A., A. Voronkov. The design and implementation of Vampire, AI Communications, 15(2-3):91–110, 2002.
[24] Sheeran, M., S. Singh, G. Stalmarck. Checking safety properties using induction and a SAT-solver, FMCAD 2000.

Solver Test	Boolector clock=0/1	MathSAT clock=0/1	iProver pre-inst clock=0/1	iProver Skolemize clock=0/1	iProver precise clock=0/1
FUB 1	0.39 / 0.18	13.0 / 3.5	1.6 / 11.6	1.3 / 1.2	9.1 / 12.8
FUB 2	0.36 / 0.3	7.7 / 4.7	1.9 / 7.6	6.7 / 11.6	42 / 78.3
FUB 3	0.05 / 0.04	0.8 / 0.9	1.4 / 0.4	3.6 / 1.8	4.1 / 4.7
FUB 4	0.13 / 138.5	26 / t-o	4.8 / 51.1	44 / t-o	42 / t-o
FUB 5	5861.26 / 3.2	160.15 / 31.75	179.24 / 13.3	t-o / 680.94	1132.36 / 329.1
TOTAL	5862.19 / 142.04	207.65 / t-o	**188.94** / **84**	t-o / t-o	1229.56 / t-o

Fig. 2. Equivalence checking UNSAT problem instances with long bit-vectors.

Solver Test	Boolector clock=0/1	MathSAT clock=0/1	iProver pre-inst clock=0/1	iProver Skolemize clock=0/1	iProver precise clock=0/1
FUB 1	0.28 / 0.18	12.0 / 3.8	1.6 / 6.4	6.6 / 40.3	8.6 / 11
FUB 2	0.32 / 0.34	14.7 / 11.5	1.8 / 19.0	9.0 / 43.1	19.6 / 31.3
FUB 3	0.04 / 0.04	0.8 / 0.9	1.2 / 0.4	3.6 / 1.9	4.1 / 4.7
FUB 4	0.13 / 138.8	t-o / t-o	t-o / t-o	43.7 / t-o	42.2 / t-o
FUB 5	t-o / 2.98	158.94 / 31.71	149.88 / 11.08	t-o / 592.3	1084.7 / 320.33
TOTAL	t-o / **142.34**	t-o / t-o	t-o / t-o	t-o / t-o	**1159.2** / t-o

Fig. 3. Equivalence checking UNSAT problem instances with shorter bit-vectors.

Solver Test	Boolector clock=0/1	MathSAT clock=0/1	iProver pre-inst clock=0/1	iProver Skolemize clock=0/1	iProver precise clock=0/1
FUB 1	0.14 / 0.14	36.3 / 34.9	5.1 / 11.2	1.4 / 1.2	9.0 / 29.3
FUB 2	0.22 / 0.24	50 / 38.9	5.3 / 15.3	6.3 / 11.5	24.4 / 57.5
FUB 3	0.04 / 0.04	3.2 / 3.3	15.1 / 0.9	26.4 / 1.7	6.2 / 2.7
FUB 4	0.14 / 0.42	t-o / t-o	147.5 / t-o	39.7 / t-o	46.7 / 46.4
FUB 5	1.66 / 1.56	t-o / t-o	63.65 / 62.57	46.58 / 48.51	379.68 / 439.95
TOTAL	**2.2** / **2.4**	t-o / t-o	236.65 / t-o	120.38 / t-o	465.98 / 575.85

Fig. 4. Equivalence checking SAT problem instances with long bit-vectors.

Solver Test	Boolector clock=0/1	MathSAT clock=0/1	iProver pre-inst clock=0/1	iProver Skolemize clock=0/1	iProver precise clock=0/1
FUB 1	0.14 / 0.16	42.8 / 36.9	5.0 / 10.4	5.7 / 71.7	8.3 / 11.9
FUB 2	0.21 / 0.26	92.2 / 48.4	6.1 / 11.4	10.3 / 47.9	10.5 / 32.3
FUB 3	0.04 / 0.04	3.1 / 3.2	15.3 / 1.0	26.6 / 1.6	6.0 / 2.7
FUB 4	0.14 / 0.38	t-o / t-o	129.5 / t-o	44.0 / t-o	44.1 / 47.4
FUB 5	1.66 / 1.54	t-o / t-o	291.48 / 92.93	43.61 / 42.89	424.71 / 511.34
TOTAL	**2.19** / **2.38**	t-o / t-o	447.38 / t-o	130.21 / t-o	493.61 / 605.64

Fig. 5. Equivalence checking SAT problem instances with shorter bit-vectors.

Combinational Techniques for Sequential Equivalence Checking

Hamid Savoj[1] David Berthelot[1] Alan Mishchenko[2] Robert Brayton[2]

Envis Corporation[1] and Department of EECS, University of California, Berkeley[2]

{hamid, david}@envis.com and {alanmi, brayton}@eecs.berkeley.edu

Abstract

Often sequential logic synthesis can lead to substantially easier verification problems, compared to the general-case for sequential equivalence checking (SEC). We prove some general theorems about when SEC can be reduced to combinational equivalence checking (CEC). These can be applied to many sequential clock gating transforms, where correctness is argued intuitively using a finite unrolling of a sequential design. A method based on these theorems was applied to six large industrial examples. It completed on all examples and was about 30x faster on the three examples where the conventional engine was able to finish.

1 Introduction

To motivate this work, consider a sequential circuit, *A*, which is to be optimized by a *k*-step unrolling process; then combinational synthesis is applied to the first frame while the other *k*-1 frames are left untouched. This synthesis is done so that no difference between the two circuits is observed i.e. neither at the POs of each of the *k* frames nor at the flip-flop (FF) inputs of the final frame (see Figure 1). The last *k*-1 copies of *A* are used only to produce "ODCs" for transforming the combinational part of *A* into the combinational part of a new sequential circuit *B*.

Several questions arise in similar types of synthesis:

1. Is the derived circuit *B sequentially* equivalent to *A*? This is not obvious because it is the *k*-1 copies of *A* that provide the observability don't cares (ODCs), for *B*, and not *B* producing those ODCs as would be the case during the sequential operation of machine *B*. Although there is a known 1-1 correspondence between FFs in *A* and *B*, their state-transition functions are not necessarily the same.
2. Suppose *A* is unrolled *n* times and the *last* copy of *A* is synthesized using satisfiability don't cares (SDCs) provided by the first *n*-1 copies of *A*. Are *A* and *B* sequentially equivalent? As in Question 1, this is not obvious.
3. More generally, suppose *A* is unrolled *n* + *k* times and the n^{th} copy of *A* is synthesized, using both ODCs and SDCs, to produce machine *B*. Is *B* sequentially equivalent to *A*? This is not only not obvious, but generally incorrect.

In Section 2, we answer these questions, affirmatively for the first two with Theorems 1 and 2, and give a counterexample for the last one. The theorems are stated for general SEC and give sufficient conditions when it can be solved by a CEC method. Theorem 1 might be expected to apply when a synthesis transform can be argued from non-observability principles and Theorem 2 when non-controllability is used. In Section 3, we discuss relevant literature and related parallels to the results obtained in this paper, and in Section 4, we give some experimental results illustrating how these methods can make sequential equivalence checking (SEC) much more effective and practical on certain types of problems. Section 5 summarizes and poses some open questions for future research.

2 Sequential Equivalence

Let *A* be a sequential circuit and A^1 denote the combinational part of *A*. Let A^n denote the *combinational* circuit obtained by connecting *n* copies of A^1 at the FF inputs and outputs. The outputs of A^n are the set of *n* POs of *A* one for each time frame plus the final FF input signals after the n^{th} frame. The inputs of A^n are the set of *n* PIs of each frame, plus the initial FF output signals at the start of the first frame.

Let *A* and *B* be two sequential circuits with the same PIs and POs, and the same number of FFs. (B^n, A^k) denotes the combinational circuit where the outputs corresponding to the final FF inputs of B^n are connected to the inputs corresponding to the initial FF outputs of A^n. The connection is done using some 1-1 mapping between the FF of *A* and *B*. We overload notation by dropping the superscript in A^1 when the context is clear, as in (B, A^k).

In this paper, it is always assumed that a single initial state is given for a sequential machine. We are not concerned with initializing sequences etc., but follow the philosophy articulated in [1]. Thus two machines are considered sequentially equivalent if starting at their respective initial states they produce the same sequence of POs for any sequence of PIs. This is usually equivalence checked by forming a miter (which creates a single output formed by ORing XORs of corresponding POs) of the two circuits to obtain one machine with a single output. Then it is to be proved that the output is always 0 for all time if the miter machine is started in the initial state given by the initial states of the two machines.

For two sequential circuits, *A* = *B* denotes that the circuits are sequentially equivalent starting from the two given initial states. If *C* and *D* are combinational circuits, the *C* = *D* means that they are combinationally equivalent, i.e for any input, their outputs match.

The first question in Section 1 concerns equivalence of two related *combinational* circuits, i.e. does $A^k = (B, A^{k-1})$ imply *A* = *B*? This is depicted in Figure 1 where *k* = 3. We emphasize that to create the related combinational circuit (B, A^{k-1}) from A^1 and B^1, it is necessary that there is a 1-1 correspondence between the FFs of *A* and *B*. In some applications, this can be relaxed by inserting dummy FFs in one of the circuits.

Theorem 1: Suppose two sequential circuits *A* and *B* have the same PIs and POs. Using some 1-1 mapping between the FF of *A* and *B* to form (B^n, A^k), suppose that $(B^n, A^k) = A^{n+k}$. Then *A* = *B*, for *any* common initial state.

Note that *A* and *B* are initialized with the same initial state.

Proof:[1] Assume that $(B^n, A^k) = A^{n+k}$. Consider the following infinite sequence of lines.

$$\rightarrow\rightarrow \text{ time frame number } \rightarrow\rightarrow$$

$$\xrightarrow{S^*} A^n \xrightarrow{S_0(n)} A^n \xrightarrow{S_0(2n)} A^n \cdots$$

$$\xrightarrow{S^*} B^n \xrightarrow{S_1(n)} A^n \xrightarrow{S_1(2n)} A^n \xrightarrow{S_1(3n)} \cdots$$

$$\xrightarrow{S^*} B^n \xrightarrow{S_2(n)} B^n \xrightarrow{S_2(2n)} A^n \xrightarrow{S_2(3n)} \cdots$$

$$\vdots$$

$$\xrightarrow{S^*} B^n \xrightarrow{S_j(n)} B^n \xrightarrow{S_j(2n)} B^n \xrightarrow{S_j(3n)} B^n \cdots$$

It is assumed that all lines at each time-frame receive the same sequence of common PI inputs. Denote the POs of line j by $PO_j(t)$, where $t \in \{1,2,3,\cdots\}$ and $j \in \{0,1,2,3,\cdots\}$. Similarly for the states; $S_j(t)$ denotes the state of line j at time t, $t \in \{0,1,2,3,\cdots\}$ where $S_j(0) = S*$, the set of all states. Since $(B^n, A^k) = A^{n+k}$, then $PO_0(t) = PO_1(t)$ for $t \in \{1,2,3,\cdots,n+k\}$. Note that for all $t > n+k$, this is also true because $S_0(n+k) = S_1(n+k)$ and the circuit copies in both lines are A from then on. Now compare lines 1 and 2. Clearly $PO_1(t) = PO_2(t)$, $t = 1,...,n$ since in both lines, the inputs are to n copies of B, and by using the template, $(B^n, A^k) = A^{n+k}$, but applying it starting at the end of frame n, we have $PO_1(t) = PO_2(t)$ for all t, by the same argument that established that $PO_0(t) = PO_1(t)$ for all t. Thus by transitivity, $PO_0(t) = PO_2(t)$. Continuing, we get $PO_0(t) = PO_j(t)$ for all lines $j \in \{1,2,3,\cdots\}$. Thus line 0 and line ∞ always produce the same sequence of POs for all time no matter what is the initial state. Since the miter for $A \oplus B$ is proven to be UNSAT, we have $A = B$. **QED**.

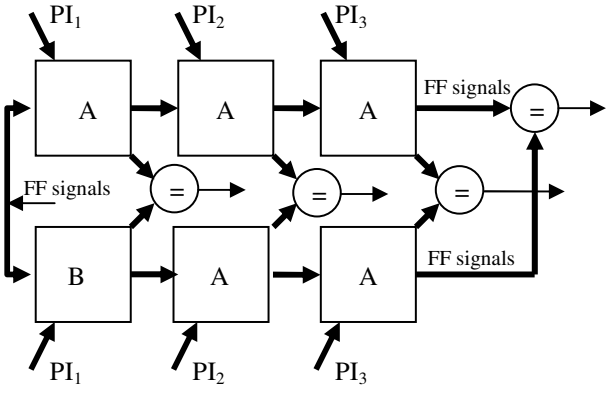


Figure 1. SEC by unrolling and CEC. POs are compared at each time frame as well as FF inputs *after* the last frame.

Note that nothing is assumed about how B derived. Also, if $(B^n, A^k) \neq A^{n+k}$, one can still try to prove $A = B$ by increasing k or n, and a false negative may go away.

[1] It has been suggested by several people (including one reviewer) that the theorems of this paper can be proved more elegantly by induction. However, we prefer the more graphical proofs (which are basically induction).

Note also that no initial state information was used in proving this theorem, i.e. $S_j(0) = S*$ is the set of all states. However, we could use a subset $\hat{S} \subset S*$ as long it is guaranteed that $S_0(n), S_1(2n), S_2(3n), \cdots$ are all **subsets** of $\hat{S}$. Thus a corollary of the theorem would be that $[(B^n, A^k) = A^{n+k}]_{\hat{S}} \Rightarrow A = B$ for any initial state $s \in \hat{S}$, where $[(B^n, A^k) = A^{n+k}]_{\hat{S}}$ denotes that combinational equivalence need only hold on state inputs in $\hat{S}$.

A variation of Theorem 1 states that SEC holds after n cycles of A.

Theorem 2: $[(A^n, B^k) = A^{n+k}] \Rightarrow A = B$ *on any initial state chosen from the subset of states that can be reached by A after n cycles, denoted* S_n^A.

Proof: We use the fact that the state space is finite, and therefore its diameter, D, is bounded. Thus after D time frames, every possible state has been seen under all possible inputs. The proof is similar to that of Theorem 1, except we proceed backwards from time-frame $T = D + (k\lceil D \div k \rceil)$. We first apply the template, $(A^n, B^k) = A^{n+k}$, to insert k B's just before $t = T$. This is iterated $\lceil D \div k \rceil$ times until we arrive at a line with n A's followed by all B's up to $t = T$. At each iteration j, we are assured that $PO_1(t) = PO_j(t)$ for all time. At this point we know that all states and all PIs for these states have been seen and for all of these the PO's agree. Thus starting at any state in S_n^A, $A = B$. **QED.**

To illustrate the need to start only on the states reachable by an initial sequence of A's, consider the example of Figure 2 (a bubble at an input to a gate denotes inversion).

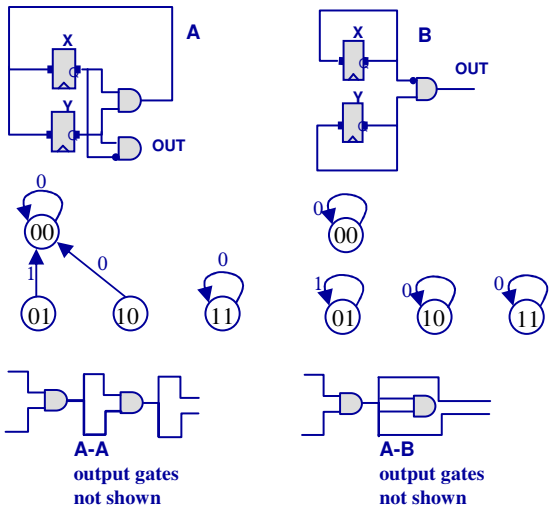


Figure 2. $A, B = A^2$, but sequential equivalence occurs only after one cycle of A.

It is easy to check that (A,A) = (A,B) from the STGs shown, but $A \neq B$. The counterexample is that if A and B start in State 01, the PO sequences for A and B are not the same. However, note that starting from any state that can be reached after one clock cycle of A (i.e. States 00 and 11), then $A = B$.

The first theorem is essentially an observability theorem and the second a controllability theorem. One might conjecture that analgous combined controllability and observability theorems hold. Indeed we have the following.

Theorem 3: $[(B,A,A)=(B,B,A)] \Rightarrow A=B$ *on any initial state chosen from the subset of states that can be reached by B after one cycle, denoted* S_1^B.

Proof: Consider the sequence of transformations shown below.

$$B,A,A,A,A,\cdots$$
$$B,B,A,A,A,\cdots$$
$$B,B,B,A,A,\cdots$$
$$B,B,B,B,A,\cdots$$
$$\cdots$$

Each new line is obtained by using $(B,A^2)=(B^2,A)$. At each line note that $PO_j(t)=PO_0(t),\forall t$. Thus after the first time frame, the set of states that can exist is S_1^B and after that, we have

$$A,A,A,A,A,\cdots=B,B,B,B,B,\cdots$$

Thus, $A=B$ on S_1^B. **QED**

Note that in Figure 2, $(B^2,A)\neq(B,A^2)$, which can be seen by starting at State (01), so it is not a counterexample to Theorem 3.

One could consider this as a *B*-controllable, *A*-observable theorem and the first two theorems as *A*-observable and *A*-controllable theorems respectively. What about an *A*-controllable, *A*-observable theorem, where we consider $(A,B,A)=(A,A,A)$?

Such a result does not hold. Consider the STG example shown in Figure 3, which has no inputs; the label on the edges denotes the output value. Although $(A,A,A)=(A,B,A)$, one can check that $A\neq B$, even on the states that *A* can reach after one cycle, e.g. starting at State 01 *A* 110… and *B* outputs 111….

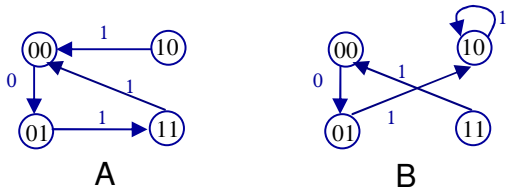


Figure 3. Although $(A,A,A)=(A,B,A)$ combinationally, *A* and *B* are not sequentially equivalent.

Although such a theorem does not hold, it still might be useful to synthesize (A,A,A) into (A,B,A) to derive a new sequential machine *B*. This is easier to do than obtaining a new machine *B*, for example, by synthesizing (A,A,A) into (B,B,A) or (A,B,B). These twi cases can guarantee equivalence using Theeorems 1 and 2 respectively. However, it is possible in the synthesis into (A,B,A), that the SDC or ODC don't cares actually used would be produced also by *B*. We can try to check $A^3=(B,A,A)$ or $A^3=(A,B,B)$ using Theorems 1 or 2, or $(B,A,A)=(B,B,A)$ or $(A,A,B)=(A,B,B)$ using Theorem 3. If any of these cases hold, then $A=B$. For the last three checks, it needs to be checked also that the initial state is in the appropriate subspace.

3 Relations to Previous Work

One of the pragmatic aspects of sequential synthesis is that it is insufficient to provide synthesis software, which may even use formally-proved[2] transforms because the software that embodies these may have bugs. Even if the software has withstood the test of time having been applied to many examples, most companies insist on formally verifying the result against the original design. Equivalence checking of combinational netlists (CEC) is practical for most industrial designs and, partly because of this, combinational synthesis is readily accepted. Also, resolution proofs [9] can be used for CEC.[3]

However, the PSPACE-complete complexity of SEC often discourages the use of sequential synthesis. In special cases, the complexity of SEC is simpler, e.g. if synthesis is restricted to one set of combinational transformations followed by one retiming (a sequential synthesis step) or vice versa, the problem is provably simpler - *only* NP-complete. If retiming and resynthesis are iterated, the problem is PSPACE complete [3]. Like CEC, SEC becomes simpler in practice if there are many structural or functional similarities (cut-points) between the two circuits being compared.

There are instances where SEC can be transformed into a CEC problem on which today's commercial CEC engines usually can be successful, even on very large problems. One is where *sequential signal* equivalences (signals that are equivalent on the reachable state set) are derived using induction [5] and used in the synthesis process. If equivalence checking is done immediately after this *without* other transformations intervening, SEC can be proved by CEC methods.

Another example is where a history of synthesis is recorded as a redundant sequential circuit [6]. In most cases, this history circuit provides a set of intermediate equivalences, which can be proved inductively, and these are enough prove SEC . Also, the concept of speculative reduction [7] can be used to make the equivalence checking problem even easier in this case.

Several papers have used an (explicitly or implicitly) unrolled version of the circuit to derive redundancies for synthesizing an improved sequential circuit. These papers do not address the formal SEC of the synthesized result. All deal with the case where the redundancies derived are independent of any initial state, similar to the theorems in the present paper. These types of results come mostly from the testing community, where a signal is redundant if the good and faulty (with a *stuck-at* fault inserted) machines can not be distinguished for any initial state.

There is a subtle distinction between *untestable* faults and *redundant* faults. If s_f and s are the initial states of the faulty and good machines respectively, then a fault is *untestable* if $\forall(I)\exists(s,s_f)[Z(I,s)=Z^f(I,s_f)]$ and it is *redundant* if $\forall(I,s_f)\exists(s)[Z(I,s)=Z^f(I,s_f)]$. $Z(I,s)$ is the trace, starting at state *s,* of POs under the sequence *I* of PI inputs. Using redundancy in synthesis means that when the good machine is replaced with the "faulty" (redundancy removed) machine, no difference can be observed externally because no matter what state s_f the faulty machine starts in, there is an equivalent state in which the good machine could have started in. Such a replacement is *safe*[4] [10] and *compositional*. In contrast, if the fault is merely untestable, then there could exist a *pair* of states in which the two machines could start, such that the difference between the two machines could not observed. However, there

[2] There are cases where "proved" methods in the literature have been shown to have counterexamples.

[3] We know of no similar capability for SEC.

[4] A safe replacement is one for which there is no possibility of externally detecting any difference from the original.

could be a state in either machine which has no equivalent in the other, and if one of the machines happened to start in such a state, the two machines would have different observable behaviors. Such a (untestable) replacement is not safe and is not compositional, and its use in synthesis is problematic. A good discussion on the difference between undetectable faults and redundant faults is in [3].

From Theorem 1, if $(B^n, A^k) = A^{n+k}$, then the synthesized circuit is a safe replacement for the original one. Safe replacements are useful because safety implies that every synchronization sequence for the original design also synchronizes the replacement. This is often desired because it is not necessary to re-derive a new synchronizing sequence for initializing the synthesized machine.

A useful notion is *c*-cycle redundancy [2] where the two circuits' outputs need not match for the first *c* cycles after power-up. This allows more flexibility in synthesizing a circuit because the behavior of the machine need only be preserved on states that can be reached after *c* cycles as long as initialization is preserved. Several papers make use of this and determine a bound *k* and a new circuit with the redundancies removed (called a *k*-delayed replacement) [4]. In [2] such redundancies are identified, one is then removed, and new ones identified. This is repeated until no more can be found. In [4], a set of "compatible" redundancies is found and removed simultaneously.

The method of [4] derives a constant *n* which is the difference between the time frame of an identified redundancy and the least time frame needed to infer this redundancy. Their theorem states that if the redundancy is used to create the new circuit, then it is an *n*-delayed replacement of the original. Note that in *n*-delayed replacement, it is *B* that is delayed for *n* cycles before equivalence can be guaranteed, but in Theorem 2 it is *A* that is delayed *n* cycles.

A sequential ATPG engine can be used to determine if a test vector sequence can be found which justifies a state that activates the fault in *n* cycles and then propagates the fault effect to a PO in *k* cycles. If none can be found, the fault might be redundant, but three things can go wrong; (i) undetectable faults are not necessarily redundant, (ii) the justification and propagation conditions are usually done on the good machine, and (iii) finite values for *n* and *k* were used. Such a fault is a good candidate for redundancy removal, but the result must be sequentially verified, possibly by applying Theorems 1-3, which may work if *A* or *B* are supplying a sufficient set of SDCs or ODCs. An interesting discussion of some incorrect "proofs" in the literature related to the use of ATPG for redundancy removal can be found in [3], as well as limitations of some other methods.

4 Experimental Results

A few experimental results are shown in Table 1. They were designed to compare the efficiency of applying the new SEC approach of this paper with the general SEC method of the ABC system. Six large industrial benchmarks were synthesized using sequential clock-gating transforms, based intuitively on sequential ODC arguments, but not formally proved. The synthesized versions are denoted by *B* and the originals by *A*. Columns 1-5 give the sizes of the circuits. The entries in column 6-11 give the times in minutes taken to verify equivalence. The columns labeled *New* denote the use of Theorem 1 and Berkeley's ABC system CEC algorithm to prove SEC. The column *ABC general* denotes that the ABC command *dsec* was used. Columns *seq-j* denote experiments where (B, A^j) was compared combinationally with A^{j+1} to illustrate how CEC run-times might scale as *j* increases. The items marked with * or **, denote that the corresponding equivalence checking problem timed out.

Observations.

1. In general, *New* is significantly faster than *General,* as expected (about 30 times faster when *General* could complete. The fact that *General* could actually complete on three out of six large problems was surprising to us).
2. Except for Design 4, CEC times scale approximately linearly with the size of the CEC problem.

5 Conclusions and Future Work

Some sequential synthesis transforms do not use the initial state information but preserve a circuit's behavior starting from any initial state. Such transforms may use sequential observability [2] [4] and can be practical because they do not use state space search or can be argued using structural information as in the case of many clock-gating methods. These contrast with transforms that extract ODCs using reachability analysis such as BDD reachability, interpolation or SAT-based induction [5].

In the sequential observability case, it may be possible that sequential equivalence can be proved by combinational equivalence checking methods, making SEC much easier. This can have a significant impact in applications where parts of the circuit are changed based on a local view of the circuit.

We have given a method for SEC, which can be effective in certain special cases, leading to considerable reduction in computation effort. The method is conservative; it fails no information is obtained. Some conditions under which it can be expected to succeed include sequential clock-gating methods and methods that alter pipeline behavior. Experimental results were given on a six large industrial SEC problems, comparing the sequentially synthesized design against the original design. It was demonstrated that the new SEC method was about 30 times faster than in the general case. In addition, it was able to check three examples where general SEC could not complete.

Our theorems are stated in terms of having a one-to-one correspondence between the FFs of *A* and *B*. This was necessary for combinational circuits (*A,B*) or (*B,A*) to be formed where signals in the first circuit are wired to their corresponding signals in the second circuit. However, some clock-gating transforms require that a signal be delayed one or more time-frames. In such cases, FFs must be introduced in *B* that have no correspondence in *A*. This can be handled by introducing dummy FFs in *A* with no fanout.

We conjecture, more generally, that it is sufficient to find two cuts of the same size, one in *A* and the other in *B*. The signals in the cuts can be a mixture of internal wires and FFs. It may be that the only requirement is that the cuts are feedback arc sets, i.e. cutting them makes each circuit acyclic. This would allow applications of the theorems to retimed circuits.

Also, it would be desirable to have a practical method to check general *k*-delayed equivalence, such as for designs produced by the methods of [2][4]. These situations are cases of local sequential synthesis being done. Note that if $S_B^k \subseteq S_A^k$, then Theorem 2 applies and can be used to prove *k*-delayed equivalence. It is possible that Theorem 3 can be used in such cases, although at the moment, we have no experimental results on this.

Theorem 1 legitimizes sequential synthesis based on unrolling a sequential machine *A*, *k* times, and combinationally synthesizing the first copy of *A* to obtain a new equivalent sequential machine

B. However, we have not done experiments on how effective this might be in terms of improved quality of the synthesis result.

Acknowledgements

This work was supported by SRC contract 1444.001, NSF grant CCF-0702668, and the (now-defunct) California Micro program.

References

[1] J. Baumgartner, H. Mony, V. Paruthi, R. Kanzelman and G. Janssen, "Scalable Sequential Equivalence Checking across Arbitrary Design Transformations." International Conference on Computer Design, San Jose, CA. October 2006.

[2] M. A. Iyer, D. E. Long, and M. Abramovici, "Identifying sequential redundancies without search," *DAC'96*, pp. 457-462.

[3] M. A. Iyer, D. E. Long, and M. Abramovici, ''Surprises in Sequential Redundancy Identification,'' *EDTC'96*.

[4] A. Mehrotra, S. Qadeer, V. Singhal, R. K. Brayton, A. Aziz, and A. L. Sangiovanni-Vincentelli. "Sequential optimisation without state space exploration". *ICCAD'97*, pp. 208-215.

[5] A. Mishchenko, M. L. Case, R. K. Brayton, and S. Jang, "Scalable and scalably-verifiable sequential synthesis", *ICCAD'08*. http://www.eecs.berkeley.edu/~alanmi/publications/2008/iccad08_seq.pdf

[6] A. Mishchenko and R. K. Brayton, "Recording synthesis history for sequential verification", *FMCAD'08*, pp. 27-34. http://www.eecs.berkeley.edu/~alanmi/publications/2008/fmcad08_haig.pdf

[7] H. Mony, J. Baumgartner, V. Paruthi, and R. Kanzelman, "Exploiting suspected redundancy without proving it". *DAC'05*.

[8] "Scalable Sequential Equivalence Checking across Arbitrary Design Transformations."

[9] S. Chatterjee, A. Mishchenko, R. Brayton, and A. Kuehlmann. "On Resolution proofs for combinational equivalence", *DAC'07*.

[10] V Singhal and C. Pixley. "The verification problem for safe replaceability," *CAV'94*, LNCS, Vol. 818, pp. 311-323.

Table 1. Experimental results.

Design	Statistics				Seq-1		Seq-2	Seq-3	Seq-4	Seq-5
	Ands	Flops	PI	PO	New	General	New	New	New	New
1	39282	6506	51	83	0.68	15.16	0.98	1.34	1.55	1.78
2	18932	10544	96	115	0.51	18.88	0.7	0.88	1.06	1.25
3	31103	7276	105	79	0.78	*60.55	1.29	1.51	1.69	1.63
4	81782	13822	394	703	1.61	*152.21	2.34	17.22	72.93	267.83
5	45241	11595	1741	301	0.94	25.63	1.26	1.63	2.37	**6.18
6	114824	15284	857	804	2.05	*112.83	3.26	4.09	4.79	6.22

Notes: * General sequence equivalence in ABC timed out. Although time-out was set to 1 hour, we were curious to see if the problem could complete if more time was given. Hence the irregular time-out times reported.
** Unresolved by ABC combinational equivalence checking
Entries in columns 6-11 denote run times in minutes.
Seq-*j* denotes the CEC problem where *j* copies of *A* are used, i.e. (B,A^j) is compared to A^{j+1}.

Automatic Verification of Estimate Functions with Polynomials of Bounded Functions

Jun Sawada

IBM Austin Research Laboratory
Austin, Texas 78758
Email: sawada@us.ibm.com

***Abstract*—The correctness of some arithmetic functions can be expressed in terms of the magnitude of errors. A reciprocal estimate function that returns an approximation of $1/x$ is such a function that is implemented in microprocessors. This paper describes an algorithm to prove that the error of an arithmetic function is less than its requirement. It divides the input domain into tiny segments, and for each segment we evaluate a requirement formula. The evaluation is carried out by converting an arithmetic function to what we call a polynomial of bounded functions, and then its upper bound is calculated and checked if it meets the requirement. The algorithm is implemented as a set of rewriting rules and computed-hints of the ACL2 theorem prover. It has been used to verify reciprocal estimate and reciprocal square root estimate instructions of one of the IBM POWER™ processors.**

I. Introduction

Formal verification has been used to verify floating-point arithmetic logic in the past. Especially, verifying primitive floating-point arithmetic operations, such as multiply or add operations, can be handled by automatic equivalence checking [1]. The results of floating-point addition or multiplication are well-defined in the IEEE 754 floating-point standard [2], and it is not hard to define their reference model. Running equivalence checking between a hardware implementation and its reference model may require a number of tricks [3], [4], such as using proper case-splitting and variable ordering for BDD [5] representations, but today's formal verification tools can handle it pretty well. Because the equivalence checking of these operations does not rely on the equivalence of the intermediate results, the reference model can be developed independently of hardware implementations, making it less likely to have the same defects in both. The reference model can be reused over and over for different projects, which makes the reference model even more trustworthy. Furthermore, the reference model itself can be formally checked by theorem proving technology, which can be done once and for all [6].

On the other hand, verifying micro-coded floating-point operations, such as divide and square-root, is not as easy. First of all, applying the same equivalence checking approach for primitive floating-point operations does not work well. Micro-coded operations are far more complex, and it is intractable to symbolically simulate an entire microcode sequence and perform equivalence checking. Industrial equivalence checking tools are very good at comparing two similar net-lists, by taking advantage of internal equivalent points. However, such internal equivalent points do not exist in general between the hardware execution of micro-code and its reference model.

One approach to solve this problem is writing a *high-level model* which mimics the hardware behavior and using it as a stepping stone for the verification. Since the high-level model is specifically built to have the same intermediate results as hardware, the equivalence checking becomes more tractable. One must also prove that the high-level model is correct. Since the high-level model is built to mimic the behavior of the hardware, both may contain the identical algorithmic defects.

The proof of a high-level model usually requires theorem proving or similar techniques. In the past, theorem provers have been used to verify divide and square root algorithms [7], [8], [9], [10]. However, the verification of a high-level model using mechanical theorem proving takes a lot of time and expertise, and it has not been used widely in the industrial setting. Some early work used mathematical analysis such as derivatives [11] or series approximations [12] to verify the algorithms, but the use of analysis further complicates the proof. It would be desirable if one can automatically verify a high-level model.

In this paper, we will consider reciprocal and reciprocal square root estimate instructions in order to study the automation of high-level model validation. Floating-point instructions `fre` and `frsqrte` in the POWER architecture [13] are examples of such instructions. Estimate instructions are somewhat similar to micro-coded instructions from the perspective of verification. An estimate instruction returns a number close to the actual reciprocal or the reciprocal of a square root, but not exact one. Their correctness is given as a relative error being less than a certain value. Therefore, there is no single reference model that could be used for the equivalence checking against any implementations. The verification needs to be carried out by first creating a high-level model, verifying its correctness, and then checking the equivalence against the hardware. The equivalence part is relatively easy because estimate instructions are much simpler than that of divide or square root. The high-level model verification is a key for successful verification.

We developed a new algorithm to verify the high-level models of estimate instructions. This algorithm runs automatically with no human guidance. We used the new algorithm to verify

the estimate instructions implemented in an industrial processor. In theory, this algorithm can be applied to other arithmetic operations whose correctness is expressed in terms of relative error size. For example, the correctness of divide and square root algorithms using the Newton-Raphson procedure can be represented by a relative error requirement.

In Section II, we describe our evaluation algorithm used to verify the high-level model of estimate instructions, and in Section III, we describe its ACL2 implementation. In Section IV, we apply the verification algorithm to instructions of an industrial processor. In Section V, we discuss future improvements of our algorithm and its implementation.

II. Verification Algorithm

A. Overall Verification Scheme

The overall verification framework is as shown in Fig. 1. The *design under test* (DUT) is typically a hardware implementation of an arithmetic function, and it is written in a hardware description language such as VHDL or Verilog. DUT may be implemented as microcode or firmware, essentially a piece of software working with the hardware.

In order to verify DUT, one has to provide a high-level model of the algorithm. It should precisely define arithmetic operations performed by DUT, but it may not capture implementation details such as what type of adder or multiplier implementations are used.

There are two paths to check the correctness of the arithmetic operation. The first path employs an algorithm evaluation process to check that a high-level model satisfies a desired mathematical property. Then, the second path uses an equivalence checker to compare the high-level model and the DUT. If both paths succeed, the operation of the DUT is guaranteed to meet the mathematical property. We assume in this paper that a verified mathematical property is written in an inequality like the maximum relative error requirement for an estimate instruction.

The second path operation of verification is the well-studied equivalence checking problem that is straightforward to skilled engineers. The high-level model must be translated into a bit-level net-list so that a bit-level equivalence checker can be used. We may need to perform symbolic simulation on the DUT to obtain the symbolic result of the arithmetic operation, which is then compared to the net-list representation of the high-level model. Equivalence checking tools are widely used in hardware verification in industry [14], [15].

If one is only interested in checking the algorithm, not hardware, we can only use the algorithm evaluation process alone, and skip the equivalence checking path altogether. In the rest of the paper, we will focus on the algorithm evaluation process to check the high-level model.

B. Formal Specification of Mathematical Operations

In a high-level model, an arithmetic operation is represented as a function of rational numbers. It should be specified in a polynomial of bounded functions (*PBF*) as defined below using a Backus-Nauer form:

$$\begin{aligned}\text{PBF} ::= {} & Constant \mid Variable \mid \text{PBF} + \text{PBF} \\ & \mid \text{PBF} \times \text{PBF} \mid \text{ERR\_FUNC}(\text{PBF}, \ldots, \text{PBF}) \\ & \mid \text{USR\_FUNC}(\text{PBF}, \ldots, \text{PBF})\end{aligned}$$

Here, the ERR_FUNC is a set of functions such that, for any function $e \in$ ERR_FUNC, there is a polynomial function $B(x_1, \ldots, x_n)$ and:

$$\forall i. |x_i| \leq \Delta_i => |e(x_1, \ldots, x_n)| \leq B(\Delta_1, \ldots, \Delta_n).$$

Since functions in ERR_FUNC are used to represent the errors of primitive arithmetic operations, we call them *error functions*. We restrict the bounding function $B(x)$ to be a polynomial function in order to make it easy to compute the upper bound of an error function when its argument domains are also bounded. The error function will be treated as an uninterpreted function during the evaluation process. USR_FUNC is the set of user-defined functions. One can define a new function f to be $f(x_1, \ldots, x_n) = g$, where g is a PBF with variables $x_1, \ldots, x_n$.

PBF is used to model mathematical operations that can be approximated with a polynomial, and we found that many arithmetic operations used in hardware designs can be nicely represented as a PBF.

For example, the nearest mode rounding defined in the IEEE 754 standard rounds a value to the closest representable floating-point number. For 53-bit double-precision numbers,

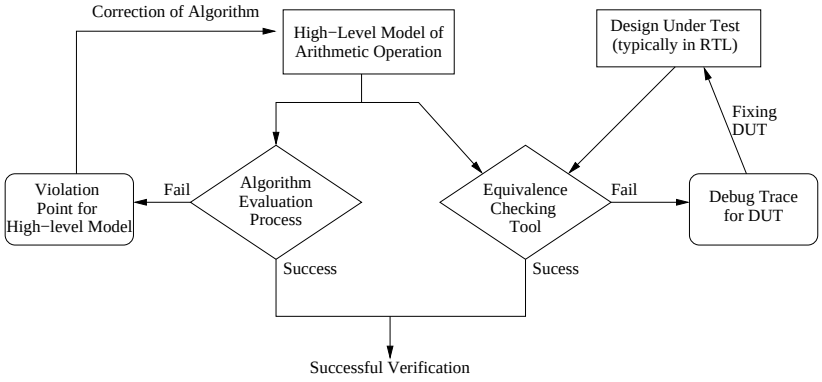


Fig. 1. Overall Scheme of the Verification

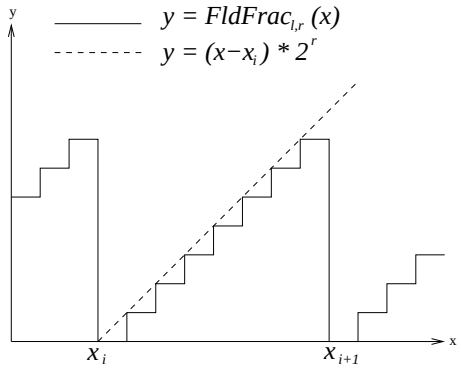


Fig. 2. Fraction field value and its approximation.

the nearest mode rounding can be defined as a user-defined function:

$$\text{near}_{\text{dp}}(x) = x + E_{\text{near},53}(x)$$

with an associated bounding condition $|E_{\text{near},53}(x)| \leq 2^{-53} \times \Delta$ for any $|x| \leq \Delta$.

Next, the double-precision floating-point add operation with the nearest mode rounding can be defined as:

$$\text{add}(x, y) = \text{near}_{\text{dp}}(x + y),$$

and similarly the multiply operation is:

$$\text{mult}(x, y) = \text{near}_{\text{dp}}(x \times y).$$

Another example is an operation to extract a bit-field from the fraction of a floating-point number. A floating-point number x is typically represented with sign bit *sgn*, exponent *expo* and fraction *frac*, where $x = (-1)^{sgn} \times 2^{expo} \times frac$. We assume x is normalized, which means that *frac* satisfies $1 \leq frac < 2$. Thus the binary representation of *frac* looks like $1.b_1b_2b_3\cdots$, where b_i is 1 or 0. Let us consider a function $FracFld_{l,r}(x)$ that returns binary integer $b_lb_{l+1}\cdots b_{r-1}b_r$. In Fig. 2, the $FracFld_{l,r}$ function is represented as a solid line.

We can approximate the $FracFld_{l,r}$ with $(x - x_i) \times 2^r$ represented as a dashed line in a domain $[x_i, x_{i+1})$ such that $x_i = 2^{-l} \times i$, and $1 \leq x_i < 2$. Thus $FracFld_{l,r}$ can be represented as:

$$FracFld_{l,r}(x) = (x - x_i) \times 2^r + E_{FracFld_{l,r}}(x), \qquad (1)$$

with an error function $E_{FracFld_{l,r}}(x)$ satisfying:

$$E_{FracFld_{l,r}}(x) \leq 1.$$

Note that $FracFld_{l,r}(x)$ is a user-defined function of PBF because 2^r is a constant, and $x - x_i$ is a shorthand of $x + (-1) \times x_i$.

Similarly, we can consider truncating lower bits of an integer as yet another example. Let us suppose x is an m-bit binary integer and $\text{truncate}_n(x)$ removes the lower n-bit of x and returns $(m - n)$ bit integer. This truncation can be represented as:

$$truncate_n(x) = x \times 2^{-n} + E_{truncate_n}(x)$$

Simplify$(P(x), [x_i, x_j))$

1) Substitute $x_i + \delta$ for x in $P(x)$. Variable δ satisfies $|\delta| \leq x_j - x_i$.
2) Replace user defined functions with the corresponding PBF,
3) Expand and simplify to normalize the the polynomial.
4) Move the constant term to the right of $\leq$ and non-constant terms to the left. Return the resulting inequation $Q_i(\delta) \leq C_i$.

Fig. 3. PBF Simplification Algorithm

where $E_{truncate_n}(x)$ satisfies

$$E_{truncate_n}(x) \leq 1.$$

We can represent a number of operations used in the implementation of numerical operations as functions of PBF. Sometimes restricting the domain of input variables is a key. For example, floating-point divide and square root operations using the Newton-Raphson algorithm is usually implemented by combining an initial table look-up, floating-point multiply-and-add operations, and a final rounding operation. If we narrow the input range so that the table look-up value is a constant, the entire algorithm except the final rounding is represented by PBF. Then the correctness of the algorithm can be given by a formula bounding the error of the final approximation before rounding.

Some arithmetic operations are hard to be represented as a PBF. For example, representing the SRT division algorithm as a PBF is difficult. The SRT division algorithm guesses the next quotient digit by table look-up using some bits of an intermediate remainder as an index. Depending the guess, the next intermediate remainder can be completely different, making it hard to approximate using a polynomial.

C. Algorithm to Verify a Property of Formal Specification

In this subsection, we consider an algorithm to verify a PBF formula of the form:

$$\text{Formula} ::= \text{PBF} \leq \text{PBF}.$$

This type of formula can be used to represent relative error requirements such as the correctness statement of an estimate function. Let us consider verifying formula $P(x)$, with the assumption that the input variable x satisfies $x_0 \leq x < x_n$, and the error functions appearing in $P(x)$ may take any values as long as they satisfy the associated bounding condition. We assume x is the only free variable in $P(x)$, but we can easily extend the algorithm to a multiple-variable formula.

The evaluation algorithm is carried out by splitting segment $[x_0, x_n)$ into non-overlapping segments, $[x_0, x_1), [x_1, x_2), \ldots, [x_{n-1}, x_n)$, and simplifying and evaluating $P(x)$ in each sub-segment. The algorithm *Simplify*$(P(x), [x_i, x_j))$ in Fig. 3 is used to simplify the original formula $P(x)$ in segment $[x_i, x_j)$.

The first step of the simplification algorithm produces $P(x_i + \delta)$, a formula of variable δ where $|\delta| < x_j - x_i$. Let us define $\Delta_{ij} = x_j - x_i$. After expanding user-defined functions, Step 3 normalizes the polynomial by applying the associativity, commutativity and distributivity laws of $\times$ and

$\mathrm{U}(c, \Delta) = |c|$ where c is a constant.
$\mathrm{U}(\delta, \Delta) = \Delta$ where δ is a variable.
$\mathrm{U}(Q_1 + Q_2, \Delta) = \mathrm{U}(Q_1, \Delta) + \mathrm{U}(Q_2, \Delta)$.
$\mathrm{U}(Q_1 \times Q_2, \Delta) = \mathrm{U}(Q_1, \Delta) \times \mathrm{U}(Q_2, \Delta)$.
$\mathrm{U}(e(Q_1), \Delta) = P(\mathrm{U}(Q_1, \Delta))$
where $|x| \leq b \Rightarrow |e(x)| \leq P(b)$ holds for error function e.

Fig. 4. Rules to calculate Upper Bound $\mathrm{U}(Q(\delta), \Delta)$

```
Evaluate(P(x),[x_0, x_n)) {
   S := {[x_0, x_1), [x_1, x_2), ..., [x_{n-1}, x_n)}
   While ( S ≠ ∅ ) {
      Pick a segment [x_i, x_j) ∈ S, and S := S/[x_i, x_j).
      (Q_i ≤ C_i) := Simplify(P(x), [x_i, x_j)).
      If C_i < 0, then return fail with x_i as a failure point.
      If U(Q_i, x_j − x_i) > C_i, then S := S ∪ {[x_i, x_k), [x_k, x_j)}
      for some new k.
   }
   return success
}
```

Fig. 5. Algorithm to verify $P(x)$ over $[x_0, x_n]$

$+$, and combining monomials of the same kind. Finally, constants are moved to the right of $\leq$ symbol, and non-constant monomials are moved to the left. The simplification algorithm *Simplify*$(P(x), [x_i, x_j))$ preserves the semantics in the sense that the original formula $P(x)$ holds if the final formula $Q_i(\delta) \leq C_i$ does.

In the final product of simplification, $Q_i(\delta) \leq C_i$, the left-hand side is a PBF without any constant terms or duplicate monomials. Note that $Q_i(0) = 0$ if we assume all error functions appearing in Q_i take zero values. Thus the proof of $P(x)$ fails if $C_i < 0$. For $Q_i(\delta) \leq C_i$ to hold regardless of the values of the error functions, C_i must be a non-negative number.

If C_i is indeed a non-negative number, we compute an upper bound $\mathrm{U}(Q_i, \Delta_{ij})$ of $Q_i(\delta)$ for $|\delta| < \Delta_{ij}$, and compare it against C_i. Fig. 4 shows the recursive rules to compute $\mathrm{U}(Q_i, \Delta_{ij})$. If $\mathrm{U}(Q_i, \Delta_{ij}) \leq C_i$, $Q_i(\delta) \leq C_i$ holds for all δ such that $|\delta| < \Delta_{ij}$.

The entire algorithm to verify $P(x)$ over the domain $[x_0, x_n)$ is given in Fig. 5. Evaluate($P(x)$,$[x_0, x_n)$) starts by splitting the entire domain into n segments $[x_i, x_j)$. We require that $P(x)$ is a PBF in each segment. For each segment, it applies the simplification algorithm in Fig. 3 to $P(x)$, obtaining $Q_i \leq C_i$. If $C_i < 0$, it fails and returns x_i as a failure point. Otherwise it computes the upper bound of Q_i. If the upper bound is less than or equal to C_i, $P(x)$ holds for the segment and we continue. Otherwise, it splits the segment $[x_i, x_j)$ into $[x_i, x_k)$ and $[x_k, x_j)$, adds them to S, and repeats the analysis on the refined segments. The choice of x_k is arbitrary, but we think $x_k = (x_i + x_j)/2$ would work in most cases.

If the algorithm returns with $S = \emptyset$, then all segments are verified, thus $P(x)$ holds for $[x_0, x_n)$. There is no general guarantee that the algorithm terminates because the bound of the error functions may not be strong enough to either compete the proof or refute it. In general, we should define error functions appearing in $P(x)$ so that error functions take relatively small magnitude compared to the main computation. If it is done properly, the smaller the segment gets, the more likely the evaluation succeeds or reports a failure point.

D. An Example of Verifying a Reciprocal Estimate Function

In this Section, we discuss a simple example of reciprocal estimate functions that illustrates the use of our algorithm. Instructions `fre` in the POWER processor returns a reciprocal estimate of a given number. Precisely speaking, for a given double-precision floating point number x, it returns a double-precision number $\mathrm{fre}(x)$ that is an approximation of $1/x$. The architectural definition of the POWER processor requires the following formula to be met:

$$\left| \frac{1/x - \mathrm{fre}(x)}{1/x} \right| \leq 1/256. \quad (2)$$

Similarly the `frsqrte` instruction returns a reciprocal square root estimate $\mathrm{frsqrte}(x)$, an approximation of $1/\sqrt{x}$. It must satisfy the following relative error requirement.

$$\left| \frac{1/\sqrt{x} - \mathrm{frsqrte}(x)}{1/\sqrt{x}} \right| \leq 1/32.$$

Depending on the processor implementation, it may be required that the upper bound of the relative errors should be 2^{-14}, not 1/256 or 1/32.

The result of the estimate instructions are considered correct as long as it satisfies the formula above, and there is no single correct answer. The idea is that these numbers are later used for software divide and square root routines based on an iterative algorithm. Such an algorithm is self-correcting and is not sensitive to minor differences of the initial estimate values.

Let us consider a reciprocal estimate function $\mathrm{fre}(x)$ that is implemented by a piecewise linear approximation. For the input $1 \leq x < 2$, $\mathrm{fre}(x)$ first computes an index i by taking the most significant several bits of the fraction of x, looks up tables to obtain a_i and b_i, and returns $a_i x + b_i$ as the answer. Once fre(x) is defined for $1 \leq x < 2$, then we can extend it to the entire non-zero domain by using equations $\mathrm{fre}(x \times 2) = \mathrm{fre}(x)/2$ and $\mathrm{fre}(-x) = -\mathrm{fre}(x)$. Furthermore, these equations can be used to extend the proof of Equation 2 for $x \in [1, 2)$ to the the entire domain of x. Thus, we focus on the domain $x \in [1, 2)$ in the following discussion.

Now let us consider a piecewise linear approximation that uses eight linear functions for segment [1,2):

$$F(x) = \begin{cases} \frac{967}{512} - \frac{455}{512} \times x & \text{for } x \in \left[1, \frac{9}{8}\right) \\ \frac{1729}{1024} - \frac{91}{128} \times x & \text{for } x \in \left[\frac{9}{8}, \frac{10}{8}\right) \\ \vdots & \\ \frac{8463}{8192} - \frac{273}{1024} \times x & \text{for } x \in \left[\frac{15}{8}, 2\right) \end{cases}$$

It is easy to prove by analysis that the relative error of $F(x)$ is less than 1/256. However, $F(x)$ is not what can be implemented as hardware, because hardware can implement only finite precision operations. A realistic implementation

of fre(x) should take the finite bits of x and compute the approximation of the linear function. Following definition fre(x) is one such example:

$$\text{fre}(x)=\begin{cases} \frac{967}{512}-\frac{455}{512}\times\left(1+\frac{\textit{FracFld}_{4,10}(x)}{2^{10}}\right) & \text{for } x\in\left[1,\frac{9}{8}\right) \\ \frac{1729}{1024}-\frac{91}{128}\times\left(\frac{9}{8}+\frac{\textit{FracFld}_{4,10}(x)}{2^{10}}\right) & \text{for } x\in\left[\frac{9}{8},\frac{10}{8}\right) \\ \vdots & \\ \frac{8463}{8192}-\frac{273}{1024}\times\left(\frac{15}{8}+\frac{\textit{FracFld}_{4,10}(x)}{2^{10}}\right) & \text{for } x\in\left[\frac{15}{8},2\right) \end{cases}$$

When x is in $[1, 9/8)$, the binary representation of x looks like $1.000b_4b_5\cdots$. Function $\textit{FracFld}_{4,10}(x)$ returns $b_4b_5\cdots b_{10}$ as a 7-bit integer. This fre(x) function can be implemented using a tiny multiply-adder. Mathematical analysis cannot be used for fre(x) because fre(x) is not a continuous function with a derivative. We illustrate how our algorithm can prove the relative error requirement of fre(x).

First, we convert the requirement formula $|(\text{fre}(x) - 1/x)/1/x| < 1/256$ to its equivalent formula $255/256 \leq F(x)\times x \leq 257/256$, as our algorithm can take only a formula of PBF. Since $F(x)$ is almost always larger than $1/x$, we will focus on the second $\leq$ comparison for the rest of the arguments.

Our algorithm first divides the target domain of x into sub-segments. Let us assume that the domain of x is divided into the segment of size 1/128, and we will explain how the algorithm works for the sub-segment [1,129/128).

The Simplify algorithm in Fig. 3 first substitutes $1+\delta$ for x in the original formula:

$$\text{fre}(x)\times x \leq 257/256 \tag{3}$$

resulting in:

$$\text{fre}(1+\delta)\times(1+\delta)\leq 257/256$$

Second, it replaces fre and $\textit{FracFld}_{4,10}$ using the definition of fre and Equation 1.

$$(-\frac{455}{512}\times(1+\delta+E_{\textit{FracFld}_{4,10}}(\delta)\times 2^{-10})+\frac{967}{512})\times(1+\delta)\leq\frac{257}{256}$$

Now we expand, combine same monomials, and move constants to the right of $\leq$ and non-constant monomials to the left, putting into the format of $Q_i \leq C_i$

$$\frac{57}{512}\delta-\frac{455}{2^{19}}E_{\textit{FracFld}_{4,10}}(\delta)-\frac{455}{512}\delta^2-\frac{455}{2^{19}}\delta E_{\textit{FracFld}_{4,10}}(\delta)\leq\frac{1}{256}$$

Since the right-hand side is a positive number, we compute the upper bound of the left-hand side, by the rules in Fig. 4.

$$\begin{aligned}&\text{U}\left(\frac{57}{512}\delta-\frac{455}{2^{19}}E_{\textit{FracFld}_{4,10}}(\delta)-\frac{455}{512}\delta^2-\frac{455}{2^{19}}\delta E_{\textit{FracFld}_{4,10}}(\delta)\right)\\ =&\frac{57}{512}\times\frac{1}{128}+\frac{455}{2^{19}}\times 1+\frac{455}{512}\times\frac{1}{128^2}+\frac{455}{2^{19}}\times\frac{1}{128}\times 1\\ =&\frac{120703}{2^{26}}<\frac{1}{256}\end{aligned}$$

This shows that the upper bound of the left-hand side is less than 1/256, finishing the proof the original equation 3 for [1,129/128).

Similarly we can apply the same simplification and upper-bound calculation to other sub-segments. It turns out that, for the segment [33/32,133/128), the upper bound is larger than the right-hand side constant. However, in this case, dividing the segment into 4 sub-segments of size 1/512 and repeating the process will successfully prove the inequation.

The evaluation attempt also fails for the sub-segment [133/128,67/64). However, this time, further refining the sub-segment to smaller segments does not work. The proof attempt of a refined segment [4283/4096,1071/1024) will produce a C_i less than 0. In fact, the relative error of fre$(4283/4096)$ is larger than 1/256. In order to correct it, fre(x) must use 9 bits instead of 7 bits of the fraction of x for the linear function calculation. With this fix, our algorithm can successfully verify that fre(x)'s relative error is less than 1/256 over the entire domain.

III. ACL2 Implementation of Algorithm

We implemented our verification algorithm on the ACL2 theorem prover [16]. The ACL2 theorem prover is a widely-used open-source theorem prover, that has been used for hardware and software verification in both academia and industry [17]. This section assumes some knowledge of the ACL2 theorem prover. Readers who are not interested in implementation details may skip to the next section.

There are a number of advantages for us to implement the algorithm on the ACL2 theorem prover. First, the results of our verification algorithm of the high-level design can be augmented with other mechanical proofs. For example, the verification of fre(x) in the previous section in segment [1,2) can be extended to all non-zero domain of x with interactive theorem proving. Second, we can use the theorem prover's rewriting engine to manipulate polynomials, instead of writing a polynomial simplifier from scratch. Third, the ACL2 theorem prover provides an interface named clause-processor [18] to call external formal verification tools such as a bit-level equivalence checker. We can use this feature to run an equivalence checker between the high-level model of an algorithm and its hardware implementation.

One disadvantage of the implementation using ACL2 is speed. Especially, our implementation of the verification algorithm is not optimized for speed. It is possible to implement our algorithm in a typical programming language, and call it from the ACL2 theorem prover using the clause-processor mechanism. However, the clause-processor mechanism does not guarantee the soundness of the newly integrated system, because the called program may contain flaws.

We used an approach to implement the algorithm using computed-hints [19] and a set of rewriting rules. A computed-hint is a user-defined functions that is used to steer the direction of a proof, but the proof itself is carried out by the pure ACL2 proof engine. It is somewhat like "strategy" [20] in the PVS theorem prover [21], or "tactics" in the HOL theorem prover [22], although ACL2 computed-hints may not be able to specify the proof step-by-step. Unlike clause processors, computed-hints do not introduce unsoundness to the ACL2

```
(defthm fre-error-1-2
    (implies (and (rationalp b) (<= 1 b) (< b 2))
             (and (<= 255/256 (* b (fre b)))
                  (<= (* b (fre b)) 257/256)))
  :hints
  ((case-split-segment 'b 1 2 (expt 2 -7))
   (when-pattern
     (if (not (< B (@ low))) (< B (@ high)) 'nil)
     :use (:instance xd-decomp (x b) (x0 (@ low)))
     :in-theory (enable fre))
   (bound-poly-prover id clause world)))
```

Fig. 6. An ACL2 command to verify fre(x).

prover. If a computed-hint is not implemented correctly, the proof using the hint may fail, but it never proves an untrue statement as a theorem.

In our implementation, a single ACL2 command shown in Fig. 6 proves the error requirement for fre(x) from the previous section. A typical ACL2 command `(defthm` *name expr* `:hints` *hints*`)` attempts to prove *expr* with the guidance provided by *hints*, and, if successful, stores the theorem with the *name*. In Fig. 6, three computational hints are provided after `:hints`. Briefly speaking, the first computed-hint starting with `case-split-segment` splits the domain of x into tiny segments. The second computed-hint named `when-pattern` applies the simplification algorithm in Fig. 3, and the last computed-hint `bound-poly-prover` builds a proof based on the upper bound computed as in Fig. 4.

The first computed-hints using `case-split-segment` splits the domain [1, 2) of `b` into small sub-segments of size 2^{-7}. This generates 128 independent sub-goals. The rest of the computed hints are applied to each sub-goal.

The second computed-hint `(when-pattern` *pattern* `.` *hints*`)` attempts to pattern match sub-terms of the target to *pattern*, and if there is a match, *hints* are applied to the ACL2 proof. This computed-hint is an extension described in [23]. Expressions starting with an `@` sign is a pattern variable, and can be matched to any expression. Then the pattern variables in *hints* are substituted accordingly.

In our example in Fig. 6, computed-hint `when-pattern` looks for an `if`-expression that matches the pattern. The previous case-splitting hint must have produced a term `(and (<=` x_i `x) (< x` x_j`))`, and its internal representation is an `if`-expression that looks exactly like the pattern. As a result, `(@ low)` and `(@ high)` are matched to x_i and x_j respectively.

Then the :use hint instantiates a theorem named `xd-decomp` given as follows:

```
(defthmd xd-decomp
  (implies (and (rationalp x)
                (rationalp x0))
           (and (rationalp (xd x x0))
                (equal x  (+ x0 (xd x x0))))))
```

where `xd` is a function defined as:

```
(defund xd (x x0) (rfix (- x x0)))
```

The definition of `xd` is disabled and the ACL2 theorem prover treats it as an uninterpreted function. This effectively has the same effect as replacing x with $x_i+\delta$, with term `(xd x x0)` serving the role of variable δ.

The `when-pattern` hint also calls an `:in-theory` hint, which opens up the definition of `fre`. By setting up proper rewriting rules, the ACL2 term rewriter applies the same simplification as the simplification algorithm in Fig. 3. For this purpose, we need to disable all the built-in rewrite rules of ACL2, and enable specific rules that mimic the simplification algorithm, using a script that is not shown here.

There are two sets of rewriting rules needed for the task. First is to normalize polynomials. This includes typical rewrite rules for the commutativity, associativity, and distributivity of $+$ and $\times$, unicity, and rules to combine coefficients. For example, we need a rewriting rule `(+ (* c0 x) (* c1 x))` $\rightarrow$ `(* (+ c0 c1) x))` that is applied only if `c0` and `c1` are constants. In addition to that, we need a few tricky rewriting rules which combine monomial of the same kind located far apart in a polynomial. Such rewriting rules can be written using the `syntaxp` heuristic filter of ACL2.

The other set of rules are used to bring constants to the right of $\leq$, and non-constant terms to the left. All rules have to be carefully coded and ordered so that it works well with the ACL2's rewriting algorithm.

The final computed-hint `bound-poly-prover` in Fig. 6 computes U(Q) of the left-hand side of $\leq$ after the previous step produces $Q \leq C$. If the U(Q) is less than the right-hand side constant C, then it constructs a proof of $Q \leq C$. This is done by instantiating theorems stating $|a \times b| = |a||b|$, $|a+b| \leq |a|+|b|$, and the inequality bounding error functions, for each step of the bounding proof.

For example, suppose $E_f(x)$ is an error function that satisfies $|E_f(x)| \leq 1 + \Delta$ for $|x| \leq \Delta$, and we want to prove $|\delta| \leq 1 => \delta + \delta E_f(\delta) \leq 3$. The proof can be given by the following three inequations:

$$|\delta| \leq 1 => |E_f(\delta)| \leq 2$$

$$|\delta| \leq 1 \wedge |E_f(\delta)| \leq 2 => |\delta E_f(\delta)| \leq 2$$

$$|\delta| \leq 1 \wedge |\delta E_f(\delta)| \leq 2 => |\delta + \delta E_f(\delta)| \leq 3$$

When `bound-poly-prover` is called, it adds these three instantiated theorems to the target goal as assumptions. In essence, bound-poly-prover elaborately provides the proof steps to ACL2, so that ACL2 only needs to perform propositional logic inference to finish.

We did not implement dynamic adjustment of the segment size. Currently we manually adjust the segment size as an argument to the `case-split-segment` computed-hint.

IV. Applications

We applied the ACL2 implementation of our verification algorithm to a couple of industrial examples. The first example is a reciprocal estimate instruction of one of the POWER processors. This particular algorithm uses a piecewise linear approximation with segment size of 2^{-6} for the input between 1 and 2. It uses 22 bits of the fraction of an input to compute a reciprocal estimate. The relative error must be

less than 2^{-14}. It is implemented using a look-up table, a small multiplier, adders, bit field extractions, and shifters. It is more complicated than our toy example in Subsection II-D, but the principle remains the same and it can be modeled as a PBF function. One notable point is that the algorithm uses one's complement to negate intermediate results, not 2's complement. Since this is just an estimate instruction, the difference in the last 1 bit should not matter, but our high-level description models precisely such behavior.

Our algorithm successfully verifies this high-level model of the reciprocal estimate function in the segment [1,2), dividing it to 4048 sub-segments of size 2^{-12}. If the segment size gets bigger than 2^{-11}, proof failed for some segments. The verification took 3652 seconds using a 2.93GHz processor with the ACL2 version 3.6 running on Clozure Common Lisp. We minimized printing by ACL2 intermediate proof goals, as it would have printed out huge expressions and consumed a sizable amount of CPU time. No interactive human inputs are required during this proof.

After proving that the estimate function meets its requirement in [1,2), an interactive theorem proving extends the domain from [1,2) to the entire non-zero values, by proving the theorem:

```
(defthm fre-correct
    (implies (and (rationalp b)
                  (not (equal b 0)))
     (<= (abs (/ (- (fre b) (/ 1 b)) (/ 1 b)))
                 1/16384)))
```

In this theorem, `fre` represents the verified reciprocal estimate instruction. Also we used our *ACL2SIX* framework [24] and IBM verification tool *SixthSense* [25] to check that the algorithm matches the hardware implementation. Briefly speaking, ACL2SIX translates an ACL2 expression to VHDL, and runs SixthSense to verify whether the property holds for DUT. If the verification fails, a waveform is produced to assist debugging. If successful, the ACL2 theorem prover continues the proof with the fact that the expression is true.

We also applied the same algorithm to the reciprocal square root estimate instruction of the same processor. This particular implementation should have less than 2^{-14} relative errors, meaning reciprocal estimate $s(x)$ should satisfy:

$$\left|\frac{s(x) - 1/\sqrt{x}}{1/\sqrt{x}}\right| \leq 2^{-14}.$$

This is equivalent to proving $(1 - 2^{-14})^2 \leq s(x)^2 \times x \leq (1 + 2^{-14})^2$.

The verification required the input domain [1,4) divided into 3072 segments of size 2^{-10}. The algorithm successfully verified this algorithm, taking 13953 seconds to complete. Although the number of sub-segments are smaller than that of the reciprocal estimate instruction, evaluation of each segment generates far more complex polynomials, thus taking more time to finish.

In the verification of both estimate instructions, the segment has been split into a fixed size at the beginning of the evaluation, and we did not dynamically sub-divide segments. Such improvements will certainly speed up the verification algorithm.

V. Discussion

Our verification algorithm has been successfully applied to verify two estimate instructions in an industrial processor. Unlike many theorem-proving based methods, our algorithm runs automatic and can be applied to different examples with minimum human effort.

The current implementation of the verification algorithm is given as a set of ACL2 computed hints, and it has not been optimized for performance. Especially the segment size is fixed at the beginning of the verification, and it does not automatically adjust the size. Our investigation shows that adjusting the sub-segment size is likely to improve the verification speed significantly. For example, the frsqrte verification used the segments of 2^{-10} for the entire domain of inputs, but segment size of 2^{-8} is sufficient to verify most of the segments except a few.

More improvements can be implemented for the `bound-poly-prover` computed hint. The current implementation adds a large number of irrelevant and duplicated inequalities to the target, by instantiating lemmas for each proof step. A smarter implementation can, for example, avoid repeating the same proof steps for common sub-expressions.

We relied on the ACL2 theorem prover for most of the heavy work. Majority of the time is spent on the simplification of polynomials and propositional reasoning of the bounding proof, both of which are performed by ACL2. It is possible to implement a standalone program to accelerate our verification algorithm.

It is interesting to compare our approach to MetiTarski [26], which applied series's of upper and lower bounds of polynomials given by Daumas et. al. [27] to a resolution theorem prover, and proved many inequalities with analytical functions such as trigonometric functions and logarithm. It does not split input domains to smaller segments explicitly like our algorithm. We think their system is more suitable for verifying inequalities used for the control system of, for example, avionics, than hardware implementation of arithmetic circuits. One reason is that it may not handle non-continuous functions such as rounding and bit-extractions. However, we can combined their techniques to ours for the verification of trigonometric functions.

One may ask whether our automatic verification algorithm scales to other problems. Next natural targets of our algorithm are divide and square root algorithms using an iterative algorithm such as Newton-Raphson procedure. Their correctness can be stated that the error of the final approximation before rounding is less than a quarter or a half of the *unit of the last position* (ULP), especially if the algorithm uses a special hardware rounding mechanism. The ULP of 1 is 2^{-23} for single precision and 2^{-52} for double precision operations. Since our reciprocal estimate verification required 2^{10} or 2^{12} segment-wise analysis to prove the 2^{-14} relative error requirement,

one may wonder that our procedure never scales for double-precision algorithms. Certainly, if we need to analyze, for example, 2^{50} individual segments, the verification algorithm never finishes in a reasonable amount of time.

Fortunately, our preliminary analysis shows that we do not need that many segments to be evaluated. In this analysis, we plugged in our reciprocal square root estimate function in Section IV to double-precision Goldschmidt's algorithm[28], and saw what kind of polynomial $Q(\delta) \leq C$ is generated from $((x_0+\delta) - s(x_0+\delta)^2) \leq 2^{-53}$ at $x_0 = 1$, where $s(x)$ is the square root approximation before the final rounding. We found that $Q(\delta)$ is a polynomial with very small coefficients for low-degree monomials. For example, the coefficient of δ in Q is close to 2^{-48} and that of δ^2 is close to 2^{-40}, while C is about 2^{-53}. This suggests that we need to evaluate at least 2^7 segments, but a similar number of segmentation used for the estimate instructions is likely to be sufficient to complete the square root verification.

The real challenge in verifying the square root algorithm is the size of the polynomial. If we naively expand all the terms to obtain simplified inequation $Q \leq C$, then Q will be a polynomial with millions or more monomials. This is because the algorithm uses about 10 instructions, and each instruction inserts an error function, which is essentially a new variable of the polynomial. As a result, the size of Q grows exponentially as the number of instruction increases.

In order to mitigate the growth of polynomial Q, we must implement a better polynomial simplification approach. One approach is using sub-expression sharing using unique pointers. Another approach is improving our polynomial simplification and evaluation process, so that we do not require full expansion of the polynomial. If none of these approaches work, we can still use theorem proving to finish the proof while using our verification algorithm to check intermediate results. In any case, full or semi-automatic validation of the high-level model is critical to enhance the use of formal techniques in the industrial setting.

REFERENCES

[1] Y.-A. Chen and R. E. Bryant, "Verification of floating-point adders," in *CAV '98: Proceedings of the 10th International Conference on Computer Aided Verification*. London, UK: Springer-Verlag, 1998, pp. 488–499.

[2] *IEEE Standard for Binary Floating-Point Arithmetic*, ANSI/IEEE Std. 754-1985, 1985.

[3] C. Jacobi, K. Weber, V. Paruthi, and J. Baumgartner, "Automatic formal verification of fused-multiply-add FPUs," in *DATE '05: Proceedings of the conference on Design, Automation and Test in Europe*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 1298–1303.

[4] A. Slobodová, *Challenges for Formal Verification in Industrial Setting*, ser. Lecture Notes in Computer Science. Springer, 2007, vol. 4346, pp. 1–22.

[5] R. E. Bryant, "Graph-based algorithms for boolean function manipulation," *IEEE Trans. Comput.*, vol. 35, no. 8, pp. 677–691, 1986.

[6] J. O'Leary, X. Zhao, R. Gerth, and C.-J. H. Seger, "Formally verifying IEEE compliance of floating-point hardware," *Intel Technology Journal*, vol. Q1, Feb. 1999.

[7] J. S. Moore, T. W. Lynch, and M. Kaufmann, "A mechanically checked proof of the AMD5K86™ floating-point division program," *IEEE Trans. Comput.*, vol. 47, no. 9, pp. 913–926, 1998.

[8] D. Russinoff, "A mechanically checked proof of IEEE compliance of a register-transfer-level specification of the AMD-K7 floating-point multiplication, division, and square root instructions," *London Mathematical Society Journal of Computation and Mathematics*, vol. 1, pp. 148–200, December 1998, http://www.onr.com/user/russ/david/k7-div-sqrt.html.

[9] J. Harrison, "Formal verification of IA-64 division algorithms," in *TPHOLs '00: Proceedings of the 13th International Conference on Theorem Proving in Higher Order Logics*. London, UK: Springer-Verlag, 2000, pp. 233–251.

[10] ——, "Formal verification of square root algorithms," *Form. Methods Syst. Des.*, vol. 22, no. 2, pp. 143–153, 2003.

[11] ——, "Verifying the accuracy of polynomial approximations in HOL," in *TPHOLs '97: Proceedings of the 10th International Conference on Theorem Proving in Higher Order Logics*. London, UK: Springer-Verlag, 1997, pp. 137–152.

[12] J. Sawada and R. Gamboa, "Mechanical verification of a square root algorithm using taylor's theorem," in *FMCAD '02: Proceedings of the 4th International Conference on Formal Methods in Computer-Aided Design*. London, UK: Springer-Verlag, 2002, pp. 274–291.

[13] *POWER ISA™ Version 2.06*, International Business Machines Corporation, 2009, See URL http://www.power.org/resources/downloads.

[14] J. Baumgartner, H. Mony, V. Paruthi, R. Kanzelman, and G. Janssen, "Scalable sequential equivalence checking across arbitrary design transformations," in *Proc. ICCD '06*, 2006.

[15] J. Baumgartner, H. Mony, M. L. Case, J. Sawada, and K. Yorav, "Scalable conditional equivalence checking: An automated invariant-generation based approach," in *FMCAD*, 2009, pp. 120–127.

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

[17] M. Kaufmann, J. S. Moore, and P. Manolios, Eds., *Computer-Aided reasoning: ACL2 case studies*. Norwell, MA, USA: Kluwer Academic Publishers, 2000.

[18] M. Kaufmann, J. S. Moore, S. Ray, and E. Reeber, "Integrating external deduction tools with ACL2," *Journal of Applied Logic*, vol. 7, no. 1, pp. 3–25, Mar. 2009.

[19] M. Kaufmann and J. S. Moore, "ACL2 home page," See URL http://www.cs.utexas.edu/users/moore/acl2.

[20] S. Owre and N. Shankar, "Writing PVS proof strategies," in *Design and Application of Strategies/Tactics in Higher Order Logics (STRATA 2003), number CP-2003-212448 in NASA Conference Publication*, 2003, pp. 1–15.

[21] S. Owre, J. M. Rushby, and N. Shankar, "PVS: A prototype verification system," in *CADE-11: Proceedings of the 11th International Conference on Automated Deduction*. London, UK: Springer-Verlag, 1992, pp. 748–752.

[22] M. J. C. Gordon and T. F. Melham, Eds., *Introduction to HOL: a theorem proving environment for higher order logic*. New York, NY, USA: Cambridge University Press, 1993.

[23] J. Sawada, "ACL2 computed hints: Extension and practice," in *ACL2 Workshop 2000 Proceedings, Part A*. The University of Texas at Austin, Department of Computer Sciences, Technical Report TR-00-29, Nov. 2000.

[24] J. Sawada and E. Reeber, "ACL2SIX: A hint used to integrate a theorem prover and an automated verification tool," in *FMCAD '06: Proceedings of the Formal Methods in Computer Aided Design*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 161–170.

[25] H. Mony, J. Baumgartner, V. Paruthi, R. Kanzelman, and A. Kuehlmann, "Scalable automated verification via expert-system guided transformations," in *FMCAD*, 2004, pp. 159–173.

[26] B. Akbarpour and L. C. Paulson, "MetiTarski: An automatic theorem prover for real-valued special functions," *J. Autom. Reason.*, vol. 44, no. 3, pp. 175–205, 2010.

[27] M. Daumas, D. Lester, and C. Muoz, "Verified real number calculations: A library for interval arithmetic," *Computers, IEEE Transactions on*, vol. 58, no. 2, pp. 226 –237, feb. 2009.

[28] P. Markstein, "Software division and square root using Goldschmidt's algorithms," in *In 6th Conference on Real Numbers and Computers*, 2004, pp. 146–157.

A Framework for Incremental Modelling and Verification of On-Chip Protocols

Peter Böhm
Oxford University Computing Laboratory
Oxford, OX1 3QD, England
peter.boehm@comlab.ox.ac.uk

***Abstract*—Arguing formally about the correctness of on-chip communication protocols is an acknowledged verification challenge. We present a generic framework that tackles this problem using an incremental approach that interleaves model construction and verification.**

Our protocol models are based on abstract state machines formalized in Isabelle/HOL. We provide abstract building blocks and generic composition rules to support incremental addition of protocol features to a parameterized endpoint model. This structured approach controls model complexity. We can refine data structures and develop control independently, to create a concrete instantiation.

To make the verification effort feasible, we combine interactive theorem proving with symbolic model checking using NuSMV. The theorem prover is used to reason about generic correctness properties of the abstract models given some local assumptions. We can use model checking to discharge these assumptions for a specific instantiation. We show the utility and breadth of the framework by sketching two case studies: modelling a bus protocol, and modelling the PCI Express point-to-point protocol.

I. Introduction

Formal verification of high-performance on-chip communication protocols is widely acknowledged to be hard. Modern multi- or many-core architectures require highly complex protocols to handle the performance bottleneck due to communication. These protocols implement sophisticated features to provide the needed performance.

Traditionally, monolithic models are created and proven correct using post-hoc verification. Given the complexity of the features and the size of the distributed system, this approach became often infeasible in time and effort.

We propose a new methodology based on incremental modelling and step-wise verification to tackle this challenge. The idea is to encapsulate the complexity of the features into independent modelling steps and add these features incrementally to the model, starting with a very simple model. At the same time, we reduce the verification effort with two main strategies: first the verification process can be spread over the modelling process such that in each step we only need to verify the parts added to the model that implement the new feature. Second we use generic building blocks for which we have shown correctness results. The verification can be restricted to discharging local assumptions on the building blocks.

In previous contributions [1], [2], we have illustrated the approach on two case studies. To explore the utility and breadth of the approach, we chose two rather different examples: first the ARM AMBA Advanced High-performance Bus (AHB) protocol [3], an arbiter-based master-slave bus protocol. As a second protocol, we picked the PCI Express protocol [4], a modern point-to-point protocol. We will briefly summarize the application of the general framework to these protocols in Section VI.

The development of the framework was driven by these case studies. After completing the work on both protocols, we were able to create a protocol independent formalization of a mature framework which we present here. All the models have been formalized in higher order logic using the Isabelle theorem prover [5].

For the case studies, we realized the modelling and verification idea using interactive theorem proving only. However, our final aim was to reduce the theorem proving part to further increase the feasibility of the approach. Here, we show how to integrate automatic tools into the verification workflow. On the one hand, we use automatic theorem provers for subgoals in first-order logic using the sledgehammer interface of Isabelle/HOL (Isabelle 2009-1). Sledgehammer invokes the provers E, SPASS, and Vampire and, if successful, returns a proof script for the Metis theorem prover.

On the other hand, we integrate the NuSMV symbolic model checker in the workflow. To use the model checker, we adapt the oracle-based IHaVeIt interface [6] to Isabelle 2009. We can invoke NuSMV to prove LTL and CTL formulas from Isabelle. This is especially useful when we instantiate generic composition operators and need to discharge local assumptions, such as fairness constraints. We will detail the integration of automated tools in Section V.

Our main contributions can be summarized as follows:

- A framework based on (abstract) state machines for modelling and verification of on-chip protocols.
- Generic building blocks and composition rules to create models incrementally covering basic components such as buffers to specialised composition schemes.
- A verification methodology that handles the complexity by restricting the effort to local constraints and global, generic correctness results.
- Integration of the NuSMV model checker in the verification process to further reduce the verification effort with focus on automatically discharging the local constraints.

These contributions result in a promising prototype system that has been successfully applied to a variety of protocol features.

A. Related Work

Modelling systems with automata in general is a well studied field and can be found, for example, in the widely-cited book by Robert Kurshan [7]. Formalizing state machines in Isabelle/HOL goes back to at least Nipkow and Slind [8]. They formalized I/O automata and developed a meta-theory to represent them as objects in the logic. We restrict our state machine framework to a simpler formalization specialised to our requirements. Formal verification of protocols using I/O automata in theorem provers has a long history, e.g. [9], [10]. Our goal is not to provide yet another specific protocol verification using I/O automata and a theorem prover, but the formalization of a methodology.

Suhaib *et al.* [11] propose an incremental methodology for developing formal models called XFM. An extendable set of LTL properties is used to incrementally create a model that satisfies the set of properties. Their approach focuses on building prescriptive formal models that capture the behaviour of natural language specifications. Our methodology tries to capture specific features in independent models.

Another related approach is the B Method [12], an event-based method for a refinement-based specification, design and implementation of software components. Abrial *et al.* [13] apply the method to the incremental development of the IEEE 1394 tree identify protocol. Cansell *et al.* present an incremental proof of the producer/consumer property for the PCI protocol using the approach. Besides being tailored to software and being event-based, our approach is not only restricted to refinement steps.

Schmaltz *et al.* [14] present a generic network on chip model as a framework for correct on-chip communication. They identify key constraints on architectures and show protocol correctness given these constraints. However, their work focuses on the topologies in general, whereas this work aims at the verification of sophisticated endpoints.

Chen *et al.* [15] propose a modular, refinement based approach to verify transaction-based hardware implementations against their specification models and illustrate their methodology using a cache coherency protocol. Their approach is tailored to a different application area: verifying implementations against specifications.

Müffke [16] 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.

General hardware verification based on refinement checking or simulation relations has a long history. Finn and Fourman [17] present the toolset LAMBDA, a refinement based general-purpose design assistant using mathematical logic to represent and manipulate system behaviour. Abadi and Lamport [18] show the existence of refinement mappings in their widely-cited article. McMillan [19] proposes a compositional rule for hardware verification based on local refinements which can be efficiently model checked.

The combination of Isabelle/HOL and NuSMV using the IHaVeIt tool has been applied to a variety of hardware verification instances. Schmaltz [20] applies it to the area of clock domain crossing and the time-triggered hardware implementing it. Alkassar *et al.* [21] use the tool to show the correctness of a fault-tolerant real-time scheduler and its hardware implementation. In both cases, the authors apply a similar strategy: they use theorem proving to argue about real-time, asynchronous properties of the system, and the model checker to prove properties of finite state machines which are used to model the hardware implementation. A more general overview of *hybrid verification approaches* can be found in the survey from Bhadra *et al.* [22]

An overview of existing work on specific protocol verification such as PCI Express or the AMBA protocol can be found in our previous contributions covering the case studies [1], [2]. As this work focuses on the protocol independent, general methodology, we omit the protocol specific work here.

II. Basics

A. Notation

To represent data, we often use the option datatype and records which we introduce in the following. Moreover, we define discrete time signals and introduce correctness properties.

a) Option Type: To specify a possibly undefined value, we use the *option datatype* that is well known from functional programming languages. For an element of type $(\alpha)option$, we write $\mathrm{Some}\ x$ for $x \in \alpha$ and None for the two constructors. The selection operator $\mathbf{the}$ is used to access a value: $\mathbf{the}(\mathrm{Some}\ x) = x$ and $\mathbf{the}(\mathrm{None})$ is left unspecified.

b) Records: We use the Isabelle notation for *records*. Let $\mathcal{S}_1, \ldots, \mathcal{S}_n$ be sets and $l_1, \ldots, l_n$ be labels. The record $\mathcal{R} = (\!|\, l_1 : \mathcal{S}_1, \ldots, l_n : \mathcal{S}_n \,|\!)$ yields the set of all tuples $(l_1 = s_1, \ldots, l_n = s_n)$ where $s_i \in \mathcal{S}_i$. For $r \in \mathcal{R}$, we refer to the field l_i with $r.l_i$. An *update* to a field l_i by a value $s_i \in \mathcal{S}_i$ is denoted $r(\!|\, l_i := s_i \,|\!)$. If the context is obvious, we write l_i for $r.l_i$. For $i \in [1, n]$, the domain of field l_i is given by $\mathbf{dom}(l_i, \mathcal{R}) = \mathcal{S}_i$. For a label l, we define the element operator $l \tilde{\in} R$ as $\exists j \in [1, n].\ l = l_j$.

Given the usual Cartesian product for sets, $\prod_{i\in[1,n]} \mathcal{S}_i = \mathcal{S}_1 \times \ldots \times \mathcal{S}_n$, we define an analogous operation for records: $\widetilde{\prod}_{i\in[1,n],\mathcal{L}} \mathcal{R}_i = (\!|\, l_1 : \mathcal{R}_1, \ldots, l_n : \mathcal{R}_n \,|\!)$ where $\mathcal{L} : [1 : n] \to \{l_i \mid i \in [1, n]\}$ is a *labelling function* that assigns a label to each index. This labelling function can be given as an explicit function or a set of pairs. We also use this operator for sets as arguments, i. e. to create a record from sets. A disjoint union operator over records is given by the set of record fields: $\widetilde{\biguplus}_{i\in[1,n]} \mathcal{R}_i = \{l_{i,j} \mid l_j \tilde{\in} \mathcal{R}_i\}$. Finally, we define the concatenation of records: $\mathcal{R}_i = (\!|\, l_{i,0} : \mathcal{S}_{i,0}, \ldots, l_{i,m_i} : \mathcal{S}_{i,m_i} \,|\!)$: $\widetilde{\bigodot}_{i\in[0,n]} \mathcal{R}_i = (\!|\, l_{0,0} : \mathcal{S}_{0,0}, l_{0,1} : \mathcal{S}_{0,1}, \ldots, l_{n,m_n} : \mathcal{S}_{n,m_n} \,|\!)$. We use $R_0 \tilde{\uplus} R_1$, $R_0 \tilde{\times}_{\mathcal{L}} R_1$, and $R_0 \tilde{\odot} R_1$ for the binary variants.

c) Signals: A *signal sig* is a function from discrete time to a signal data type α, i.e. $sig : \mathbb{N} \to \alpha$. We denote the value of a signal sig at time t with sig^t.

d) Correctness Properties: In the context of this paper, a *correctness property* P is either a propositional logic formula, an LTL formula, or a CTL formula given a state machine model M, an external input signal i, and a set of assumptions $\mathcal{A}$. If the model and the input signal satisfies the correctness property given the assumptions, we write $\langle M, i\rangle \models_{\mathcal{A}} P$.

B. Communicating Abstract State Machines

We use standard Mealy machines to represent components in the framework. A state machine is specified by an *initial state*, a *transition function*, and an *output function* together with sets for the *state space*, the *input space*, and the *output space*.

Definition 1 (Mealy Machine): A Mealy machine is given by a 6-tuple $(S, I, O, s0, \delta, \omega)$ with domains S, I, O, initial state $s0 \in S$, transition function $\delta : S \times I \to S$, and output function $\omega : S \times I \to O$. We denote the next state $s' = \delta(s, i) \in S$ and the current output is $\omega(s, i) \in O$.

If not stated otherwise, we assume that the domain spaces are given as records. Given a state machine and an input signal, we can define the *execution trace* and the *output signal*.

Definition 2 (Execution Trace and Output Signal): Given a state machine $M = (S, I, O, s0, \delta, \omega)$ and an input signal $in : \mathbb{N} \to I$, we define the execution trace τ as $\tau^0_{M,in} = s0$ and $\tau^t_{M,in} = \delta(\tau^{t-1}_{M,in}, in^{t-1})$ for $t > 0$. The output signal, $out_{M,in} : \mathbb{N} \to O$, is given by $out^t_{M,in} = \omega(\tau^t_{M,in}, in^t)$.

To define composition operators in Section IV, we introduce a model of synchronous communication among state machines. We model uni-directional communication from a *source* to a *destination* by connecting an output of the source to an input of the destination. This 'connection' is modelled by defining the input component using the output function of source state machine. To illustrate the general approach, assume we want to model a communication from output $x \tilde{\in} O_s$ to input $y \tilde{\in} I_d$. Given an input signal $i^t_d \in I_d$, we use the following definition for its record component y instead of considering it as an environment input: $i^t_d.y = (\omega_s(\tau^t_{M_s,in_s}, in^t_s)).x$. We can generalize this approach by introducing a global communication function for a set of abstract state machines.

Definition 3 (Communication Function): Given a set of state machines $\mathcal{M} = \{M_0, \ldots, M_n\}$. We define communication as a partial function $\mathbf{com}_{\mathcal{M}} : \widetilde{\biguplus}_i I_i \to \widetilde{\biguplus}_i O_i$ such that $\mathbf{com}_{\mathcal{M}}(y_i) = x_j$ if output x of M_j is connected to y of M_i and undef otherwise. We call an input y of M_i *external* with respect to $\mathcal{M}$ iff $\mathbf{com}_{\mathcal{M}}(y_i) = \mathrm{undef}$ and *internal* otherwise.

C. Standard Interface

We use a simple *handshake* protocol to realise a uni-directional communication between two state machines using three signals: a *valid* and a *data* signal provided by the sender, and a *busy* from the receiver. The basic idea is that a sender provides data on the data signal and raises the valid signal to indicate so. If the receiver's busy signal is low, it is an acknowledgement that the receiver samples the data in the same time step. If the busy signal is active, the receiver cannot sample the data yet and the sender has to keep its signals stable. We refer to the three signals with $b^t \in \mathbb{B}$, $v^t \in \mathbb{B}$, and $d^t \in \mathcal{D}$ where $\mathcal{D}$ is the set of data elements to be communicated. We use a suffix o to refer to an output, and a suffix i for an input.

We formalise the protocol in terms of two assumptions: one that defines *valid outputs* provided by a sender and one that specifies the *correct sampling* behaviour of a receiver.

Assumption 1 (valid outputs): M provides *valid output* at time t iff $vo^t \implies (do^t = x) \wedge \big(bi^t \implies vo^{t+1} \wedge (do^{t+1} = x)\big)$ for some data element $x \in O$.

Assumption 2 (correct sampling): M has to sample input data $x = di^t$ at time t iff $vi^t \wedge \neg bo^t$.

We use the option datatype to model the data signal and omit the valid signal. The interface consists of two signals: a busy signal $b^t \in \mathbb{B}$ and a data signal $d^t \in \mathcal{D}\ option$. The valid signal can be obtained by $v^t \equiv (d^t \neq \mathrm{None})$. Next, we generalize the concept to specify the input and output records of a state machine implementing n *input interfaces*—the data signal is an input—and m output interfaces—the data signal is an output. We use the following labelling convention for the signal names: the k-th data input is di_k together with the k-th busy output bo_k; analogously for the output interface. Thus, we restrict the input and output records to the following generalised constructs:

$$
\begin{aligned}
I &= \Big(\widetilde{\prod}_{m,\rho_{bi,m}} \mathbb{B}\Big) \mathbin{\tilde{\odot}} \Big(\widetilde{\prod}_{i\in[1,n],\rho_{di,n}} \mathcal{D}_i\ option\Big) \\
&= \mathcal{BI}^m \mathbin{\tilde{\odot}} \mathcal{DI}^n \\
O &= \Big(\widetilde{\prod}_{n,\rho_{bo,n}} \mathbb{B}\Big) \mathbin{\tilde{\odot}} \Big(\widetilde{\prod}_{j\in[1,m],\rho_{do,m}} \mathcal{D}_j\ option\Big) \\
&= \mathcal{BO}^n \mathbin{\tilde{\odot}} \mathcal{DO}^m
\end{aligned}
$$

where the labelling $\rho_{l,k}$ is given by $\rho_{l,k}(i) = l_i$ for $i \in [1, k]$. Given an element $i \in I$, we use $\mathcal{BI}^m(i)$ to refer to the m busy inputs and $\mathcal{DI}^n(i)$ to refer to the n data inputs. For $o \in O$, we use $\mathcal{BO}^n(o)$ and $\mathcal{DO}^m(o)$.

Since we use Mealy machines to model abstract components, it is possible to create combinatorial loops when we compose state machines using this handshaking protocol. We define two interface properties which prevent this. When we introduce operators for state machine composition, we will use these properties as local constraints.

Assumption 3 (busy-independent data): Given a state machine M and an input signal i. M provides busy-independent data output signals iff ω satisfies for all $k \in [1, m]$: $\omega\ (\tau^t_{M,i}, i^t).do_k = \omega\ (\tau^t_{M,i}, i^t(\!|\, bi_k := \mathsf{T} \,|\!)).do_k$

Assumption 4 (data-independent busy): Given a state machine M and an input signal i. M provides data-independent busy signals iff ω satisfies for all $k \in [1, n]$: $\omega\ (\tau^t_{M,i}, i^t).bo_k = \omega\ (\tau^t_{M,i}, i^t(\!|\, di_k := \mathrm{None} \,|\!)).bo_k$

III. Abstract Components

In this section, we introduce basic building blocks: a polymorphic buffer of finite size for arbitrary data elements that obeys the standard interface, a component for data modification, and one for signal routing.

A. Buffer

We model a buffer using a list and specify the basic operations for buffers or queues: an *enqueue* operation enq to add a

new element, a *dequeue* operation deq to remove the oldest element, and a *top* operation top to obtain the oldest element. We use predicates $empty$ and $full$ for corresponding buffer states. Using lists, all these operations are straightforward. We put the buffer in a state machine wrapper implementing the standard interface.

Definition 4 ($(\alpha)buffer$ of finite size): A generic buffer for datatype α and finite size $s \in \mathbb{N}$ is given by:

$$
\begin{aligned}
&S = (\!|\, buf : \alpha\,\mathbf{list}, size : \mathbb{N} \,|\!), \quad s0 = (\!|\, buf = \mathrm{Nil}, size = s \,|\!)\\
&I = (\!|\, bi : \mathbb{B}, di : \alpha\, option \,|\!), \quad O = (\!|\, bo : \mathbb{B}, do : \alpha\, option \,|\!)\\
&\delta = \lambda s \in S.\ \lambda i \in I.\ \mathrm{let}\\
&\qquad s' = \mathrm{if}\ \neg(i.bi \vee \mathbf{empty}\ s)\ \mathrm{then}\ \mathbf{deq}\ s\ \mathrm{else}\ s\\
&\qquad \mathrm{in\ if}\ (i.di = \mathrm{Some}\ x \wedge \neg\mathbf{full}\ s')\ \mathrm{then}\ \mathbf{enq}\ s'\ x\\
&\qquad\quad \mathrm{else}\ s'\\
&\omega = \lambda s \in S.\ \lambda i \in I.\ \mathrm{let}\\
&\qquad d = \mathrm{if}\ \neg(i.bi \vee \mathbf{empty}\ s)\ \mathrm{then\ Some}\ (\mathbf{top}\ s)\\
&\qquad\quad \mathrm{else\ None}\\
&\qquad \mathrm{in}\ (\!|\, bo = \mathbf{full}\ s,\ do = d \,|\!)
\end{aligned}
$$

We refer to such a buffer with $(\alpha, s)\ Buf$.

B. Data Modification

The data modification component is a minimalistic state machine implementing modifications to the data element. We abstract from the modification and model it as a function $f : S \times \alpha \to \beta$. A typical use of the component is the extension of a data element with a sequence number or a check sum. An optional element opt is added to provide required data; together with an initial state opt^0 and an update function δ_{opt}.

Definition 5 (Data Modification): Given a function $\mathbf{f} : S \times \alpha \to \beta$, a data modification is a simple state machine with:

$$
\begin{aligned}
&S = (\!|\, opt : Opt \,|\!), \quad s0 = (\!|\, opt = opt^0 \,|\!)\\
&I = (\!|\, bi : \mathbb{B}, di : \alpha\, option \,|\!), \quad O = (\!|\, bo : \mathbb{B}, do : \beta\, option \,|\!)\\
&\delta = \lambda s \in S.\ \lambda i \in I.\ (\!|\, opt = (\delta_{opt}\ (s, i)) \,|\!)\\
&\omega = \lambda s \in S.\ \lambda i \in I.\ \mathrm{let}\\
&\qquad d = \mathrm{if}\ (i.di = \mathrm{Some}\ x)\ \mathrm{then\ Some}\ (\mathbf{f}\ (s, x))\\
&\qquad\quad \mathrm{else\ None}\\
&\qquad \mathrm{in}\ (\!|\, bo = i.bi, do = d \,|\!)
\end{aligned}
$$

C. Routing

The goal of the routing component is to distribute control or data flow. Typical applications are the arbitration among data elements or the generation of messages while stalling incoming data. We also use an optional state component opt with initial state opt^0 and update function δ_{opt}. The core of the building block are two functions f_b and f_d which represent the modification of the busy and data signals.

Definition 6 (Routing Component): Let $f_b : S \to I \to \widetilde{\prod}_{n,\rho_{bo,n}} \mathbb{B}$ and $f_d : S \to I \to \widetilde{\prod}_{j\in[1,m],\rho_{do,m}} \mathcal{D}_j\ option$ be *routing functions*. A routing component is given by:

$$
\begin{aligned}
&S = (\!|\, opt : Opt \,|\!), \quad s0 = (\!|\, opt = opt^0 \,|\!)\\
&I = \left(\widetilde{\prod}_{m,\mathcal{L}_{bi}} \mathbb{B}\right)\ \tilde{\odot}\ \left(\widetilde{\prod}_{i\in[1,n],\mathcal{L}_{di}} \mathcal{D}_i\ option\right)\\
&O = \left(\widetilde{\prod}_{n,\mathcal{L}_{bo}} \mathbb{B}\right)\ \tilde{\odot}\ \left(\widetilde{\prod}_{j\in[1,m],\mathcal{L}_{do}} \mathcal{D}_j\ option\right)\\
&\delta = \lambda s \in S.\ \lambda i \in I.\ (\!|\, opt = (\delta_{opt}\ (s, i)) \,|\!)\\
&\omega = \lambda s \in S.\ \lambda i \in I.\ (f_b\ s\ i) \tilde{\odot} (f_d\ s\ i)
\end{aligned}
$$

IV. Composition of Abstract Components

In this section, we detail ways to compose state machines in order to incrementally build complex systems. We introduce two standard operations: *parallel* and *sequential* composition. Parallel composition can be used to combine send and receive parts of an endpoint model, for example. A typical application for sequential composition is the modelling of interconnects or the composition of stack layer models. An overview of the sequential composition is shown in Fig. 1(a).

Definition 7 (Parallel Composition): The parallel composition $M_1 \mathbf{par} M_2$ with $M_i = (S_i, I_i, O_i, s0_i, \delta_i, \omega_i)$ is given by:

$$
\begin{aligned}
&S = S_1 \tilde{\times}_{\mathcal{L}} S_2, \quad s0 = (\!|\, m_1 = s0_1,\ m_2 = s0_2 \,|\!)\\
&I = I_1 \tilde{\times}_{\mathcal{L}} I_2, \quad O = O_1 \tilde{\times}_{\mathcal{L}} O_2\\
&\delta = \lambda s.\ \lambda i.\ (\!|\, m_1 = \delta_1\ (s.m_1, i.m_1),\ m_2 = \delta_2\ (s.m_2, i.m_2) \,|\!)\\
&\omega = \lambda s.\ \lambda i.\ (\!|\, m_1 = \omega_1\ (s.m_1, i.m_1),\ m_2 = \omega_2\ (s.m_2, i.m_2) \,|\!)
\end{aligned}
$$

where $\mathcal{L} = \{(1, m_1), (2, m_2)\}$ is the labelling.

To define the sequential composition in a compact way, we need to define some internal signals. Assume we want to compose M_1 sequentially with M_2, i. e. $M_1\, \mathbf{seq}\, M_2$. As illustrated in Fig. 1(a), the busy signals of M_1 are connected to the busy outputs of M_2, and vice versa for the data signals. Since there is a *cyclical* dependency, we need that either M_1 provides busy-independent data outputs (Assumption 3) or M_2 provides data-independent busy outputs (Assumption 4). We first define the sequential composition in Definition 8 assuming the internal signals $bint$ and $dint$ as depicted. We define these signals in Definition 9.

Definition 8 (Sequential Composition): Given M_1, M_2 with $M_i = (S_i, I_i, O_i, s0_i, \delta_i, \omega_i)$ where $I_1 = \mathcal{BI}^m \tilde{\odot} \mathcal{DI}^n$, $O_1 = \mathcal{BO}^n \tilde{\odot} \mathcal{DO}^m$ and $I_2 = \mathcal{BI}^p \tilde{\odot} \mathcal{DI}^m$, $O_2 = \mathcal{BO}^m \tilde{\odot} \mathcal{DO}^p$. Then, the sequential composition $M_1\, \mathbf{seq}\, M_2$ is defined as:

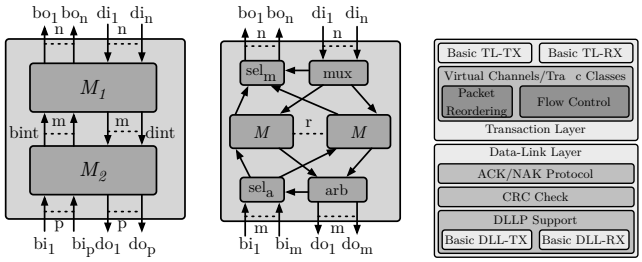
$$
\begin{aligned}
&S = S_1 \tilde{\times}_{\mathcal{L}} S_2, \quad s0 = (\!|\, m_1 = s0_1,\ m_2 = s0_2 \,|\!)\\
&I = \mathcal{BI}^p \,\tilde{\odot}\, \mathcal{DI}^n, \quad O = \mathcal{BO}^n \,\tilde{\odot}\, \mathcal{DO}^p\\
&\delta = \lambda s.\ \lambda i.\ (\!|\, m_1 = \delta_1\ (s.m_1, bint \tilde{\odot} \mathcal{DI}^n(i)),\\
&\qquad m_2 = \delta_2\ (s.m_2, \mathcal{BI}^p(i) \tilde{\odot} dint) \,|\!)\\
&\omega = \lambda s.\ \lambda i.\ \mathcal{BO}^n(\omega_1\ (s.m_1, bint \tilde{\odot} \mathcal{DI}^n(i))) \,\tilde{\odot}\\
&\qquad \mathcal{DO}^p(\omega_2\ (s.m_2, \mathcal{BI}^p(i) \tilde{\odot} dint))
\end{aligned}
$$

where the labelling $\mathcal{L}$ is $\{(1, m1), (2, m2)\}$.

Definition 9 (Internal Signals): Let $M = M_1\, \mathbf{seq}\, M_2$ be the sequential composition of M_1 and M_2, and $i^t \in I$ be an input signal. Moreover, let $nod = \widetilde{\prod}_{m,\rho_{m,di}} \mathrm{None}$ and $allb = \widetilde{\prod}_{n,\rho(n,bi)} \mathsf{T}$. Then, $bint = \mathcal{BO}^m(\omega_2\ (s.m2, int_2))$ and $dint = \mathcal{DO}^m(\omega_1\ (s.m1, allb \tilde{\odot} \mathcal{DI}^n(i)))$ if Assumption 3 holds. In case of Assumption 4, $bint = \mathcal{BO}^m(\omega_2\ (s.m2, \mathcal{BI}^p(i) \,\tilde{\odot}\, nod)$ and $dint = \mathcal{DO}^n(\omega_1\ (s.m1, int_1))$.

A. Replication

The replication operator is the first non-standard composition that we introduce. The goal is the controlled, parallel execution of r copies of a component while maintaining the external input and output interfaces. When we summarise the case studies in Section VI, we present examples how this operation is applied. The basic schematics is depicted in Fig. 1(b).



(a) Sequential Comp. (b) Replication (c) PCI Express Study

Fig. 1. Overviews of Composition Operators and PCI Express Case Study

The four main components are a multiplex function mux, an arbitration function arb, the state machine M to be replicated, and and additional component opt with initial state opt^0 and step function δ_{opt}. An instance of the operator for a state machine M is given by $\mathbf{rep}(r, OPT, mux, arb)$ where

- r is the number of replications,
- $OPT = (Opt, opt^0 \in Opt, \delta_{opt})$ is the optional part,
- $mux : Opt \times \mathcal{DI}^n \to ([1,r] \to \mathcal{DI}^n)$ is the multiplex function, and
- $arb : Opt \times ([1,r] \to \mathcal{DO}^m) \to (\!|\, w : [1,r],\, do : \mathcal{DO}^m \,|\!)$ is the arbitration function.

Definition 10: Let M be the state machine to be replicated. Then, $M_r = \mathbf{rep}(n, OPT, mux, arb)\, M$, is given by:

$$
\begin{aligned}
S_r &= (\!|\, ms : [1,n] \to S,\ opt : Opt \,|\!), \quad I_r = I, \quad O_r = O \\
s0_r &= (\!|\, ms = (\lambda w \in [1,n].s0),\ opt = opt^0 \,|\!) \\
\delta_r &= \lambda s.\, \lambda i.\, \mathrm{let} \\
&\qquad dii = (mux(s.opt, \mathcal{DI}^n(i))) \\
&\qquad doi = \lambda w.\, \omega(s.ms\ w, (\widetilde{\textstyle\prod}_{m,\rho_{m,bi}} \mathsf{T}) \tilde{\odot} dii\ w) \\
&\qquad arbo = arb\,(s.opt, \lambda w.\, \mathcal{BO}^m(doi\ w)) \\
&\qquad bii = sel_a\,(\mathcal{BI}^m(i), arbo.w) \\
&\qquad ms' = \lambda w.\, \delta(s.ms\ w, (bii\ w) \tilde{\odot} (dii\ w)) \\
&\quad \mathrm{in}\ (ms = ms',\ opt = \delta_{opt}(s.opt)) \\
\omega_r &= \lambda s.\, \lambda i.\, \mathrm{let} \\
&\qquad dii = (mux(s.opt, \mathcal{DI}^n(i)) \\
&\qquad doi = \lambda w.\, \omega(s.ms\ w, (\widetilde{\textstyle\prod}_{m,\rho(m,bi)} \mathsf{T}) \tilde{\odot} dii\ w) \\
&\qquad bo = sel_m\,(\lambda w.\, \mathcal{BO}^n(doi\ w), dii.w)) \\
&\quad \mathrm{in}\ bo \,\tilde{\odot}\, arb\,(s.opt, \lambda w.\, \mathcal{DO}^m(doi\ w))
\end{aligned}
$$

where $sel_m : ([1,n] \to \mathcal{BO}_n \times [1,n]) \to \mathcal{BO}_n$, $sel_a : (\mathcal{BI}_m \times [1,m]) \to ([1,m] \to \mathcal{BI}_m)$ are the busy select functions. Note that the construction requires M to satisfy Assumption 3.

We conclude the specification of the replication operator by stating two assumptions which ensures that the construction *makes sense*. The first one states that the multiplex function selects a unique internal component for a given data element.

Assumption 5 (Valid Multiplex Function): Let $id = mux\,(opt, i)$. A valid multiplex function mux satisfies:

$$
\begin{aligned}
\exists!\ w \in [1:r]. \quad & (id\ w = i) \wedge \\
& \forall k \neq w.\, (id\ k = \widetilde{\textstyle\prod}_{n,\rho(n,di)} \mathrm{None})
\end{aligned}
$$

The second assumption states similarly that the output of the arbitration function is coherent.

Assumption 6 (Coherent Arbitration Function): Let $(w, do) = arb\,(opt, idos)$. A valid arbitration function arb satisfies:

$$\forall idos \in [1,r] \to \mathcal{DO}^m.\, (do = idos\ w)$$

B. Multiplex/Arbitrate

The multiplex/arbitrate composition is a similar, but more general, construct as the replicate operator. The goal is to parallelise n arbitrary components in a structured way. Thus we are not restricted to an instantiation with n copies of a state machine. In order to define a generic construction and allow arbitrary components, the input and output interface have to change. We only allow components with the same number of input and output interfaces. We also assume some *logical* relation between the i-th data component of each state machine. Then, the i-th external input or output interface provides data elements from the union of the all the i-th internal interfaces.

Moreover, the multiplex component does not have to be unique anymore and can select more than one internal state machine. For example, it might split a data element and input the two parts to two internal state machines. Similarly, the arbitration function may select more than one internal component, but is still only allowed to produce a single output, of course. Again, an idea may be the data elements that have been split by the multiplex function are combined again. Because of space restrictions, we omit the formal definition here as it is analogous to Definition 10.

C. Communication Channels

To conclude the section on component composition, we introduce models of communication channels. Having modelled the endpoints, we need to be able to interconnect them. We introduce both: a model for point-to-point topologies, and a model for communication busses.

The goal is to specify interconnects with a transmission delay $d \in \mathbb{N}$ and a capacity of $c \leq d \in \mathbb{N}$ data elements. To define such a point-to-point channel, we can use previously specified components: sequential composition, a buffer, a data modification, and a routing component. The idea is as follows: we use a buffer of size c to provide the capacity. To model the delay, we first use a data modification to add a sequence number $seq \in [0,d)$ to a data element. After the buffer, we use a routing component to check if a counter value $cnt \in [0,d)$ is equal to the sequence number and only in case it is equal, the data is passed on. Both the sequence number and the counter are increased in every time step, whether there is a new data element or not. Note that this works because a buffer generates a delay of at least one and we ensure that at all times the sequence number is equal to the counter value.

Definition 11 (Point-to-Point Channel): Let M_{seq} be the state machine obtained from instantiating a data modification unit with $Opt = [0,d)$, $opt^0 = (\!|\ opt = 0\ |\!)$, $\delta_{opt} = \lambda s, i.\, s.opt + 1$, and $f = \lambda s, x.\ (x, s.opt)$. Moreover, let $M_{buf} = (\mathcal{D} \times [0,d), c)\, Buf$. Finally, let M_{del} be a routing unit with $n = m = 1$, $Opt = [0,d)$, $opt^0 = (\!|\ opt = 0\ |\!)$,

and $\delta_{opt} = \lambda s, i.\ s(\!|\ opt := s.opt + 1\ |\!)$. The routing functions are $f_b = \lambda s, i.\ s.opt \neq \mathbf{snd}(i.di)$ and $f_d = \lambda s, i.$ if $\neg f_b\ (s, i)$ then Some $(\mathbf{fst}\ (i.di))$ else None. Then, $(d, c, \mathcal{D})\ Chan = (M_{seq}\, \mathbf{seq}\, M_{buf}\, \mathbf{seq}\, M_{del})$.
In order to define a communication bus, we use the channel and simply compose it with two more routing units to generate the inputs and outputs to the channel.

Definition 12 (Communication Bus): Let $g : I^v \to I$ be an arbitration function to select among v bus inputs the one that is allowed to use the bus. A bus with delay d, capacity $c \leq d$, inputs v, and outputs w, $(d, c, v, w, g)\ Bus$, is constructed as $M_{arb}\, \mathbf{seq}\, (d, c)\ \ Chan\, \mathbf{seq}\, M_{mux}$. M_{arb} is a routing instance with $(n, m) = (v, 1)$ using g to select an input. M_{mux} is a routing instance with $(n.m) = (1, w)$ that forwards the input to all w outputs.

V. Verification and Automatic Tool Support

In this section, we detail our verification approach and the generic correctness properties of the framework components. By integrating a model checker and using automated theorem prover, we react to concerns regarding the feasibility of the theorem proving approach. The support for automated verification tools aims mainly at simplifying the discharging of the local assumptions. But we were also able to apply them to parts of the generic correctness argumentation.

Here, we focus on the integration and use of the model checker. Since our modelling approach is based on state machines, we can use NuSMV to reason about LTL and CTL properties. The main use of the model checker is to discharge the local assumptions when applying one of the specific composition operators. A good example for the merits of the model checker is the arbitration function in the replication composition. As we will see in Section V-B, we require the arbitration function to be fair with respect to the data signals. Instead of a tedious induction proof using interactive theorem proving, we can use the model checker: in the simplest case we can check whether a given arbitration function satisfies

$$\mathbf{G}(i.di_k = \text{Some } x) \implies \mathbf{F}\ (arb.w = i)$$

where $\mathbf{G}$ and $\mathbf{F}$ are the standard LTL operators for *globally* and *finally*.

A proven LTL or CTL property can easily be translated to the execution trace semantic from Definition 2. This way, we can integrate model checking in the theorem proving workflow. Since we use NuSMV mainly to discharge local assumptions when we apply the framework to a specific protocol, the 'end-user' verification part benefits from the integration. It is a very promising first step to the final goal of reducing the theorem prover to a knowledge management system only and large parts of the framework application steps can already be automated using NuSMV.

A. Basic Components

In order to argue about correctness in a reasonable way, we have to introduce an environment assumption first. We assume in the following that the busy signal provided by the environment, i. e. by the host system, is fair in the sense that it is not constantly active. Thus, the environment allows progress. Assumption 7 formulates this by stating that a busy signal is only constantly active for a finite time interval.

Assumption 7 (Fair busy Signals): For all external busy signals b holds: $\forall t.\ \exists k.\ \neg b^{t+k}$
Note that this is a common assumption for inputs with a semantics similar to the busy signal. We can easily see that the definition of the buffer satisfies Assumptions 1, 2, 3, and 4. Here, we show that the buffer provides stable output signals as long as the busy input is active. It is basically a conclusion from Assumption 1. Then, we state the two main buffer theorems: a liveness and an ordering (FIFO) property.

Lemma 1 (Stable Buffer Outputs): Given a generic buffer $B = (S, I, O, s0, \delta, \omega)$ and an input signal $i^t \in I$, B satisfies:

$$\forall x \in \mathbf{dom}(do, O).\ bi^t \wedge (do^t = \text{Some } x) \implies \exists k.\ \neg bi^{t+k} \wedge (\forall k' \leq k.\ do^{t+k'} = \text{Some } x)$$

Proof: With the integration of the NuSMV model checker, the lemma is automatically shown by re-stating it as an LTL formula $((do = \text{Some } x)\, \mathbf{Until}\, (\neg bi))$. It is then easily translated to our execution trace semantics in HOL. ■

Theorem 1 (Buffer Liveness): Given a fair busy signal, a buffer satisfies the following liveness property:

$$\forall x \in \mathbf{dom}(di, I).\ \neg bo^t \wedge (di^t = \text{Some } x) \implies \exists k.\ (do^{t+k} = \text{Some } x)$$

Proof: This theorem can be automatically shown with NuSMV using the assumption of a fair environment. We show that $\neg bo \wedge (di = \text{Some } x) \implies \mathbf{F}(do = \text{Some } x)$ and translate it to the execution trace semantics. ■
Note that for a simple buffer, $\mathbf{dom}(di, I) = \mathbf{dom}(do, O)$ holds since the basic buffer construct does not implement any data modification. Therefore, we can quantify over $\mathbf{dom}(di, I)$ and argue about the data output.

Theorem 2 (Buffer FIFO Property): A buffer preserves the ordering of its input data. Let $t < t'$ such that $\neg bo^t$ and $\neg bo^{t'}$, then it holds that:

$$\forall x, y \in \mathbf{dom}(di, I).\ (di^t = \text{Some } x) \wedge (di^{t'} = \text{Some } y) \implies \exists k, k'.\ (do^{t+k} = \text{Some } x) \wedge (d_o^{t+k+k'} = \text{Some } y)$$

where the delay values k, k' are given by Theorem 1.

Proof: The liveness part of the statement is shown with Theorem 1. Since buffers are modelled using lists, the ordering property is shown using the ordering property of lists. ■

B. Composition Operators

Given the correctness of the basic building blocks, we need to argue about the composition operators. Our main goal is to show that the properties for the basic components are *preserved* by the compositions. Informally, the idea is that if a component satisfies a correctness property P, we aim at showing that a composed system satisfies a correctness property P' that can be derived from P only using the construction of the composition.

For the parallel composition, the correctness property is straightforward: the composed system satisfies the conjunction

of the individual correctness properties. Since parallel composition only executes the two state machines simultaneously without any control or data modification, one can easily see that this is the case.

Lemma 2 (Parallel Composition Correctness): Given state machines M_1, M_2 and corresponding input signals $i_1^t \in I_1$ and $i_2^t \in I_2$. Moreover, let $i = \lambda t \in \mathbb{N}.\ (\!|\, m_1 = i_1^t,\ m_2 = i_2^t \,|\!)$ be the input signal for the parallel composition $M_1 \mathbf{par} M_2$. Then, the following holds:

$$\langle M_1, i_1\rangle \models_{\mathcal{A}_1} P_1 \wedge \langle M_2, i_2\rangle \models_{\mathcal{A}_2} P_2 \implies \langle M_1 \mathbf{par} M_2, i\rangle \models_{\mathcal{A}_1 \cup \mathcal{A}_2} P_1 \wedge P_2$$

Proof: The proof is straightforward by applying the definition of parallel composition (Definition 7). ∎

The corresponding lemma for the sequential composition is slightly more complicated since not every input or output of the individual system is still an external input or output in the composed system (cf. Fig. 1(a)). Thus, we need to make the respective substitutions in the correctness statements. To describe a substitution of x by y in a formula P in the following, we use the common notation $P[x/y]$.

Lemma 3 (Sequential Composition Correctness): Given M_1, M_2 and corresponding input signals $i_1^t \in I_1$, $i_2^t \in I_2$. Let $bint$ and $dint$ be the internal signals from Definition 9, and let $i = \lambda t \in \mathbb{N}.\ \mathcal{BI}^p(i_2^t) \tilde{\odot} \mathcal{DI}^n i_1^t$ be the input signal to the sequential composition $M = M_1 \mathbf{seq} M_2$. Then, it holds:

$$\langle M_1, i_1\rangle \models_{\mathcal{A}_1} P_1 \wedge \langle M_2, i_2\rangle \models_{\mathcal{A}_2} P_2 \implies \langle M, i\rangle \models_{\mathcal{A}} P_1[\mathcal{BI}(i_1^t)/bint] \wedge P_2[\mathcal{DI}(i_2^t)/dint]$$

where $\mathcal{A}$ is the union of $\mathcal{A}_1$ and $\mathcal{A}_2$ with the respective input signal substitutions.

Proof: Similar to the proof of Lemma 2, the proof is basically an application of the definition of sequential composition (Definition 8). The proof can be obtained using automatic theorem proving via sledgehammer. ∎

Next, we will provide generic correctness results for the replication operator. We will state assumptions that have to be discharged when the composition is instantiated. Given these assumptions, we can show a generic correctness theorem. Since the replication operation is more restrictive than the multiplex/arbitrate composition, we can derive more correctness properties for the former. Therefore, we also give the former preference over the latter in the case studies wherever possible.

The assumptions for the replication operator can be summarized as: (i) the inner components are correct and ensure liveness, (ii) the multiplex function is *correct*, i.e. it multiplexes valid inputs to some inner component (Assumption 5), and (iii) the arbitration is *fair* with respect to an active data signal from an inner component. The following assumption states the first point. We omit the fairness of the arbitration function here due to space limitations.

Assumption 8 (Inner Component): Let M be the state machine to be replicated using the replication operator. Then, M has to satisfy the busy-independent output assumption and has to provide stable output signals, i. e. $\forall i \in I.\ \langle M, i\rangle \models_{\mathcal{A}}$ Assumptions 1 and 3, where $\mathcal{A}$ is the fair environment assumption. Moreover, M has to satisfy liveness:

$$\forall i \in [1,n].\ \forall x \in \mathbf{dom}(di_i, I).\ \neg b_o^t \wedge (di_i^t = \text{Some } x) \implies (\exists j \in [1,m], k \in \mathbb{N}.\ do_j^{t+k} = \text{Some } f_{i,j}(x))$$

where $f_{i,j} : \mathbf{dom}(di_i, I) \to \mathbf{dom}(do_j, O)$ is a potential data transformation applied by the inner component.

The following theorem states that given Assumption 8 and the assumptions on the multiplex and arbitration functions, the derived system satisfies liveness

Theorem 3 (Correctness of Replication): If the inner state machine satisfies Assumption 8, the system obtained using the replication operator satisfies this assumption again if the multiplex and arbitration functions ensure the previously mentioned assumptions.

Proof: The Isabelle proof of Theorem 3 is mainly obtained by unfolding definitions and assumptions. An induction is needed to conclude the stable input signals for the time interval from Assumption 8 and the fairness of the arbitration function. ∎

VI. Case Studies

The development of the framework was driven by the work on two case studies covering rather different protocols: first, the ARM AMBA High-performance Bus (AHB) protocol, an arbiter-based master-slave bus protocol for system on chips. Second, the PCI Express protocol, an off-chip point-to-point high-performance protocol implementing many sophisticated features of current and future on-chip communication protocols. By choosing two case studies covering a wide range of protocol features as well as bus and point-to-point network topologies, we show the utility and breadth of the framework.

A. AMBA High-performance Bus

The AHB protocol is a bus protocol where masters access data stored in slaves, all connected to a bus. Bus access is regulated by an arbiter. The bus itself consists of an address and data bus. Each transfer is split into two, in the simple case, consecutive phases: an address and a data phase. In two steps, we add pipelined transfers and burst support.

Pipelined bus transfers are realised using the replication operator and executing two copies of the sequential master in parallel. The address and data bus outputs of the sequential masters are arbitrated such that address and data phases on the bus are pipelined. Burst transfers are added to either a sequential or pipelined master by the sequential composition of a control flow instance and the master. The idea is to generate a sequence of transfers with an incrementing address counter.

We were able to model two crucial and widely-used features of bus protocols with only two framework components.

B. PCI Express

The PCI Express case study was more extensive than the initial AHB study. Our goal was to investigate the approach using an industrial-sized protocol which implements a series of features that are used in modern multi- and many-core communication

architectures. The protocol is specified using three stack layers, each implementing an abstraction layer: a transaction layer (TL), a data-link layer (DLL), and a physical layer. Our case study considered the two upper layers with a focus on the TL. An overview of the case study is shown in Fig. 1(c).

As an example of reducing a complex feature to a small set of framework operations, we outline the receiver part of the flow control system. Using simple buffer, a small instance of the multiplex/arbitrate composition, and finally an instance of the replication operator, we are able to construct a receiver model supporting flow control. It is a very nice example of constructing a complex system incrementally starting with a very simple one in structured way, applying the generic correctness results to verify each modelling step.

With the small set of composition operators and basic building blocks presented here, we were able to model almost all of the features mentioned in Fig. 1(c). Only for the ACKNAK protocol we used an ad-hoc modelling approach. But we were still able to encapsulate the feature in a transformation and use sequential composition to integrate it in the DLL model.

VII. Conclusion

We have formalized a framework for the modelling and verification of on-chip communication protocols in the Isabelle/HOL theorem prover. Our modelling approach is based on abstract state machines and we specify initial, basic building blocks. Using composition rules, especially the replication and multiplex/arbitrate operators, we can incrementally compose more complex systems. In previous contributions, we have shown how to apply these principles to model protocol features independently. With a small set of basic building blocks and composition rules, we were able to model a broad variety of protocol features. The framework is flexible enough so that it is applicable to two very different protocol types covered by the case studies.

We managed to reduce the verification effort by spreading it over the modelling process and integrating automated tools into the methodology such as the NuSMV model checker. We also prove generic correctness properties for the composition rules so that we can restrict the verification to discharging local assumptions when we use the framework to model concrete feature extensions. By reducing the manual theorem proving parts compared to out previous case studies, we tackled frequent concerns regarding the usability of our approach. Future work includes an even further reduction of the theorem proving, ideally to a point where automated tools are sufficient to apply the framework to a specific case study.

Future work in the short-term focuses around two points: integrating even more automatic tools and linking the models to hardware descriptions. For the former, we plan on investigating the integration of an SMT Solver in the approach, for example the current Isabelle version provides a link to the Z3 SMT solver. Also the integration of a SAT solver will be considered. To link actual hardware descriptions to the models, we aim at further integrating the IHaVeIt interface in Isabelle 2009 as it also provides generators for Verilog and VHDL code.

Our larger-scale, long-term aim is to provide a feasible approach to modelling and verification of complex on-chip protocols as an alternative to monolithic, ad-hoc modelling and post-hoc verification. We aim at increasing the efficiency of the model building process, and providing a final model that has significant merits against ad-hoc models. The merits of our models is that they are already functionally verified and independent from the actual implementation or design architecture which can act as a longer-term reference model.

VIII. Acknowledgements

This work is funded by the Engineering and Physical Sciences Research Council and a donation from Intel Corporation.

References

[1] P. Böhm, "Incremental Modelling and Verification of the PCI Express Transaction Layer," in *MEMOCODE'09*. IEEE, 2009, pp. 36–45.
[2] P. Böhm and T. Melham, "A Refinement Approach to Design and Verification of On-Chip Communication Protocols," in *FMCAD'08*. IEEE, 2008, pp. 136–143.
[3] *AMBA Specification Revision 2.0*, ARM, 1999.
[4] PCI-SIG, *PCI Express Base Specification Revision 2.0*, December 2006.
[5] T. Nipkow, L. C. Paulson, and M. Wenzel, *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. LNCS vol. 2283, Springer, 2002.
[6] S. Tverdyshev, "Formal verification of gate-level computer systems," Ph.D. dissertation, Saarland University, Computer Science Department, 2009. [Online]. Available: http://www-wjp.cs.uni-saarland.de/publikationen/Tv09.pdf
[7] R. P. Kurshan, *Computer-Aided Verification of Coordinating Processes: the automata-theoretic approach*. Princeton University Press, 1994.
[8] T. Nipkow and K. Slind, "I/O automata in Isabelle/HOL," in *TYPES'94*, ser. LNCS, vol. 996. Springer, 1995, pp. 101–119.
[9] L. Helmink, P. A. Sellink, M., and W. Vaandrager, F., "Proof-checking a data link protocol," in *TYPES'93*. Springer, 1994, pp. 127–165.
[10] V. Luchangco, E. Söylemez, S. J. Garland, and N. A. Lynch, "Verifying timing properties of concurrent algorithms," in *Formal Description Techniques VII*. Chapman & Hall, Ltd., 1995, pp. 259–273.
[11] S. M. Suhaib, D. A. Mathaikutty, S. K. Shukla, and D. Berner, "Xfm: An incremental methodology for developing formal models," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 10, no. 4, pp. 589–609, 2005.
[12] J. R. Abrial, M. K. O. Lee, D. S. Neilson, P. N. Scharbach, and I. H. Sørensen, "The B-method," in *VDM'91*, ser. LNCS, vol. 552. Springer, 1991.
[13] J.-R. Abrial, D. Cansell, and D. Méry, "A Mechanically Proved and Incremental Development of IEEE 1394 Tree Identify Protocol," *Formal Aspects of Computing*, vol. 14, no. 3, pp. 215–227, April 2003.
[14] J. Schmaltz and D. Borrione, "A functional formalization of on chip communications," *Form. Asp. Comp.*, vol. 20, no. 3, pp. 241–258, 2008.
[15] X. Chen, S. M. German, and G. Gopalakrishnan, "Transaction Based Modeling and Verification of Hardware Protocols," in *FMCAD'07*. IEEE, 2007, pp. 53–61.
[16] F. Müffke, "A Better Way to Design Communication Protocols," Ph.D. dissertation, University of Bristol, May 2004. [Online]. Available: http://www.cs.bris.ac.uk/Publications/Papers/2000199.pdf
[17] S. Finn and M. Fourman, "The LAMBDA Logic. Abstract Hardware Limited, September 1993. In LAMBDA 4.3 Reference Manuals." 1993.
[18] M. Abadi and L. Lamport, "The existence of refinement mappings," *Theor. Comput. Sci.*, vol. 82, no. 2, pp. 253–284, 1991.
[19] K. L. McMillan, "A compositional rule for hardware design refinement," in *CAV '97*. Springer, 1997, pp. 24–35.
[20] J. Schmaltz, "A formal model of clock domain crossing and automated verification of time-triggered hardware," in *FMCAD '07*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 223–230.
[21] E. Alkassar, P. Böhm, and S. Knapp, "Correctness of a fault-tolerant real-time scheduler and its hardware implementation," in *MEMOCODE'08*. IEEE Computer Society, June 2008, pp. 175–186.
[22] J. Bhadra, M. S. Abadir, L.-C. Wang, and S. Ray, "A survey of hybrid techniques for functional verification," *IEEE Des. Test*, vol. 24, no. 2, pp. 112–122, 2007.

Modular Specification and Verification of Interprocess Communication

Eyad Alkassar*, Ernie Cohen†, Mark Hillebrand†, and Hristo Pentchev*
*Saarland University, Saarbrücken, Germany
{eyad,pentchev}@wjpserver.cs.uni-saarland.de
†European Microsoft Innovation Center (EMIC GmbH), Aachen, Germany
{ecohen,mahilleb}@microsoft.com

***Abstract*—The usual goal in implementing IPC is to make a cross-thread procedure call look like a local procedure call. However, formal specifications of IPC typically talk only about data transfer, forcing IPC clients to use additional global invariants to recover the sequential function call semantics. We propose a more powerful specification in which IPC clients exchange knowledge and permissions in addition to data. The resulting specification is polymorphic in the specification of the service provided, yet allows a client to use IPC without additional global invariants. We verify our approach using VCC, an automatic verifier for (suitably annotated) concurrent C code, and demonstrate its expressiveness by applying it to the verification of a multiprocessor flush algorithm.**

I. Introduction

Procedural abstraction—the ability for the caller of a procedure to abstract a procedure call to a relation between its pre- and poststates—is one of the most important structuring mechanisms in all of programming methodology. The central role of procedural abstraction is reflected in the fact that it is built into not only all modern imperative languages, but also into most program logics and verifiers for such languages. However, in a concurrent or distributed system, procedure calls between threads are provided only indirectly through system calls or libraries for interprocess communication (IPC). This begs the question of how such libraries might be specified so as to provide procedural abstraction to their clients, and how such libraries can be verified to meet these specifications. In this paper, we consider the problem in the context of multithreaded C software, with threads executing in a single shared address space.

To see why this problem is nontrivial, consider a simple implementation where all data is passed through shared memory, and where each ordered pair of caller-callee threads share a mailbox at a fixed address. The caller makes a call by creating a suitable call record in memory (including identification of which procedure to execute, values of the call parameters, and a place to put the return value), writes the address of this record into the mailbox going to the callee, and calls an IPC function to signal the callee. The callee, on receiving the signal, reads the address of the call record from the mailbox, reads the memory to get the call parameters, executes the call, and signals the caller. Note that all memory accesses are sequential; the only synchronization necessary is provided by the IPC layer.

Now, it's not hard to see that the IPC layer is providing functionality similar to a split binary semaphore, with the call records playing the role of the lock-protected data, and the data invariant given by the semantics of the various procedure calls. Thus, a specification for semaphores would provide a natural starting point for a specification for IPC. However, in classical program verification, semaphore operations are specified by their effect on global ghost state; making use of such a specification requires additional global invariants to capture how the clients use each semaphore. Using this kind of specification for IPC would force the client of the remote procedure call to use these global invariants on both call and return. This fails to faithfully capture the local character of procedural abstraction.

A second possibility is to encapsulate these global invariants inside the IPC layer. For example, the IPC specification could be strengthened to include the pre- and post-conditions of the procedure call. This is the sort of specification one would find in a local logic, such as concurrent separation logic (CSL). But such logics typically cannot specify generic semaphores, because the semaphore code has to be polymorphic in both the encapsulated data and the data invariant.[1] Similarly, taking this approach with the IPC code requires the specification of the code to be polymorphic in the specification of the material being passed between caller and callee.

We propose a different approach to specifying and verifying IPC that allows the recovery of procedural abstraction. The key idea is that IPC routines transfer ghost objects that own the call records, and whose invariants capture the pre- and post-conditions of the procedures. (This is possible because we allow object invariants to mention arbitrary parts of the state, with a semantic consistency check that guarantees the stability of each object invariant while the object exists.) The "contract" between caller and callee is expressed in ghost data as a binary relation between call objects and return objects. The IPC routines can transfer ownership of the ghost objects without knowing their types, making the transport suitably polymorphic. This ghost scaffolding, combined with the (fixed) specification of the IPC routines, yields for the client the sequential procedural abstraction provided by the

[1] A recent proposal [7] extends CSL with a facility similar to VCC ghost objects, which should allow to do constructions similar to the one in this paper.

application function.

We have used this approach to specify and verify an IPC layer, and illustrate its application to a multiprocessor flush algorithm. The implementation was derived from a real verification target, the inter-processor interrupt (IPI) routines of Microsoft's Hyper-V™ hypervisor. All specification and proofs given here have been carried out using VCC, an automatic verifier for (suitably annotated) concurrent C.[2] VCC provides the first-class ghost objects needed to carry out our approach, while allowing the approach to be applied to real implementation code.

A. Related Work

The correctness of IPC has been tackled in the context of microkernel verification. For example, the IPC implementations of the seL4 [9] and VAMOS [6] kernels have been formally verified against their respective ABIs. These projects focused on implementation correctness rather than client usability, and specify solely data transfer.

The application of VAMOS IPC provided in [1] shows the shortcomings of this approach: there, correctness statements of the remote procedure calling (RPC) library argue simultaneously on the sender/receiver pair instead of using thread-local reasoning.

A number of formalisms were applied to specification and verification of interprocess communication in the context of the RPC-Memory Specification Case Study [3]. None of the submitted solutions attempted to provide general-purpose sequential procedural abstraction.

In [8] a verification framework for threads and interrupt handlers based on CSL is described. This work is similar to ours, as both the implementation of (thread-switching) primitives and clients using them, are verified. When threads switch, ownership is transfered and some global invariant on shared data is checked. In contrast to our work the client code is interactively verified in two different logics, whereas in our approach both are verified seamlessly and automatically in the same proof context.

B. Overview

The paper is structured as follows. In Section II we outline main VCC concepts. In Section III we present an IPC algorithm with polymorphic specification, which we use in Section IV to implement and verify a TLB flush protocol. In Section V we extend these results to multiple senders and receivers as required for the implementation of interprocessor interrupt (IPI) protocols used in real, multiprocessor hypervisors. In Section VI we conclude.

II. VCC Overview

In this section, we give a brief overview of VCC. More detailed information and references can be found through the VCC homepage [10]. To understand the VCC view of the world, it is helpful to think of verification in a pure object model, which is used to interpret the C memory state. Thus, we first describe VCC concepts in terms of objects, and then describe how this is applied to C.

Table I shows a syntax overview of the constructs required for our IPC design presented in the following sections.

A. Objects

In VCC, the state is partitioned into a collection of objects, each with a number of fields. Objects have addresses, so fields can be (typed) object references. Each object has a 2-state invariant, which is expected to hold over any state transition. These invariants can mention arbitrary parts of the state. However, when checking an atomic update to the state, instead of checking the invariants of *all* objects we want to check the invariants of only the updated objects. We justify this by checking, for each object type, that starting from a state in which all object invariants hold, a transition that breaks the invariant of an object of that type must break the invariant of some modified object (not necessarily of that type); such invariants are said to be *admissible*. (In addition, we have to check that stuttering from the poststate of a transition preserves all invariants of all objects.) Both requirements are checked for each object type when the type is defined; this check makes use of type definitions, but not of program code. Details can be found in [5].

Within an object invariant, the (2-state) invariant of other objects can be referred to.[3] A commonly used form of this is *approval*: we say that an object *o approves* changes to another object's field $p \rightarrow f$, if p has a 2-state invariant stating that $p \rightarrow f$ stays unchanged or the invariant of o holds. In other words, any change to $p \rightarrow f$ requires checking the invariant of o. Approval is used to express object dependencies or build object hierarchies, e.g., VCC's ownership model.

Since it is unrealistic to expect objects to satisfy interesting invariants always (e.g., before initialization or during destruction), we add to each object a Boolean ghost field ***closed*** indicating whether the object is in a "valid" state. Implicitly, the 2-state invariants declared with an object type are meant to hold only across state transitions in which the object is closed in the prestate and/or the poststate. Each object field is classified as either sequential or volatile. Volatile fields can change while the object is closed, while sequential fields cannot. (That is, for each sequential field, there is an implicit object invariant that says that the field does not change while the object is closed.)

Each object has an *owner*, which is itself an object. It is a global system invariant that open objects are owned only by threads, which are regular objects. In the context of a thread t, a closed object owned by t is said to be *wrapped*, while an open object owned by t is said to be *mutable*. Threads themselves have invariants; essentially, the invariant of a thread t says that any transition that does not change the state of t leaves unchanged (i) the set of objects owned by t, (ii) the fields of its mutable objects, (iii) the sequential fields of its wrapped

[2]Sources are available at http://www.verisoftxt.de/PublicationPage.html.

[3]This implicitly makes object invariants recursive; to guarantee that all object invariants have a consistent interpretation, we allow such references to occur only with positive polarity.

VCC Keyword	Description
Basics	
this	self-reference to object (used in type invariants)
invariant(*p*)	type invariant with property *p*
old(*e*)	evaluates *e* in prestate (of function, loop, or 2-state invariant)
closed(*o*)	object *o* closed; invariants of *o* guaranteed to hold
inv(*o*)	evaluates to (2-state) invariant of *o*
approves($o, f_1, \ldots, f_n$)	changes of fields $f_1,...,f_n$ require check of *o*'s invariant: $(\bigvee_i \textbf{\textit{old}}(f_i) \neq f_i) \implies \textbf{\textit{inv}}(o)$
atomic($o_1, \ldots, o_n$){*s*;}	marks atomic execution of *s*; updates only volatile fields of $o_1, \ldots, o_n$
ref_cnt(*o*)	number of claims that depend on *o*
claims(*c*,*p*)	invariant of claim *c* implies *p*
spec(. . .)	wraps ghost code and parameters
∀(*T t*;. . .)	universal quantification
∃(*T t*;. . .)	existential quantification

(a) Basic Keywords

VCC Keyword	Description
Ownership	
owner(*o*)	owner of object *o*
owns(*o*)	set of objects owned by object *o*
wrapped(*o*)	*o* closed and owned by current thread
mutable(*o*)	*o* not closed and owned by current thread
set_closed_owner(*o*,*o'*)	sets owner of *o* to *o'* and extends ownership of *o'* by *o*
giveup_closed_owner(*o*,*o'*)	make *o* wrapped and remove it from the ownership of *o'*
Function Contracts	
requires(*p*)	precondition
ensures(*p*)	postcondition
writes($o_1, \ldots, o_n$)	function writes to objects o_i
Spec Types	
mathint	mathematical integers
claim_t	claim pointers
T2 map[T1]	map from *T1* to *T2*
λ(T1 t1;. . .)	lambda expression over *t1*

(b) Ownership, Function Contracts, Spec Types

TABLE I: VCC Keywords

objects, and (iv) the (volatile) fields of closed objects approved by *t* (we call such fields *thread-approved*). Each object *o* implicitly contains an invariant that says that its owner (as well as its owner in the prestate) approves any change to the field *o*→***closed*** and to the set of objects owned by *o*.[4]

The *sequential domain* of a closed object is the smallest set of object fields that includes the sequential fields of the object and, if its set of owned objects is declared as nonvolatile, the elements of the sequential domains of the objects that it owns. Intuitively, the values of fields in the sequential domain of *o* are guaranteed not to change as long as *o* remains closed.

Within program code, each memory access is classified as ordinary or atomic. An ordinary write is allowed only to fields of mutable objects; an ordinary read is allowed only to fields of mutable objects, to nonvolatile fields of in the sequential domain of a wrapped object, and to volatile fields of objects that are closed if changes to the field are approved by the reading thread. In an atomic operation, all of the objects accessed have to be known to be mutable or closed (i.e., not open and owned by some other thread), only volatile fields of closed objects may be written, and the update must be shown to preserve the invariants of all updated objects. Before each atomic operation, VCC simulates running other threads by forgetting everything it knows about the state outside of its sequential domain; standard reduction techniques [4] can be used to show that we can soundly ignore scheduler boundaries at other locations.

B. Ghost Objects

VCC verifications make heavy use of *ghost* data and code (surrounded by ***spec()***), used for reasoning about the program but omitted from concrete implementation. VCC provides ghost objects, ghost fields of structured data types, local ghost variables, ghost function parameters, and ghost code. C data types are limited to those that can be implemented with bit strings of fixed length, but ghost data can use additional mathematical data types, e.g., mathematical integers (***mathint***) and maps. VCC checks that information cannot flow from ghost data or code to non-ghost state, and that all ghost code terminates; these checks guarantee that program execution including ghost code simulates the program with the ghost data and ghost code removed.

C. Claims

A ghost object can be used as a first-class chunk of *knowledge* about the state, because the invariant of the object is guaranteed to hold as long as the object is closed. In particular, the owner of the object does not have to worry about the object being opened by the actions of others, so it can make use of the object invariant whenever it needs it. Being a first-class object, the chunk can be stored in data structures, passed in and out of functions, transfered from thread to thread, etc. Because they are so useful, VCC provides syntactic support for these chunks of knowledge, in the form of *claims*. Claims are similar to counting read permissions in separation logic [2], but are first-class objects; this allows claims to approve changes, be claimed, or even claim things about themselves.

Typically, a claim depends on certain other objects being closed; it is said to "claim" these objects. Since objects are usually designed to be opened up eventually, these "claimed" objects must be prevented from opening up as long as the claim is closed. Concretely, this can be implemented in various ways, the most obvious being for the dependee to track the count ***ref_cnt***(*o*) of claims that claim *o*, and allowing *o* to be opened only when ***ref_cnt***(*o*) is zero, cf. [5]. In constructing a claim, the user provides the set of claimed objects and invariant of the claim; VCC checks that this invariant holds and is

[4]By default, the set of objects owned by *o* is nonvolatile, and so cannot change while *o* is closed. This can be overridden by declaring ***vcc***(*volatile_owns*) in the type definition of *o*.

preserved by transitions under the assumption that the claimed objects are closed (this check corresponds to the admissibility check if the claim was declared with an explicit type). Any predicate implied by this invariant is said to be "claimed" by the claim; this allows a client needing a claim guaranteeing a particular fact to use any claim that claims this fact (without having to know the type of the claim); to make this convenient, VCC gives all claims the same type (***claim_t***); we can think of an additional "subtype" field as indicating the precise invariant.

D. Function Contracts and Framing

Verification in VCC is *function-modular*; when reasoning about a function call, VCC uses the specification of the function, rather than the function body. A function specification consists of preconditions (of the form ***requires***(*p*)), postconditions (of the form ***ensures***(*p*), where *p* is a 2-state predicate, the prestate referring to the state on function entry), and writes clauses (of the form ***writes***(*o*), where *o* is an object reference or a set of object references). VCC generates appropriate verification conditions to make sure that the writes clauses are not violated.

E. Binding to C

The discussion above assumed that we are in a world of unaliased objects. To deal with the real C memory state, VCC maintains in ghost state a global variable called the *typestate* that keeps track of where the "real" objects are; these objects correspond to instances of C aggregate types (structs and unions). (Variables of primitive types that are not fields of such objects are put into artificial ghost objects or ghost arrays.) There are system invariants that (i) each memory cell is part of exactly one object in the typestate, (ii) if a struct is in the typestate, then each of its subobjects (e.g., fields of aggregate type) are in the typestate, and (iii) if a union is in the typestate, then exactly one of its subobjects is in the typestate. These invariants guarantee that if two objects overlap, then they are either identical or one object is a descendant of the other in the object hierarchy. When an object reference is used (other than as the target of an assignment), it is asserted that reference points to an object in the typestate. Thus, the typestate gets rid of all of the "uninteresting" aliasing (like objects of the same type partially overlapping).

III. A Polymorphic Specification of IPC

In this section we verify the implementation of a simple communication algorithm between two threads. The threads exchange data over a shared but sequentially accessed message box to which they synchronize access with a Boolean volatile notification flag. To verify the implementation's memory safety, an ownership discipline must be realized in which the ownership of the message box is transferred back and forth between the two threads. We extend this pattern by passing claims between the two threads, which we store in the message box. The properties of these claims can be configured by the clients, thus providing the desired polymorphic procedure call semantics for IPC.

```
spec(typedef struct vcc(record) InOut {
  unsigned val; mathint gval; claim_t cl; } InOut;)

typedef struct MsgBox {
  unsigned in, out;
  spec(InOut input, output;)
  invariant(input.val≡ in ∧ output.val≡ out)
  invariant(input.cl≠ output.cl ∧
    input.cl ∈ owns(this) ∧ ref_cnt(input.cl)≡ 0 ∧
    output.cl ∈ owns(this) ∧ ref_cnt(output.cl)≡ 0) } MsgBox;
```

Listing 1: Message Box Type with Invariants

There are various ways to structure annotations and, in particular, the definitions of ghost objects and invariants. At their core, all of these share information via volatile fields, pass on knowledge via claims or object invariants, and make use of thread-approved state for the two communication partners. We chose here a way that is easy to present but also extends cleanly to multiple senders and receivers (cf. Section V).

A. Scenario

We consider the scenario of two threads (0 and 1) exchanging data over a shared message box (of type *MsgBox*). The message box contains two fields (*in* and *out*) which are used for sending a request to the other thread and receiving back a response, respectively. The fields of the message box are nonvolatile and accessed sequentially. The message box is contained in another structure (of type *Mgr*), which additionally holds a volatile Boolean notification flag *n* used to synchronize access to the message box. Given the canonical conversion of Booleans to integers (where *0* and *1* are mapped to ***false*** and ***true***, respectively), this flag identifies the currently acting thread. If set, thread 1 is acting, i.e., preparing a response for thread 0 and posting a new request, and thread 0 may not access the message box. Otherwise, thread 0 is acting and thread 1 may not access the message box. Thread 0 may not clear the flag, and thread 1 may not set it.

The implementation has two functions. Both take a *Mgr* pointer and a thread identifier *a*. The function *snd()* is meant to be called by thread *a* when the notification flag equals *a*. It negates the notification flag, thus sending the response and a new request contained in the message box at that time to the other thread. The function *rcv()* waits in a busy loop until the notification flag equals *a* again, thus receiving the other thread's response (to a preceding *snd()* call) and a new request.

B. Message Box

Listing 1 shows the annotated definition of the message box type. As outlined above, we want to generalize information exchange to beyond the mere transferral of data (the fields *in* and *out* in the message box). We therefore define an abstract I/O type (*InOut*) that carries a ghost value *gval* of unbounded integer type, and a claim pointer *cl* in addition to the implementation data value *val* being transmitted.

An abstract input and output each are an invariant stored in ghost fields of the message box. We maintain an invariant that the input's and output's *val* fields match their implementation

```
spec(typedef struct vcc(volatile_owns) Actor {
  struct Mgr *mgr;
  volatile bool w;
  volatile InOut l_input, r_input;
  invariant(closed(this) ∨ ¬closed(mgr))
  invariant(approves(owner(this), w, l_input, r_input))
  invariant(approves(mgr, owns(this), w, l_input, r_input)) } Actor;)
```

Listing 2: Actor Type and Invariants

counterpart. We also require that the claims pointed to by the input's and output's *cl* fields do not alias and are owned by the message box with a zero reference count.

The latter fact is particularly important. Whoever owns the message box also controls the contained claims, and may make use of the knowledge / property they hold or destroy them. The main functionality of the verified algorithm is thus the transferal of ownership of the message box between the two threads, making sure that the contained data has the desired properties, as instantiated by the client.

C. Actors

The *Actor* type keeps track of the protocol state of a protocol participant. Listing 2 shows the annotated definition of this type. The actor has a nonvolatile pointer *mgr* to the manager, which will hold all protocol invariants. For admissibility reasons, the actor must promise to stay closed longer than *mgr*. All others fields are volatile and may be atomically updated while the actor remains closed. Such updates, however, must be approved by two parties: the manager *mgr*, which checks all the protocol invariants, and the owner of the actor, which is one of the communicating threads and exclusive writer of the fields. The actor is also used as an intermediate owner of the message box during ownership transferral. For this purpose, its owns set is also declared volatile as well as approved by *mgr* but *not* thread-approved, to enable foreign updates by other threads.

The three regular fields of the actor are used as follows. The wait flag *w* is active when the thread owning the actor is waiting for a response from the other thread. The fields *l_input* and *r_input* buffer (abstract) local and remote inputs, i.e., input to the last request sent to or received from the other thread (or, in other words: the evaluation of the call parameters from the caller's and callee's perspective, respectively). In contrast to the *input* fields of the message box itself, which may be opened and updated sequentially by the owning thread, these buffers can be admissibly referred to all the time and used in the protocol invariants.

D. Manager

Listing 3 shows the annotated declaration of the *Mgr* type. In addition to the implementation fields, we also add some ghost components for the verification: the maps *InP* and *OutP* encoding pre- and postconditions for the message exchange, and a two-element array *A* of actors.

The predicates are declared nonvolatile, which allows clients to deduce that they remain unchanged as long as the manager

```
typedef struct Mgr {
  volatile bool n;
  MsgBox msgBox;
  spec(Actor A[2];
    bool InP[bool][InOut];
    bool OutP[bool][InOut][InOut];)
  invariant(∀(unsigned a; a < 2 ⟹ closed(&A[a]) ∧ A[a].mgr≡ this))
  invariant(A[¬n].w)
  invariant(A[n].w
    ? &msgBox ∈ owns(&A[n]) ∧ OutP[n][A[n].l_input][msgBox.output]
      ∧
      A[¬n].l_input≡ msgBox.input ∧ InP[n][msgBox.input]
    : A[n].r_input≡ A[¬n].l_input) } Mgr;
```

Listing 3: Manager Type and Invariants

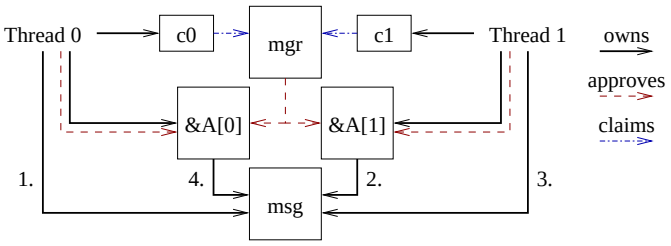


Fig. 1: Object Structure and Ownership Transfer

object is closed. They take a Boolean parameter identifying the actor and one resp. two abstract input-output values. The intention is that *InP[a][i]* is true iff *i* is a valid request for thread *a* (i.e., if the request meets the precondition of the service), and *OutP[a][i][o]* is true iff *o* is a valid response to a (valid) request *i* made by thread *a* (i.e., if the response meets the postcondition corresponding to the call). The IPI transport code is polymorphic with respect to these predicates, the concrete definition of which can be provided by the client at initialization.

We now describe the manager's invariants. As described above, each protocol partner *a* owns its corresponding actor *&A[a]*. The first invariant states that both actors remain closed and point back to the manager, which (in combination with the actor's approval invariants) allows us to admissibly talk about the actors in invariants here.

The remaining invariants define the protocol behavior. For an overview, refer to Fig. 1 depicting object structure and a protocol run starting from thread 0. In addition to the objects already introduced, each (client) thread i owns a claim ci that guarantees the manager structure to be closed. In phase 1, thread 0 owns the message box and may prepare its response and new request. In phase 2, ownership of the message box has passed from thread 0 to the actor of thread 1, waiting to be processed. Phases 3 and 4 are symmetrical: in phase 3 thread 1 prepares its response, which is then waiting to be processed in phase 4.

In addition to ownership, the protocol invariants restrict values for the actor fields. The second invariant states the non-acting thread, identified by the negated notification flag, must be waiting, i.e., have the wait flag of its actor set.

The third invariant refers to the acting thread, given by the

```
void snd(struct Mgr *mgr, bool a spec(claim_t c))
  requires(wrapped(c))
  requires(claims(c,closed(mgr)))
  requires(wrapped(&mgr→A[a]))
  ensures(wrapped(&mgr→A[a]))
  requires(¬mgr→A[a].w)
  requires(wrapped(&mgr→msgBox))
  requires(mgr→OutP[¬a][mgr→A[a].r_input][mgr→msgBox.output])
  requires(mgr→InP[¬a][mgr→msgBox.input])
  writes(&mgr→msgBox,&mgr→A[a])
  ensures(mgr→A[a].l_input≡ old(mgr→msgBox.input))
  ensures(mgr→A[a].w)
{
  atomic (c, mgr, &mgr→A[0], &mgr→A[1]) {
    assert(¬mgr→A[a].w ∧ mgr→A[¬a].w ∧ mgr→n≡ a);
    mgr→n = ¬a;
    spec(mgr→A[a].l_input = mgr→msgBox.input;
      mgr→A[a].w = true;
      set_closed_owner(&mgr→msgBox, &mgr→A[¬a]);
      bump_vv(&mgr→A[a]); /* technicality */
    )
  }
}
```

Listing 4: Send function with contract

```
void rcv(struct Mgr *mgr, bool a spec(claim_t c))
  requires(wrapped(c))
  requires(claims(c,closed(mgr)))
  requires(wrapped(&mgr→A[a]))
  ensures(wrapped(&mgr→A[a]))
  requires(mgr→A[a].w)
  writes(&mgr→A[a])
  ensures(¬mgr→A[a].w)
  ensures(wrapped(&mgr→msgBox))
  ensures(mgr→OutP[a][old(mgr→A[a].l_input))][mgr→msgBox.output])
  ensures(mgr→A[a].r_input≡ mgr→msgBox.input)
  ensures(mgr→InP[a][mgr→msgBox.input])
{
  unsigned tmp;
  do
    invariant(mgr→A[a].w)
    invariant(wrapped(&mgr→A[a]))
    invariant(mgr→A[a].l_input≡ old(mgr→A[a].l_input))
    atomic (c, mgr, &mgr→A[0], &mgr→A[1]) {
      tmp = mgr→n;
      spec(if (tmp≡ a) {
        mgr→A[a].r_input = mgr→A[¬a].l_input;
        mgr→A[a].w = false;
        giveup_closed_owner(&mgr→msgBox, &mgr→A[a]);
        bump_vv(&mgr→A[a]); /* technicality */
      })
    }
  while (tmp≠ a);
}
```

Listing 5: Receive function with contract

notification flag. The fact that the acting thread is waiting indicates that the message box is still waiting to be processed by the acting thread. It holds a response to the acting thread's last request in the *output* field and a new request in the *input* field. In the corresponding invariant we state that (i) the message box is owned by the current actor, (ii) its output is valid with respect to the acting thread's last / locally-stored request, and (iii) the new input equals the local input buffer of the other thread and is valid for the acting thread. If the acting thread is not waiting, we require local and remote input buffers of the current and non-current actors, respectively, to match. Note that these input buffers are approved by the acting and non-acting threads, respectively. Thus, this condition states that request inputs may not be changed while the request has not yet been processed.

E. Operations

The (annotated) implementation and the contracts for the send and receive function are given in Listings 4 and 5. Both functions take a manager pointer *mgr*, an actor identifier *a*, and a claim *c* supplied as a ghost parameter stating that the manager is closed. They maintain that the identified actor is wrapped. To send to the other thread, the current thread's actor must be flagged as non-waiting, the message box must be wrapped and hold valid outputs and inputs to the other thread, just as we have seen in the manager invariant for the acting thread. Afterwards, the message box is unknown to be wrapped (the ***writes*** clause on *&mgr→msgBox* destroys that knowledge), but the input sent to the other thread is buffered in the local input field of the actor (and the current thread's actor is flagged as waiting).

Given a waiting actor, the receive function is guaranteed to return a wrapped message box, that contains a valid response for the old local request and a new valid request.

As a verification example consider the *snd()* function from Listing 4. VCC automatically verifies that its implementation fulfills the contract. The code consists of a single atomic update on the actors and the manager (where the closedness of the manager and the foreign actor is guaranteed by the claim *c*). The precondition on the wait flag, its thread-approval, and the manager's invariant allow to derive that the current thread is still not waiting, the other thread is waiting, and the notification flag equals *a* just before the atomic operation.[5] Also, the message box, which is in the sequential domain of the thread, must still be wrapped and continues to satisfy the communication preconditions. The notification flag is then flipped (changing the 'acting' thread) and the ghost updates ensure that the atomic update satisfies the manager's invariant (e.g., by transferring ownership of the message box from the current thread to the other thread's actor).

The verification of the *rcv()* function is similar. In addition to the atomic statement, appropriate invariants have to specified for the loop that polls on the notification flag.

IV. TLB Flush Example

We implement and verify a protocol for flushing translation look-aside buffers (TLBs) based on the communication algorithm from the previous section, demonstrating the expressiveness of its polymorphic specification.

TLBs are per-processor hardware caches for translations from virtual to physical addresses. These translations are defined by page tables stored in memory, which are asynchronously and non-atomically gathered by the TLBs (requiring multiple reads and writes to traverse the page tables). Since

[5]The assertion is for illustration only; VCC deduces it automatically.

```
spec(typedef struct Tlb {
    volatile mathint invalid, current;
    invariant(invalid ≤ current ∧
      old(invalid) ≤ invalid ∧ old(current) ≤ current)
} Tlb;)
```

Listing 6: TLB Model

```
typedef struct FlushMgr {
  struct Mgr mgr;
  spec(struct Tlb tlb;)
  invariant(&tlb ∈ owns(&mgr)))
  invariant(mgr.InP≡ λ(bool a; InOut i;
    a ⟹ claims(i.cl, i.gval ≤ tlb.current)))
  invariant(mgr.OutP≡ λ(bool a; InOut i, o;
    a ∨ claims(o.cl, i.gval ≤ tlb.invalid)))
} FlushMgr;
```

Listing 7: Flush Manager Type and Annotations

translations are not automatically flushed in response to edits to page tables, operating systems must implement procedures to initiate such flushes on their own.

We think of page-table reads being marked with unique (increasing) identifiers and model each TLB as an object with two volatile counters,[6] cf. Listing 6. The *current* counter increases as the TLB gathers new translations. The *invalid* counter is a watermark for invalidated translations and is bumped (i.e., copied from the *current* field) when the associated processor issues a TLB flush.

Consider the scenario of two threads, the *caller* (thread 0) requesting the flush and the *callee* (thread 1) performing the flush. We implement this as follows: the caller sends a flush request by invoking the send primitive and subsequently polls for the answer by calling the receive primitive. On callee side, the thread polls via receive for new flush requests. When a flush request has been received, the callee issues a TLB flush operation, and signals back that the flush has been performed using the send primitive. After a completed flush operation, the flush client (e.g., the memory manager) wants to derive that the callee TLB's current *invalid* counter is larger or equal than the callee's *current* counter at the time of the flush operations.

We realize this scenario by embedding the IPC manager (and callee's TLB) into a *flush manager*, as shown in Listing 7. Apart from ownership, the invariants give meaning to the input and output predicates of the communication manager. The ghost value *i.gval* transmitted from the caller to the callee encodes which translations are meant to be flushed. For the callee ($a \equiv$ ***true***), the input predicate states that this value is less or equal than the *current* field of its TLB (since the callee could not possibly flush translation 'from the future', i.e., such a request could not be handled by the TLB flush semantics). For the caller ($a \equiv$ ***false***), the output predicate then states that the *invalid* field of the callee's TLB is greater or equal than the value, i.e., the requested flush has been performed. For the other cases the input and output predicates are trivially true.

[6]While this model is sufficiently detailed to express the semantics of (full) TLB flushes, extensions are needed for applications that go beyond that.

Based on this definition, the correctness of the functions *sendFlush()* and *receiveFlush()* at caller and callee side, respectively, can be proven. The main postcondition that is established by *sendFlush()* for the flush manager *fmgr* then is ***old***$(fmgr{\rightarrow}tlb.current) \leq fmgr{\rightarrow}tlb.invalid$.

V. Interprocessor Interrupts

Interprocessor interrupts (IPI) are used in multicore operating systems or hypervisors to implement different synchronization and communication protocols. Via IPIs a thread executing on one processor can trigger the execution of interrupt handlers (here: NMI handlers) on other processors. Using IPIs, a communication protocol can be implemented, in which a caller thread sends work requests to other processors, the callees. Such an IPI protocol is part of the Verisoft XT academic hypervisor, where it may be used for different work types, e.g., for TLB flushing. Thus a polymorphic specification is desirable.

By expanding the simple communication pattern introduced previously, we specified and verified the IPI protocol (and on top of it a TLB flushing protocol) for the academic hypervisor. There are several differences between the previous version of the algorithm and the IPI protocol:

- **More communication partners.** In the simple case we had a single sender and a single receiver. Now we have multiple communication partners, where one sender may invoke an IPC call on many receivers, and where each receiver may be invoked by many senders at the same time.
- **No receiver polling.** The callees in the IPI scenario do not poll for messages. Rather the caller invokes the callee by triggering an IPI. This is done by writing registers of the advanced programmable interrupt controller (APIC), which delivers the interrupts to other processors. In the work at hand we do not yet model this hardware device.
- **More concurrency.** In the new setting we have another source of concurrency, NMI handlers which may interrupt the execution of ordinary threads. Basically, the NMI handler code always acts as receiver or callee and the thread code as sender or caller.
- **Interlocked hardware operations.** Interlocked bit operations are required to atomically access bit vectors which may be written and read concurrently by many threads/ handlers.

A. Implementation

Since multiple senders can send requests to multiple receivers, we need a notification bit for each sender/receiver pair. This is implemented by introducing one notification mask per processor. Each bit of such a mask is associated with a specific sending processor. Thus, a sender signals a request by setting its bit in the receiver's notification mask. When finishing the work, the receiver clears that bit. Many senders and receivers can write the same notification mask in parallel, requiring the use of interlocked bit operations.

Similarly, we need one mailbox for each sender/receiver pair. Note, that for each processor pair we need two mailboxes, since both may send messages to each other simultaneously.

In the sending code a while-loop iterates over the set of intended receivers (encoded in a bit mask). In each iteration, first the mailbox is prepared, and then by using an interlocked OR-operation, atomically, the corresponding bit in the receiver mask is set to *1* and the mask is compared with *0*. If this check evaluates to true, an IPI for the receiving processor is triggered via the APIC. Otherwise, nothing has to be done, since some other sender already triggered the interrupt, and the handler has not returned yet.

In the receiving code (implemented as an NMI handler) a while-loop iterates on (possibly multiple) sender requests as long as the receiver's notification mask is not *0*. Once the work for one sender is done, the corresponding bit in the notification mask is cleared by an interlocked AND-operation.

B. Specification

The specification pattern of Section III can be straightforwardly applied to the IPI protocol. The number of ghost objects scales linearly with the number of processors. The structure and the invariants of message boxes (with their ghost fields encoding input/output claims) and actors introduced in the simple protocol can be reused almost identically in the new setting.

If n is the number of processors, $2 \cdot n$ actor objects are required, since each processor may act both as sender—when running thread-code—or as receiver—when running NMI handler code. Though executed on the same processor, both code portions are two logically different entities, possibly residing in different protocol states, and owning different sets of mailboxes. That is also how we deal with thread and NMI handler concurrency: each of the NMI handler and the thread code own (and thus approve) separate actors. Note that in the IPI case, a single actor may communicate with many other partners, requiring it to maintain protocol state (the wait flag, and the remote and local input fields) per processor. The invariants of the manager are similar to those from the simple protocol.

C. Multiprocessor TLB Flush

The TLB abstraction and specification is similar to the previous section, but with a separate TLB for each processor.

D. IPIs in Microsoft's Hyper-V™ Hypervisor

In the context of the Verisoft project we also studied the correctness of the IPI mechanism implemented in Microsoft's Hyper-V hypervisor. Though comparable in complexity to the IPI routine of the academic hypervisor, there are several differences:

- **Efficiency.** By introducing additional protocol variables sequential access to some of the shared data can be ensured, and thus fewer (costly) interlocked operations are required.
- **Lazy work.** The interrupt handler signals the receipt of the request and the accomplishing of the work separately. This allows for implementing less blocking caller code.

We have verified the implementation against a non-generic specification in VCC and are confident that this effort can be easily adapted to the generic specification used here.

VI. Conclusion

The verification presented here achieves the desired goal—it allows IPC clients to reason about IPCs like local procedure calls. As future work, the structure presented in Section III can be made modular even with respect to the set of functions provided via IPC. We can improve the structure slightly by changing the *Mgr* type; instead of the maps *InP* and *OutP*, the *Mgr* could hold a mapping of function tags to function objects, where each function object has its own *InP* and *OutP* maps. This would allow function objects to be reused in different managers, or even dynamically registered for IPC.

In principle, the technique presented here could also be applied to RPC, where the caller and callee execute in different address spaces. This requires translating the claims representing the pre- and post-conditions from one address space to the other. One possible way to achieve this effect would be to take the claim in the caller space, couple this to a second state in a way that captures the guarantees of the RPC, and existentially quantify away the caller space.

Acknowledgment

This work was partially funded by the German Federal Ministry of Education and Research (BMBF) in the Verisoft XT project under grant 01 IS 07 008. Work of the third author done while at DFKI GmbH, Saarbrücken, Germany.

References

[1] Eyad Alkassar, Sebastian Bogan, and Wolfang Paul. Proving the correctness of client/server software. *Sādhanā: Academy Proceedings in Engineering Sciences*, 34(1):145–191, 2009.

[2] Richard Bornat, Cristiano Calcagno, Peter W. O'Hearn, and Matthew J. Parkinson. Permission accounting in separation logic. In Jens Palsberg and Martín Abadi, editors, *POPL 2005*, pages 259–270. ACM, 2005.

[3] Manfred Broy, Stephan Merz, and Katharina Spies, editors. *Formal Systems Specification*, volume 1169 of *LNCS*. Springer, 1996.

[4] Ernie Cohen and Leslie Lamport. Reduction in TLA. In Davide Sangiorgi and Robert de Simone, editors, *CONCUR 1998*, number 1466 in LNCS, pages 317–331. Springer, 1998.

[5] Ernie Cohen, Michał Moskal, Wolfram Schulte, and Stephan Tobies. Local verification of global invariants in concurrent programs. In Tayssir Touili, Byron Cook, and Paul Jackson, editors, *CAV 2010*, volume 6174 of *LNCS*, pages 480–494. Springer, 2010.

[6] Matthias Daum, Jan Dörrenbächer, and Burkhart Wolff. Proving fairness and implementation correctness of a microkernel scheduler. *JAR*, 42(2–4):349–388, 2009.

[7] Thomas Dinsdale-Young, Mike Dodds, Philippa Gardner, Matthew J. Parkinson, and Viktor Vafeiadis. Concurrent abstract predicates. In Theo D'Hondt, editor, *ECOOP 2010*, volume 6183 of *LNCS*, pages 504–528. Springer, 2010.

[8] Xinyu Feng, Zhong Shao, Yu Guo, and Yuan Dong. Certifying low-level programs with hardware interrupts and preemptive threads. *JAR*, 42(2–4):301–347, 2009.

[9] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, et al. seL4: Formal verification of an OS kernel. In *SOSP 2009*, pages 207–220. ACM, 2009.

[10] Microsoft Corp. VCC: A C Verifier. http://vcc.codeplex.com/, 2009.

Large-scale Application of Formal Verification: From Fiction to Fact

Viresh Paruthi
IBM Systems and Technology Group, Austin TX, USA.
e-mail: *vparuthi@us.ibm.com*

***Abstract*—Formal verification has matured considerably as a verification discipline in the past couple of decades, becoming a mainstream technology in industrial design and verification methodologies and processes. In this paper we chronicle the evolution of formal verification at IBM from being a specialized side activity with a narrow focus, to achieving a broad-based usage as a core verification technology helping to significantly improve design and verification productivity. We showcase what is possible in the application of formal verification in a commercial/industrial setting by highlighting the success we had in leveraging the technology extensively on IBM's POWER7^{TM} microprocessor and systems. We touch upon the methodology and execution aspects of the unprecedented use of formal verification on the POWER7 program, and depict ways in which the technology positively impacted pre-silicon design quality and facilitated root causing of bug escapes to silicon. Furthermore, we outline where we see applied formal verification evolving towards at IBM, and the challenges thereof.**

I. Introduction

IBM has a rich history developing robust formal and semi-formal verification technologies, and applying those effectively to the verification of microprocessor designs and systems. Since its advent almost a decade and a half ago Formal Verification (FV), inclusive of functional formal verification and sequential equivalence checking, has evolved from being a specialized technology in the hands of experts, to a widely deployed technology with a broadened user base to include design and functional verification engineers.

Functional Formal Verification (FFV) at IBM dates back to the POWER3 (1996) microprocessor where it was applied on an experimental basis, followed by a larger and more defined effort on the POWER4 [13] program. On both of these projects the application was limited to small-sized logic partitions requiring the creation of intricate testbenches comprising complex "environmental assumptions". Dramatic improvements to the FFV toolset, and the debut of semi-formal technologies, allowed for increased application and leverage on subsequent programs such as POWER5 [21] and POWER6. The mode of application in all of these programs was similar with FFV being a standalone side activity driven by skilled formal verification engineers - albeit with scaling to bigger logic partitions, and greater portions of the chip logic subjected to FFV analysis. Sequential Equivalence Checking (SEC) technology [4] became available around the 2004 time frame and quickly became a huge productivity advantage. It facilitated proving non-functional design changes (e.g., timing, power) without the need to rerun (lengthy) regression buckets, and enabled key new methodologies (e.g., sequential synthesis, infer clock-gating opportunities).

The mandate coming into the POWER7 program, based on analysis of bugs on past projects, was to apply formal verification technology more extensively to improve pre-silicon design quality and minimize bug escapes into silicon. The result was a step function of integrated and broader usage resulting in the largest and most successful ever application of FV on any project at IBM. FV assumed a central role in the verification of large parts of the design culminating in flushing out hundreds of bugs, many of which would have been extremely difficult to find using traditional verification methods, and ensuring correctness of the logic by way of obtaining proofs. FV was exploited at all levels of the design hierarchy encompassing all areas of the chip. Such a widespread use of the technology has been enabled by IBM's suite of state-of-the-art formal and semi-formal verification tools, SixthSense [18] and RuleBase PE [5], [22], which are fully integrated into the methodology.

In this paper we describe large-scale application of functional formal and semi-formal verification and sequential equivalence checking with experiences from leveraging the technologies on the POWER7 microprocessor and systems. POWER7 [11] is a complex high-end eight-core processor chip with four-way Simultaneous Multi-Threading (SMT4) per core, scalability to 32 sockets, and an aggressive memory subsystem design. It implements a modular structure with heavy use of asynchronous interfaces, and new power-management and RAS (Reliability, Availability, and Serviceability) mechanisms across the chip and system.

FFV and SEC application on the project can be best summarized as a combination of an up-front defined methodology, and results from deploying the methodology during project execution. We start with a brief description of the verification methodology as it relates to FV in the next section. We then outline aspects of FFV and SEC application on POWER7 and the benefits realized in sections 3 through 6. We conclude with glimpses into our strategy for further leveraging FV to address future challenges.

II. Verification Methodology

The base of all verification disciplines inclusive of simulation, hardware accelerated simulation and formal and semi-formal verification is a cycle-based execution model of the design under test (Figure 1). Having one single, consistent interpretation model of the RTL specification, regardless of

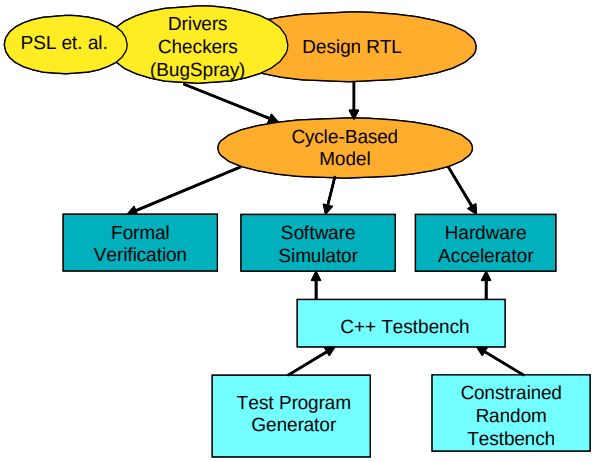


Figure 1. Core Methodology Flow

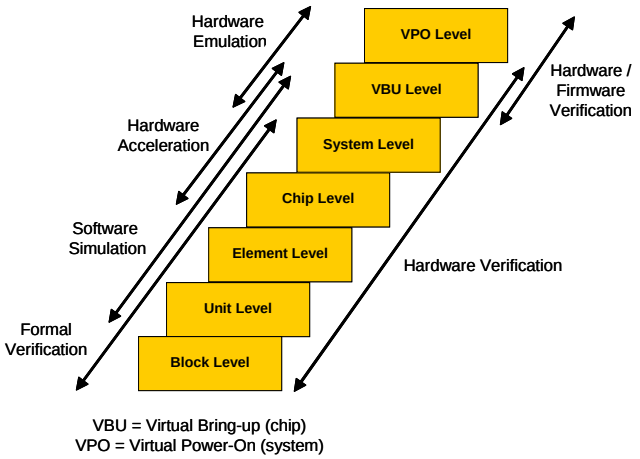


Figure 2. Verification progression

which tool or technology is used for a particular verification task enables reuse of verification objects (assertions, coverage events) and results (e.g., waveform signatures) across disciplines. Such a tight integration coupled with dramatically increased model capacity of the formal and semi-formal toolset has allowed the application of the technology to span designer-level verification, deep verification of design blocks, sequential equivalence checking and formal verification of large design partitions like the complete Floating-Point Unit (FPU) dataflow.

Internally IBM uses an extension of VHDL for functional coverage and assertion instrumentation, called *BugSpray*. BugSpray is used by the design and verification teams alike to efficiently annotate the RTL with assertion and coverage events. BugSpray enables verification objects to be portable across verification disciplines and across hierarchies, and allows for their reuse with design. For example, majority of the coverage events are provided by the design team with the goal of grading the verification effort given their knowledge of the design implementation. In addition, Property Specification Language (PSL) [9], standardized as IEEE 1850, may be used for the purposes of design instrumentation.

Extensive verification is undertaken at all levels of the design hierarchy [13], depicted in Figure 2. Verification at the lower levels of the hierarchy tends to be more productive due to the smaller size of the design under test, and greater controllability of the interfaces yielding higher state coverage and exploration of corner cases/boundary conditions. Verification objects from lower levels are selectively enabled at the higher levels. Formal methods are leveraged at various levels of the design/verification hierarchy to achieve different goals.

At the *block level* FFV is applied widely to prove design components. This may be driven in part by the designers themselves by way of assertion-based verification, and FFV environments inherit all of the designer assertions and coverage events. The downside of verifying logic at the block level is the need to model complex interface interactions between logic blocks requiring intricate testbenches (environment assumptions and properties), and coping with churn at those interfaces as the design evolves. Clear and precise documentation at this level may be somewhat lacking, making verification a more laborious process.

Increased speed and capacity of formal and semi-formal verification toolset has allowed FFV to scale to the *unit level* selectively. This enables taking advantage of relatively stable and well-documented interfaces, creation of simpler constraint-based [20], [16] drivers, and focus on more encompassing micro-architectural properties. The checking may entail creation of a reference model to equivalence check the implementation against, such as the IEEE floating point specification to verify the FPU, or specify a rich set of properties to constitute a specification of the unit as a whole.

The *element level* comprises design elements such as the processor core, cache and memory sub-system. FFV is primarily employed to verify multi-unit interactions and architectural aspects at this level, for instance hangs and stalls, starvation, bus protocols. We endeavor to reuse simulation RTL models by exposing only the logic of interest and effectively deleting logic not pertaining to the verification task at hand. Portions of the logic may need to be abstracted, and replaced with behavioral models to reduce logic size and complexity. The *chip level* incorporates multiple processor cores along with interconnect and storage sub-system, and the *system level* consists of multiple chips, memory and I/O chips per actual machine configurations. At these levels high level mathematical reasoning, manual proof techniques, and specialized models (e.g. Murphi model [8]) are used to verify features such as chip/system deadlock/livelock, cache and memory coherence, message routing and traffic flows across (asynchronous) interfaces.

"Pervasive" logic (e.g., initialization, scan/debug, RAS, power-management) which may span block, unit, element and chip boundaries poses numerous challenges to verification as it can be sequentially very deep, and may have large numbers of inputs to be verified effectively with simulation. FFV has demonstrated strength in this domain [12], and is applied at all levels of the hierarchy to verify complex pervasive logic.

Because the cost of finding a bug is lower at lower levels of the hierarchy every major bug found at a higher level is treated

as an escape of the lower levels, and every attempt is made to reproduce it at the lower levels. This helps to "harden" the lower level environments and make them more resilient. This translates into formulating design assertions at the block level to expose the flaw with FFV wherever applicable, and prove conclusively that the logic fix fixes it.

In a similar fashion, FFV is extensively applied to recreate post-silicon test floor failures at the block level, and verify fixes thereof. Typical fails on the test floor require many events to line up (as otherwise the problems would have been caught in pre-silicon verification), and it is non-trivial to produce the sequence of events leading up to the fail with higher level environments as those don't have direct control over the interfaces coming into a logic block. FFV has unique strengths to quickly root cause a defect once it is understood and the general area of the logic exhibiting the failure has been localized.

III. Integrated Approach

The cornerstone of large-scale application of formal and semi-formal verification is the pursuit of an integrated approach with design and simulation.

In the past FFV was a side activity with an execution plan owned and managed solely by the FFV team working closely with the designers to infer correctness properties, and to obtain interface specs to facilitate creation of FFV environments. The apparent disconnect with simulation-based verification can be attributed to a focus on verification at the block level with FFV, and at higher levels of the design hierarchy with simulation. This permitted a limited interaction with simulation teams, and coordination of the overall verification process across disciplines.

We changed all that on POWER7 by positioning FFV synergistically alongside simulation and making unit verification teams responsible for defining and owning (project managing) FFV plans based on their respective needs. This allowed for the plan to be dynamically altered, addressing specific requirements and deficiencies in real time, to make FFV application more effective. It was our endeavor to apply FFV early on complex logic blocks identified for formal checking in an attempt to provide value-add upfront to complement simulation-based verification. The widespread application of FFV at various levels of the hierarchy furthered this interlock. A rigorous process was instituted whereby FFV environments were reviewed closely with designers, architects, FV experts and functional verification engineers to ensure completeness of the checking and correctness of the assumptions.

The portability of synthesizable coverage and assertion specification between simulation, hardware accelerated simulation and formal and semi-formal verification was critical to our approach to cross-link the different verification efforts more effectively (Figure 3). It enabled us to aggressively drive an assertion-based verification paradigm which brought together designer-level verification, formal verification and simulation in a unified integrated methodology. For example, designer assertions and coverage events used to grade simulation environments were utilized in formal verification

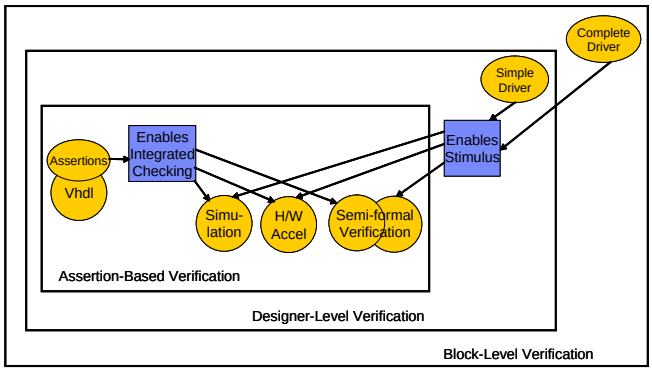


Figure 3. Integrated (with Design) Approach

environments. In turn, formal environment checkers (properties) and assumptions were added to the simulation environment to be cross-checked and cross-leveraged. Artifacts of simulation-based verification, such as biases to specify relative probabilities of inputs switching (e.g., a reset line should not be active frequently as it would restart the simulation effectively), or initializations based on machine configurations, are taken advantage of in formal and semi-formal verification (as applicable) to make it more productive.

We have been pushing for a broad adoption of FFV natively by design and verification teams and achieved significant progress on POWER7. FFV was leveraged extensively by designers as an assertion-based verification vehicle to check their designs before making them available to verification teams. This helped to improve productivity significantly by breaking the costly cycle of: designer checks RTL code into the repository, simulation runs with the design and uncovers errors and logs issues, the designer fixes the design and again makes it available. In a number of cases designers developed comprehensive block level FFV environments to prove the design. For others the designers inherited the environments from FFV experts and took ownership to continue to regress with those, and enhance the environments as needed. Simulation verification teams, for their part, took advantage of FFV to fill gaps in their verification testplans - e.g. chip-wide networks to transmit debug information which require large numbers of patterns to verify with simulation are easy for FFV to handle.

IV. Demonstrated Best Practices

We continue to leverage and extend successful applications of FFV from past projects. The POWER7 program saw substantial deployment of these proven engagements.

Complex logic blocks on the different units were identified and prioritized for a targeted "deep dive" verification by FV experts [13], [21]. The core strengths of block level verification are the small size of the design under test, and direct control on testing as the driving stimulus is applied directly to the interfaces of the block with no "filtering" effect of upstream logic. The former translates into proofs to ensure correctness of the aspects verified. The latter facilitates exploring all areas of the interface's state space equally easily, whereas some of those areas would have been exercised rarely in the larger

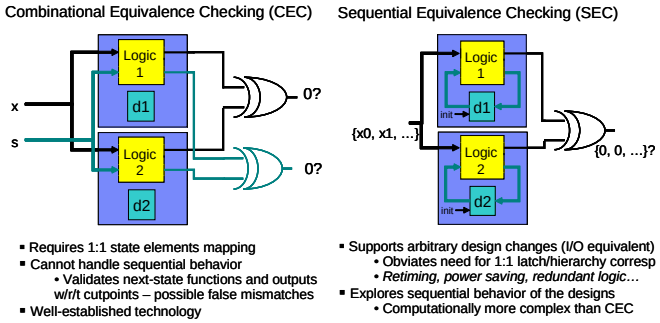


Figure 4. Combinational and Sequential Equivalence Checking

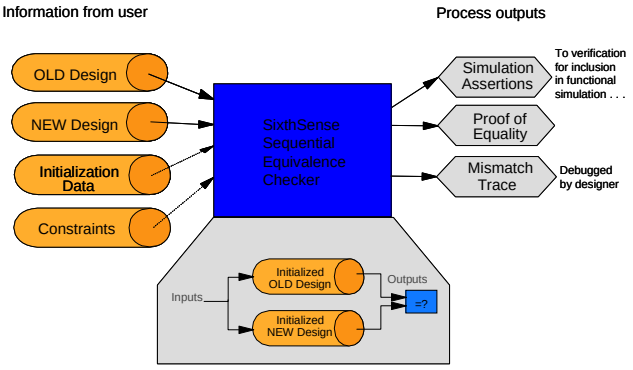


Figure 5. Sequential Equivalence Checking Set-up

context - e.g., with the block connected in the unit. This is invaluable to exercise low level interactions and window conditions given challenging micro-architecture features (e.g., simultaneous multi-threading, out-of-order execution).

The blocks are chosen in consultation with design and verification teams based on their complexity, difficulty in verifying them with other verification disciplines, and logics that have exhibited late bugs on previous projects. Block level verification ranges from white-box checking of the logic by FFV experts based on a deep understanding of the micro-architecture (e.g., instruction prefetch/fetch, memory management unit), to end-to-end checking to verify conformance of the function against a specification (e.g., queues, error correcting codes), to verification of the algorithmic correctness of the logic (e.g., least recently used cache replacement). Our endeavor is to make the verification high level to the extent possible by formulating properties that are conceptual/architectural, hence independent of the implementation. Application areas include all areas of the chip (core units, caches, pervasive) and hierarchies (blocks, unit, element, chip).

POWER7 saw more blocks verified on more functional units than on any previous project.

Proven and established methodologies were utilized to verify (complete) function of logics such as FPU dataflow [14] and arithmetic functions (e.g., adder, multiplier, divider), by comparing the high performance implementation against a high level reference model. We expanded the list of pervasive logics verified with FFV building atop past successes [12] to include additional areas (e.g., chip-wide sensor networks), and created automated methodologies to improve productivity – e.g., Debug Bus verification via automated testbench creation given a specification in a custom language.

In a number of cases FFV was the technology of choice to be relied upon heavily/solely to ensure correctness of, often times critical, logic (e.g., arithmetic dataflow, arbitration, least recently used). The environments created to verify blocks proved very useful later to quickly root cause post-silicon problems seen on the laboratory floor. FFV has assumed a central role over the years in triaging bugs in silicon, and assuring the correctness of the logic fixes.

V. Sequential Equivalence Checking

Sequential transformations are widespread in hardware design flows to address needs such as performance, power, area, debug and test. Established frameworks such as Combinational Equivalence Checking (CEC) are unable to handle such sequential changes as it requires a 1:1 pairings of the state elements. Industrial demand for robust Sequential Equivalence Checking (SEC) solutions is thus becoming increasingly prevalent. SEC is a paradigm to help offset the limitations of CEC (Figure 4). SEC performs a true sequential check of input/output equivalence, hence is not limited to operation on designs with 1:1 state element pairings.

SEC [4] is widely deployed at different stages of the project to achieve various goals, such as verify non-functional design changes (e.g., power, timing, area), verify external IP conversion over to IBM's clocking and latching methodology, ensure mode latches indeed revert the design back to a previous function. SEC has also been key to several new methodologies which leverage the power of sequential transformations [25], [10], [4].

With POWER7 we made this easy-to-use yet powerful technology available in the hands of the designers to improve productivity substantially by obviating the need to rerun costly regressions to verify non-functional changes to the design. In later stages of the project when all function was completed, we instituted a rigorous end-to-end SEC process (Figure 6) starting at the macros and working its way up to the chip level (with black-boxing lower levels of the hierarchy) to establish that inadvertent functional changes did not get introduced in subsequent releases of the RTL. This facilitated avoiding (tools and technology) capacity issues with running large partitions, and enabled designers to run SEC on interfaces/design sections they are the most familiar with. It helped to improve the debug cycle by way of producing short, precise and localized mismatch traces, as opposed to needing large numbers of cycles before mismatches propagate to an observable outputs.

The process is flexible enough to permit definition of hierarchies to run SEC at, which may or may not align with the design hierarchy. This allows to verify behavior-preservation of changes across design entities, such as moving logic between entities with bundling the entities together in a custom wrapper, which is then equivalence checked and black-boxed at higher levels.

Any assumptions required to get the equivalence check to succeed at any level of the hierarchy are independently

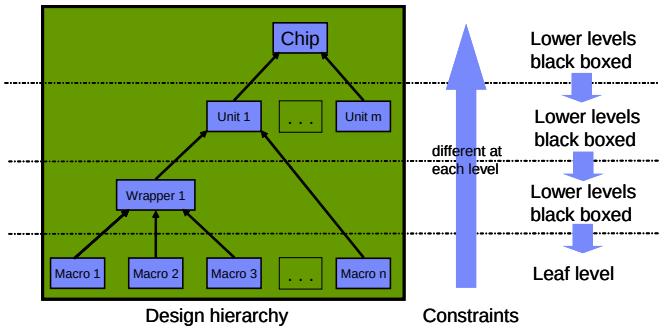


Figure 6. Hierarchical Decomposition

validated with functional verification by converting the constraints into assertions (Figure 5) which are then pulled into the simulation/FFV environments.

This end-to-end SEC process has proved to be an invaluable tool on multiple projects to verify remaps of the design to a newer technology without the need to set-up functional verification environments yielding huge resource savings, productivity improvement and fast turnaround. Low level functional verification environments are done away with completely and a few higher level ones are used for the purposes of a quick sanity check and to verify SEC assumptions – the portability of assertions across design hierarchies allows those to be carried to higher level verification environments.

VI. New Exploration Areas

We continue to push the envelope to scale FV application to areas which hitherto have been beyond the scope of FV. We endeavored to push the (capacity) limits of FV application by exploring several new areas on the POWER7.

We pursued creation of (constraint-based) environments at higher levels of abstraction, for example (sub-) unit level, to take advantage of well-defined interfaces, and increase interplay with simulation. The latter is achieved by leveraging FFV to weed out unreachable coverage events, or obtain hints to hit tough coverage events to help enhance simulation test buckets, and dramatically increase state-space coverage via semi-formal verification.

We investigated verification of system level aspects in several contexts. These were, for example, timing protection windows to ensure coherence by enumerating system topologies and studying multi-chip interactions, and deadlock free operation of the system using direct mathematical analysis on dedicated high level models. These efforts helped flush out architectural issues early on, and assisted in setting-up effective simulation testbenches to check for violations. As another example, asynchronous interfaces are a formidable challenge to functional verification due to the fact that they are not exercised adequately at the system level. We attempted to reason about the effect of asynchronous interfaces (e.g., unpredictable traffic flow across the interface may manifest as buffer overflows) by modeling these interfaces using system level models with design details suitably abstracted.

We pioneered various innovative and reusable techniques/methodologies by way of creating "off-the-shelf" verification IP which can be applied out-of-the-box for similar logics on other parts of the chip, or on future products. Examples include a method to expose starvation in complex arbitration logics using successive property strengthening and underapproximations [1], and to verify correctness and performance of such arbiters by accurately computing request-to-delay bounds and ascertaining the fairness requirements of the arbitration scheme [15]; systematic methods to verify complex 64Byte error correcting codes, least recently used replacement and hardware data structures such as linked lists, queues, buffers, etc.

Block level FFV testbenches may require modeling complex interfaces which can make them non-trivial and error prone. Specifying those at higher levels of abstraction can help alleviate this to a great extent. Towards this goal we undertook the creation of a rich library of functions (implemented as VHDL packages) to enable specifying testbenches at higher levels. The parameterized functions encapsulate commonly used logic constructs (such as counters, zero/one-hot detectors/generators, oscillators, biased non-deterministic generators, to list a few) and synthesize correct-by-construction logic implementing the functionality. While it is desirable to raise the level of abstraction of the design logic itself, it is not-so-easy for optimized custom logic used in high-end microprocessors, more so given the interwining with "non-mainline" functions such as pervasive logic. We are selectively applying the concept of functions pursued in the context of testbenches to the design domain by creating a library of functions optimized with respect to desired features such as area, timing, logic depth.

VII. Future Directions

With having established FFV and SEC firmly as a mainstream verification discipline integrated in the design and verification methodology, we expect to derive increasing leverage from its application on future projects. Following are example areas where we foresee investments.

We plan to build upon the integrated approach further and position formal and semi-formal platform as the technology of choice for Designer-level Verification (DLV). Towards this goal we have enhanced the technology and the supporting infrastructure to cover the entire spectrum of DLV from block level simulation with applying deterministic patterns to study input-output behavior, to selectively randomizing signals, to creating comprehensive FFV environments to reason conclusively about assertions and coverage events.

Given the successes and the mind share FFV has to be an effective verification paradigm, we will continue down the path of "booking" the verification of more logics in FFV, and take those off the simulation plate altogether. This allows to maximize productivity across the various disciplines by avoiding duplicate work. We have undertaken a detailed analysis of testplans across FFV, simulation and performance verification to optimize them by making trade-offs with regard to what aspects of the logic get checked where, with the intent of taking maximal advantage of the strengths of the various technologies.

We expect to build upon the theme of "off-the-shelf" checkers to verify logics in an implementation agnostic manner. We

have created a persistent compendium of verification IP and we plan to generalize it to verify the logics end-to-end with high level micro-architectural and architectural checkers. In some cases we are attempting to package the IP as a parameterized library which can be applied easily and productively to logics implementing the same function. We will continue to evolve reusable and automated methodologies to make verification of certain logics push-button.

More and more verification transcends checking for functional correctness of the logic to include aspects such as performance, throughput, power, etc. FFV has unique strengths to be able to provide insights into the various aspects by approaching the verification task in a unified manner. An example is the combined verification of performance and correctness of arbiters as described in [15] by establishing an upper bound on the request-to-grant delay. Another example is examining traffic throughput across an (asynchronous) interface to decide on machine configurations/settings such as to not clog buffers on the receive side and queue up traffic in the system. Our goal is to leverage FFV and SEC to provide value add beyond checking for correctness of the logic, including as a general purpose reasoning engines to enable new methodologies.

It is our endeavor to pursue a "formal design" paradigm, especially for newly designed logics, to evolve methods to guarantee its correctness. This can be achieved by formally verifying a high level model independently, and ensuring the implementation conforms to the verified model by virtue of an equivalence check. The high performance RTL may be derived from the high level model via automated (iterative) transformations which are verified with SEC. Alternatively, we may decompose the design into smaller modular pieces in a manner such that each piece can be reasoned about exhaustively standalone, and the proofs of the individual pieces imply correctness of the logic.

We continue to expect to innovate and evolve methods to scale to the complex logics showing up on the next generations of our systems, e.g., wide operand non-linear arithmetic such as used in cryptography accelerators. The power of theorem proving augmented with model checking [24] is a key ally in scaling to such tough problems, as demonstrated in [23].

System level issues such as deadlocks/livelocks are a particular concern on large multi-processor systems as simulation methods cannot produce the kinds of traffic the hardware would experience, and the kinds of interactions between the various traffic sources as a consequence. High level analysis and ways to model traffic in such multi-chip systems, especially given asynchronous interfaces, to study traffic flows or lack of forward progress is an effective method to uncover problems, and will be a focus area.

Significant improvements to the speed and capacity of the formal and semi-formal toolset in the form of improvements to the core engines [17], [3], [19], addition of new algorithms [7], [6] and significant features (e.g., native array support [2]), has enabled FV to address the outlined applications. We continue to expect to see rapid advances to scale to future challenges.

ACKNOWLEDGMENTS

The authors would like to thank the members of the FV community at IBM for their relentless pursuit of leverage from the technology. Thanks to Wolfgang Roesner, Klaus-Dieter Schubert, Jason Baumgartner and Ali El-Zein for helping shape FV methodology, and to members of the FFV and SEC team, Gadiel Auerbach, Mark Firstenberg, Paul Roessler, Jo Lee, Shiri Moran, David Levitt, Fady Copty, Steven German, Krishnan Kailas for their contributions to the POWER7 effort.

REFERENCES

[1] G. Auerbach, F. Copty, and V. Paruthi. Formal verification of arbiters using property strengthening and underapproximations. In *FMCAD*. IEEE, Oct. 2010.

[2] J. Baumgartner, M. Case, and H. Mony. Coping with Moore's law (and more): Supporting arrays in state-of-the-art model checkers. In *FMCAD*. IEEE, Oct. 2010.

[3] J. Baumgartner, H. Mony, and A. Aziz. Optimal constraint-preserving netlist simplification. In *FMCAD*, Nov. 2008.

[4] J. Baumgartner, H. Mony, V. Paruthi, R. Kanzelman, and G. Janssen. Scalable sequential equivalence checking across arbitrary design transformations. In *ICCD*. IEEE, Oct. 2006.

[5] S. Ben-David, C. Eisner, D. Geist, and Y. Wolfsthal. Model checking at IBM. *Formal Methods in System Design*, 22(2):101–108, 2003.

[6] M. Case, A. Mishchenko, R. Brayton, J. Baumgartner, and H. Mony. Invariant-strengthened elimination of dependent state elements. In *FMCAD*, Nov. 2008.

[7] M. Case, H. Mony, J. Baumgartner, and R. Kanzelman. Enhanced verification through temporal decomposition. In *FMCAD*, Nov. 2009.

[8] X. Chen, S. German, and G. Gopalakrishnan:. Transaction based modeling and verification of hardware protocols. In *FMCAD*. IEEE, Nov. 2007.

[9] C. Eisner and D. Fisman. *A Practical Introduction to PSL*. Integrated Circuits and Systems. Springer-Verlag, 2006.

[10] C. Eisner, A. Nahir, and K. Yorav. Functional verification of power gated designs by compositional reasoning. In *CAV*, July 2008.

[11] Wikipedia The Free Encyclopedia. Power7. http://en.wikipedia.org/wiki/POWER7.

[12] T. Gloekler, J. Baumgartner, D. Shanmugam, R. Seigler, G. Huben, H. Mony, P. Roessler, and B. Ramanandray. Enabling large-scale pervasive logic verification through multi-algorithmic formal reasoning. In *FMCAD*, Nov. 2006.

[13] J. Ludden et al. Functional verification of the POWER4 microprocessor and POWER4 multiprocessor systems. *IBM Journal of Research and Development*, Jan. 2002.

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

[15] K. Kailas, V. Paruthi, and B. Monwai. Formal verification of correctness and performance of random priority-based arbiters. In *FMCAD*. IEEE, Nov. 2009.

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

[17] H. Mony, J. Baumgartner, A. Mishchenko, and R. Brayton. Speculative-reduction based scalable redundancy identification. In *DATE*, Apr. 2009.

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

[19] V. Paruthi, C. Jacobi, and K. Weber. Efficient symbolic simulation via dynamic scheduling, don't caring, and case splitting. In *CHARME*, 2005.

[20] Carl Pixley. Integrating model checking into the semiconductor design flow. In *Electronic Systems Technology & Design*, 1999.

[21] R. Gott and J. Baumgartner and P. Roessler and S. Joe. Functional formal verification on designs of pseries microprocessors and communication subsystems. *IBM Journal of Research and Development*, July. 2005.

[22] IBM Research. Rulebase parallel edition. https://www.research.ibm.com/haifa/projects/verification/RB_Homepage/.

[23] J. Sawada. Automatic verification of estimate functions with polynomials of bounded functions. In *FMCAD*. IEEE, Oct. 2010.

[24] J. Sawada and E. Reeber. Acl2six: A hint used to integrate a theorem prover and an automated verification tool. In *FMCAD*. IEEE, 2006.

[25] A. Seigler, G. Van Huben, and H. Mony. Formal verification of partial good self-test fencing structures. In *FMCAD*. IEEE, Nov. 2007.

A Single-Instance Incremental SAT Formulation of Proof- and Counterexample-Based Abstraction

Niklas Een, Alan Mishchenko
EECS Department, University of California
Berkeley, USA.

Nina Amla
Cadence Research Laboratroy
Berkeley, USA.

***Abstract*—This paper presents an efficient, combined formulation of two widely used abstraction methods for bit-level verification: *counterexample-based abstraction (CBA)* and *proof-based abstraction (PBA)*. Unlike previous work, this new method is formulated as a single, incremental SAT-problem, interleaving CBA and PBA to develop the abstraction in a bottom-up fashion. It is argued that the new method is simpler conceptually and implementation-wise than previous approaches. As an added bonus, proof-logging is not required for the PBA part, which allows for a wider set of SAT-solvers to be used.**

I. Introduction

Abstraction techniques have long been a crucial part of successful verification flows. Indeed, the success of SAT-solving can largely be attributed to its inherent ability to perform localization abstraction as part of its operations. For this reason so called bug-hunting, or BMC, methods can often be applied on a full design directly, thereby deferring the abstraction work to the SAT-solver. However, computing an abstraction explicitly is often more useful for hard properties that require a mixture of different transformation and proof-engines to complete the verification.

In our formulation, both CBA and PBA compute a localization in the form of a set of flops. An abstracted flop is in essence replaced by a primary input (*PI*), thus giving more behaviors to the circuit. Both methods work by analyzing, through the use of SAT, a k-unrolling of the circuit. However, they differ as follows:

- CBA works in a bottom-up fashion, starting with an empty abstraction (all flops are replaced by PIs) and adding flops to refute the counterexamples as they are enumerated for successively larger k.
- PBA, in contrast, considers the full design and a complete refutation of all counterexamples of depth k (in the form of an UNSAT proof). Any flop not syntactically present in the proof of UNSAT is abstracted.

The two methods have complementary strengths: CBA by virtue of being bottom-up is very fast, but may include more flops than necessary. PBA on the other hand does a more thorough analysis and almost always gives a tighter abstraction than CBA, but at the cost of longer runtime.

In this work, it is shown how the two methods can be seamlessly combined by applying PBA, not on the full design, but on the latest abstraction produced by CBA. This solution has a very elegant incremental SAT formulation, which results in a simple, scalable algorithm that has the strength of both methods.

In the experimental section it is shown how a design with 40,000 flops and 860,000 AND-gates is localized to a handful of flops in just 4 seconds (much faster than any previous method), and how this abstraction is instantaneously solved by the interpolation-based proof-engine [10], whilst the original unabstracted design took 2 minutes to verify, despite the inherent localization ability of interpolation.

II. Related Work

Counterexample-based abstraction was first introduced by Kurshan in [8] and further developed by Clarke et. al. in [3]. Proof-based abstraction was coined by McMillan [11], and independently proposed by Gupta et. al in [7].

The work most closely related to ours is Gupta's work of [7] and McMillan et. al's work of [1]. In both approaches, abstract counterexamples are concretized using a SAT-solver. When concretization fails, the UNSAT proof guides the abstraction refinement. Our work does not rely on a SAT-solver to refute counterexamples, but instead uses a simpler and more scalable method based on ternary simulation (section IV-A).

Gupta's approach does not rely on BDD reachability to produce abstractions; although BDDs are used to form a complete proof-procedure. Like our method, it tries to limit the amount of logic that is put into the SAT-solver when unrolling the circuit, thereby improving scalability. It differs, though, in that the initial unrolling is done on the concrete design (our method starts with an empty abstraction), and that PBA is used to shrink the abstract model in a more conservative manner, requiring the PBA result to stabilize over several iterations.

The work of McMillan et. al. mixes PBA for refuting all counterexamples of length k with proof-analysis of counterexamples from the BDD engine, refuting individual (or small sets of) counterexamples. Unlike Gupta's work, BDDs are an integral part of the abstraction computation.

The approach proposed in this paper differs further from previous work in that it does not constitute a complete proof-procedure. There are many different ways of using an abstraction method as part of a verification flow. A simple use-model would be: Run the abstraction computation until some resource limit is reached, then output the best abstraction found so far and put the method on hold. If the abstraction turns out not to

be good enough for the downstream flow, resume abstraction computation with a higher resource limit, and produce a more refined abstraction. Obviously, this use-model can be further improved by multi-threading on a multi-core machine.

In the experimental evaluation, we choose to pass abstractions to an interpolation-based proof-engine. This particular setup relates to the work of [9] and [2].

III. Assumptions and Notation

In the presentation, the following is assumed:

- The design is given as a set of next-state functions expressed in terms of current state variables (flops) and primary inputs (PIs).
- The design has only one property, which is a safety property.
- All flops are initialized to zero, and are running on the same clock (hence acting as unit delays).

It is further assumed that the logic of the next-state functions is represented as a combined And-Inverter-graph, with the single property being the output of a particular AND-gate. As customary, the negation of the property is referred to as the *bad* signal.

An "abstraction" is identified with a set of flops. If a flop is not part of the abstraction, it is treated as a PI in the abstract model of the design. By this semantics, adding a flop to the current abstraction means *concretizing* it in the abstract model: replace the PI by a flop and connect it to the appropriate input signal.

IV. Algorithm

How does the proposed algorithm work? It starts by assuming the empty abstraction, treating all flops as PIs. It then inserts one time-frame of the design into the SAT-solver, and asks for a satisfying assignment that produces TRUE at the *bad* signal. The SAT-solver will come back SAT[1] and the counterexample is used to concretize some of the flops (= CBA). When enough flops have been concretized, the SAT-problem becomes UNSAT, which means that all counterexamples of length 0 have been refuted (unless there is a true counterexample of length zero). The algorithm can now move on to depth 1, but before doing so, any flop that did not occur in the UNSAT proof is first removed from the abstraction (= PBA). The procedure is repeated for increasing depths, resulting in an incremental sequence of SAT calls that looks something like

depth 0: SAT, SAT, SAT, SAT, **UNSAT**
depth 1: SAT, **UNSAT**
depth 2: SAT, SAT, SAT, **UNSAT**
⋮

with each sequence of calls at a given depth ending in an "UNSAT" result that prunes the abstraction built up by analyzing the preceding "SAT" counterexamples.

[1]The very first query only comes back "UNSAT" if the property holds combinationally, a corner case we ignore here.

The algorithm terminates in one of two ways: either (i) CBA comes back with the same set of flops as were given to it, which means we have found a true, justified counterexample, or (ii) it runs out of resources for doing abstraction and stops. The resulting abstraction is then returned to the caller to be used in the next step of the verification process.

A. Counterexample-based refinement

Assume that for the current abstraction A the last call to SAT returned a counterexample of length k. The counterexample is then analyzed and refined by the following simple procedure[2] in order to refute it:

> **CBA refinement.** Loop through all flops not in A. Replace the current value of the counterexample with an X (the undefined value) and do a three-valued simulation. If the X does not reach the *bad* signal, its value is unimportant for the justification of the counterexample, and the corresponding flop is kept as a PI. If, on the other hand, X propagates all the way to *bad*, we undo the changes made by that particular X-propagation and add the corresponding flop to A.

The order in which flops are inspected does matter for the end result. It seems like a good idea to consider multiple orders and pick the one producing the smallest abstraction. But in our experience it does not improve the overall algorithm. The extra runtime may save a few flops temporarily, but they are typically added back in a later iteration, or removed by PBA anyway, resulting in the same abstraction in the end.

B. Incremental SAT

Incremental SAT is not a uniquely defined concept. The interpretation used here is a solver with the following two methods:

- ***addClause***(*literals*): This method adds a clausal constraint, i.e. $(p_0 \vee p_1 \vee \ldots \vee p_{n-1})$ where $p_i \in$ *literals*, to the SAT-solver. The incremental interface allows for more clauses to be added later.
- ***solveSat***(*assumps*): This method searches for an assignment that satisfies the current set of clauses under the unit assumptions *assumps* = $a_0 \wedge a_1 \wedge \ldots \wedge a_{n-1}$. If there is an assignment that satisfies all the clauses added so far by calls to ***addClause***(), as well as the unit literals a_i, that model is returned. If, on the other hand, the problem is UNSAT under the given assumptions, *the subset of those assumptions used in the proof of UNSAT* is returned in the form of a final conflict clause.

The extension of ***solveSat***() to accept a set of unit literals as assumptions, and to produce the subset of those that were part

[2]This procedure (implemented by Alan Mishchenko in ABC [6]), has been independently discovered by one of our industrial collaborators, and probably by others too. A similar procedure is described in [13].

of the UNSAT proof, can easily be added to any modern SAT-solver.[3] This is in contrast to adding proof-logging, which is a non-trivial endeavor. For that reason, the proposed algorithm is stated entirely in terms of this interface and does not rely on generating UNSAT proofs.

C. Refinement using activation literals

Unlike the typical implementation of PBA, this work uses *activation literals*, rather than a syntactic analysis of resolution proofs, to determine the set of flops used for proving UNSAT. For each flop f that is concretized, a literal a is introduced in the SAT-instance. As the flop input f_{in} at time-frame k is tied to the flop output at time-frame $k+1$, the literal is used to activate or deactivate propagation through the flop by inserting two clauses stating:

$$a \rightarrow (f[k+1] \leftrightarrow f_{in}[k])$$

The set of activation literals is passed as assumptions to ***solveSat***(), and for UNSAT results, the current abstraction can immediately be pruned of flops missing from the final conflict clause returned by the solver.

This PBA phase is very affordable. The same SAT-problem would have to be solved in a pure CBA based method anyway. The cost we pay is only that of propagating the assumption literals. Because abstractions are derived in a bottom-up fashion, with the final abstraction typically containing just a few hundred flops, the overhead is small.

V. Implementation

This section describes the combined abstraction method in enough detail for the reader to easily and accurately reproduce the experimental results of the final section. The pseudo-code uses the following conventions:

- Symbol **&** indicates pass-by-reference.
- The type **Vec**⟨**T**⟩ is a dynamic vector whose elements are of type T.
- The type **Netlist** is an extended And-Inverter-graph. It has the following gate types: AND, PI, FLOP, CONST. Inverters are represented as complemented edges. Flops act as unit delays. Every netlist *N*, has a special gate *N.True* of type CONST.
- The type **Wire** represents an edge in the netlist. Think of it as a pointer to a gate plus a "sign" bit. It serves the same function as a *literal* w.r.t. a variable in SAT. Function ***sign***(*w*) will return TRUE if the edge is complemented, FALSE otherwise. By w_0 and w_1 we refer to the left and right child of an AND-gate. By w_{in} we refer to the input of a flop.
- The type **WSet** is a set of wires.

[3]Two simple things should be done: (i) the decision heuristic has to be changed so that the first n decisions are made on the assumption literals; and (ii) if a conflict clause is derived that contradicts the set of assumptions, that clause has to be further analyzed back to the decision literals rather than the first UIP. For more details, please review the ***analyzeFinal()*** method of MiniSAT [5].

```
class Trace {
 – Private variables:
      Netlist&              N;
      Netlist               F;
      SatSolver             S;
      WSet_N                abstr;      – publicly read-only

      Vec⟨WMap_N⟨Wire_F⟩⟩    n2f;
      WMap_F⟨Lit⟩           f2s;
      WMap_N⟨Lit⟩           act_lits;
 – Private functions:
      Lit  clausify   (Wire_F f);
      void insertFlop (int frame, Wire_N w_flop, Wire_F f);
 – Constructor:
      Trace(Netlist& N);
 – Public functions:
      Wire_F insert   (int frame, Wire_N w);
      void   extendAbs (Wire_N w_flop);
      bool   solve     (WSet_F f_disj);
      Cex    getCex    (int depth);
};

class Cex { ... };        – stores a counter-example
```

Figure 1. *Interface of the "Trace" class.* The class handles the BMC unrolling of the design *N*. Netlist *F* will store the structurally hashed unrolling of *N*. SAT-solver *S* will store a CNF representation of the logic in *F*.

- The type **WMap**⟨**T**⟩ maps wires to elements of type T. For practical reasons, the sign bit of the wire is *not* used. For map *m*, *m*[*w*] is equivalent to *m*[¬*w*]. Unmapped elements of *m* are assumed to go to a distinct element T_UNDEF (e.g. LIT_UNDEF for literals, or WIRE_UNDEF for wires).
- The type **lbool** is a three-valued boolean that is either *true*, *false*, or *undefined*, represented in the code by: LBOOL_0, LBOOL_1, LBOOL_X.
- Every SAT-instance *S* (of type **SatSolver**) has a special literal *S.True* which is bound to *true*. Method *S.**newLit***() creates a new variable and returns it as a literal with positive polarity. Clauses are added by *S.**addClause***() and method *S.**satSolve***() commences the search for a satisfying assignment.

Because the pseudo-code deals with two netlists *N* and *F*, wire-types are subscripted Wire_N and Wire_F to make clear which netlist the wire belongs to. The same holds for WSet and WMap.

A. BMC Traces

To succinctly express the SAT analysis of the unrolled design, the class **Trace** is introduced (see Figure 1). It allows for incrementally extending the abstraction, as well as lengthening the unrolled trace. Its machinery needs the following:

- A reference *N* to the input design (read-only).
- A set of flops *abstr*, storing the current abstraction. Calling ***extendAbs***() will grow this set. Calling ***solve***()

may shrink it through its built-in PBA.

- A netlist F to store the unrolling of N under the current abstraction. Gates are put into F by calling ***insert***(*frame*, *w*). Only the logic reachable from gate *w* of time-frame *frame* is inserted. For efficiency, netlist F is kept structurally hashed.
- A SAT-instance S to analyze the logic of F. Calling ***solve***(*f_disj*) will incrementally add the necessary clauses to model the logic of F reachable from the set of wires *f_disj*. The user of the class does not have to worry about how clauses are added; hence ***clausify***() is a private method. The SAT-solving will take place under the assumption $f_0 \vee f_1 \vee \ldots \vee f_{n-1}$. The method ***solve***() has two important side-effects:
  - For satisfiable runs, the satisfying assignment is stored so that ***getCex***() can later retrieve it.
  - For unsatisfiable runs, the flops not participating in the proof are *removed* from the current abstraction.
- Maps *n2f* and *f2s*. Expression "*n2f*[*d*][*w*]" gives the wire in F corresponding to gate *w* of N in frame *d*. Expression "*f2s*[*f*]" gives the literal in S corresponding to gate f of F.
- Map *act_lits*. Expression "*act_lits*[*w_flop*] gives the activation literal for flop *w_flop*, or WIRE_UNDEF if none has been introduced.

B. The main procedure

The main loop of the abstraction procedure is given in Figure 2. Trace instance T is created with an empty abstraction. For increasing depths, the following is done:

- If the SAT-solver produces a counterexample, it is analyzed (by ***refineAbstraction***()) and flops are added to the abstraction to rule out this particular counterexample.
- If UNSAT is returned, the depth is increased. The ***solve***() method will have performed proof-based abstraction internally and may have removed some flops from the abstraction.

For each new depth explored, a new *bad* signal is added to *bad_disj*. This disjunction is passed as an assumption to the *solve* method of T, which means we are looking for a counterexample where the property fail in at least one time frame. It is not enough to just check the last time frame because of PBA.

C. Unrolling and SAT solving

Figure 3 details how ***insert***() produces an unrolling of N inside F, and Figure 4 describes how ***solve***() translates the logic of F into clauses and calls SAT. Great care is taken to describe accurately what is implemented, as the precise incremental SAT formulation is important for the performance and quality. For the casual reader who may not want to delve into details, the following paragraph summarizes some properties of the implementation:

As the procedure works its way up to greater and greater depth, only the logic reachable from the *bad* signal is introduced into the SAT-solver, and only flops that have been concretized bring in logic from the preceding time-frames. Constant propagation and structural hashing is performed on the design, although constants are not propagated across time-frames due to proof-based abstraction (PBA). Concrete flops are guarded by activation literals, which are used to implement PBA. One literal guards all occurrences of one flop in the unrolling. Flops that are removed by PBA will not be unrolled in future time-frames. However, fanin-logic from removed flops will remain in F and in the SAT-solver, but is disabled using the same activation literals.

VI. Evaluation and Conclusions

The method of this paper was evaluated along two dimensions: (i) how does the new abstraction procedure fare in the simplest possible verification flow, where a complete proof-engine (in this case interpolation [10]) is applied to its result versus applying the same proof-engine without any abstraction; and (ii) how does it compare to previous hybrid abstraction methods—in our experiments, the implementation of CBA and PBA inside ABC [6], and the hybrid method of McMillan et. al. [1].

The examples used were drawn from a large set of commercial benchmarks by focusing on designs with local properties containing more than 1000 flops.[4] Experiments were run on an 2 GHz AMD Opteron, with a timeout of 500 seconds. The results are presented in Table I.

For all methods, the depth was increased until an abstraction good enough to prove the property was found. ABC has a similar CBA implementation to the one presented in this work (based on ternary simulation), but restarts the SAT-solver after each refinement. ABC's PBA procedure is separate from CBA, so we opted for applying it once at the end to trim the model returned by CBA. This flow was also simulated in our new algorithm by delaying the PBA filtering until the final iteration (reported in column *New'*). This approach is often faster due to the fewer CBA refinement steps required, but there seems to be a quality/effort trade-off between applying PBA at every step, or only once at the end. In particular for the *S* series, interleaved CBA/PBA resulted in significantly smaller abstractions. We have observed this behavior on other benchmarks as well.

The McMillan hybrid technique was improved by replacing BDDs with interpolation, which led to a significant and consistent speedup. However, our new method, and the similar techniques of ABC, still appear to be superior in terms of scalability. This is most likely explained by the expensive concretization phase of the older method, which requires the full design to be unrolled for the length of the counterexample.

The effect of an incremental implementation can be seen by comparing columns *New'* and *ABC*. We have observed that the

[4] In other words, we've picked examples for which abstraction should work well. There are many verification problems where abstraction is *not* a useful technique, but here we investigate cases where it is.

```
WSet_N ⊎ Cex combinedAbstraction(Netlist N) {
    Trace  T(N);
    Wire_N  bad = ¬N.getProperty();
    WSet_F  bad_disj = ∅;

    for (int depth = 0;;) {
        if (⟨reached resource limit⟩)
            return T.abstr;

        bad_disj = bad_disj ∪ {T.insert(depth, bad)};

        if (T.solve(bad_disj)) {      – Found counter-example; refine abstraction:
            int n_flops = T.abstr.size();
            refineAbstraction(T, depth, bad);
            if (T.abstr.size() == n_flops)      – Abstraction stable ⇒ counter-example is valid:
                return T.getCex(depth);
        }else
            depth++;
    }
}

void refineAbstraction(Trace& T, int depth, Wire_N bad) {
    Cex                    cex = T.getCex(depth);
    Vec⟨WMap_N⟨lbool⟩⟩ sim = simulateCex(T.N, T.abstr, cex);      – 'sim[d][w]' = value if gate 'w' at frame 'd'

    WSet_N to_add;
    for all flops w not in T.abstr {
        for (int frame = 0; frame ≤ depth; frame++) {
            simPropagate(sim, T.abstr, frame, w, LBOOL_X);

            if (sim[depth][bad] == LBOOL_X) {
                – 'X' propagated all the way to the output; undo simulation and add flop to abstraction:
                for (; frame ≥ 0; frame−−)
                    simPropagate(sim, T.abstr, frame, w, cex.flops[frame][w]);
                to_add = to_add ∪ {w};
                break;
            }
        }
    }
    for w ∈ to_add
        T.extendAbs(w);
}

Vec⟨WMap_N⟩⟩ simulateCex(Netlist N, WSet_N abstr, Cex cex) {
    return ⟨ternary simulate counter-example 'cex' on 'N' under abstraction 'abstr'⟩
}

void simPropagate(Vec⟨WMap_N⟨lbool⟩⟩& sim, WSet_N abstr, int frame, Wire_N w, lbool value) {
    ⟨incrementally propagate effect of changing gate 'w' at time-frame 'frame' to 'value'⟩
}
```

Figure 2. *Main procedure.* Function ***combinedAbstraction***() takes a netlist and returns either (i) a counter-example (if the property fails) or (ii) the best abstraction produced at the point where resources were exhausted. We leave it unspecified what precise limits to use, but examples include a bound on the depth of the unrolling, the CPU time, or the number of propagations performed by the SAT solver. Function ***refineAbstraction***() will use the latest counterexample stored in T (by ***solve***(), if the last call was SAT) to grow the abstraction. Ternary (or X-valued) simulation is used to shrink the support of the counterexample. Abstract flops that could be removed from the support (i.e. putting in an X did not invalidate the counterexample) are kept abstract; all other flops are concretized. When simulating under an abstraction, abstract flops don't use the value of their input signal, but instead the value of the counterexample produced by the SAT solver (where the flop is a free variable).

			Abstr. Size (flops)				**Abstr. Time** (sec)				**Proof Time** (sec)				
Bench.	#Ands	#Flops	New	New'	ABC	Hyb.	New	New'	ABC	Hyb.	New	New'	ABC	Hyb.	No Abs.
T0	57,560	1,549	2	4	2	6	0.1	0.1	0.3	0.5	0.1	0.1	0.1	0.2	0.4
T1	57,570	1,548	15	15	15	15	1.1	**0.7**	2.3	9.7	0.9	0.9	0.9	2.8	3.5
S0	2,351	1,376	**112**	157	174	–	**0.1**	0.3	2.0	–	8.7	129.7	**5.3**	–	21.2
S1	2,371	1,379	**136**	170	167	–	0.1	0.1	0.6	–	**57.8**	162.9	104.9	–	188.1
S2	3,740	1,526	**83**	123	113	187	0.3	**0.1**	0.6	26.0	**1.1**	37.1	11.8	106.7	4.3
D0	8,061	1,026	107	112	**106**	–	**3.0**	3.3	15.9	–	6.9	19.6	**4.9**	–	7.9
D1	7,262	1,020	139	139	139	139	1.2	1.2	4.4	**0.9**	0.3	0.3	0.3	2.9	0.6
M0	17,135	1,367	179	179	180	**178**	6.8	**6.3**	18.5	206.5	0.2	0.2	0.2	6.3	0.7
I0	1,241	1,104	59	57	**50**	–	0.5	**0.1**	0.6	–	2.0	1.9	**0.7**	–	5.8
I1	395,150	25,480	24	21	21	33	5.5	1.3	**1.1**	16.3	0.0	0.0	0.0	0.3	22.1
I2	5,589	1,259	45	**44**	51	–	1.5	**0.5**	1.5	–	6.2	**5.7**	6.8	–	18.0
I3	5,616	1,259	49	**47**	52	–	1.2	**0.4**	1.5	–	**5.9**	6.5	6.2	–	19.1
I4	394,907	25,451	79	**72**	100	–	64.3	**19.3**	30.9	–	**5.1**	15.0	17.9	–	–
I5	5,131	1,227	49	44	**38**	59	0.5	**0.1**	0.4	202.2	2.2	**0.2**	0.4	20.2	1.6
A0	35,248	2,704	**61**	68	95	81	1.8	**1.6**	6.3	6.9	18.9	**12.0**	35.7	18.3	43.2
A1	35,391	2,738	56	56	62	83	2.3	**1.7**	4.9	11.6	15.7	13.1	31.1	**6.9**	29.5
A2	35,261	2,707	8	8	18	24	0.1	0.1	0.2	0.8	0.0	0.0	0.0	0.2	0.6
A3	35,416	2,741	**59**	70	79	83	**2.2**	2.4	7.9	104.0	21.2	**11.5**	79.0	12.3	52.2
A4	35,400	2,741	**63**	65	67	101	2.5	**2.1**	4.4	34.4	**11.9**	20.0	36.2	12.1	34.6
F0	863,248	40,849	3	3	3	–	**1.0**	2.0	3.5	–	0.0	0.0	0.0	–	48.2
F1	863,251	40,850	4	8	4	–	**1.5**	4.7	7.0	–	0.0	2.2	0.0	–	100.6
F2	863,254	40,851	5	9	5	–	**3.9**	6.1	9.4	–	0.0	2.4	0.0	–	110.1

Table I. *Evaluation of abstraction techniques.* Four implementations of hybrid counterexample- and proof-based abstraction were applied to 22 benchmarks of more than 1000 flops, all for which the property holds. In *New'*, PBA was only applied to the final iteration (to be closer to the ABC implementation). The first section of the table shows the size of the designs. The second section shows, for each implementation, the size of the smallest abstraction it produced that was good enough to prove the property. The third and fourth sections show the time to compute the abstraction, and the time to prove the property using interpolation based modelchecking, with the very last column showing interpolation on the original unabstracted design. Benchmarks with the same first letter denote different properties of the same design. Timeout was set to 500 seconds.

speedup tends to be more significant for harder problems with higher timeouts.

The overall conclusion is that small abstractions help the proof-engine. However, there are cases where a tighter abstraction led to significantly longer runtimes than a looser one (although that effect did not manifested itself in this benchmark set). This can partly be explained by the underlying random nature of interpolant-based model checking, but it should also be recognized that replacing flops with PIs introduces more behaviors, which means the SAT-solver has to prove a more general theorem. Occasionally this can be detrimental, and offset the benefit of the reduced amount of logic that needs to be analyzed. Altogether, it emphasizes that abstraction should be used in good orchestration with other verification techniques.

VII. Acknowledgments

This work was supported in part by SRC contracts 1875.001 and 2057.001, NSF contract CCF-0702668, and industrial sponsors: Actel, Altera, Calypto, IBM, Intel, Intrinsity, Magma, Mentor Graphics, Synopsys (Synplicity), Tabula, Verific, and Xilinx.

References

[1] N. Amla and K. McMillan. **A Hybrid of Counterexample-based and Proof-based Abstraction**. In *FMCAD*, 2004.

[2] N. Amla and K. McMillan. **Combining Abstraction Refinement and SAT-based Model Checking**. In *TACAS*, 2007.

[3] P. Chauhan, E. Clarke, J. Kukula, S. Sapra, H. Veith, and D. Wang. **Automated Abstraction Refinement for Model Checking Large State Spaces Using SAT-based Conflict Analysis**. In *FMCAD*, 2002.

[4] N. Een, A. Mishchenko, and N. Sorensson. **Applying Logic Synthesis for Speeding Up SAT**. In *SAT07*, volume 4501 of *LNCS*, 2007.

[5] Niklas Een and Niklas Sörensson. **The MiniSat Page**. *http://minisat.se*.

[6] Berkeley Logic Synthesis Group. **ABC: A System for Sequential Synthesis and Verification**. *http://www.eecs.berkeley.edu/~alanmi/abc/*, v00127p.

[7] A. Gupta, M. Ganai, Z. Yang, and P. Ashar. **Iterative Abstraction Using SAT-based BMC with Proof Analysis**. In *ICCAD*, 2003.

[8] R. P. Kurshan. **Computer-Aided-Verification of Coordinating Processes**. In *Princeton Univ. Press*, 1994.

[9] B. Li and F. Somenzi. **Efficient Abstraction Refinement in Interpolation-Based Unbounded Model Checking**. In *TACAS*, 2006.

[10] K. McMillan. **Interpolation and SAT-based Model Checking**. In *CAV*, 2003.

[11] K. McMillan and N. Amla. **Automatic Abstraction without Counterexamples**. In *TACAS*, 2003.

[12] D. Vroon P. Manolios. **Efficient Circuit to CNF Conversion**. In *SAT*, 2007.

[13] D. Wang, P. Jiang, J. Kukula, Y. Zhu, T. Ma, and R. Damiano. **Formal property verification by abstraction refinement with formal, simulation and hybrid engines**. In *DAC*, 2004.

```
Trace::Trace(Netlist& N0) {
    N = N0;
    f2s[F.True] = S.True;
}

Wire_F Trace::insert(int frame, Wire_N w) {
    Wire_F ret = n2f[frame][w];
    if (ret == WIRE_UNDEF) {
        if      (w == N.True)      { ret = F.True; }
        else if (type(w) == PI)    { ret = F.add_PI(); }
        else if (type(w) == AND)   { ret = F.add_And(insert(frame, w_0), insert(frame, w_1)); }
        else if (type(w) == FLOP)  { ret = F.add_PI(); if (w ∈ abstr) insertFlop(frame, w, ret); }
        n2f[frame][w] = ret;
    }
    return ret ^ sign(w);                    – interpretation: (w ^ b) ≡ (b ? ¬w : w)
}

void Trace::insertFlop(int frame, Wire_N w_flop, Wire_F f) {
    Wire_F f_in = (frame == 0) ? ¬F.True : insert(frame−1, w_in);
    Lit p = clausify(f_in);
    Lit q = clausify(f);
    Lit a = act_lits[w_flop];
    if (a == LIT_UNDEF) {
        a = S.newLit();
        act_lits[w_flop] = a; }
    S.addClause({¬a, ¬p, q});
    S.addClause({¬a, p, ¬q});                – we've now added: a → (p ↔ q)
}

void Trace::extendAbs(Wire_N w_flop) {
    abstr = abstr ∪ {w_flop};
    for (int frame = 0; frame < n2f.size(); frame++) {
        Wire_F f = n2f[frame][w_flop];
        if (f != WIRE_UNDEF)                 – f is either undefined or a PI
            insertFlop(frame, w_flop, f);
    }
}
```

Figure 3. *Unrolling the netlist.* Method ***insert***() will recursively add the logic feeding *w* to netlist *F*. Flops that are concrete will be traversed across time-frames, but not abstract flops. Each flop that is introduced to *F* is given an *activation literal*. If this literal is set to TRUE, the flop will connect to its input; if it is set to FALSE, the flop acts as a PI. Activation literals are used to implement the proof-based abstraction, and to disable flops when the abstraction shrinks. At frame 0, flops are assumed to be initialized to zero. The purpose of ***extendAbs***() is to grow the abstraction by one flop, adding the missing logic for all time frames.

```
Lit Trace::clausify(Wire_F f) {
    Lit ret = f2s[f];              – map ignores the sign of 'f'
    if (ret == LIT_UNDEF) {
        if (type(f) == PI)
            ret = S.newLit();
        else if (type(f) == AND) {
            – Standard Tseitin clausification
            Lit x = clausify(f_0);
            Lit y = clausify(f_1);
            ret = S.newLit();
            S.addClause({x, ¬ret});
            S.addClause({y, ¬ret});
            S.addClause({¬x, ¬y, ret});
        }
        f2s[f] = ret;
    }
    return ret ^ sign(f);
}

bool Trace::solve(WSet_F f_disj) {
    Lit q = S.newLit();
    S.addClause({¬q} ∪ {clausify(f) | f ∈ f_disj});
    assumps = {q} ∪ {act_lits[w] | act_lits[w] != LIT_UNDEF && w ∈ abstr};
    bool result = S.solve(assumps);
    if (result) ⟨store SAT model⟩
    else        abstr = abstr \ {w | type(w) == FLOP && w ∉ S.conflict};      – this line does PBA
    S.addClause({¬q});     – forever disable temporary clause
    return result;
}

Cex Trace::getCex(int depth) {
    return ⟨use maps 'n2f' and 'f2s' to translate the last SAT model
            into 0/1/X values for the PIs and Flops of frames 0..depth⟩
}
```

Figure 4. *SAT-Solving.* Method ***clausify***() translates the logic of F into CNF for the SAT-solver using the Tseitin transformation. The above procedure can be improved, e.g., by the techniques of [4], [12]. Method ***solve***() takes a disjunction of wires in F and searches for a satisfying assignment to that disjunction. Because only unit assumptions can be passed to ***solveSat***(), a literal q is introduced to represent the disjunction, and a temporary clause is added. Disabling the clause afterwards will in effect remove it. The activation literals of the current abstraction are passed together with q as assumptions to ***solveSat***(). The SAT-solver will give back either a satisfying assignment (stored for later use by ***getCex***()), or a conflict clause expressing which of the assumptions were used for proving UNSAT. This set is used to perform PBA. In computing *assumps*, we note that "&& $w \in abstr$" is necessary if PBA has shrunken the abstraction. In the experimental section, a variant (column *New'* in Table I) is evaluated where PBA is not applied inside ***solve***(). The set of redundant flops is still computed as above, and remembered. When the resource limit is reached, those flops that were redundant in the final UNSAT call are removed. In essence, the variant corresponds to an incremental CBA implementation with a final trimming of the absraction by PBA.

Predicate Abstraction with Adjustable-Block Encoding

Dirk Beyer
Simon Fraser University / University of Passau

M. Erkan Keremoglu
Simon Fraser University, B.C., Canada

Philipp Wendler
University of Passau, Germany

***Abstract*—Several successful software model checkers are based on a technique called single-block encoding (SBE), which computes costly predicate abstractions after every single program operation. Large-block encoding (LBE) computes abstractions only after a large number of operations, and it was shown that this significantly improves the verification performance. In this work, we present adjustable-block encoding (ABE), a unifying framework that allows to express both previous approaches. In addition, it provides the flexibility to specify any block size between SBE and LBE, and also beyond LBE, through the adjustment of one single parameter. Such a unification of different concepts makes it easier to understand the fundamental properties of the analysis, and makes the differences of the variants more explicit. We evaluate different configurations on example C programs, and identify one that is currently the best.**

I. Introduction

Software model checking has been proven successful for increasing the quality of computer programs [2]. Several fundamental concepts were invented in the last decade which made it possible to scale the technology from tiny examples to real programs, e.g., device drivers [4], and to significantly improve the analysis precision, compared to traditional data-flow analyses. Predicate abstraction was introduced as an appropriate abstract domain [17], counterexample-guided abstraction refinement (CEGAR) makes it possible to automatically learn new facts to track [12], lazy abstraction performs expensive refinements only on relevant program paths [19], and interpolation is a sucessful technique to identify a small number of predicates that suffice to eliminate imprecise paths [15], [18].

The software model checker BLAST is an example of a tool that implements all of the above-mentioned concepts [7]. Such a tool implementation performs a reachability analysis along the edges of the control-flow automaton (CFA). The program counter is explicitly represented, and the data state is symbolically represented using predicates. The intermediate results are stored in an abstract reachability graph (ARG). Abstract successor states are obtained by computing the predicate abstraction of the strongest postcondition for a program operation, which involves querying a theorem prover. This category of implementing predicate abstraction can be characterized as *single-block encoding* (SBE), because every single control-flow edge of the program is transformed into a formula that is used for computing the abstract successor state. For a more detailed illustration of the general SBE approach on a concrete example, we refer the reader to the overview article [7].

* This research was supported in part by the Canadian NSERC grant RGPIN 341819-07.

Recently, a new approach was introduced which encodes many CFA edges into one formula, for computing the abstract successor. This approach is called *large-block encoding* (LBE) [6], and transforms the original CFA into a new, summarized CFA in which every edge represents a large subgraph (of the original CFA) that is free of loops. Solvers for satisfiability modulo theories (SMT) had continuously improved their expressiveness and performance, but the SBE approach did not take advantage of this additional power. Therefore, it was time to explore LBE, where a large part of the computational burden of the reachability analysis is delegated to an SMT solver. The experiments showed that LBE not only has a much better performance, but even a better precision (because it is feasible to use boolean instead of cartesian predicate abstraction). However, LBE has two drawbacks: First, it operates on a modified CFA which makes combinations with other abstract domains that operate on single edges impossible. Second, LBE is just one particular choice for how much of the program is encoded in one block and this choice is hard-coded into the verifier and cannot be changed. Our work addresses the need to explore the large space of choices from SBE to LBE, and also beyond LBE.

This article contributes a new approach that is called *adjustable-block encoding* (ABE), which unifies SBE and LBE in one single formalism and fills the gap of missing configurations. This new formalism, together with the corresponding tool implementation, makes it possible to perform experiments which were not possible before, i.e., in which the block encoding is adjustable as a parameter. ABE works on the original CFA and constructs the formulas for large blocks *on-the-fly* during the analysis, and in parallel to other domains (product domains). The number of operations that are encoded in one formula per abstraction step is freely adjustable using a so called block-adjustment operator. By modifying this parameter, ABE can not only operate like SBE or LBE, but can also express configurations with block encodings between SBE and LBE, as well as block encodings larger than LBE.

In our predicate analysis with adjustable-block encoding, every abstract state has two formulas to store the abstract data state: an abstraction formula and a path formula. The successor computation can operate in two different modes, either in abstraction mode or in non-abstraction mode. In a first step (same for both modes) the strongest postcondition for the path formula of the predecessor and the program operation is (syntactically) constructed as formula. In non-abstraction mode, this formula is stored as the path formula in the new state, and the abstraction formula is just copied.

```
1 int main() {
2    int i = 0;
3    while (i < 2) {
4       i++;
5    }
6    if (i != 2) {
7       ERROR: return 1;
8    }
9 }
```

Fig. 1. Simple example program

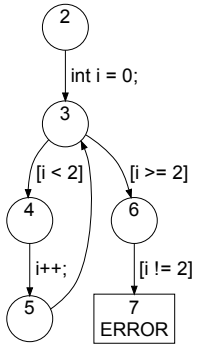


Fig. 2. Corresponding CFA

In abstraction mode, the boolean predicate abstraction of the formula is computed and stored as abstraction formula in the new state, and the path formula is set to *true*. At meet points in the control flow, and if the analysis is in non-abstraction mode, the path formulas of the two branches are combined via disjunction (resulting in a disjunctive path formula). In other words, as long as the analysis operates in non-abstraction mode, a disjunctive path formula is constructed that represents all program operations since the last abstraction formula was computed in abstraction mode. The mode is determined by the block-adjustment operator (analysis parameter).

Availability. Our experiments (implementation, benchmarks, logs) are available at `http://www.sosy-lab.org/~dbeyer/cpa-abe`. The archive includes an executable copy of the CPACHECKER system. For the complete system, cf. the CPACHECKER website.

Example. We illustrate ABE on the simple program in Fig. 1. Figure 2 shows the corresponding CFA (assume(p) is represented by [p]; we removed irrelevant parts from which the error location is not reachable). Nodes represent program locations and arrows represent program operations. We consider a predicate precision (the set of predicates that are tracked) that contains the predicates $i = 0$, $i = 1$, and $i = 2$. First we consider a block-adjustment operator that implements LBE on-the-fly, i.e., abstracting at loop heads and at the error location. The abstract reachability graph (ARG) is shown in Fig. 3. Nodes represent abstract states, and the numbers in the node are the CFA program location (top) and the unique state identifier (bottom). Nodes that are filled in grey represent abstraction states, and their abstraction formula is shown in the box attached to the abstraction state. Nodes with dashed circles represent abstract states that the analysis determines as unreachable (i.e., the result of the abstraction compuation is $false$). Such states are not added to the set of reachable states, therefore they do not have a unique state identifier. Note that the number of grey nodes shows exactly how many (costly) abstraction computations were necessary.

The analysis starts in non-abstraction mode, and is initialized with the formula *true* for both the abstraction formula ψ and the path formula φ. The analysis explores the path from location 2 to 3, creating abstract state $\frac{3}{2}$. Since location 3 is a loop head, state $\frac{3}{2}$ is an abstraction state and the computed abstraction formula is $i = 0$, the path formula φ is re-set to *true*. Locations 4 and 5 are no loop heads, so no abstraction

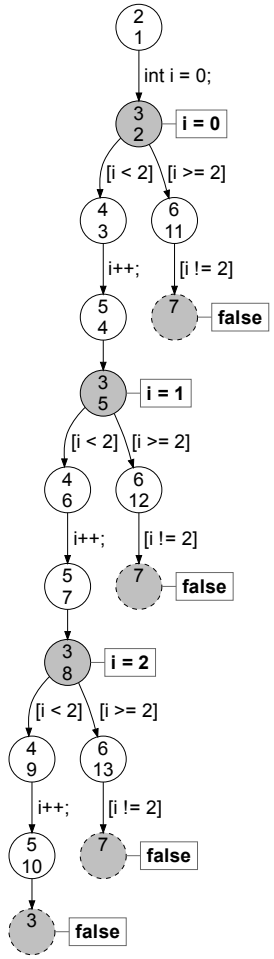


Fig. 3. ARG after analysis with large-block encoding (LBE)

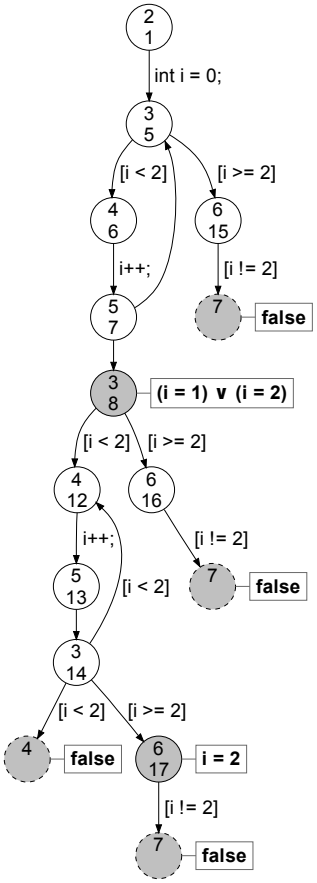


Fig. 4. ARG after analysis with blocks of length 7

is computed and instead only the path formula is extended by the operations on the edges to states $\frac{4}{3}$ and $\frac{5}{4}$. The abstraction formula is copied from their respective predecessor. When the analysis re-encounters location 3, an abstraction is computed again, this time with $i = 1$ as the result (state $\frac{3}{5}$). This process continues until the result of the abstraction computation is *false* (for the successor of $\frac{5}{10}$), which means that the new abstract state is not reachable and analysis can stop exploring this path. Note that $\frac{4}{9}$ and $\frac{5}{10}$ are already unreachable, but the analysis does not detect this, because the abstraction formula is not computed for such non-abstraction states. However, this does not cause a problem because all computations needed for the construction of non-abstraction states like $\frac{5}{10}$ are inexpensive compared with the cost of abstraction computations. The exploration of the remaining paths (those through location 6) is similar. At location 7, an abstraction is always computed because it is the error location, and thus the analysis checks the reachability of abstract states at this location. No such abstraction state is reachable, thus the program is safe.

Now we consider a block-adjustment operator that forces an abstraction computation if the longest path represented by the current path formula has length 7. The ARG is shown in Fig. 4. The analysis starts similarly to the previous example.

However, when it first encounters location 3 it does not compute an abstraction because the condition of the block-adjustment operator (length 7) is not yet fulfilled. Instead, it creates a non-abstraction state $\frac{3}{2}$ (which does not occur in the figure because it is later subsumed by the result of a merge). The same holds for $\frac{4}{3}$ and $\frac{5}{4}$. When the analysis reaches location 3 again, it creates state $\frac{3}{5}$ and immediately merges it with the existing state $\frac{3}{2}$, because both share the same location and the same abstraction formula (which is still the initial one). Therefore, abstract state $\frac{3}{2}$ is removed. The path formula of the new (merged) abstract state is the disjunction of the path formulas of both states, i.e., representing both the paths 2-3 and 2-3-4-5-3. The same happens at locations 4 and 6, creating states $\frac{4}{6}$ and $\frac{5}{7}$. But when the analysis encounters location 3 for the third time, the path formula represents the paths 2-3-4-5-3 and 2-3-4-5-3-4-5-3. The latter path contains 7 edges, thus an abstraction is computed. At this abstraction state, either the predicate $i = 1$ or the predicate $i = 2$ is true. Continuing, the analysis constructs the non-abstraction states $\frac{4}{9}$, $\frac{5}{10}$, $\frac{3}{11}$, $\frac{4}{12}$, $\frac{5}{13}$ and $\frac{3}{14}$. Again, the former three states are removed from the set of reached states because they are merged into the latter three states. All these six states are not merged with the previous states although some of them share a common program location, because the abstraction formula of the new states differs from the abstraction formula of the previous states. Also, abstraction states like $\frac{3}{8}$ are never changed by merge operations. The path formula of $\frac{3}{14}$ represents the paths 3-4-5-3 and 3-4-5-3-4-5-3. Thus, when the successors of this state are created, the length of the longest path represented by the path formula reaches 7 and an abstraction is computed. The successor at location 4 has the abstraction formula *false*, thus it is not added to the reached states. The abstraction formula of $\frac{6}{17}$ is $i = 2$. The analysis continues with the remaining paths, correctly determining that all paths leading to the error location are infeasible. Therefore the program is again reported as safe.

By choosing a good block-adjustment operator, the size of the blocks (the regions of the ARG that do not contain abstraction states) and the number of abstraction computations can be optimized. Larger blocks lead to fewer costly abstraction computations, but the problems given to the SMT solver are harder because the path formulas are more complex. With ABE, the reachable states do not necessarily form a tree, like for SBE and for LBE with preprocessing. However, note that the abstraction states still form a tree in both examples. In fact, this is true for all choices of the block-adjustment operator.

Related Work. Our work is based on the idea of stepwise exploring the reachable states of the program, using CEGAR to refine the abstraction, and symbolic techniques to operate on abstract data states. Existing example implementations of this category are SBE-based (SLAM [4] and BLAST [7]) or LBE-based [6]. The goal of our ABE-based approach is to make the configuration of the algorithm flexible, i.e., (1) to subsume the previous approaches (SBE, LBE) and (2) enable even larger encodings such that it is freely adjustable how much of the state-space exploration is done symbolically by the SMT solver. A different category of verification tools is based on the idea of performing a fully symbolic search. Examples are the model checker SATABS [14], which is based on CEGAR but operates fully symbolically, and the bounded model checker CBMC [13], which is targeted at finding bugs instead of proving safety. Fully symbolic search is also applied to large generated verification conditions, for example in the extended static checkers CALYSTO [1] and SPEC# [5]. The algorithm of McMillan is also based on the idea of lazy abstraction, but never performes predicate abstraction-based successor computations [21]. Our approach can be characterized as based on predicate abstraction [17], CEGAR [12], lazy abstraction [19], and interpolation [18].

II. Preliminaries

A. Programs and Control-Flow Automata

We restrict the presentation to a simple imperative programming language, where all operations are either assignments or assume operations, and all variables range over integers.[1] We represent a program by a *control-flow automaton* (CFA). A CFA $A = (L, G)$ consists of a set L of program locations, which model the program counter l, and a set $G \subseteq L \times Ops \times L$ of control-flow edges, which model the operations that are executed when control flows from one program location to another. The set of program variables that occur in operations from Ops is denoted by X. A *program* $P = (A, l_0, l_E)$ consists of a CFA $A = (L, G)$ (models the control flow of the program), an initial program location $l_0 \in L$ (models the program entry), and a target program location $l_E \in L$ (models the error loc.).

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

A *path* σ is a sequence $\langle (op_1, l_1), ..., (op_n, l_n) \rangle$ of pairs of operations and locations. The path σ is called *program path* if σ starts with l_0 and for every i with $0 < i \leq n$ there exists a CFA edge $g = (l_{i-1}, op_i, l_i)$, i.e., σ represents a syntactical walk through the CFA. The *concrete semantics for a program*

[1] Our implementation CPACHECKER works on C programs that are given in CIL intermediate language [22]; non-recursive function calls are supported.

path $\sigma = \langle (op_1, l_1), ..., (op_n, l_n) \rangle$ is defined as the successive application of the strongest postoperator for each operation: $\mathsf{SP}_\sigma(\varphi) = \mathsf{SP}_{op_n}(...\mathsf{SP}_{op_1}(\varphi)...)$. The formula $\mathsf{SP}_\sigma(\varphi)$ is called *path formula*. The set of concrete states that result from running σ is represented by the pair $(l_n, \mathsf{SP}_\sigma(true))$. A program path σ is *feasible* if $\mathsf{SP}_\sigma(true)$ is satisfiable. A concrete state (l_n, c_n) is called *reachable* if there exists a feasible program path σ whose final location is l_n and such that $c_n \models \mathsf{SP}_\sigma(true)$. A location l is reachable if there exists a concrete state c such that (l, c) is reachable. A program is *safe* if l_E is not reachable.

B. Predicate Precision and Boolean Predicate Abstraction

Let $\mathscr{P}$ be a set of predicates over program variables in a quantifier-free theory $\mathscr{T}$. A *formula* φ is a boolean combination of predicates from $\mathscr{P}$. A *precision for formulas* is a finite subset $\pi \subset \mathscr{P}$ of predicates. A *precision for programs* is a function $\Pi : L \to 2^{\mathscr{P}}$, which assigns to each program location a precision for formulas. The *boolean predicate abstraction* $(\varphi)^\pi$ *of a formula* φ is the strongest boolean combination of predicates from the precision π that is entailed by φ. Such a predicate abstraction of a formula φ, which represents a region of concrete program states, is used as an *abstract data state* (i.e., an abstract representation of the region) in program verification. For a formula φ and a precision π, the boolean predicate abstraction $(\varphi)^\pi$ of φ can be computed by querying an SMT solver in the following way: For each predicate $p_i \in \pi$, we introduce a propositional variable v_i. Now we ask the solver to enumerate all satisfying assignments of $v_1, ..., v_{|\pi|}$ in the formula $\varphi \wedge \bigwedge_{p_i \in \pi}(p_i \Leftrightarrow v_i)$. For each satisfying assignment, we construct a conjunction of all predicates from π whose corresponding propositional variable occurs positive in the assignment. The disjunction of all such conjunctions is the boolean predicate abstraction for φ. An abstract strongest postoperator for a predicate abstraction with precision π and a program operation *op*, which transforms an abstract data state φ into its successor φ', can be defined by applying first the strongest postcondition operator and then the predicate abstraction, i.e., $\varphi' = (\mathsf{SP}_{op}(\varphi))^\pi$. For more details, we refer the reader to the work of Ball et al. and Lahiri et al. [3], [20].

III. Adjustable-Block Encoding

In ABE, the predicate abstraction is not computed after every CFA edge, but only at certain abstract states, which we call *abstraction states* (the other abstract states are called non-abstraction states). On paths between two abstraction computations, the strongest postcondition of the path(s) is stored in a second formula of the abstract state, which we call *disjunctive path formula*. Therefore, every abstract state of ABE contains two formulas ψ and φ, where the *abstraction formula* ψ is the result of an abstraction computation and the disjunctive path formula φ represents the strongest postcondition since the last abstraction state was computed. Given a CFA edge $g = (l, op, l')$ and an abstract state with ψ and φ, the abstract successor either extends the path formula φ only (which is a purely syntactical operation), or computes a new abstraction formula ψ and resets φ. Where to compute abstractions (and thus the block size) is determined by the so-called block-adjustment operator blk as follows: If $\mathsf{blk}(e, g)$ returns *false* (no abstraction computation, i.e., the abstract state e is a non-abstraction state), the abstract successor contains ψ (unchanged) and $\mathsf{SP}_{op}(\varphi)$ (as the new φ). If $\mathsf{blk}(e, g)$ returns *true* (e is abstraction state), the abstract successor contains the formula that results from the abstraction of $\psi \wedge \varphi$ as the new abstraction formula and *true* as the new disjunctive path formula. If $\psi \wedge \varphi$ is unsatisfiable for an abstract state e, then e is not reachable.

A. CPA for Adjustable-Block Encoding

We formalize adjustable-block encoding (ABE) as a configurable program analysis (CPA) [8]. This allows us to use the flexibility of the CPA operators to describe how the analysis operates without changing the general iteration algorithm (cf. Alg. *CPA*). The configurable program analysis for adjustable-block encoding $\mathbb{D} = (D, \rightsquigarrow, \mathsf{merge}, \mathsf{stop})$ consists of an abstract domain D, a transfer relation $\rightsquigarrow$, a merge operator merge, and a stop operator stop, which are defined as follows. (Given a program $P = (A, l_0, l_E)$, we use X for denoting the set of program variables occuring in P, $\mathscr{P}$ for the set of quantifier-free predicates over variables from X, and $\Pi : L \to 2^{\mathscr{P}}$ for the precision of the predicate abstraction.)

1. The **abstract domain** $D = (C, \mathscr{E}, [\![\cdot]\!])$ is a tuple that consists of a set C of concrete states, a semi-lattice $\mathscr{E} = (E, \top, \sqsubseteq, \sqcup)$, and a concretization function $[\![\cdot]\!] : E \to C$. The lattice elements $e \in E$ are also called abstract states, and are tuples $(l, \psi, l^\psi, \varphi) \in (L \cup \{l_\top\}) \times \mathscr{P} \times (L \cup \{l_\top\}) \times \mathscr{P}$, where l models the program counter, the abstraction formula ψ is a boolean combination of predicates that occur in Π, l^ψ is the location at which ψ was computed, and φ is a disjunctive path formula representing some or all paths from l^ψ to l. Note that an abstraction state has always $l = l^\psi$ and $\varphi = true$. The top element of the lattice is the abstract state $\top = (l_\top, true, l_\top, true)$. The partial order $\sqsubseteq \subseteq E \times E$ is defined such that for any two elements $e_1 = (l_1, \psi_1, l^{\psi}_1, \varphi_1)$ and $e_2 = (l_2, \psi_2, l^{\psi}_2, \varphi_2)$ from E the following holds:

$$e_1 \sqsubseteq e_2 \iff (e_2 = \top) \vee ((l_1 = l_2) \wedge (\psi_1 \wedge \varphi_1 \Rightarrow \psi_2 \wedge \varphi_2))$$

The join operator $\sqcup : E \times E \to E$ yields the least upper bound of the two operands, according to the partial order.

2. The **transfer relation** $\rightsquigarrow \subseteq E \times G \times E$ contains all tuples (e, g, e') with $e = (l, \psi, l^\psi, \varphi)$, $e' = (l', \psi', l^{\psi'}, \varphi')$ and $g = (l, op, l')$ for which the following holds:

$$\begin{cases} (\varphi' = true) \wedge \left(\psi' = (\mathsf{SP}_{op}(\varphi \wedge \psi))^{\Pi(l')}\right) \wedge (l^{\psi\prime} = l') \\ \qquad\qquad\qquad\qquad\qquad\qquad \text{if } \mathsf{blk}(e, g) \vee (l' = l_E) \\ (\varphi' = \mathsf{SP}_{op}(\varphi)) \wedge (\psi' = \psi) \wedge (l^{\psi\prime} = l^\psi) \qquad \text{otherwise} \end{cases}$$

The 'mode' of the transfer relation, i.e., when to compute abstractions, is determined by a block-adjustment operator

Algorithm 1 $CPA(\mathbb{D}, e_0)$ (taken from [8])

Input: a CPA $\mathbb{D} = (D, \rightsquigarrow, \mathsf{merge}, \mathsf{stop})$,
an initial abstract state $e_0 \in E$, where E denotes
the set of elements of the lattice of D
Output: a set of reachable abstract states
Variables: a set reached of elements of E,
a set waitlist of elements of E

```
1: waitlist := {e_0}
2: reached := {e_0}
3: while waitlist ≠ ∅ do
4:    choose e from waitlist
5:    waitlist := waitlist \ {e}
6:    for each e' with e ⇝ e' do
7:       for each e'' ∈ reached do
8:          // combine with existing abstract state
9:          e_new := merge(e', e'')
10:         if e_new ≠ e'' then
11:            waitlist := (waitlist ∪ {e_new}) \ {e''}
12:            reached := (reached ∪ {e_new}) \ {e''}
13:      if ¬ stop(e', reached) then
14:         waitlist := waitlist ∪ {e'}
15:         reached := reached ∪ {e'}
16: return reached
```

$\mathsf{blk} : E \times G \to \mathbb{B}$, which maps an abstract state e and a CFA edge g to *true* or *false*. The operator blk is given as parameter to the analysis. The second case does not compute an abstraction, but purely syntactically assembles the precise strongest postcondition [2]. Thus, the choice of blk determines the block-encoding (i.e., how much to collect in the path formula before abstraction). Most instances of the block-adjustment operator will eventually return *true* for every path through the CFA, otherwise the analysis might not terminate if the program contains loops. The precision of the predicate abstraction can vary between program locations (parsimonious precision [7]).

3. The **merge operator** $\mathsf{merge} : E \times E \to E$ for two abstract states $e_1 = (l_1, \psi_1, {l^\Psi}_1, \varphi_1)$ and $e_2 = (l_2, \psi_2, {l^\Psi}_2, \varphi_2)$ is defined as follows: $\mathsf{merge}(e_1, e_2) =$

$$\begin{cases} (l_2, \psi_2, {l^\Psi}_2, \varphi_1 \vee \varphi_2) & \text{if } (l_1 = l_2) \wedge (\psi_1 = \psi_2) \wedge ({l^\Psi}_1 = {l^\Psi}_2) \\ e_2 & \text{otherwise} \end{cases}$$

This operator combines the two abstract states using a disjunctive path formula, if the location of the abstract states is the same and they were derived from the same abstraction states, i.e., the abstraction formulas are equal and were computed at the same program location [3].

4. The **stop operator** $\mathsf{stop} : E \times 2^E \to \mathbb{B}$ checks if e is covered by another state in the reached set:

$$\forall e \in E, R \subseteq E : \ \mathsf{stop}(e, R) \ = \ \exists e' \in R : \ (e \sqsubseteq e')$$

[2] The strongest postcondition as defined in the preliminaries in fact contains existential quantifiers. However, our implementation of the transfer relation uses an SP that operates on SSA-like quantifier-free formulas.

[3] Two identical abstraction states never exist in the reached set due to the stop operator, which would eliminate the second instance of the same abstraction state before insertion into the reached set. Thus, the ARG restricted to abstraction states still represents a tree (ART).

B. Discussion: SBE, LBE, BMC, and in Between

A fundamental improvement of adjustable-block encoding over the previous work with blocks hard-coded in the pre-processed CFA is that now other abstract domains that the ABE analysis is combined with, can work each with different (perhaps also adjustable) block sizes, which are not dictated by the pre-processed CFA. The great flexibility of ABE results from the possibility to freely choose the blk operator. Two particular possibilities are blk^{sbe} and blk^{lbe}. The operator blk^{sbe} returns always *true*, and thus the transfer computes the predicate abstraction after every CFA edge. The operator blk^{lbe} returns *true* if the successor location of the given edge is the head location of a loop, and thus the transfer computes the predicate abstraction at loop heads. Therefore, the analysis can easily be configured to behave exactly like SBE or LBE. Another possible choice for blk is to compute an abstraction if the length of the longest path represented by the path formula φ of the abstract state exceeds a certain threshold. But the decision made by blk does not necessarily have to be based only on statically available information. We could for example measure the memory consumption of the path formula and compute abstractions if the path formulas become too large. Or we could measure the time needed to compute the abstractions and adjust the block encoding such that a single computation does not take more than a certain amount of time. Thus, one could write a block-adjustment operator that is tailored to the SMT solver that is used, i.e., to delegate problems to the solver that are large enough to benefit from the SMT technology, and at the same time small enough to not overwhelm the solver. In our experiments, we demostrate the usefulness of the ABE approach using a few simple choices for the operator blk. In particular, the experiments indicate that useful block-adjustment operators should respect the control-flow structure of the program.

We did not describe how the precision Π for programs is computed, because we use a standard approach that is based on CEGAR, lazy abstraction, and Craig interpolation. We consider only abstraction states for the abstract reachability tree (ART). The merge operator ensures that the abstraction states form a tree (abstraction states are never changed by merge). The formulas of the error path are the (disjunctive) path formulas that were constructed during the creation of the abstraction states along this path and which were used as inputs for the abstraction computation. The only difference is that now a single formula represents one or several paths between two arbitrary locations of the CFA, and not necessarily only one CFA edge as before. The interpolation will then produce predicates for those locations at which an abstraction was computed, so the new predicates will be used in the next iteration of the analysis if the block-adjustment operator returns the same value. The possiblity of dynamic block-adjustment operators, i.e., determining different abstract states as abstraction states depending on the overall progress of the analysis, raises the interesting question of where to add the predicates extracted from the interpolants. Currently, we refine

the predicate precision of the program only at the abstraction states, but we could in principle also add the predicates to all locations between the previous and current abstraction state.

IV. Experiments

Implementation. We implemented adjustable-block encoding in CPACHECKER, which is a software-verification framework based on configurable program analysis. The tool accepts programs in C Intermediate Language [22] (other C programs can be pre-processed with the tool CIL). CPACHECKER uses MATHSAT [11] and CSISAT [9] as SMT solvers.

Our implementation uses the following optimizations: (1) The feasibility of abstract paths is checked only at abstraction states. This does not negatively affect the precision of the analysis because abstract states with the error location are always abstraction states. (2) Instead of constructing postconditions that include existential quantifiers, we leave the variables in the path formulas and use a simple form of skolemization, which is equivalent to static single-assignment (SSA) form [16] (well-known from compilers). A tutorial-like example of this process is provided in an article about BLAST [7]. (3) When the operator stop checks if an abstract state e is covered by a non-abstraction state e', we do not perform a full SMT check for the implication that the partial order requires; instead we do a quick syntactical check that compares the path formulas of both states. This check will correctly detect that e is covered by e' if e was merged into e', but may fail in other situations. This is sound, and faster than a full SMT check.

Benchmark Programs. We experimented with three groups of C programs, which are similar to those previously used [6]. The first group (`test_locks_*`) was artificially created to show that SBE leads to exponentially many abstract states. Several nested locks are acquired and released in a loop. The number in the name indicates the number of locks in the program. The second group contains several (parts of) drivers from the Windows NT kernel. The third group (`s3_*`) was taken from the SSH suite. The code contains a simplified version of the state machine handling the communication according to the SSH protocol. Both the NT drivers and the SSH examples were pre-processed manually in order to remove heap accesses, and automatically with CIL v1.3.6. The examples with `BUG` in the name have artificially inserted bugs that cause assertions to fail. All examples are included in the CPACHECKER repository together with the used configurations.

All experiments were performed on a machine with 2.8 GHz and 4 GB of RAM. The operating system was Ubuntu 9.10 (64 bit), using Linux 2.6.31 as kernel and OpenJDK 1.6 as Java virtual machine. We used CPACHECKER, branch 'abe', revision 1457, with MATHSAT 4.28 as SMT solver. The times are reported in seconds and rounded to three significant digits. In cases where CPACHECKER needed either more than 1800 s or more than 4 GB of RAM, the analysis was aborted, indicated by "> 1800" or "MO", respectively. CPACHECKER reports the correct verification result in all cases, i.e., a counterexample for all programs with `BUG`, and safety for all other programs.

Configurations. We experimented with different choices of the block-adjustment operator of ABE, which we classify into three categories: (1) we repeat the experiments with LBE from previous work [6], (2) we perform new experiments to explore the spectrum of encodings between SBE and LBE, and (3) we explore new encodings larger than LBE. Our implementation supports functions, and thus we extend the operator blk^{lbe} such that it returns *true* at loop heads *and* function entries/returns. We measure the length of a block that is encoded in an abstract state e as the length (in ops) of the longest path represented in the disjuncive path formula of e.

We have also experimented with cartesian vs. boolean abstraction, and not only re-confirm the results from previous experiments [6] (SBE: cartesian works best, LBE: boolean is best); we conclude that cartesian abstraction becomes unusably imprecise as soon as the block length is more than 1 op. Thus, boolean abstraction must be used for all non-SBE encodings.

A. LBE: Pre-Processed versus On-the-Fly

Due to the overhead that is caused by the on-the-fly encoding and some additional abstraction computations, a certain performance loss is expected. We started our experiments with confirming that the negative impact of the overhead on the performance is not dramatic. The results are reported in

Program	Pre-proc. LBE	LBE (blk^{lbe})
test_locks_5.c	.170	.483
test_locks_6.c	.370	.398
test_locks_7.c	.237	.875
test_locks_8.c	.305	.437
test_locks_9.c	.202	.510
test_locks_10.c	.266	.746
test_locks_11.c	.256	.416
test_locks_12.c	.248	.486
test_locks_13.c	.240	.769
test_locks_14.c	.227	.787
test_locks_15.c	.466	.896
cdaudio1.sim.c	11.7	51.5
diskperf1.sim.c	537	146
floppy3.sim.c	7.04	20.1
floppy4.sim.c	8.35	32.2
kbfiltr1.sim.c	1.27	2.57
kbfiltr2.sim.c	1.73	3.75
cdaudio1_BUG.sim.c	5.26	32.5
floppy3_BUG.sim.c	2.97	11.1
floppy4_BUG.sim.c	4.58	20.1
kbfiltr2_BUG.sim.c	1.96	2.28
s3_clnt_1.sim.c	15.9	14.6
s3_clnt_2.sim.c	12.8	35.4
s3_clnt_3.sim.c	19.5	17.8
s3_clnt_4.sim.c	36.6	9.59
s3_srvr_1.sim.c	16.6	31.2
s3_srvr_2.sim.c	107	86.7
s3_srvr_3.sim.c	109	14.1
s3_srvr_4.sim.c	441	160
s3_srvr_6.sim.c	456	45.7
s3_srvr_7.sim.c	321	136
s3_srvr_8.sim.c	>1800	21.2
s3_clnt_1_BUG.sim.c	1.22	2.81
s3_clnt_2_BUG.sim.c	2.12	2.06
s3_clnt_3_BUG.sim.c	1.26	3.14
s3_clnt_4_BUG.sim.c	2.03	2.54
s3_srvr_1_BUG.sim.c	1.43	1.62
s3_srvr_2_BUG.sim.c	1.55	2.71

TABLE I
COMPARISON OF PRE-PROCESSED LBE WITH ADJUSTED LBE (blk^{lbe})

Program	SBE	$k=10$	$k=20$	$k=30$	$k=40$	$k=50$	$k=60$	$k=70$	$k=80$	$k=90$	$k=100$	LBE
test_locks_5.c	6.36	3.42	1.02	1.29	.367	.695	.397	.292	.587	.468	.507	.483
test_locks_6.c	13.1	3.03	1.90	1.36	.690	.334	.527	.428	.637	.790	.323	.398
test_locks_7.c	34.8	5.71	1.30	3.26	.516	1.06	.800	.326	.591	.355	.807	.875
test_locks_8.c	102	25.8	3.82	1.86	1.20	1.27	.414	.392	.670	.575	.680	.437
test_locks_9.c	298	67.7	12.4	6.97	1.67	1.63	.543	.454	.551	.667	.705	.510
test_locks_10.c	1250	109	7.53	6.59	3.79	1.24	.679	.588	.805	.845	.993	.746
test_locks_11.c	>1800	244	26.7	4.71	6.73	1.71	.906	.992	1.20	.905	.418	.416
test_locks_12.c	>1800	>1800	88.5	20.6	3.08	5.31	1.32	1.04	.995	1.35	.728	.486
test_locks_13.c	>1800	MO	134	71.5	7.12	2.78	2.29	1.48	1.77	1.05	1.09	.769
test_locks_14.c	>1800	MO	>1800	580	19.6	17.6	4.61	2.25	2.07	1.13	.915	.787
test_locks_15.c	>1800	>1800	>1800	>1800	32.2	22.3	23.1	5.56	2.71	2.46	1.40	.896
cdaudio1.sim.c	MO	210	119	51.9	52.9	54.5	49.0	52.6	58.1	53.8	53.5	51.5
diskperf1.sim.c	MO	855	155	171	163	168	158	152	154	146	167	146
floppy3.sim.c	559	80.5	23.6	19.3	25.5	23.1	20.0	21.0	21.1	19.8	17.9	20.1
floppy4.sim.c	MO	212	54.0	28.8	41.4	39.2	35.2	31.7	32.6	32.8	44.6	32.2
kbfiltr1.sim.c	48.2	10.1	3.72	2.66	3.28	2.82	2.15	2.49	1.83	1.89	2.81	2.57
kbfiltr2.sim.c	128	59.1	10.2	5.26	5.90	7.93	4.12	4.56	4.19	4.67	3.94	3.75
cdaudio1_BUG.sim.c	158	106	151	32.6	36.7	38.6	32.7	35.5	40.0	33.4	31.9	32.5
floppy3_BUG.sim.c	75.8	45.9	13.9	12.1	13.9	11.3	11.2	9.38	9.11	10.5	10.4	11.1
floppy4_BUG.sim.c	77.4	150	39.0	16.9	30.4	31.3	26.1	21.3	22.2	23.3	23.1	20.1
kbfiltr2_BUG.sim.c	156	16.5	3.25	4.16	3.22	3.37	2.89	3.22	2.19	2.36	2.27	2.28
s3_clnt_1.sim.c	MO	MO	MO	MO	MO	41.3	27.8	13.4	10.8	445	45.0	14.6
s3_clnt_2.sim.c	MO	MO	MO	MO	MO	34.4	45.0	16.2	12.8	569	49.1	35.4
s3_clnt_3.sim.c	MO	MO	MO	MO	MO	45.7	238	309	24.6	MO	36.4	17.8
s3_clnt_4.sim.c	MO	MO	MO	MO	MO	38.1	17.7	24.2	9.53	441	28.4	9.59
s3_srvr_1.sim.c	>1800	MO	MO	MO	MO	43.7	MO	712	113	MO	47.8	31.2
s3_srvr_2.sim.c	MO	MO	MO	MO	MO	462	33.2	MO	340	MO	98.5	86.7
s3_srvr_3.sim.c	>1800	MO	MO	MO	MO	32.9	11.5	31.1	24.7	MO	MO	14.1
s3_srvr_4.sim.c	>1800	MO	MO	MO	MO	325	56.4	12.1	22.0	MO	45.6	160
s3_srvr_6.sim.c	>1800	MO	MO	MO	MO	MO	83.8	638	MO	50.8	MO	45.7
s3_srvr_7.sim.c	>1800	MO	MO	MO	MO	MO	133	458	MO	MO	315	136
s3_srvr_8.sim.c	>1800	MO	MO	MO	MO	MO	18.9	42.7	26.8	155	565	21.2
s3_clnt_1_BUG.sim.c	667	67.5	20.4	8.78	11.8	3.82	2.39	2.25	2.16	7.87	4.19	2.81
s3_clnt_2_BUG.sim.c	677	135	26.6	14.2	10.2	3.93	3.05	1.94	2.59	6.47	3.58	2.06
s3_clnt_3_BUG.sim.c	653	55.3	18.6	19.6	6.20	3.62	2.72	3.27	1.93	9.87	3.45	3.14
s3_clnt_4_BUG.sim.c	646	78.0	33.8	15.2	5.42	3.73	2.35	1.86	2.68	8.91	3.64	2.54
s3_srvr_1_BUG.sim.c	42.2	14.9	9.44	2.03	3.01	1.49	2.64	2.32	2.35	5.22	1.47	1.62
s3_srvr_2_BUG.sim.c	35.6	60.4	6.18	2.72	4.80	2.65	1.09	2.03	1.61	5.00	3.32	2.71

TABLE II
RESULTS FOR blk^{sbe}, FOR blk_k^{lbe} WITH k FROM 10 TO 100, AND FOR blk^{lbe}

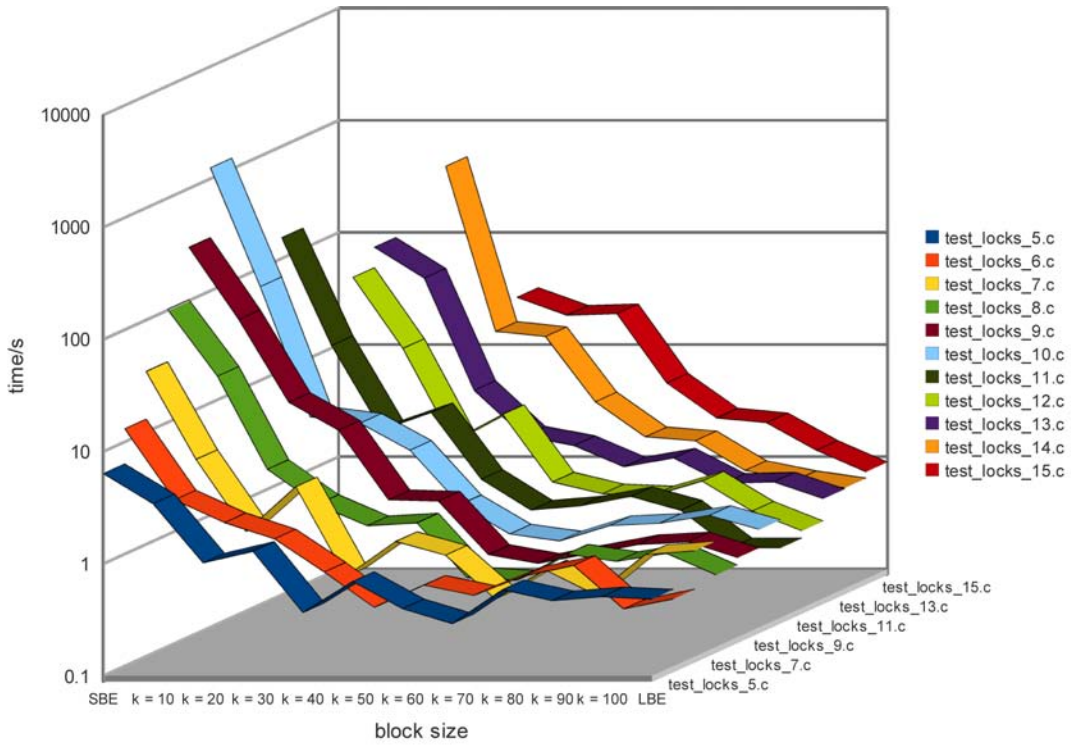


Fig. 5. Results for blk^{sbe}, for blk_k^{lbe} with k from 10 to 100, and for blk^{lbe}

Program	LBE		$k=50$		$k=100$		$k=150$		$k=200$		$k=250$		$k=300$	
cdaudio1_BUG.sim.c	32.5	26989	451	49240	168	6428	11.1	444	12.9	114	15.9	105	13.4	181
floppy3_BUG.sim.c	11.1	4059	51.4	4697	20.1	1207	6.17	217	7.81	99	10.9	112	4.03	43
floppy4_BUG.sim.c	20.1	11066	87.7	7734	43.7	2491	11.8	177	14.2	167	10.4	161	6.63	55
kbfiltr2_BUG.sim.c	2.28	660	3.75	424	1.94	57	1.45	52	2.22	24	1.07	0	1.10	0
s3_clnt_1_BUG.sim.c	2.81	16	14.6	743	MO	-	5.39	38	3.63	33	4.67	10	6.57	20
s3_clnt_2_BUG.sim.c	2.06	20	3.70	146	2.94	51	3.41	28	MO	-	4.48	9	5.07	18
s3_clnt_3_BUG.sim.c	3.14	22	14.6	795	MO	-	MO	-	13.1	44	7.06	7	5.27	7
s3_clnt_4_BUG.sim.c	2.54	22	5.21	218	4.46	78	MO	-	13.2	125	5.32	9	6.17	19
s3_srvr_1_BUG.sim.c	1.62	13	6.01	303	1.28	38	3.24	10	MO	-	1.90	2	2.13	7
s3_srvr_2_BUG.sim.c	2.71	13	4.61	306	MO	-	2.24	10	MO	-	1.97	2	2.01	7
test_locks_5.c	.483	4	1.21	22	1.03	10	2.05	7	1.09	5	1.65	3	2.49	2
test_locks_6.c	.398	4	71.4	2258	2.26	84	2.22	40	2.18	12	1.11	1	1.47	1
test_locks_7.c	.875	4	3.17	120	2.48	17	1.11	1	1.66	10	7.16	26	5.90	60
test_locks_8.c	.437	4	3.33	180	.578	1	1.16	14	8.71	144	1.78	1	5.49	7
test_locks_9.c	.510	4	2.37	89	.880	2	11.2	192	3.49	19	2.62	10	15.5	47
test_locks_10.c	.746	4	1.83	118	.677	10	4.97	144	32.1	114	11.2	38	1.57	1
test_locks_11.c	.416	4	3.85	164	1.59	22	3.80	27	1.75	18	8.51	95	3.19	14
test_locks_12.c	.486	4	33.0	1985	MO	-	1.12	1	17.2	98	1.29	1	185	537
test_locks_13.c	.769	4	24.4	285	324	4628	.626	1	37.8	470	1.48	3	24.1	232
test_locks_14.c	.787	4	179	79	77.1	1202	1.01	2	10.0	220	4.43	22	27.8	107
test_locks_15.c	.896	4	1580	1268	154	2234	1.80	10	4.93	37	367	1255	1.28	1

TABLE III
RESULTS AND NUMBER OF ABSTRACTIONS FOR blk^{lbe} AND FOR blk_k WITH k FROM 50 TO 300

Table I. Column 'Pre-proc. LBE' reports the performance of the previous implementation [6], which first transforms the CFAs of the program in a pre-processing step into new CFAs that reflect the large-block encoding, and then the analysis is performed. Column 'LBE (blk^{lbe})' reports the performance of the new, more flexible implementation, with the block operator adjusted to blk^{lbe}. The simple old implementation is faster on most example programs, as expected, but the difference is not dramatic, and also inconsistent, i.e., there are several examples on which the new approach performs as good or even better. Thus, although it might seem wasteful to explore a huge number of extra states without performing any abstraction, just to assemble the strongest-postcondition formula for the encoded block on-the-fly, this table shows that in fact the overhead is not dramatic.

B. Block Sizes between SBE and LBE

The second set, of novel configurations, is based on the new block-adjustment operator $\mathsf{blk}_k^{lbe} : E \times G \to \mathbb{B}$, which is defined as the disjunction $\mathsf{blk}^{lbe} \vee \mathsf{blk}_k$. The operator blk_k^{lbe} returns *true* if either the longest path represented by the disjunctive path formula of the abstract state is longer than $k \in \mathbb{N} \setminus \{0\}$ or the successor location is a loop/function head. Only a few examples have blocks longer than 100 ops when analyzed with LBE, thus, we analyze the examples for $k \in \{10, 20, 30, 40, 50, 60, 70, 80, 90, 100\}$.

Table II shows the results for the three groups of example programs. The examples `test_locks_*` and the NT drivers show an exponential performance improvement with growing blocks. The diagram with a logarithmic time axis in Fig. 5 illustrates this for the first group of examples. The blocks of blk_k^{lbe} are never longer than in LBE, and thus, there is no further performance improvement beyond a certain program-dependent threshold. The SSH programs with artificial bugs follow the same trend. The safe SSH programs do not follow such a clear trend and instead, their performance extremely depends on the block size. This indicates the superiority of blk^{lbe} over blk_k^{lbe} for blocks that are smaller than in the LBE encoding: the operator blk_k cuts off the formulas at an arbitrary position, ignoring the structure of the control flow and thus destroying the advantage of encoding larger blocks. LBE, i.e., the encoding with the largest blocks, is the only configuration that can verify all examples, and is the best for most examples.

It is important that the block-encoding encloses 'whole structures', i.e., that a block not only contains a few branches, but that it actually contains the branches until they meet again (for example, a whole 'if' structure). This is demonstrated by the fact that encodings smaller than LBE are sometimes not performing well (cf. `s3_clnt_3.c` with $k = 50$ vs. $k = 60$; in particular, note the MO for $k = 90$). This is particularly important for loop unrollings: coverage checks can be done most efficiently at abstraction states, thus, abstraction computations should ideally occur at matching locations of the loop body.

C. Block Sizes larger than LBE

In the third set of experiments, we evaluated new configurations with blocks larger than LBE. It is known that shallow bugs can efficiently be found by a technique called bounded model checking, where programs are unrolled up to a given bound of the length, and a formula is constructed which is satisfiable iff one of the modeled program paths reaches the error location. We apply a similar technique to find bugs using our ABE approach: in this set of experiments, we use the block-adjustment operator $\mathsf{blk}_k : E \times G \to \mathbb{B}$, which returns *true* if the block is k operations long, with $k \in \{50, 100, 150, 200, 250, 300\}$.

Table III reports the time and number of abstraction computations needed to find the error, in the first part of the table. The benefit of ever larger block encodings is clearly indicated. The performance of this configuration for finding bugs is almost comparable to the performance of a highly tuned tool for bounded model checking (BMC) [10]: we analyzed the programs with CBMC [13] and the runtimes were less than 6 s for every NT driver example with bug. It is interesting to consider the number of abstractions in our table; there are even two cases where the large size of the block encoding makes it possible to find the bug without any abstraction computation or refinement step (this would be equivalent to BMC).

As the results look promising, very large block encodings might be a way to reduce the number of abstraction computations, which in turn improves both precision *and* performance. Therefore, we also experimented with the examples that do not have bugs. The performance of the examples `test_locks_*` are shown in the second part of Table III. The results show that the performance can dramatically decrease if the block is terminated after a certain number of operations regardless of the control-flow structure. The performance of the NT driver examples without bug did not improve, because the effect of the loop bodies seems efficiently represented by the abstractions at the loop heads (or loop bodies are not relevant for the property to verify). The results for the SSH programs were mixed. Only some configurations provide better performance than LBE for a few programs. However, there were many examples for which much more time is needed, or the analysis even fails to terminate. Almost all time is spent by the SMT solver while computing Craig interpolants, and the resulting formulas are sometimes huge. The SMT solvers seem to be overwhelmed by the complexity of the large disjunctive path formulas. A comparison of different SMT solvers (MATHSAT, CSISAT) shows that different solvers perform well on different examples. Thus, we can conclude that there is much room for improvement when a new generation of SMT solvers is available which can handle large interpolation queries (only three interpolating SMT solvers are currently available: MATHSAT [11], CSISAT [9], FOCI [21]).

V. Conclusion

Software model checking largely depends on automated theorem proving, and the efficiency and precision have significantly improved over the last years due to ever better theorem provers. We have designed and implemented a model-checking approach which makes it possible to flexibly choose how much of the state-space exploration is delegated to a theorem prover. A previous project had already indicated that it is highly beneficial to design the model-checking process such that *larger* queries can be given to a theorem prover, and less state-space is explored by the software model checker itself [6]. Our work provides answers to several new experimental questions: (1) We should generally use boolean abstraction in software model checking, because cartesian abstraction is feasible only for one (the traditional, SBE) configuration. (2) On the full spectrum between single-block encoding (SBE) and large-block encoding (LBE), there is no configuration of the block size that is generally better than LBE. (3) Encodings in the spectrum far beyond LBE can significantly improve the performance for finding bugs, similar to bounded model checking. We have also identified room for improvement for block encodings larger than LBE. [4] We leave it for future work to explore improvements of interpolation procedures, and to assemble structurally better encoding formulas that are 'easier' for theorem provers, restrict the size of the interpolation queries, or help the SMT solver where needed by keeping structures explicit. We found it convenient to formalize our concept of ABE using the framework of configurable program analysis [8]; but we have only specified explicitly what ABE means for a predicate-analysis domain, and have not yet designed any other abstract domains (e.g., numerical domains) with adjustable-block encoding.

[4] For example, the "very large block" encodings should not be defined as a strict k-bound, but respect the control structure of the program (e.g., compute an abstraction after each control-flow subgraph of size more than k) — the ABE approach opens a large spectrum of possibilities.

References

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

[2] T. Ball, B. Cook, V. Levin, and S. Rajamani, "SLAM and Static Driver Verifier: Technology transfer of formal methods inside Microsoft," in *Proc. IFM*, LNCS 2999. Springer, 2004, pp. 1–20.

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

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

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

[6] D. Beyer, A. Cimatti, A. Griggio, M. E. Keremoglu, and R. Sebastiani, "Software model checking via large-block encoding," in *Proc. FMCAD*. IEEE, 2009, pp. 25–32.

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

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

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

[10] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu, "Symbolic model checking without BDDs," in *Proc. TACAS*, LNCS 1579. Springer, 1999, pp. 193–207.

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

[12] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Counterexample-guided abstraction refinement for symbolic model checking," *J. ACM*, vol. 50, no. 5, pp. 752–794, 2003.

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

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

[15] W. Craig, "Linear reasoning. A new form of the Herbrand-Gentzen theorem," *J. Symb. Log.*, vol. 22, no. 3, pp. 250–268, 1957.

[16] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadek, "Efficiently computing static single-assignment form and the program dependence graph," *ACM Trans. Program. Lang. Syst.*, vol. 13, no. 4, pp. 451–490, 1991.

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

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

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

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

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

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

Modular Bug Detection with Inertial Refinement

Nishant Sinha
NEC Research Labs, Princeton, USA

***Abstract*—Structural abstraction/refinement (SAR) [4] holds promise for scalable bug detection in software since the abstraction is inexpensive to compute and refinement employs pre-computed procedure summaries. The refinement step is key to the scalability of an SAR technique: efficient refinement should avoid exploring program regions irrelevant to the property being checked. However, the current refinement techniques, guided by the counterexamples obtained from constraint solvers, have little or no control over the program regions explored during refinement. This paper presents *inertial* refinement (IR), a new refinement strategy which overcomes this drawback, by *resisting* the exploration of new program regions during refinement: new program regions are incrementally analyzed only when no error witness is realizable in the current regions. The IR procedure is implemented as part of a generalized SAR method in the F-SOFT verification framework for C programs. Experimental comparison with a previous state-of-the-art refinement method shows that IR explores fewer program regions to detect bugs, leading to faster bug-detection.**

I. INTRODUCTION

Modular program analyzers [30], [28], [12], [6], [32], [31], [4], [7] that exploit the program structure are more scalable since they avoid repeated analysis of program regions by computing reusable summaries. Traditional modular methods [30], [28] target proofs of program assertions by computing and composing summaries in an intertwined manner. For example, to compute a summary for a function F, the methods need to compute and compose the summaries of all the callees of F, even if many of these callees are irrelevant to checking the property at hand. Recent methods based on structural abstraction/refinement (SAR) [4], [31], [12] alleviate this problem by dissociating summary composition from computation: function summaries and verification conditions [16] are first *computed locally* by skipping the analysis of callees (abstraction phase) and then *composed lazily* with callee summaries (refinement phase). Refinement is property-driven and employs an efficient constraint solver (e.g., [15], [13]) for the program logic. In contrast to other abstraction/refinement methods, e.g., predicate abstraction [17], [5], computing a structural abstraction is relatively inexpensive, and refinement is done incrementally via pre-computed function summaries. Owing to these advantages, several recent methods [31], [4], [2], [7] have exploited the idea of SAR for scalable bug detection.

By dissociating summary computation from composition, SAR has the ability to *select* which regions to explore during the refinement phase for checking properties efficiently. The selective refinement strategy determines the efficiency of an SAR-based verification method. Ideally, we desire an *optimal* strategy, which explores (composes with) exactly those program regions which are relevant to a given property. Optimal refinement is as hard as the (undecidable) program verification problem since it may require a knowledge of the complete program behavior for making a selection. Consequently, researchers employ heuristics [4], [31] for performing refinement, guided by the counterexamples obtained when the solver checks the abstract model [23], [10]. The solver, however, is oblivious of the program structure, and may produce spurious counterexamples that continuously drive the refinement towards newer program regions, even though a witness may exist undetected in the currently explored regions. Redundant refinement of this form burdens the solver with irrelevant summary constraints, leading to dramatic increase in solving times, and, in many cases, to the failure of an SAR-based method.

This paper presents a new structure-aware method, called *inertial refinement* (IR) to overcome this drawback. The IR method *resists* exploring new program regions during refinement, as much as possible, in hope of finding a witness *within* the currently explored regions. Given a program assertion A, our method computes an initial abstract *error condition* ϕ for violating A, by exploring program paths in a small set of regions relevant to A, while abstracting the other adjacent regions. To check if ϕ is feasible, IR first symbolically *blocks* all unexplored program regions involved in ϕ, by adding auxiliary constraints to the solver. This forces the solver to find witnesses to ϕ that avoid the unexplored regions. If such a witness exists, IR succeeds in avoiding the costly analysis of the unexplored regions. Otherwise, IR explores a *minimal* set of new program regions that may admit an error witness. The minimal set of regions are computed in a property-driven manner by analyzing the proofs of infeasibility inside the solver (based on the notion of *minimal correcting sets* [24]), which provide hints as to why the currently explored regions are inadequate for checking the property. IR has multiple advantages as a refinement method. IR improves the scalability of SAR-based methods by restricting search to a small set of program regions, leading to more *local* witnesses than other methods. Moreover, IR exploits the fact that most bugs can be detected by analyzing a small number of program regions [3], [26].

All previous methods based on SAR [31], [4], [2], [7] restrict structural abstraction to function boundaries. This paper proposes a *generalized* SAR scheme that may abstract (and later refine on-demand) arbitrary program regions, including loops. As a result, SAR can exploit the entire modular program structure to make a more fine-grained selection of regions to explore for checking properties efficiently. A consequence of this generalization is that we do not statically unroll loops and recursive functions for checking properties; they are dynamically unrolled in a property-driven manner by inertial refinement. The paper makes the following main contributions:

- We present a modular bug detection method based on a new *generalized* structural abstraction/refinement (SAR) approach, which fully exploits the modular structure of a program (functions, loops and conditionals) to perform an efficient analysis.
- We propose a new structural refinement method, called inertial refinement, which avoids exploring new program regions until necessary. The technique is property-guided and employs *minimal correcting sets* [24] produced by

```
1  int x,y;
2  void foo (int *p, int c) {
3  if (p == NULL)
4    x = c;
5  else x = bar (*p);
6  ...
7  assert (x > c);
8  ...
9  if (x == c)
10   y = neg(x);
11 else y = 0;
12 assert (y >= 0);
13 }
```

```
1  int neg (int a) {
2    if (a > 0) return -a;
3    else return a;
4  }

1  void loopf (int n) {
2    int i=0, j=0;
3    while (i < n) {
4      j = j + 2 * i;
5      i++;
6    }
7    assert (n >= 0 &&
8            j < 2*n);
9  }
```

Fig. 1: Motivating Examples. The complex function `bar` is not described.

constraint solvers [15], [13] to efficiently select new regions to explore.

- The SAR method with inertial refinement is implemented in the F-SOFT verification framework for C programs [19]. Experimental results on real-life benchmarks show that the method explores fewer regions than a state-of-the-art refinement technique [4], and outperforms the previous approach on larger benchmarks.

II. OVERVIEW

We illustrate the key ideas of inertial refinement for checking the function `foo` in Fig. 1: `foo` contains a call to a complex function `bar` (line 5) and two assertions at lines 7 and 12, respectively. Consider the assertion (say A) at line 7 in `foo`: to check this assertion, SAR first computes an *error condition* (EC) under which A is violated. This EC, say ϕ, represents the feasibility condition for all program executions in `foo` which terminate at A and violate A. To compute ϕ, our method explores `foo` locally (cf. Sec. III) by performing a precise data-flow analysis: a form of forward symbolic execution [20] with data facts being merged path-sensitively at *join* nodes [21], [3]. The analysis propagates data of form (ψ, σ) through `foo`: ψ is the path condition at the current program location (summarizing the set of incoming paths to the location symbolically) and σ is a map from program variables to their path-sensitive (symbolic) values at the current location.

To avoid exploring `bar` at line 5, the method performs structural abstraction of `bar` during propagation: the effect of `bar` is abstracted by a tuple $(\pi_b, [ret_{bar} \mapsto \lambda_{b,ret}])$, where the placeholder (essentially, a free variable) π_b abstracts the set of paths through `bar` symbolically, and the placeholder $\lambda_{b,ret}$ abstracts the return value of `bar`. For example, the value of x computed at line 6 (obtained by merging data from the branches of the conditional at line 3) is $x_1 = ite(p \neq 0 \wedge \pi_b,\ \lambda_{b,ret},\ c)$ and the path condition is $\psi' = ((p \neq 0 \wedge \pi_b) \vee (p = 0))$. The EC ϕ computed for A at line 7 (ψ' conjoined with the negated assertion) is $\phi = \psi' \wedge (x_1 \leq c)$. Note that ϕ depends on the two unconstrained placeholders π_b and $\lambda_{b,ret}$ corresponding to `bar`. Now, ϕ is checked with a constraint solver, e.g., [15], [13] using structural refinement. We will see how the placeholder π_b plays a crucial role to avoid exploring paths into `bar`.

Checking ϕ with the solver may return a witness (lines 2-5-6-7) that includes a call to the complex function `bar`. This witness relies on the abstraction of `bar` by π_b and $\lambda_{b,ret}$ and hence may be spurious, e.g., if `bar` returns a value always greater than c. To check if the witness is an actual one, refinement will expand π_b and $\lambda_{b,ret}$ with the corresponding precise summaries from `bar`. Note, however, that line 4 sets x to c, and hence an actual witness for ϕ exists inside `foo` (line 2-3-4-6-7) that does not require exploring `bar`. However, this is not apparent from ϕ syntactically and a naive SAR checker will perform spurious refinement by expanding both the placeholders.

More sophisticated refinement procedures may also succumb to spurious refinement. For example, the state-of-the-art structural refinement strategy (referred to as `DCR`) [4], [3] uses the satisfying model from the constraint solver to compute a set of irrelevant placeholders, to avoid expanding them subsequently. Since `DCR` is guided only by the structure-unaware solver, it may expand placeholders spuriously even if a witness exists in the current regions. For example, suppose the solver generates the following model for the EC ϕ above: $(p \neq 0)$ and $(\pi_b = true)$. `DCR` analyzes the expression for ϕ guided by this model and concludes that both π_b and $\lambda_{b,ret}$ are relevant to ϕ being satisfiable. Therefore, `DCR` must perform the costly expansion of both the placeholders. Similarly, another structural refinement procedure [31] driven only by models from a constraint solver may also explore `bar` when trying to concretize an abstract counterexample.

In contrast, our inertial refinement (`IR`) procedure (cf. Sec. IV) *resists* expansion and checks if a proof/witness to ϕ exists within the currently explored region. To this goal, the analysis *blocks* paths leading to the unexplored function `bar` by adding a constraint $\neg\pi_b$ to ϕ and then checks for a solution. If a solution is found, as in this case, the method is able to avoid the cost of a spurious refinement. Otherwise, `IR` selects a *minimal* set of new regions to explore, which may admit an error witness (cf. Sec. IV).

Most bug finding approaches [4], [9], [32] statically unroll the loops to a fixed depth, which may lead to several errors being missed. Although loops may be also handled as tail-recursive functions in SAR (as in [31]), conventional static analysis [11] seldom does so. We propose a structural abstraction specific to loops, so that inertial refinement corresponds to *dynamically* unrolling loop iterations in a property-driven manner (cf. Sec. IV-A). As a result, our method can check non-trivial assertions, e.g., the assertion at line 7 in the `loopf` function in Fig. 1 is violated only when $n \geq 3$.

III. GENERALIZED STRUCTURAL ABSTRACTION

We start with describing our *generalized* structural abstraction, which forms the basis of our SAR method and may abstract arbitrary program *regions*, as defined below.

Program Regions. A program region R corresponds to a structural unit of the program syntax, i.e., a function body, a loop or a conditional statement. To formalize regions precisely, we view a sequential C program $\mathcal{P}$ as a hierarchical *recursive state machine* (RSM) M [1]. The RSM M consists of a set of *regions*: each region contains a control flow graph, which in turn consists of (i) a set of *nodes* (labeled by assignments), (ii) a set of *boxes* (each box is, in turn, mapped to a region), and (iii) control flow edges among nodes and boxes (labeled with guards). Each region also has special *entry* and *exit* nodes. An *unfolding* [1] of M is obtained by recursively inlining

```
SAR (Program P)
R := Partition P into regions
foreach region R ∈ R do
    (ψ_R, σ_R, Φ_R) := LOCSUMMARIZE(R)
    Φ := HOIST(Φ_R)
    foreach φ ∈ Φ do
        res := REF(φ)
        /* Report witness if res is SAT */

REF(φ)
while φ contains placeholders do
    if CHECK (φ) = UNSAT then
        return UNSAT
    Pick a placeholder λ in φ
    t = GETSUMMARY (λ)
    φ := φ[λ ↦ t]
return SAT
```

Algorithm 1: A generic modular analysis algorithm SAR.

each box by the corresponding region. An edge from a node to a box is said to be a *call edge*. A program region R_1 is said to *precede* another region R_2, if R_1 contains a box that maps to R_2, i.e., control flow enters R_2 on leaving R_1. We also say that R_2 *succeeds* R_1 in this case. We assume that assertions for property checking, e.g., dereference safety, array bound violations, etc., are modeled as special *error nodes* in the RSM M; the reachability of error nodes implies that the corresponding assertion is violated.

For example, the program fragment in Fig. 1 consists of the following top-level regions: function bodies `foo`, `neg` and `loopf`. Region `foo` contains two boxes mapped to *if-then-else* (conditional) regions C_1 (lines 3-5) and C_2 (lines 9-11); both regions succeed region *foo*. C_1 and C_2, in turn, contain boxes mapped to `bar` and `neg` function regions, respectively. Similarly, the `loopf` function region contains a box corresponding to the *loop body* region (lines 3-6). For ease of description, we will refer to an inlined instance of a region in a box as a region also. In the following, we use the standard program analysis terminology [30], [12], extended to RSM regions in a straightforward manner. In the following, we will assume that the regions corresponding to conditionals are inlined in the corresponding boxes; we will only differentiate between function and loop regions.

Side-effects. For each program region R, the *side-effects set* $\mathcal{M}(R)$ denotes the set of program variables that may be modified on executing R (together with its successors) under all possible calling contexts. The *inputs* to region R consist of the set of variables that are referenced in R. To compute side-effects for programs with pointers, we assume that the heap size is bounded (to handle dynamic allocation and recursive data structures) and employ a whole-program side-effect analysis [29], [31] to compute the side-effects.

Error Conditions. Given an error node eb in the program RSM and a set of paths T terminating at eb, the formula representing the feasibility condition for the set T is said to be an *error condition* (EC). In contrast to verification conditions (VCs) [16], which express sufficient conditions for existence of proofs, the satisfiability of ECs implies existence of assertion violations. We say that an EC ϕ has a witness, if ϕ has a satisfying solution; otherwise, we say that the EC has a proof. Note that an infinite number of ECs may be derived from a location eb (due to loops and recursion). Our *under-approximate* analysis checks only a finite subset of all the ECs and therefore, guarantees only the soundness of bugs detected; the proofs do not imply that eb is unreachable (cf. Theorem 1).

Structural Abstraction. Analyzing all program regions may neither be feasible for a given program analysis nor necessary for checking a given property. Structural abstraction enables a property-driven modular analysis of programs while avoiding the analysis of undesired regions, e.g., one or more nested successor regions of a region Q can be abstracted during analysis of Q. The structural abstraction of a region R is a tuple (π_R, σ_R), where (i) π_R is a *Skolem constant* (basically, a fresh variable) summarizing the paths in R and (ii) σ_R is a map with entries of form $(v \mapsto \lambda_{v,R})$, where $v \in \mathcal{M}(R)$ is a side-effect of R and $\lambda_{v,R}$ is a Skolem constant which models arbitrary modifications of v in R. In the following, the Skolem constants π_R and $\lambda_{v,R}$ are jointly referred to as *placeholder* variables. We also refer to placeholders of form π_R as *π-variables*. The set of placeholders in the range of σ_R are said to *depend* on π_R, and are denoted by $Dep(\pi_R)$. For example, the call to `bar` in the function `foo` in Fig. 1 (cf. Sec. II) is abstracted by the tuple $(\pi_b, [ret_{bar} \mapsto \lambda_{b,ret}])$, where $\lambda_{b,ret} \in Dep(\pi_b)$.

When the analysis encounters a call to R in a preceding region Q, it conjoins the placeholder π_R with the current path condition ψ, updates the current value map σ with σ_R, and continues analyzing Q. If R is later found relevant to an assertion in Q, the initial abstraction of R is refined on-demand. The abstraction has several advantages: first, it is cheap (computation of side-effects $\mathcal{M}(R)$ is done once for the whole program); second, it allows on-the-fly refinement using a summary of R, which is computed only once, and finally, it allows us to analyze program *fragments* in absence of the whole program. Note that our formalization generalizes the earlier approaches [31], [4] to handle all modular units of a program, i.e., functions, loops and conditionals, uniformly. As a result, SAR can perform a more fine-grained selection of program regions to explore when checking an EC.

Alg. 1 presents a generic modular algorithm SAR for checking assertions in a program $\mathcal{P}$, having these phases:

- The algorithm first partitions the program into a set of regions $\mathcal{R}$.
- For each region $R \in \mathcal{R}$, a procedure LOCSUMMARIZE is used to compute a *local* summary (by a forward data flow analysis over program expressions [21], [3] or using weakest preconditions [14], [16], [4]) while abstracting all the successor regions of R as above. The local summary $(\psi_R, \sigma_R, \Phi_R)$ consists of the predicate ψ_R summarizing the paths in R, the map σ_R summarizing the outputs (side-effects) of R in terms of symbolic expressions over inputs to R, and a set of error conditions (ECs) Φ_R which correspond to assertion violations in R.

- The ECs Φ_R are local to R; in order to find violating executions starting from the program entry function, these ECs are *hoisted* [3], [7], [16] to the entry function of the program by the HOIST procedure, which computes weakest preconditions of ECs with respect to a bounded set of calling contexts [30] to the region R. Note that HOIST may also use structural abstraction during backward propagation [3].
- Finally, the procedure REF is used to check each hoisted EC ϕ using structural refinement based on a constraint solver, e.g., an SMT solver [13], [15]. REF proceeds iteratively by choosing a placeholder λ in ϕ, *expanding* λ using its summary expression t (computed by the GETSUMMARY procedure), and checking if the resulting ϕ is satisfiable. The procedure REF terminates when the solver finds the EC ϕ unsatisfiable (UNSAT) or if ϕ does not contain any placeholders and is satisfiable (SAT).

In this paper, we assume a partition of the program into only function and loop regions, i.e., conditionals are inlined in the predecessor regions. The details of the LOCSUMMARIZE, GETSUMMARY and HOIST procedures can be found elsewhere [3], [21], [7], [16]; we will only concern ourselves with the REF procedure, which is the prime bottleneck for the SAR method.

SAR is an *under-approximate* analysis, i.e., it analyzes only a subset of all possible paths reaching an assertion violation. Hence, it can only detect bugs soundly (cf. Theorem 1). SAR can natively handle programs with arbitrary recursive functions and loops: however, it may not terminate if an unbounded number of iterations of `IR` are needed during the check.

Example 1. Recall the program fragment shown in Fig. 1. Our analysis first partitions the fragment into four regions: `foo`, `neg`, `loopf` functions, and the loop body region (lines 3-6 in `loopf`). The procedure LOCSUMMARIZE then summarizes each region, e.g., the summary of `foo` (shown below) consists of path and side-effect summaries, $\psi_{\texttt{foo}}$ and $\sigma_{\texttt{foo}}$, resp., and a set of ECs $\Phi_{\texttt{foo}}$. To summarize `foo`, the calls to `bar` and `neg` are abstracted by placeholder pairs (π_b, $\lambda_{b,ret}$) and (π_n, $\lambda_{n,ret}$) respectively.

$\psi_{\texttt{foo}}$	$(\psi_1 \wedge \psi_2)$ where $\psi_1 = (p = 0 \vee (p \neq 0 \wedge \pi_b))$, $\psi_2 = ((x_1 = c \wedge \pi_n) \vee (x_1 \neq c))$
$\sigma_{\texttt{foo}}$	$[x \mapsto x_1, y \mapsto y_1]$, where $x_1 = ite(p \neq 0 \wedge \pi_b, \lambda_{b,ret}, c)$ and $y_1 = ite(x_1 = c \wedge \pi_n, \lambda_{n,ret}, 0)$
$\Phi_{\texttt{foo}}$	$\{\Phi_1, \Phi_2\}$; $\Phi_1 = (\psi_1 \wedge x_1 \leq c)$, $\Phi_2 = (\psi_{\texttt{foo}} \wedge y_1 < 0)$

All the ECs are then hoisted to the entry functions (`foo` and `loopf` here): in this case, the ECs for `foo` are already hoisted. Finally, REF analyzes each EC ϕ in the entry function by iteratively checking ϕ and expanding placeholders.

Theorem 1: Let SAR compute an EC ϕ for an error location l after hoisting. If REF(ϕ) returns SAT, then there exists a true error witness to l.

Selective Refinement. In general, many placeholders in an EC ϕ are not relevant for finding a proof or a witness, and expanding them leads to wasteful refinement iterations along with an increased load on the solver. *Selective* refinement, therefore, focuses on selecting a subset of placeholders in ϕ that are relevant to the property. This allows REF to terminate early if there exist no relevant placeholders in ϕ. An additional benefit of selective refinement is that, in many cases, recursive programs can be analyzed without unbounded expansion of the placeholders. We now present a new strategy for selective refinement, called *inertial refinement*.

IV. INERTIAL REFINEMENT

The key motivation behind inertial refinement (`IR`) is to avoid exploring irrelevant regions during modular analysis, based on the insight that most violations involve only a small set of program regions. To this goal, `IR` first tries to find a witness/proof for an EC *inside* the program regions explored currently, say $\mathcal{R}$. If `IR` is unsuccessful, then $\mathcal{R}$ is inadequate for computing a witness or a proof. Therefore, `IR` augments $\mathcal{R}$ by a minimal set of successor program regions, which may admit a witness. The new regions are selected efficiently based on an analysis of why the current region set $\mathcal{R}$ is inadequate. In order to describe the details of `IR`, we first introduce the notion of *region blocking*.

Region blocking. Recall (cf. Sec. III) that SAR may abstract a region R (when analyzing a predecessor region Q) in form of a tuple (π_R, σ_R), where π_R is the path summary placeholder of R and σ_R maps output variables in R to unique placeholders. A *region blocking* constraint (π-constraint, in short) for a π-variable π_R is defined to be $\phi_\pi = \neg\pi_R$. Asserting ϕ_π when checking an EC ϕ in the region Q, forces the solver to find witnesses by *blocking* the program execution paths that lead from Q to R.

Figure 2 shows the `IR` procedure in form of a flow diagram. `IR` proceeds by iteratively adding or removing π-constraints, until the result is satisfiable (SAT) or unsatisfiable (UNSAT). In order to resist exploration of irrelevant regions, the procedure first asserts π-constraints (Φ_π) for all π-variables in the current EC ϕ. If ϕ remains satisfiable even after adding Φ_π, the procedure returns $true$, implying that a witness for ϕ exists that does not involve traversing the blocked regions. Otherwise (the constraints are UNSAT), a subset ϕ_π of π-constraints is computed, whose removal leads to a satisfiable solution. Note that the set ϕ_π corresponds to a set of blocked regions whose exploration may lead to the discovery of a concrete witness to ϕ. If the set ϕ_π is empty, then no witness for ϕ exists (see Theorem 2), and `IR` returns $false$. Otherwise, `IR` performs exploration of the regions corresponding to ϕ_π in the following way. First, the paths to the blocked regions are exposed by removing all π-constraints in ϕ_π. Then, `IR` refines ϕ by expanding the placeholders V_π in ϕ_π and their dependent placeholders $Dep(V_\pi)$ with the corresponding summary expressions (cf. Sec. III).

The key step in the `IR` procedure is that of computing $\phi_\pi \subseteq \Phi_\pi$ efficiently. To this goal, we employ the notion of a *correction set* (CS) of a set of constraints [24]: given an unsatisfiable set of constraints Ψ, a correction set ψ is a subset of Ψ such that removing ψ makes $\Psi \setminus \psi$ satisfiable. To obtain efficient inertial refinement, i.e., explore a small set of blocked regions, we are interested in a *minimal* correcting set (MCS), none of whose proper subsets are correction sets. The notion of correction sets is closely related to that of *maximal satisfiable subsets* (MSSs) [24], which is a generalization of the solution of the well-known Max-SAT problem [24]. An MSS is a satisfiable subset of constraints that is maximal, i.e., adding any of one of the remaining constraints would make it UNSAT. The *complement* of an MSS consisting of the remaining set of unsatisfied constraints is an MCS. For example, the UNSAT constraint set $((x), (\neg x \vee y), (\neg y))$ admits three MCSs, (x), $(\neg x \vee y)$, and $(\neg y)$, all of which are *minimum*. Note that many approaches utilize unsatisfiable cores [27] during refinement, e.g., for proving infeasibility of abstract

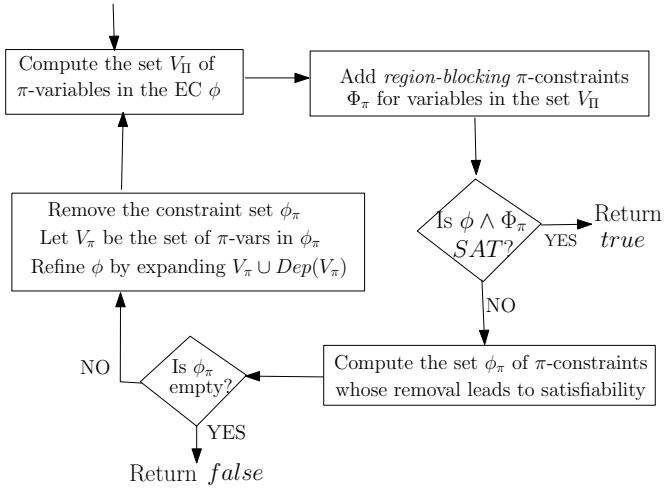


Fig. 2: Flow diagram for checking EC ϕ using *inertial* refinement.

counterexamples with predicate abstraction [18] or procedure abstraction [31]. In contrast to the above approaches which try to prove infeasibility in the concrete model (using cores), we try to obtain constraints (MCS) that allow a witness *to appear* in the abstract model. The notion of MCSs is also related to computing an *interesting* witness to a satisfiable temporal logic formula by detecting vacuous literals [22]. Note that computing MCSs is NP-hard and hence makes `IR` expensive as compared to the light-weight `DCR` method [4] (cf. Sec. II), which only needs a model from the solver. However, we expect that exploring fewer regions in `IR` will compensate for the extra cost.

An MCS for a set of constraints Φ can be computed by obtaining all the proofs of infeasibility (UNSAT cores) of Φ and then computing the minimal *hitting* literal set for this set of UNSAT cores [24]. Many modern constraint solvers, e.g., [15], allow for constraints with weights and solving Max-SAT (MSS) problems natively. Therefore, we can compute MCSs of π-constraints using these solvers by first asserting π-constraints with non-zero weights and then computing the subset of unsatisfied π-constraints in the weighted Max-SAT solution. In our experiments, however, we used the previous method of computing hitting sets: Max-SAT results obtained from [15] were unfortunately erroneous and not usable.

The `IR` procedure can be implemented efficiently using an incremental SMT solver (e.g., [15], [13]). These solvers maintain an internal *context* of constraints to provide incremental checking; constraints can be asserted or retracted iteratively from the context while checking, and the solver is able to reuse the inferred results effectively from the previous checks. Alg. 2 shows the pseudo-code of the inertial refinement algorithm REF-IR using such an SMT solver. REF-IR replaces the naive REF procedure in the overall SAR algorithm (cf. Sec. 1). The description uses the symbol ctx to denote the context of the incremental solver and the methods ASSERT and RETRACT [15] are used for adding and removing constraints to the context incrementally. The procedure starts at the **BEGIN** block by asserting the current EC ϕ in the solver's context. Depending on whether the context is satisfiable or not, the control switches to the locations labeled by **BLOCK** and **EXPAND** respectively (cf. Alg. 2). In the **BLOCK** case, the region-blocking constraints Φ_π are asserted first. If the resultant context is satisfiable, REF-IR returns with SAT result. Otherwise, the control switches to the UNSAT label. Here, a MCS ϕ_π of π-constraints is computed to check if removing any π-constraints may admit a witness to ϕ. If the MCS is empty, no witness is possible and the procedure returns UNSAT. Otherwise, all the π-constraints in the MCS are retracted and the EC ϕ is refined by expanding π-variables V_π in ϕ_π together with their dependent variables $Dep(V_\pi)$.

Theorem 2: The inertial refinement procedure REF-IR returns *SAT* while checking an EC ϕ only if there exists a concrete witness to the error node for ϕ.

Example 2. Recall the summary of the procedure `foo` (Fig. 1) presented in Example 1. The EC ϕ for the assertion at line 12 is $(\psi_{\texttt{foo}} \wedge y_1 < 0)$, where $y_1 = ite(x_1 = c \wedge \pi_n, \lambda_{n,ret}, 0)$ and $x_1 = ite(p \neq 0 \wedge \pi_b, \lambda_{b,ret}, c)$; ϕ contains two π-variables π_b (`bar`) and π_n (`neg`). (**BEGIN**) Initially, ϕ is satisfiable, and REF-IR (Alg. 2) switches to the **BLOCK** label.
(**BLOCK**) The REF-IR procedure first blocks both π_b and π_n (adds π-constraints $\neg\pi_b$, $\neg\pi_n$), and checks for a solution. No solution is found since all feasible paths in `foo` contain a function call. Therefore, the control switches to **EXPAND**.
(**EXPAND**) Here, REF-IR computes an MCS ϕ_π, which is $(\neg\pi_n)$. Since π_n corresponds to function `neg`, `IR` must explore `neg` to find a witness. The procedure then removes $\neg\pi_n$ and refines ϕ by adding summary constraints for π_n and the dependent placeholder $\lambda_{n,ret}$. These constraints ($\pi_n = true$ and $\lambda_{n,ret} = ite(x_1 > 0, -x_1, x_1)$ respectively) are generated by analyzing the `neg` function (cf. Fig. 1).
(**BEGIN**) On checking ϕ again after expansion, the solver finds a witness (lines 2-3-4-6-7-8-9-10-12), with say, $c = 1$, $p = 0$, $x_1 = 1$, $y_1 = -1$. REF-IR now checks if the witness is an actual one (**BLOCK** label) by blocking all π-variables. Note that π_b is the only π-variable remaining in ϕ and the corresponding π-constraint is already asserted. Therefore, REF-IR concludes that the witness is an actual one and terminates. Note how REF-IR avoids the redundant expansion of the complex function `bar`, guided both by the abstract EC ϕ as well as the modular program structure. Also, the efficiency of REF-IR crucially depends on the computed MCSs.

A. Example: IR with Loop-specific Abstraction

Consider the function `loopf` in Fig. 1. The assertion at line 7 checks if on loop exit, the value of j is less than $2 * n$, and is violated only when $n \geq 3$. To see this, consider the data computed at the loop exit (line 7) by a symbolic execution [20] of `loopf` after few initial iterations: (0) $(0 \not< n, j \mapsto 0; i \mapsto 0)$, (1) $(0 < n \wedge 1 \not< n, j \mapsto 0; i \mapsto 1)$ (path condition reduces to $n = 1$), (2) $(n = 2, j \mapsto 2; i \mapsto 2)$, (3) $(n = 3, j \mapsto 6; i \mapsto 3)$, respectively. Note that the value (3) violates the assertion at line 7, while (0), (1) and (2) do not.

In general, a violation like above may require an arbitrary number of iterations of the loop, depending on one or more inputs. Many bug finding methods [9], [4], [32] unroll all program loops to a fixed depth, and may miss bugs like these. The approach in [31] transforms loops to tail-recursive functions; however, conventional static analysis seldom does so. In contrast, we show how inertial refinement can be used to perform a *dynamic* property-driven unrolling of loop regions, with the help of an abstraction *specific* to loop regions. Note that methods based on refining predicate abstractions [5], [18] may detect this violation by refinement; however, constructing

REF-IR(ctx, ϕ)

BEGIN: ASSERT (ctx, ϕ)
if ctx *is satisfiable* **then**
  goto **BLOCK**
else
  goto **EXPAND**

BLOCK: V_Π := set of π-variables in ϕ
/ Assert π-constraints */*
$\Phi_\pi := \wedge\{(v = false) \mid v \in V_\Pi\}$
ASSERT(ctx, Φ_π)
if ctx *is satisfiable* **then**
  return SAT
else
  goto **EXPAND**

EXPAND: ϕ_π := MCS(ctx)
if $\phi_\pi = false$ **then**
  return UNSAT
/ Witness may exist */*
RETRACT (ctx, ϕ_π)
/ Select placeholders to refine */*
V_π := Variables in ϕ_π
$V'_\pi := V_\pi \cup Dep(V_\pi)$
foreach $\lambda \in V'_\pi$ **do**
  t = GETSUMMARY (λ)
  $\phi := \phi[\lambda \mapsto t]$
goto **BEGIN**

Algorithm 2: The REF-IR procedure for checking an EC ϕ with an incremental SMT solver using inertial refinement. The variable ctx denotes the context of the solver.

and refining predicate abstractions is expensive. In contrast, SAR with cheap abstraction and inertial refinement using loop summaries can detect such violations at a much lower cost.

SAR first computes a local loop body summary, $(i_o < n, [j \mapsto j_o + 2 * i_o;\ i \mapsto i_o + 1])$, where j_o and i_o represent the values of j and i respectively at the beginning of the body. Recall that SAR first checks `loopf` by skipping the loop region with an abstraction of form (ψ, σ); in this case, however, the abstraction is *specific* to the loop region and allows dynamic loop unrolling. More precisely, (1) $\psi = (\pi_0 \vee \pi_{1+})$, where $\pi_0 = (n \leq 0)$ and π_{1+} are path conditions after zero or ≥ 1 loop iterations, respectively, and the map (2) $\sigma = [j \mapsto ite(\pi_0, 0, \lambda_{j,1+}); i \mapsto ite(\pi_0, 0, \lambda_{i,1+})]$, where $\lambda_{j,1+}$ and $\lambda_{i,1+}$, respectively, are the values of j and i obtained after ≥ 1 loop iterations ($i = j = 0$ after zero iterations). Using the above abstraction, the symbolic data obtained at the assertion at line 7 is (ψ, σ) so that the EC ϕ is

$\phi = ((\pi_0 \vee \pi_{1+}) \wedge n \geq 0 \wedge ite(\pi_0, 0, \lambda_{j,1+}) \geq 2 * n))$

The procedure REF-IR first checks ϕ by blocking all the loop iterations, i.e., it adds a π-constraint $\neg\pi_{1+}$. The solver checks $(\phi \wedge \neg\pi_{1+})$ and returns UNSAT with the MCS $\neg\pi_{1+}$. As a result, `IR` removes $\neg\pi_{1+}$ and refines ϕ by adding summary constraints for π_{1+}, $\lambda_{j,1+}$ and $\lambda_{i,1+}$, i.e., $\pi_{1+} = (n = 1 \vee \pi_{2+})$, $\lambda_{j,1+} = ite(n = 1, 0, \lambda_{j,2+})$, etc.. `IR` again proceeds iteratively by blocking π_{2+}, π_{3+}, and so on, obtaining MCSs and refining ϕ. A satisfiable solution is obtained in the fourth iteration (with π_{4+} blocked), which corresponds to a true violation witness.

Note that if a witness requires a large number of loop unrollings, refinement using `IR` is inefficient. One solution is to expand multiple loop iterations simultaneously. However, we observed that in many real-life programs having input-dependent loops, few loop unrolls are sufficient for finding bugs; inertial refinement is effective in such cases.

V. EXPERIMENTAL EVALUATION

We implemented the modular analysis SAR (cf. Sec. III) in the F-SOFT [19] framework for verification of C programs. The framework constructs an *eager* memory model for C programs [19] by bounding the heap, flattening aggregate data types into simple types (up to depth 2 for our experiments), and modeling the effect of pointer dereferences by an explicit case analysis over the points-to sets for the pointer variables. Also, F-SOFT instruments the program for properties being checked, e.g., dereference safety (N), array bounds violation (A) and string related checks (S). Therefore, SAR is able to check multiple types of properties in an uniform manner in the F-SOFT framework. The initial model is simplified by the tool with constant folding, program slicing and other light-weight static analysis, and is then provided as an input to the SAR procedure.

We used a wide collection of open-source and proprietary industrial examples for evaluation: L2 is a Linux audio driver (`ymfpci.c`), L9 implements a Linux file-system protocol (`v9fs`), M1, M3 are modules of a network controller software, N1, N2 belong to a network statistics application, F consists of the `ftp-restart` module from the `wu-ftpd` distribution, and Spin corresponds to the SPIN model checker (without the parser front-end). The analyzed benchmarks range from LOC sizes of 1K to 19K. Our analysis focused on discovering known bugs efficiently.

Our implementation of SAR computes summaries and ECs for all program regions locally (cf. Alg. 1), stores them efficiently by representing terms as directed acyclic graphs (DAGs) and manipulates them using memoized traversal algorithms. The local ECs were hoisted up to the entry function and checked using the YICES SMT solver [15] in an incremental manner with refinement (cf. Alg. 1). To precisely model non-linear operators, e.g., modulo, which occur in many of our benchmarks, we encode all variables as bit-vectors.

We evaluated four structural refinement schemes: (i) *Naive*: expand all placeholders in the EC, (ii) `DCR`: use don't-cares for expansion, expand only *one selected* placeholder in each iteration (cf. Sec. II, similar to the state-of-the-art Calysto algorithm [4]), (iii) `DCR`$^+$, same as `DCR` except expand *all selected* placeholders (set V'_π in Alg. 2) in each iteration, and (iv) `IR`, the new inertial refinement scheme. In our experience, expanding all the selected placeholders (*set-expansion*) in each refinement iteration converges much faster than one placeholder at a time (*one-expansion*), and, therefore, is our default mode for *Naive* and `IR` schemes. The experiments were done on a Linux 2.4Ghz Core2Duo machine, with timeout of 1 hr and 8GB memory limit.

Figure 3 shows the experimental comparison between the

Bm	LOC	#EC	Naive		DC-based				IR	
					DCR		DCR^+			
			#R	T	#R	T	#R	T	#R	T
F-A	1K	48	162	73	75	282	58	**71**	**51**	78
F-N	1K	18	78	12	63	71	**32**	**11**	51	17
F-S	1.3K	54	100	2044	-	TO	27	**844**	**17**	2359
N1-N	1.2K	77	4	65	2	62	2	62	**0**	**61**
N2-S	1.4K	230	7	9	3	11	3	10	**1**	**9**
L2-A	5.4K	135	550	27	**292**	58	304	29	450	**28**
L9-A	6K	314	978	279	-	TO	549	589	**257**	**162**
L9-N	6K	124	721	22	**114**	139	144	**15**	205	27
M1-A	6K	356	906	**59**	-	TO	527	64	**408**	87
Spin	9K	233	662	2173	-	TO	295	2018	**192**	**1472**
M1-S	12K	196	800	68	338	124	354	62	**283**	**57**
M3-S	19K	419	-	TO	-	TO	253	1599	**221**	**1334**

Fig. 3: Experimental comparison of structural refinement schemes: (i) (Naive) without any selection of placeholders, (ii) DCR [4] (iii) DCR^+ with set-expansion and (iv) the new IR scheme. Benchmarks (Bm) are named in "Name-Checker" format, where Checker is either A (array bounds), N (NULL dereference) or S (string checker). LOC shows the lines of code analyzed post-simplification. #EC = the number of ECs checked for the benchmark. #R denotes the number of regions expanded. Time out (TO) of 3600s. Memory limit 8GB. Best figures are in bold.

various structural refinement techniques. The results confirm that structural abstraction methods scale to industrial benchmarks while retaining precision: many of these examples cannot be handled by other techniques, e.g., based on monolithic BMC [9] and predicate abstraction. We report the total time (T) including the summary computation and EC checking times. For each benchmark, we report the total number of regions (#R) expanded during the run. Note that ECs may have either a proof or a witness, and many of them may be checked without any refinement. Also, the set of regions explored (#R) may include the same function under multiple contexts. The results show that DCR^+ and IR clearly outperform the naive refinement scheme, which time-outs on the largest example M3-S, implying that selective refinement is essential. However, we observe that DCR time-outs in many cases where even *Naive* with set-expansion finishes. In the following, we compare DCR, DCR^+ and IR systematically.

(DCR vs DCR^+). Since DCR performs one-expansion, it calls the solver large number of times. As a result, it time-outs on 40% of the benchmarks, while DCR^+ finishes in time, showing that DCR^+ converges much faster than DCR. However, in most cases where DCR finishes, it expands fewer regions and variables than DCR^+, due to one-expansion.

(IR vs DCR). DCR time-outs on many benchmarks, especially the bigger ones, due to one-expansion, whereas IR finishes. The results show that IR outperforms DCR [4] in terms of run-times on all benchmarks. To permit a fair comparison, we augment DCR with set-expansion (DCR^+) and compare with IR below. Note, however, that for benchmarks L2-A and L9-N, DCR does expand fewer regions than IR. We discuss this below.

(IR vs DCR^+). Both these approaches use set-expansion and finish on all benchmarks. We observe that, in most cases, IR expands fewer regions than DCR^+, showing that inertial refinement is indeed useful, and that many properties can be checked while restricting to a smaller region set. For example, benchmarks N1-N and N2-S show the effectiveness of IR: in case of N1-N, DCR^+ needs to perform two expansions, while IR doesn't need any expansions. On an average, IR expands about 20% fewer (54% in the best case) regions than DCR^+. Moreover, IR outperforms DCR^+ in terms of run-times on bigger examples (e.g. Spin, M1-S, M3-S), in spite of being more computationally expensive (requires computing MCSs). Since IR expands fewer regions than DCR^+, we believe that the improvement will be more dramatic on larger benchmarks.

On a few examples (F-N, L2-A and L9-N), however, IR expands more regions than DCR^+. This is because IR depends crucially on MCSs generated during refinement, which may not be optimal; in these examples, non-optimal MCSs led to exploration of irrelevant program regions. We believe that using more sophisticated MCS computing algorithms [24], [25], based on native MAX-SAT solving inside a constraint solver (as opposed to our method of computing hitting sets of UNSAT cores, cf. Sec. IV) will lead to faster computation of MCSs and hence improve the performance significantly.

We were unable to compare thoroughly with the previous work Calysto [4], [3], since it is not available publicly and the memory models used by F-SOFT and Calysto are different. However, the refinement scheme in Calysto is similar to DCR with one-expansion; in our experience, set-expansion is more powerful since the total number of SMT solver calls are reduced. Refinement based on counterexample-driven analysis of the concrete model [31] as opposed to abstract models is orthogonal to our approach; however, these approaches can also benefit from inertial refinement.

VI. RELATED WORK

Modular methods for sequential programs have been investigated extensively: most techniques perform an over-approximate analysis to obtain proofs of assertion validity via abstract interpretation [11], [12]. In contrast, our focus is on modular bug finding methods, which perform an *under-approximate* program analysis [12]. Taghdiri and Jackson proposed a method based on *procedure abstraction* [31] for detecting bugs in Java programs. To analyze a caller function, the method automatically infers relevant specifications for all the callee functions: it starts from empty specifications, and gradually refines them using proofs derived from analyzing spurious counterexamples in the concrete program model. Babic and Hu introduced the *structural abstraction* methodology in the tool Calysto [4], [3] for analyzing large-scale C programs. Again, the method analyzes the caller by abstracting the callees with summary operators (placeholders). When

checking abstract verification conditions (VCs) having these placeholders, structural refinement *expands* the placeholders with the corresponding summaries derived from the callees. In contrast to [31], structural refinement avoids the potentially expensive analysis of the concrete model: placeholders are selected by analyzing the abstract VC using a *don't-care* analysis of the abstract counterexample [4]. Both the above approaches [31], [4] perform refinement based purely on the counterexamples produced by the solver, which is oblivious of the program structure, and hence may explore new program regions even if a witness is realizable in the current regions.

PREfix [6] performs modular bug detection using path-enumeration based symbolic execution [20] to compute bottom-up summaries. These summaries only model partial procedure behaviors and the method may succumb to path explosion. In contrast, we compute precise summaries effectively using a merge-based data flow analysis [21], [3], and employ SAR to explore all program paths relevant to the property in an incremental fashion. The tool Saturn [32] performs bit-precise modular analysis for large C programs; however, the analysis is not path-sensitive inter-procedurally, and leads to infeasible witnesses. Chandra et al. [7] employ property-driven structural refinement to incrementally expand the call graph of Java programs in the presence of polymorphism, to avoid an initial call graph explosion. The ESC/Java tool [16] introduced verification condition (VC) generation based on intra-procedural weakest precondition [14] computation but requires pre/post specifications to reason inter-procedurally. In contrast, inter-procedural VCs are generated automatically in our approach using structural abstraction (cf. Sec. III). Compositional symbolic execution [2] also uses structural abstraction of functions with uninterpreted functions to make coverage-oriented testing more scalable: inertial refinement can also benefit these methods. In context of symbolic trajectory evaluation, Chockler et al. [8] present a method to refine circuit node placeholders using the notion of *responsibility*.

VII. CONCLUSIONS

We presented a modular software bug detection method using structural abstraction/refinement, based on analyzing program *regions* corresponding to modular program constructs. A new inertial refinement procedure `IR` was proposed to address the key problem of structural refinement: `IR` resists the exploration of abstracted program regions by trying to find a witness for an assertion inside the program regions explored previously. The procedure `IR` implemented in the F-SOFT framework scales to large benchmarks and is able to check properties by exploring fewer program regions than the previous don't-care based refinement technique [4]. Future work includes combining `IR` with other schemes, e.g., `DCR`, for more effective placeholder selection. Methods to dynamically expand the heap during analysis will also be investigated. Partitioning a program automatically for efficient SAR is also an interesting open problem. Finally, we plan to perform a detailed usability study of the SAR method for finding bugs in large benchmarks.

Acknowledgements. We would like to thank Domagoj Babic and the members of the Verification group at NEC for several useful discussions. We are also indebted to the anonymous reviewers for their helpful feedback.

REFERENCES

[1] R. Alur, M. Benedikt, K. Etessami, P. Godefroid, T. W. Reps, and M. Yannakakis. Analysis of recursive state machines. *ACM Trans. Program. Lang. Syst.*, 27(4):786–818, 2005.
[2] S. Anand, P. Godefroid, and N. Tillmann. Demand-driven compositional symbolic execution. In *TACAS*, pages 367–381, 2008.
[3] D. Babic. *Exploiting Structure for Scalable Software Verification*. Dissertation, Univ. of British Columbia, Vancouver, Canada, 2008.
[4] Domagoj Babic and Alan J. Hu. Structural abstraction of software verification conditions. In *CAV*, pages 366–378, 2007.
[5] T. Ball, R. Majumdar, T. D. Millstein, and S. K. Rajamani. Automatic predicate abstraction of C programs. In *PLDI*, volume 36(5), pages 203–213. ACM Press, June 2001.
[6] W. R. Bush, J. D. Pincus, and D. J. Sielaff. A static analyzer for finding dynamic programming errors. *Softw. Pract. Exper.*, 30(7):775–802, 2000.
[7] S. Chandra, S. J. Fink, and M. Sridharan. Snugglebug: a powerful approach to weakest preconditions. In *PLDI*, pages 363–374, 2009.
[8] H. Chockler, O. Grumberg, and A. Yadgar. Efficient automatic ste refinement using responsibility. In *TACAS*, pages 233–248. Springer-Verlag, 2008.
[9] E. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In K. Jensen and A. Podelski, editors, *TACAS*, volume 2988 of *LNCS*, pages 168–176. Springer, 2004.
[10] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement for symbolic model checking. *JACM*, 50(5):752–794, Sept. 2003.
[11] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252, 1977.
[12] Patrick Cousot and Radhia Cousot. Modular static program analysis. In *CC*, pages 159–178. Springer-Verlag, 2002.
[13] L. de Moura and N. Bjørner. Z3: An efficient smt solver. In *TACAS*, pages 337–340, 2008.
[14] Edsger Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.
[15] B. Dutertre and L. de Moura. A fast linear-arithmetic solver for DPLL(T). In *CAV*, pages 81–94, 2006.
[16] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for java. In *PLDI*, pages 234–245, 2002.
[17] S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In Orna Grumberg, editor, *CAV'97*, volume 1254 of *LNCS*, pages 72–83. Springer-Verlag, June 1997.
[18] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Kenneth L. McMillan. Abstractions from proofs. In *POPL*, pages 232–244, 2004.
[19] 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.
[20] J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.
[21] A. Kölbl and C. Pixley. Constructing efficient formal models from high-level descriptions using symbolic simulation. *IJPP*, 33(6):645–666, 2005.
[22] O. Kupferman and M. Y. Vardi. Vacuity detection in temporal model checking. *STTT*, 4(2):224–233, 2003.
[23] Robert P. Kurshan. *Computer-aided verification of coordinating processes: the automata-theoretic approach*. Princeton U.P., 1994.
[24] Mark H. Liffiton and Karem A. Sakallah. Algorithms for computing minimal unsatisfiable subsets of constraints. *J. Autom. Reasoning*, 40(1):1–33, 2008.
[25] Mark H. Liffiton and Karem A. Sakallah. Generalizing core-guided max-sat. In *SAT*, pages 481–494, 2009.
[26] A. Loginov, E. Yahav, S. Chandra, S. Fink, N. Rinetzky, and M. Nanda. Verifying dereference safety via expanding-scope analysis. In *ISSTA'08*, pages 213–224, NY, USA, 2008.
[27] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. pages 530–535. ACM Press, June 2001.
[28] Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *POPL*, pages 49–61, NY, USA, 1995. ACM.
[29] B. G. Ryder, W. A. Landi, P. A. Stocks, S. Zhang, and R. Altucher. A schema for interprocedural modification side-effect analysis with pointer aliasing. *ACM Trans. Program. Lang. Syst.*, 23(2):105–186, 2001.
[30] M. Sharir and A. Pnueli. Two approaches to interprocedureal data flow analysis. In *Program Flow Analysis: Theory and Applications*, volume 5, pages 189–234. Prentice Hall, 1981.
[31] M. Taghdiri and D. Jackson. Inferring specifications to detect errors in code. *Autom. Softw. Eng.*, 14(1):87–121, 2007.
[32] Yichen Xie and Alex Aiken. Saturn: A scalable framework for error detection using boolean satisfiability. *ACM Trans. Program. Lang. Syst.*, 29(3):16, 2007.

Path Predicate Abstraction by Complete Interval Property Checking

Joakim Urdahl, Dominik Stoffel, Jörg Bormann*, Markus Wedler, Wolfgang Kunz
Dept. of Electrical and Computer Eng., U. of Kaiserslautern, Germany *Abstract RT Solutions, Munich, Germany

***Abstract*—This paper describes a method to create an abstract model from a set of properties fulfilling a certain completeness criterion. The proposed abstraction can be understood as a *path predicate abstraction*. As in predicate abstraction, certain concrete states (called *important* states) are abstracted by predicates on the state variables. Additionally, paths between important states are abstracted by path predicates that trigger single transitions in the abstract model. As results, the non-important states are abstracted away and the abstract model becomes time-abstract as it is no longer cycle-accurate. Transitions in the abstract model represent finite sequences of transitions in the concrete model. In order to make this abstraction sound for proving liveness and safety properties it is necessary to put certain restrictions on the choice of state predicates. We show that *Complete Interval Property Checking (C-IPC)* can be used to create such an abstraction. Our experimental results include an industrial case study and demonstrate that our method can prove global system properties which are beyond the scope of conventional model checking.**

I. Introduction

Even after years of progress in the area of formal property checking, simulation is still the predominant technique in industrial practice for verifying the hardware of Systems-on-Chip (SoCs). There are at least two reasons for that: first, formal techniques provide a rigorous correctness proof only for a selected piece of design behaviour. Ensuring such local correctness is considered valuable, especially in corner cases of a design. However, in many industrial flows only parts of the overall design behavior are covered by properties and confidence in the correctness of the overall design largely still depends on simulation examining global input stimuli to the design and their output responses. Sophisticated methodologies have been developed to achieve full design coverage by local properties according to rigorous completeness metrics, as we will consider them in this paper. This can indeed contribute to replace simulation for SoC module verification by formal techniques. However, even in such a scenario simulation will still be needed for chip-level verification. This is the second reason for the prevailing role of simulation in SoC hardware verification. Formulating and proving global system properties spanning across different SoC modules on the Register-Transfer-Level (RTL) of an SoC is clearly beyond the capacity of current tools. As an alternative more viable than proving global properties at the RTL we may resort to system-level verification based on abstract design descriptions as they are supported by languages like SystemC [1]. However, unless the design refinements from these high levels to lower implementation levels become fully automated and can possibly be supported by new generation equivalence checking tools it is apparent that chip-level simulation at the RTL will still be needed and will contribute substantially to the overall verification costs.

The techniques proposed in this paper intend to make a step toward proving global properties for an RTL design implementation. We will prove properties that span over multiple SoC modules and which are significantly more complex than what can be proved with available property checkers. Our approach is not based on boosting the performance of the proof engines. Instead we propose a methodology to create design abstractions based on sets of properties fulfilling a certain completeness criterion.

The proposed approach leverages the significant advances that have been made over the last years in developing *systematic* procedures of writing properties for comprehensively capturing the behavior of a design. In particular, property checking formulations such as in Symbolic Trajectory Evaluation (STE) are very suitable for a systematic approach [2], [3], [4]. STE employs a specific style of formulating properties making it natural to compose properties into a more comprehensive design description which is successfully used in industrial practice. This paper is based on a related industrial property checking formulation called *Interval Property Checking (IPC)* [5], [6]. IPC stands for proving so called *operation properties* or *interval properties* on a bounded circuit model based on satisfiability (SAT) solving. From a computational perspective it can therefore also be seen as a variant of Bounded Model Checking (BMC) [7] while STE is more related to symbolic simulation and has a different way of representing and processing state sets based on Binary Decision Diagrams (BDDs). Moreover, in this paper we state a rigorous completeness criterion for sets of IPC properties [8], [9] which is a prerequisite for the proposed abstraction. Nevertheless, we believe that the proposed methodology and formalisms could also be adapted to property checking formulations other than IPC, in particular to STE and related approaches.

Abstraction in model checking is almost as old as model checking itself. The most popular abstraction techniques can be classified in being based on localization reduction [10] and predicate abstraction [11]. There is tremendous progress to integrate these abstractions into algorithms that automatically search for an appropriate abstraction such as [12], [13], [14]. Such techniques contribute substantially to increasing the scope of model checking to designs with several hundred state variables. This is often adequate for proving properties in SoC

module verification as described above. However, if designs with thousands of state variables have to be handled and chip-level properties must be proved on the RTL additional concepts for even stronger abstractions are required.

In this paper we propose to create an abstraction based on complete sets of IPC properties. This means as a starting point of our approach we assume that individual SoC modules have first been verified using IPC and a complete set of properties is available. In [15] so called *cando-objects* were proposed as abstract but still cycle-accurate design descriptions obtained from IPC properties. In contrast, the abstraction proposed here is time-abstract and is referred to as a *path predicate abstraction*. It leads to sound models for verifying both safety and liveness properties if certain restrictions on the state predicates are fulfilled. In Section II we first introduce basic notations. Section III introduces the proposed abstraction and shows that it can be used to prove safety and liveness properties which are also valid on the concrete model. Then, in Section IV we explain how this abstraction is created through the IPC methodology. In Section V we present experimental results also including an industrial case study.

II. Notations

A Kripke model is a finite state transition structure (S,I,R,A,L) with a set of states S, a set of initial states $I \subseteq S$, a transition relation $R \subseteq S \times S$, a set of atomic formulas A, and a valuation function $L : A \mapsto S$.

We consider *state predicates*, $\eta(s)$, $S(s)$, $X(s)$, $Z(s)$, $Y(s)$, that are evaluated for any concrete state s. In Kripke models derived from Moore FSMs the state variables and input variables may serve as atomic formulas. We may distinguish between the two kinds of state variables in our notation. The *original state variables* are denoted by z_i, the *input variables* of the original Moore FSM by x_j. If $S(s)$ is expressed only in terms of input state variables then the predicate describes an input *trigger condition* denoted by $X(s)$. If $S(s)$ is expressed only in terms of original state variables then we write $Z(s)$. $Y(s)$ denotes output values of the Moore machine in a state s. $T(s,s')$ is the characteristic function of the transition relation R.

An l-sequence π_l is a sequence of $l+1$ states $(s_0,s_1,\ldots,s_l)$. An l-sequence predicate $\sigma(\pi_l) = \sigma((s_0,s_1,\ldots,s_l))$ is a Boolean function characterizing a set of l-sequences; l is called the *length* of the predicate.

Note that we allow an l-sequence predicate to be applied also to a longer sequence, i.e., to an m-sequence $(s_0,s_1,\ldots,s_l,\ldots,s_m)$ with $m > l$. The predicate is then evaluated on the l-prefix $(s_0,s_1,\ldots,s_l)$ of the sequence. In case we would like to evaluate the predicate on an l-subsequence other than the prefix we need to shift the predicate in time using the *next* operator defined as follows:

$$next(\sigma_l,n)((s_0,s_1,\ldots,s_{n-1},\ s_n,s_{n+1},\ldots,s_{n+l})) := \sigma_l((s_n,s_{n+1},\ldots,s_{n+l})).$$

The $next(\sigma_l,n)$ operator shifts the starting point of the evaluation of a predicate σ_l to the n-th state in a sequence; $next(\sigma_l,n)$ is a sequence predicate of length $(n+l)$.

The usual Boolean operators $\vee$, $\wedge$, $\neg$, $\Rightarrow$ are also applicable to l-sequence predicates. If l_{max} is the largest length of all sequence predicates in a Boolean expression built with these operators, then the value of the expression is defined only for m-sequences with length $m \geq l_{max}$.

We also define a concatenation operation $\odot$ for l-sequence predicates:

$$\sigma_l \odot \sigma_k = \sigma_l \wedge next(\sigma_k,l)$$

This predicate evaluates to true for all sequences that begin with a sequence of length l characterized by σ_l and continue with a sequence of length k characterized by σ_k, where the last state in the l-sequence is the first state in the k-sequence.

A special l-sequence predicate called $any_l(\pi_l)$ is defined to evaluate to true for every sequence π_l of length l.

Together with the transition relation of the Kripke model, an l-sequence predicate becomes an *l-path predicate*:

$$P_l(\pi_l) = P_l((s_0,s_1,\ldots,s_l)) = \sigma((s_0,s_1,\ldots,s_l)) \wedge \bigwedge_{i=1}^{l} T(s_{i-1},s_i)$$

We define the general path predicate *ispath*:

$$ispath((s_0,s_1,\ldots,s_l)) = \bigwedge_{i=1}^{l} T(s_{i-1},s_i)$$

It represents an unrolling of the transition relation into l time frames and evaluates to true if the l-sequence given as its argument is a valid path in the Kripke model.

III. Path Predicate Abstraction

A. Abstract and Concrete Kripke Model

In *path predicate abstraction* we consider a concrete Kripke model (S,I,R,A,L) and an abstract Kripke model denoted by $(\hat{S},\hat{I},\hat{R},\hat{A},\hat{L})$. The two are related to each other based on a mapping of *important states* of the concrete model to abstract states and a mapping of *finite paths between important states* to abstract transitions.

Important states are identified and characterized using state predicates $\eta_i(s)$. The vector of important state predicate values, $(\eta_1(s),\eta_2(s),\ldots)$, defines an abstract state value for every concrete state s. This is the abstraction function, $\alpha(s) := (\eta_1(s),\eta_2(s),\ldots)$ mapping a concrete state to an abstract state. The set $\hat{A}$ of atomic formulas of the abstract Kripke model comprises one state variable $\hat{a}_i$ for every important state predicate $\eta_i(s)$.

Definition 1: An *important-state predicate* $\eta_i(s)$ is a predicate evaluating to *true* for a set of concrete important states s and to *false* for all other states. The disjunction of all $\eta_i(s)$ is a state predicate $\Psi(s) = \eta_1(s) \vee \eta_2(s) \vee \ldots$ characterizing the set of all important states. Finally, we require that the η_i satisfy the *important-state requirements* stated in Def. 3, below.

Definition 2: An *operational l-path* between two important states, $s_B \in S$ and $s_E \in S$, is an l-path $(s_B,s_1,\ldots,s_{l-1},s_E)$ with $l > 0$ such that $\Psi(s_B) = true$ and $\Psi(s_E) = true$ and

all intermediate states $s_1,\ldots,s_{l-1}$ are unimportant states, i.e., $\Psi(s_1)=\ldots=\Psi(s_{l-1})=\mathit{false}$.

The important state predicates cannot be chosen arbitrarily. Instead, the choice must satisfy two constraints in order to be useful for the proposed abstraction.

Definition 3: The important-state predicates are defined to fulfill the following *important-state requirements*:

1) For all pairs of (concrete) important states $s_B, s_E \in S$ between which there exists an operational l-path, there is an $l_{\max}$ such that every operational l-path between s_B and s_E is of length $l_{\max}$ or shorter: $l \leq l_{\max}$.
2) For every pair of important-state predicates, $\eta_B(s)$, $\eta_E(s)$, such that there exists a finite operational path $(\tilde{s},\ldots,\tilde{s}')$ with $\eta_B(\tilde{s})=\mathit{true}$ and $\eta_E(\tilde{s}')=\mathit{true}$ it holds that there also exists an operational path $(s,\ldots,s')$ for *every* state s satisfying $\eta_B(s)=\mathit{true}$ and *some* state s' satisfying $\eta_E(s')=\mathit{true}$.

The first constraint requires that *all cyclic paths in the concrete model intersect an important state*, i.e., there are only finite operational paths between important states. The second constraint is more difficult to understand: it ensures that abstract paths assembled from abstract transitions can always be mapped to some concrete path, i.e., there are no false abstract paths. This requirement is "automatically" fulfilled by the operation-oriented property checking technique introduced later.

Definition 4: We consider an abstraction function α such that the important-state predicates η_i fulfill the requirements of Def. 3. Then, the transition relation $\hat{R}\subseteq\hat{S}\times\hat{S}$ of the abstract model is given by:

$$\hat{R}=\{(\hat{s},\hat{s}')|\exists l:\exists(s_0,s_1,\ldots,s_l):\mathit{ispath}((s_0,s_1,\ldots,s_l))$$
$$\wedge\alpha(s_0)=\hat{s}\wedge\alpha(s_l)=\hat{s}'\wedge\neg\Psi(s_1)\wedge\ldots\wedge\neg\Psi(s_{l-1})$$

In this definition, $(s_0,s_1,\ldots,s_l)$ denotes an operational path of length l (i.e., l transitions and $l+1$ states) in the concrete Kripke model. The transition relation contains all pairs $(\hat{s},\hat{s}')$ where $\hat{s}$ and $\hat{s}'$ are head and tail of a path between important states such that all intermediate states are non-important.

Besides mapping a set of concrete states into a single abstract state as in in standard predicate abstraction, the proposed path predicate abstraction also maps a set of concrete paths into a single abstract transition. Therefore, we refer to the proposed abstraction as a "path predicate abstraction". Note, however, that such path predicate abstraction only leads to sound models since we require certain conditions on the state predicates to be fulfilled as stated in Def. 3.

B. Model Checking on the Abstract Model

In this section we show that the proposed abstraction can be used to prove CTL safety and liveness properties of the concrete model. Similar results could be obtained for other temporal logics such as LTL.

Theorem 1: Consider a formula $\hat{f}$ for the abstract model from Table I. The formula has the form $\hat{f}=<\text{CTL operator}>\hat{p}(\hat{a}_1,\hat{a}_2,\ldots)$, with $\hat{p}$ being a Boolean formula of only $\hat{a}_i\in\hat{A}$, i.e., atomic formulas of the abstract model. The corresponding formula f from Table I for the concrete model has the form $f=<\text{CTL operator}>(\Psi\wedge p)$ or the form $f=<\text{CTL operator}>(\Psi\Rightarrow p)$, where p is the Boolean formula obtained by replacing the $\hat{a}_i$ in $\hat{p}$ by their corresponding important-state predicates: $p=\hat{p}(\hat{a}_1:=\eta_1(s),\hat{a}_2:=\eta_2(s),\ldots)$.

abstract formula $\hat{f}$	concrete formula f
$\mathsf{EF}\,\hat{p}$	$\mathsf{EF}\,(\Psi\wedge p)$
$\mathsf{EG}\,\hat{p}$	$\mathsf{EG}\,(\Psi\Rightarrow p)$
$\mathsf{AF}\,\hat{p}$	$\mathsf{AF}\,(\Psi\wedge p)$
$\mathsf{AG}\,\hat{p}$	$\mathsf{AG}\,(\Psi\Rightarrow p)$

TABLE I
ABSTRACT FORMULAS VS CONCRETE FORMULAS

If and only if the formula $\hat{f}$ holds for a state $\hat{s}\in\hat{S}$ of the abstract model then the corresponding formula f from Table I holds for the corresponding concrete states, i.e., for all states $s\in S$ of the concrete model such that $\hat{s}=\alpha(s)$.

Proof: We prove the theorem for the first row of Table I. First it is proved that if $\mathsf{EF}\,\hat{p}$ holds in an abstract state then $\mathsf{EF}\,(\Psi\wedge p)$ holds in all corresponding concrete states.

If $\mathsf{EF}\,\hat{p}$ holds in a state $\hat{s}_0$ in the abstract model then there exists a finite path $(\hat{s}_0,\hat{s}_1,\ldots,\hat{s}_n)$ of n abstract transitions such that $\hat{p}$ holds in $\hat{s}_n$. For every abstract transition $(\hat{s}_i,\hat{s}_{i+1})$, according to Def. 4, there exists an operational finite l-path $(s_{i,0},s_{i,1},\ldots,s_{i,l})$ from an important concrete state $s_{i,0}$ such that $\alpha(s_{i,0})=\hat{s}_i$ to an important concrete state $s_{i,l}$ such that $\alpha(s_{i,l})=\hat{s}_{i+1}$. Then, according to requirement 2 of Def. 3, for *every* important state s_i such that $\alpha(s_i)=\hat{s}_i$ there exists an operational l-path $(s_{i,0},s_{i,1},\ldots,s_{i,l})$ from *every* important state $s_{i,0}$ such that $\alpha(s_{i,0})=\hat{s}_i$ to some important state $s_{i,l}$ such that $\alpha(s_{i,l})=\hat{s}_{i+1}$. (Note that l may be different for every path.) The same argument holds for the important states mapped to $\hat{s}_{i+1}$. Hence, there must exist a finite path $(s_{0,0},s_{0,1},\ldots,s_{1,0},s_{1,1},\ldots,s_{n,l})$ from every important state $s_{0,0}$ such that $\alpha(s_{0,0})=\hat{s}_0$ to some important state $s_{n,l}$ such that $\alpha(s_{n,l})=\hat{s}_n$. Since $\hat{p}$ holds in $\hat{s}_n$ and, therefore, $\Psi\wedge p$ holds in all important states $s_{n,l}$ mapped to $\hat{s}_n$, this means that $\mathsf{EF}\,(\Psi\wedge p)$ holds in all important states $s_{0,0}$ mapped to $\hat{s}_0$.

We now prove that if $\mathsf{EF}\,(\Psi\wedge p)$ holds in an important concrete state then $\mathsf{EF}\,\hat{p}$ holds in the corresponding abstract state. If $\mathsf{EF}\,(\Psi\wedge p)$ holds in an important concrete state s_0 then there exists a finite path $(s_0,s_1,\ldots,s_n)$ from s_0 to an important concrete state s_n such that $(\Psi\wedge p)$ holds in s_n. The path can be split up into segments $(s_i,s_{i+1},\ldots,s_{j-1},s_j)$ such that s_i and s_j are important states and the intermediate states $s_{i+1}\ldots,s_{j-1}$ are non-important states. According to Def. 4, for every such segment of the concrete path there exists an abstract transition $(\hat{s}_i,\hat{s}_j)\in\hat{R}$ such that $\alpha(s_i)=\hat{s}_i$ and $\alpha(s_j)=\hat{s}_j$. Hence, there exists an abstract path from $\hat{s}_0$ to $\hat{s}_n$. Because $(\Psi\wedge p)$ holds in the concrete state s_n the abstract property $\hat{p}$ holds in $\hat{s}_n$. Therefore, there exists an abstract path from $\hat{s}_0$ to a state where $\hat{p}$ holds, i.e., $\mathsf{EF}\,\hat{p}$ holds in $\hat{s}_0$.

The proof for the second row of Table I for EG formulas is very similar to the above proof for EF formulas. The only

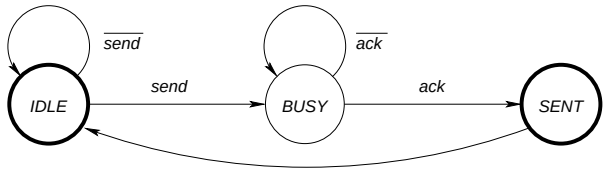


Fig. 1. Concrete FSM

difference is that in the translation from abstract properties to concrete properties, the properties are evaluated only on important states, i.e., $\hat{p}$ maps to $(\Psi \Rightarrow p)$ and vice versa. We omit this proof for reasons of space. The proof for the third row of Table I follows directly from $\mathsf{AF}\, p = \neg\mathsf{EG}\, \neg p$. Likewise, the proof for the fourth row follows from $\mathsf{AG}\, p = \neg\mathsf{EF}\, \neg p$. ■

The proposed path predicate abstraction is related to the notion of stuttering bisimulation [16]. It also decomposes infinite runs into segments of finite length that are matched segment by segment. However, we only require the important starting and ending states of the segments to be matched by the abstraction function and do not care about the intermediate state predicates. Furthermore, instead of using a theorem proving approach we use IPC to establish this weaker correlation of the models.

IV. Complete Interval Property Checking

In this section we revisit Interval Property Checking (IPC) [5], [6]. We also restate Completeness Checking for sets of IPC properties as proposed in [8], [9]. An important purpose of this paper is to show that both together can be used to create a path predicate abstraction as described in Section III.

Interval property checking is based on standard Mealy- or Moore-type finite state machine models. CTL model checking is based on Kripke models. A Moore model can be translated into a Kripke model in a straightforward way by introducing "state variables" (i.e., atomic formulas) for every input. We encode the state space of the Kripke model by the state vector $\underline{s} = (\underline{s_z}, \underline{s_x})$. The sub-state vector s_x represents the set of input values. Every state s contains in its sub-state vector s_x what combination of input values made the system transition into the state s. (This is equivalent to latching the input variables.)

In the following sections, we use the FSM of Figure 1 as a simple running example. The Moore machine stays in *IDLE* until it receives a *send* command. When the command comes it moves to state *BUSY*, sending out a request (output, not shown). In state *BUSY* it waits for an acknowledge *ack*. When the acknowledge comes it moves to state *SENT* where it signals completion to its client (output, not shown). Then it moves back to state *IDLE*.

In our IPC-based abstraction, we adopt an operational view on the design to come up with a complete set of IPC properties. An IPC operation property covers the behavior of a design moving from one *important* state to another important state within a finite time interval [6]. Operations in industrial practice typically describe one or several computational steps in a SoC module such as instructions of a processor, or processing steps of communication structures such as in transactions of a protocol. An operation (interval) property typically spans up to

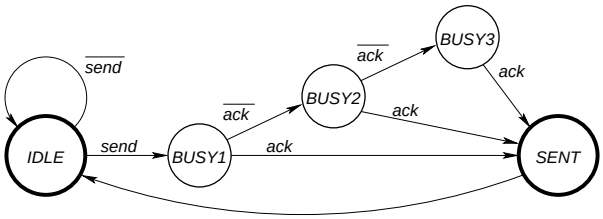


Fig. 2. Example of concrete FSM with a timing constraint on *ack*

P_1:	assume:	at t: *IDLE* ; at t: $\overline{\textit{send}}$;
	prove:	at $t+1$: *IDLE* ;
P_2:	assume:	at t: *IDLE* ; at t: *send* ; at $t+1$: *ack* ;
	prove:	at $t+2$: *SENT* ;
P_3:	assume:	at t: *IDLE* ; at t: *send* ; at $t+1$: $\overline{\textit{ack}}$; at $t+2$: *ack* ;
	prove:	at $t+3$: *SENT* ;
P_4:	assume:	at t: *IDLE* ; at t: *send* ; at $t+1$: $\overline{\textit{ack}}$; at $t+2$: $\overline{\textit{ack}}$; at $t+3$: *ack* ;
	prove:	at $t+4$: *SENT* ;
P_5:	assume:	at t: *SENT* ;
	prove:	at $t+1$: *IDLE* ;

TABLE II
Operational IPC properties

a few hundred clock cycles and can have up to a few million gates in its cone of influence. By unfolding the design into its operations IPC provides a functional view on the design that is orthogonal to the conventional structural view at the RT level. Industrial practice has proved that this is very effective in finding bugs.

A. Operations and Important States

Consider the example of Figure 1. The verification engineer chooses *IDLE* and *SENT* to be the important states — this is indicated by bold circles. We can identify three basic operations in this design: one staying in *IDLE*, one moving from *IDLE* to *SENT* and one moving back from *SENT* to *IDLE*.

Obviously, there is an infinite path in the Moore model between states *IDLE* and *SENT* that cannot be represented in an IPC property. A technical solution is to add an input constraint to the model. In our example, we assume that *ack* is asserted at most three clock cycles after entering state *BUSY*. Note that the verification as well as the abstraction of Section IV-E are based on the validity of such a constraint. In most practical cases, however, constraints can be justified by RT-level verification of other modules of the system. Figure 2 shows the Moore model resulting from the input constraint.

Table II shows a set of five IPC properties describing all possible operations between the important states *IDLE* and *SENT* in the Moore FSM of Fig. 2. Note that IPC properties are always formulated over finite time intervals, hence the requirement 1 of Def. 3 is always fulfilled if path predicate abstraction is based on IPC.

Figure 3 shows the concrete Kripke model of our example. Since there are 5 states in the constrained Moore FSM we need (at least) 3 state variables for encoding them. The state encoding is chosen as follows: *IDLE* = 000, *BUSY1* = 100, *BUSY2* = 101, *BUSY3* = 110, *SENT* = 111. We need 2 more

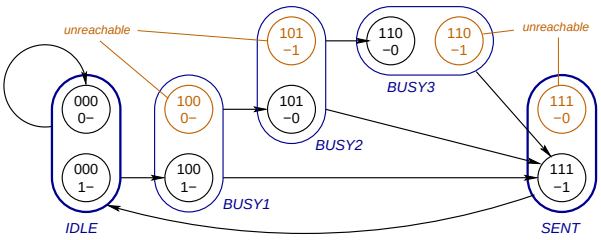


Fig. 3. Concrete Kripke model

state variables to encode the input variables *send* and *ack* . The state transition graph of the Kripke model in Figure 3 shows the 5-bit state codes inside each node.

In our examples, if we group states to abstract states this is indicated by drawing extra circles around these states. The enclosed states may themselves be abstract states. When we draw a transition edge such that it ends (or begins) at a surrounding circle we implicitly mean it to end (or begin, respectively) at every state represented by that circle.

B. Interval Property Checking

An *operation property* or *interval property* P is a pair (A_l, C_l) where both A_l (called *assumption*) and C_l (called *commitment*) are l-sequence predicates. The property checker proves that if the assumption holds on the design (given by the l-path predicate *ispath*()) the commitment does too, for all starting states s_0:

$$A((s_0,s_1,\ldots,s_l)) \wedge ispath((s_0,s_1,\ldots,s_l)) \Rightarrow C((s_0,s_1,\ldots,s_l))$$

Both sequence predicates A_l and C_l are defined over sequences of length l. The parameter l is called the *length* of the property. Since the property is implicitly checked for all possible starting states s_0 (not just the initial state of the system) it is a safety property. The implication can be rewritten in the following equivalent form:

$$ispath(\pi_l) \Rightarrow (A_l(\pi_l) \Rightarrow C_l(\pi_l))$$

where $\pi_l = (s_0,s_1,\ldots,s_l)$ is an l-sequence and $ispath(\pi_l) = \bigwedge_{i=1}^{l} T(s_{i-1},s_i)$ is the unrolling of the transition relation into l time frames. The property check can be formulated as a SAT problem that searches for a a path π_l in the Kripke model where the implication does not hold. The path π_l is then a *counterexample* of the property. It is a false counterexample if the state s_0 in the path is unreachable from the initial state.

In order to rule out unreachable counterexamples in practice, it is common to add invariants to the proof problem [6]. The strengthened proof problem looks like this:

$$(\Phi(s_0) \wedge ispath(\pi_l)) \Rightarrow (A_l(\pi_l) \Rightarrow C_l(\pi_l))$$

where $\Phi(s)$ is a state predicate characterizing an over-approximation of the reachable state set and s_0 is the head (i.e., the starting state) of the l-sequence π_l. If we re-write the implication in the following equivalent form:

$$ispath(\pi_l) \Rightarrow ((\Phi(s_0) \wedge A_l(\pi_l)) \Rightarrow C_l(\pi_l))$$

we can see that the predicate $\Phi(s_0)$ may simply be included in the assumption part of the property in order to add it to the proof.

The properties we consider in this paper have a special form. The assumption A_l of a property P is an l-sequence predicate of the form

$$A_l((s_0,s_1,\ldots,s_l)) = Z(s_0) \wedge X_l((s_0,s_1,\ldots,s_l)). \qquad (1)$$

Here, $Z(s_0)$ is a state predicate characterizing an important state from which the operation starts, and $X_l(\pi_l)$ characterizes a trigger sequence for the operation. The predicate $Z(s_0)$ is expressed only in state variables of the Moore machine, i.e., it is independent of input variables.

The commitment C_l is an l-sequence predicate of the form

$$\begin{aligned} C_l((s_0,s_1,\ldots,s_l)) = Y_l((s_0,s_1,\ldots,s_l)) \wedge Z(s_l) \\ \wedge \neg\Psi(s_1) \wedge \ldots \wedge \neg\Psi(s_{l-1}) \end{aligned} \qquad (2)$$

The state predicate $Z(s_l)$ characterizes the important state in which the operation ends. Again, $Z(s_l)$ refers only to state variables of the Moore machine and not to input variables. The output sequences produced in the operation are characterized by $Y_l(\pi_l)$. The state predicate $\neg\Psi(s_i)$ checks that every intermediate state s_i visited in the operation is an un-important state. This is not needed in conventional IPC but is inserted here to fulfill Def. 4.

Writing the properties in this way ensures that we only consider operational paths as defined in Def. 2. In practice, we can obtain the desired forms of Eq. 1 and 2 by following some coding conventions for writing properties, e.g., by defining appropriate macros as supported by commercial tools.

To continue our running example, the assumptions and commitments of the five properties are given by the l-sequence predicates listed below. In the commitments, the important-state predicate $\Psi(s)$ is given by $\Psi(s) = IDLE\ (s) \vee SENT\ (s)$.

$$\begin{aligned}
A_1((s_0,s_1)) &= IDLE\ (s_0) \wedge \neg send\ (s_1) \\
C_1((s_0,s_1)) &= IDLE\ (s_1) \\
A_2((s_0,s_1,s_2)) &= IDLE\ (s_0) \wedge send\ (s_1) \wedge ack\ (s_2) \\
C_2((s_0,s_1,s_2)) &= SENT\ (s_2) \wedge \neg\Psi(s_1) \\
A_3((s_0,s_1,s_2,s_3)) &= IDLE\ (s_0) \wedge \\
&\quad send\ (s_1) \wedge \neg ack\ (s_2) \wedge ack\ (s_3) \\
C_3((s_0,s_1,s_2,s_3)) &= SENT\ (s_3) \wedge \neg\Psi(s_1) \wedge \neg\Psi(s_2) \\
A_4((s_0,s_1,s_2,s_3,s_4)) &= IDLE\ (s_0) \wedge send\ (s_1) \wedge \\
&\quad \wedge \neg ack\ (s_2) \wedge \neg ack\ (s_3) \wedge ack\ (s_4) \\
C_4((s_0,s_1,s_2,s_3,s_4)) &= SENT\ (s_4) \wedge \neg\Psi(s_1) \wedge \neg\Psi(s_2) \wedge \neg\Psi(s_3) \\
A_5((s_0,s_1)) &= SENT\ (s_0) \\
C_5((s_0,s_1)) &= IDLE\ (s_1)
\end{aligned}$$

C. Property Language

In industrial practice, IPC properties can be formulated, for example, in SVA, or in ITL (*InTerval Language*), a proprietary language developed by OneSpin Solutions [5] that is well adapted to interval property checking. This language can be mapped to a subset of LTL as described in the following.

Definition 5: An *interval LTL formula* is an LTL formula that is built using only the Boolean operators $\wedge$, $\vee$, $\neg$ and the "next-state" operator X.

Let us define a generalized next-state operator X^t that denotes finite nestings of the next-state operator, i.e., if p is an interval LTL formula, then $\mathsf{X}^t(p) = \mathsf{X}\left(\mathsf{X}^{t-1}\right)$ for $t > 0$ and $\mathsf{X}^0(p) = p$.

Definition 6: An interval LTL formula is in time-normal form if the generalized next-state operator X^t is applied only to atomic formulas.

Since in LTL, $\mathsf{X}(a \vee b) = \mathsf{X}a \vee \mathsf{X}b$ and $\mathsf{X}(a \vee b) = \mathsf{X}a \vee \mathsf{X}b$ and $\neg \mathsf{X}a = \mathsf{X}\neg a$, any interval LTL formula can be translated to time-normal form. It is easy to see how an interval LTL formula can be used to specify an l-sequence predicate: The generalized next-state operator refers to the state variables of the system at the different "time" points in the sequence.

The ITL language can be used to specify interval LTL formulas and, hence, l-sequence predicates, using convenient syntax extensions. Consider the example of the property set shown in Table II. The "assume" and "prove" keywords are used to identify the assumption and commitment formulas, respectively. Each formula is a list of sub-formulas that are implicitly conjoined. A subformula is a Boolean expression over design variables, preceded by the definition of a time point using the "at" keyword. The time point "at t:" corresponds to the operator X^t as defined above.

For usability, ITL has many more syntactic extensions. For example, several sub-properties can be considered together disjunctively in a single property. In our example, properties P_2, P_3 and P_4 would result from a single "property" statement in ITL, succinctly describing the operation moving from *IDLE* to *SENT*. Also, expressions can be encapsulated for re-use and code structuring in so-called *macros*. For example, in our property set we have two state predicates, *IDLE* and *SENT* that have been formulated as ITL macros over the state variables of the design. They define the important states that will be the states of the abstract model.

D. Complete Interval Property Checking

In this section, we describe *Complete Interval Property Checking (C-IPC)* [8], [9]. It is based on a completeness criterion developed independently also by Claessen [17]. We will see that operation properties match well with this notion of completeness and that the completeness check becomes computationally tractable in combination with IPC.

The completeness criterion in [9], [8], [17] answers the question whether a set of properties fully describes the input/output behavior of a design implementation. The property suite is called *complete* if for every input sequence the property suite defines a unique output sequence that is to be produced by the implementation, according to some *determination requirements*. The basic idea presented in this section is to prove this inductively by considering chains of operation properties.

The determination requirements specify the times and circumstances when specific output signals need to be *determined* through the design. As an example: data on a bus only needs to be determined when the "data valid" signal is asserted. A determination requirement for the data signal could be written as "if (datavalid = true) then determined(data)". In general, a determination requirement is a pair (o, σ_s) for a signal o (here: data) and a guard σ_s given as an l-sequence predicate (here: datavalid) characterizing the temporal conditions when the signal o is to be determined. A signal is called *determined* by an operation at a certain time point if its value at this time point can be uniquely calculated from the start state Z of the operation, from its trigger condition X, or from other determined signals. These other determined signals can, for example, belong to the operands of a data path. If the operation performs an addition then the result signals are determined if the input operands are determined. It is checked for the reset state of the system that it fulfills all determination requirements. This is the induction base of an inductive proof.

In C-IPC the set of operation properties written by the verification engineer completely covers the state transition graph of the design's finite state machine. Any input/output sequence produced by the design, starting from reset, can be split up into a corresponding sequence of operations, each defined by one operation property. For each individual operation we can verify the functionality and we can check whether the determination requirements are fulfilled in that operation, provided the previous operation did also fulfill its own determination requirements. This is the induction step of an inductive proof.

Definition 7: A property set is complete if two arbitrary finite state machines satisfying all properties in the set are sequentially equivalent in the signals specified in the determination requirements at the time points characterized by the guards of the determination requirements. $\diamond$

Completeness of a set of $n+1$ properties $V = \{P_0, P_1, \ldots, P_n\}$, with P_0 being the reset property, is checked in the following way. Besides the determination requirements mentioned above, the user specifies a *property graph* $G = (V, E)$ where the nodes $V = \{P_i\}$ are the properties. Each property P_i is a pair (A_i, C_i) where both the assumption A_i and the commitment C_i are l-sequence predicates; l is called the length of the property P_i. Every property P has its own length l_P. The edges of the property graph describe the concatenation (sequencing) of operations. There is an edge $(P_j, P_k) \in E$ if the operation specified by P_k can take place immediately after the operation specified by P_j. (This is the case if operation P_j starts in the important state that is reached by operation P_k.)

Note that, in principle, the property graph could be determined automatically from the set of properties. However, for better debugging an incomplete and possibly incorrect property suite the user is required to specify the property graph which only involves a small extra effort.

The completeness engine performs three checks on the property graph G: a *case split test*, a *successor test* and a *determination test*, all described below. It is important to note that the completeness checks are carried out without consideration of the design.

1) Case Split Test: The case split test checks that all paths between important states in the design are described by at least one property in the property suite, i.e., that all input scenarios in an important state are covered. The set of important states is given by the commitments $\{C_i\}$ of the properties $\{P_i\}$. For every important state (given by a commitment C_P) reached in an operation P it is checked whether the disjunction of the assumptions $\{A_{Q_j}\}$ of all successor properties Q_j completely

covers the commitment C_P, i.e., for every path starting in a substate of the important state C_P there exists an operation property Q_j whose assumption A_{Q_j} describes the path. Let $\{A_{Q_1}, A_{Q_2}, \ldots\}$ be the set of assumptions, then the case split test checks if

$$C_P \odot any_{l_Q} \Rightarrow any_{l_P} \odot (A_{Q_1} \vee A_{Q_2} \vee \ldots)$$

In this expression, l_P is the length of property P and l_Q is the length of the longest successor property Q_j. The any_l sequence predicate defined in Section II is used to make both sides of the implication a sequence predicate of length $l_P + l_Q$.

If the case split test succeeds this means that for every possible input trace of the system there exists a chain of properties that is executed. However, this chain may not be uniquely determined. Therefore, the following successor test is performed.

2) Successor Test: The successor test checks whether the execution of an operation Q is completely determined by every predecessor operation P. For every predecessor/successor pair $(P,Q) \in E$ it is checked whether the assumption A_Q of property Q depends solely on inputs or on signals *determined* by the predecessor P.

The successor test creates a SAT instance that is satisfied if there exist two state sequences, π_1 and π_2, such that π_1 represents an execution of operation P followed by operation Q and the other represents an execution of operation P followed by another operation not being Q, with the additional constraint that the inputs and determined variables are the same in both sequences. The execution of P followed by Q is expressed through $(A_P \wedge C_P) \odot A_Q$, the execution of P followed by not-Q is expressed through $(A_P \wedge C_P) \odot \neg A_Q$. If the SAT check succeeds then, according to A_Q, triggering of the operation Q is decided non-deterministically. This is the case if the assumption A_Q was written such that it depends on some state variables other than inputs and variables determined by P.

What is most important for our work here is that the successor test (as a side product) makes sure that for all pairs $(P,Q) \in E$:

$$any_{l_P} \odot A_Q \Rightarrow C_P \odot any_{l_Q}.$$

The expression states that the successor operation Q always starts in an (important) state s_l that is reached by a predecessor operation P.

Having established that there exists a unique chain of operations for every input trace it remains to be shown that these operations determine the output signals as stated in the determination requirements. This is the task of the determination test.

3) Determination Test: The determination test checks whether each property Q fulfills its determination requirements provided the predecessor operation P, in turn, fulfilled its determination requirements.

The test creates a SAT instance that is satisfied if a determination requirement is violated. The satisfying set represents two state sequences, π_1 and π_2, that both represent an execution of operation P followed by operation Q, with the additional constraint that the inputs and the variables determined by P are the same in both sequences, such that π_1 and π_2 have different values for some signal that should be determined by Q.

The three completeness tests all contribute to an inductive proof. The induction is rooted at the reset, represented by the reset property P_0 that does not have a predecessor. The induction base is established through a separate *reset test* that checks whether reset can always be applied deterministically and whether reset fulfills all determination requirements.

E. Abstraction by C-IPC

It is now shown that C-IPC with a set of properties written in the form of Eq. 1 and Eq. 2 of Section IV-B leads to an abstract Kripke model that is a path-predicate abstraction of the design under verification according to Section III.

As described above, the methodology produces a set of properties, V, and a property graph $G = (V,E)$ for which the completeness tests have been successfully carried out. A basic element of the created abstraction are the *important states* given by state predicates that are used in the properties to characterize the starting states s_0 of an operation in the assumption and the ending states s_l of the operation in the commitment. The important-state predicates defining the abstraction function $\alpha(s)$ are given by the set of all important-state predicates $\{Z_i(s)\}$ appearing in the properties: $\alpha(s) := (Z_1(s), Z_2(s), \ldots)$. The abstraction function maps every concrete state of the design to an abstract state.

It must be shown that the transition relation $\hat{R}$ of the abstract Kripke model is given by the set of properties in the following way: there is a transition from one abstract state $\hat{s}$ to another one $\hat{s}'$ if and only if there exists a proven property P describing an operation that starts in the important state $\hat{s}$ and that ends in $\hat{s}'$ according to Def. 4. Moreover, the requirements for the state predicates of Def. 3 must be fulfilled.

The IPC proof engine, when proving the property P, verifies for a given pair of important states forming an abstract transition $(\hat{s},\hat{s}')$ that there exists a corresponding operational path as given in Def. 4. It is obvious that the first requirement of Def. 3 is always fulfilled in IPC. Since every operation is proved for all concrete important states described by a state predicate $Z_i(s)$ and a trigger condition $X_i(s)$ the Kripke model will also fulfill the second requirement of Def. 3. For *all* concrete states fulfilling $Z_i(s)$ there is a path in the Kripke model to some state fulfilling the ending state condition of the operation and the trigger condition that leads into this state.

It remains to be shown that there is a property for every abstract transition fulfilling Def. 4, and for every property there is an abstract transition. This follows from the case split test and the successor test. The case split test makes sure that for every path leaving an important state in the concrete model there is a property, i.e., an abstract transition, describing that path. The successor test makes sure that properties describe only paths actually starting in an important state reached by some other property, i.e., for every abstract transition there also exists a succeeding abstract transition.

Thus, the abstraction produced by means of C-IPC fulfills all requirements as stated in Section III and is sound to prove safety and liveness properties for the concrete system.

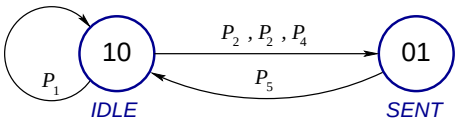


Fig. 4. Abstract Kripke model

Fig. 4 shows the abstract Kripke model of our example. The model has two important states *IDLE* and *SENT* . There is an edge between two important states if there is an IPC property describing a path between the two.

V. Experiments

Two sets of experiments were made to evaluate the usefulness of C-IPC-based abstraction. The first set is a case study on an experimental serial bus system [18]. The second set of experiments was made on a system built using Infineon's Flexible Peripheral Interconnect (FPI) bus.

In both experiments we have a set of modules communicating over a bus. *Clients* connect to the bus through *bus agents*. Each bus agent has one interface to the bus and another interface to the client. (The client could, e.g., be a CPU core or a peripheral.)

A. Serial Bus System

The communication system used in the first set of experiments is a custom-made serial bus. The protocol uses certain elements from different "real-world" serial communication protocols; for example, it uses CSMA (Carrier Sense Multiple Access) with bitwise arbitration as in CAN, and synchronization is done as in RS232 using start and stop bits.

Using C-IPC with OneSpin 360MV the bus agent was verified and a complete set of properties was obtained. The corresponding abstract state machine was manually translated into VHDL. This step will have to be automated in our ongoing work, but is here guided by a coding convention that makes the abstract states and abstract transitions obvious. Note that only the bus agents were abstracted. The clients and the interface between a client and its bus agent remained the same so that properties could be checked on the concrete and the abstract system. The clients were designed to implement a token passing mechanism among them.

Number of agents	Concrete system CPU time	Concrete system Memory	Abstract system CPU Time	Abstract system Memory
3	0.32s	78MB	0.04s	37MB
5	1.75s	158MB	0.12s	43MB
8	1min 46s	735MB	0.38s	74MB
12	54min 59s	1372MB	1.03s	109MB
15	—	—	1.89s	155MB
30	—	—	9.09s	514MB

TABLE III
IPC property checked on concrete and abstract system

Table III shows the results for checking an IPC property on different abstract system configurations using OneSpin 360MV. The design was made such that the number of bus participants can be configured by a parameter. The property checks that after reset, token passing is triggered ensuring that there is only one master in the system. Table III shows in each row the number of bus participants and the CPU time and memory consumption for checking the property on the concrete system and on the abstract system. The experiments were run on an Intel Core 2 Duo at 3GHz with 4GB main memory.

For the serial bus system, the particular strength of path predicate abstraction becomes apparent. Each individual agent in the system has 129 state variables in the concrete and 89 state variables in the abstract model. While this reduction of about 30% is not drastic the main reduction in proof complexity comes from temporal abstraction: The individual operations in the concrete model, having lengths of up to 35 cycles, are mapped to abstract single-cycle transitions. A system transaction taking more than a hundred clock cycles of serial transmission is therefore mapped to only a few transitions in the abstract model, reducing temporal length of properties by factors as low as 1/35.

Number of agents	Concrete System CPU Time	Concrete System Memory	Abstract System CPU Time	Abstract System Memory
2	10s	117MB	4s	119MB
3	26s	115MB	9s	346MB
4	1min 16s	461MB	15s	428MB
5	—	—	58s	577MB

TABLE IV
Safety property checked using induction

Table IV shows the results for checking a safety property using the induction prover built into OneSpin 360 MV. The safety property ensures that at any time there is only one master. For more than 4 agents the property cannot be proven on the concrete system, while on the abstract system it is proven in very short CPU time.

B. Industrial FPI Bus System

A more comprehensive evaluation of the proposed method using CTL model checking on the abstract model was done in an industrial case study. The Flexible Peripheral Interconnect bus (FPI bus) owned by Infineon Technologies is used for our experiments. It is an on-chip bus system similar to the industry standard AMBA. The throughput of the FPI bus is optimized by pipelining of transactions and extensive use of combinational logic. This makes it particularly interesting to examine how our approach can be used to abstract from such high-performance implementations and how a "clean" model at the transaction level can be obtained.

The FPI bus is a modular system consisting of master/slave interfaces, a BCU, an address decoder and a bus multiplexer. C-IPC was applied to obtain complete property sets for the modules. From the complete property sets we derived the abstract modules.

In our experiment we implemented our abstraction in the Cadence SMV input language. By extensive use of macros in our IPC-based verification tool (OneSpin 360 MV) the signals of the SoC modules were encapsulated and named so that a one-to-one mapping with the signals of the abstract module was obtained. The implementation of the abstraction also here was a manual step. Correctness can be ensured easily due to the one-to-one mapping between the macros created in OneSpin 360MV and the design description used for Cadence SMV. In this way, abstract modules for the master agent and the BCU were derived. For the slave agent, the address decoder and the bus multiplexer the abstract modules were not derived from C-IPC but created ad-hoc and integrated with the master agent and the BCU to form an abstract system.

	Master agent	BCU
RT code inspection, lines of code	4,000	1,500
Number of properties	17	6
Total runtime of properties	1h 19min	15s
Total runtime of completeness checks	41s	10s

TABLE V
FPI BUS MODULE VERIFICATION

Table V shows some information on the complexity of deriving the abstract modules by C-IPC. Specifically, it presents the approximate number of lines of RTL code which had to be inspected in order to create our abstract models. In general, the manual effort spent in C-IPC is about 2,000 lines of code per person month for an average verification engineer. This figure proved quite accurate also in the case study conducted here.

Based on these industrial SoC modules we assembled a system of three master agents, two slaves, the arbiter as well as bus multiplexers and address decoders. If several complete property suites are composed to completely describe a design assembled from several modules additional checks need to be applied in the completeness methodology to ensure the correctness of the integration conditions [8].

As a result of the proposed methodology the abstract model was obtained for the assembled FPI bus. While the concrete system contained 2,624 state variables only 75 state variables were included in the abstract system. We now used Cadence SMV to prove several liveness and safety properties on the abstract system. All properties are proven on the abstract model within a few minutes using less than 500 MB.

As a liveness property, we have proved that any request from a master will finish successfully within a fixed time under the constraint that a master peripheral only sends requests complying with the protocol, that the starvation prevention is switched on, and that a slave does not stay busy forever. As an example of a safety property, we prove that the bus is correctly driven at any time. Specifically, we proved that the various enable signals (data, address, ready) are one-hot-encoded. According to Theorem 1 this property holds only in the important states. By adding local properties proving that the enable signals do not change value in-between important states we obtain an unrestricted proof of the safety property that now holds for both the important and the unimportant states of the concrete model.

VI. Conclusion

In this paper we presented a methodology to leverage the results of a complete property checking methodology, C-IPC, to create abstractions for system-level verification. Our approach can be understood also as a light-weight theorem proving approach. In theorem proving, building a stack of models to prove system properties is very common. Our results show that such a paradigm is also feasible for property checking by an appropriate methodology. Future work will explore how the proposed abstraction can be integrated into a SystemC-based design and verification flow.

References

[1] D. Kroening and N. Sharygina, "Formal verification of system c by automatic hardware/software partitioning," *Formal Methods and Models for Co-Design*, 2005.

[2] J. Yang and C.-J. H. Seger, "Introduction to generalized symbolic trajectory evaluation," *IEEE Transactions on VLSI Systems*, vol. 11, no. 3, pp. 345–353, 2003.

[3] A. J. Hu, J. Casas, and J. Yangpa, "Reasoning about GSTE assertion graphs," in *Proc. CHARME*. Springer, 2003, pp. 170–184, Lecture Notes in Computer Science Vol. 2860.

[4] R. Sebastiani, E. Singerman, S. Tonetta, and M. Y. Vardi, "GSTE is partitioned model checking." in *Proc. International Conference on Computer-Aided Verification (CAV)*, 2004.

[5] Onespin Solutions GmbH, Germany. OneSpin 360MV.

[6] M. D. Nguyen, M. Thalmaier, M. Wedler, J. Bormann, D. Stoffel, and W. Kunz, "Unbounded protocol compliance verification using interval property checking with invariants," *IEEE Transactions on Computer-Aided Design*, vol. 27, no. 11, pp. 2068–2082, November 2008.

[7] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu, "Symbolic model checking without BDDs," in *Proc. Intl. Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 1999.

[8] J. Bormann, "Vollständige Verifikation," Dissertation, Technische Universität Kaiserslautern, 2009.

[9] J. Bormann and H. Busch, "Method for determining the quality of a set of properties," European Patent Application, Publication Number EP1764715, 09 2005.

[10] R. P. Kurshan, *Computer-Aided Verification of Coordinating Processes – The Automata-Theoretic Approach*. Princeton University Press, 1994.

[11] S. Graf and H. Saïdi, "Construction of abstract state graphs with PVS," in *Proc. International Conference Computer Aided Verification (CAV)*, ser. LNCS, vol. 1254. London, UK: Springer-Verlag, 1997, pp. 72–83.

[12] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Counterexample-guided abstraction refinement for symbolic model checking," *Journal of the ACM*, vol. 50, no. 5, pp. 752–794, 2003.

[13] E. Clarke, M. Talupur, H. Veith, and D. Wang, "SAT-based predicate abstraction for hardware verification," in *Conference on Theory and Applications of Satisfiability Testing*, ser. LNCS, vol. 2919. Springer, 5 2003, pp. 78–92.

[14] H. Jain, D. Kroening, N. Sharygina, and E. M. Clarke, "Word-level predicate abstraction and refinement techniques for verifying RTL Verilog," *IEEE Trans. on CAD*, vol. 27, no. 2, pp. 366–379, 2008.

[15] M. Schickel, V. Nimbler, M. Braun, and H. Eveking, "On consistency and completeness of property sets: Exploiting the property-based design process," in *Proc. Forum on Design Languages*, 2006.

[16] P. Manolios and S. K. Srinivasan, "A refinement-based compositional reasoning framework for pipelined machine verification," *IEEE Transactions on VLSI Systems*, vol. 16, pp. 353–364, 2008.

[17] K. Claessen, "A coverage analysis for safety property lists," in *Proc. International Conference on Formal Methods in Computer-Aided Design (FMCAD)*. IEEE Computer Society, 2007, pp. 139–145.

[18] H. Lu, "A case study on the verification of abstract system models derived through interval property checking," Master's thesis, University of Kaiserslautern, 2009.

Relieving Capacity Limits on FPGA-Based SAT-Solvers

Leopold Haller
Computing Laboratory
Oxford University, United Kingdom
leopold.haller@comlab.ox.ac.uk

Satnam Singh
Microsoft Research
Cambridge, CB3 0FB, United Kingdom
satnams@microsoft.com

***Abstract*—FPGA-based SAT solvers have the potential to dramatically accelerate SAT solving by effectively exploiting fine-grained pipeline parallelism in a manner which is not achievable with regular processors. Previous hardware-based approaches have relied on on-chip memory resources to store data which, similar to a CPU cache, are very fast, but are also very limited in size. For hardware-based SAT approaches to scale to real-world instances, it is necessary to utilise large amounts of off-chip memory. We present novel techniques for storing and retrieving SAT clauses using a custom multi-port memory interface to off-chip DRAM which is connected to a processor core implemented on a medium sized FPGA on the BEE3 system. Since DRAM is slower than on-chip memory resources, the parallelisation which can be achieved is limited by memory throughput. We present the design and implementation of a new parallel architecture that tackles this problem and estimate the performance of our approach with memory benchmarks.**

I. Introduction

SAT solvers have been established as popular black-box reasoning techniques in a number of application areas, most notably, formal verification of hardware and software. This can be partially attributed to the fast rise in solving efficiency over the last 15 years. One possibility of increasing solving efficiency further is to make use of the fine-grained parallelism that is offered by hardware platforms. Previous approaches have relied on on-chip memory resources which are fast and allow for parallelised access, but impose strict limits on the size of input instances.

In this paper, we explore the feasibility of building a hardware-based SAT solver that directly accesses off-chip DRAM memory resources. This has the advantage that the size of SAT instances solved by our hardware solver are orders of magnitude larger than what is possible when storing instance data using only on-chip memory. The disadvantage is that it creates a memory bottleneck due to the memory access characteristics of DRAM. We present an implementation of a Boolean constraint propagation (BCP) unit on the BEE3 multi-FPGA board.

Our design uses novel techniques for clause retrieval and propagation that utilise fine-grained parallelism in spite of this bottleneck. For the clause retrieval step, we adapt the BCP algorithm to independently access multiple memory channels. The unique advantage of our approach is that it does not impose the strict instance size limits that are common with other hardware-based SAT solvers. The evaluation of our implementation is work in progress. We present initial memory benchmarks to estimate the feasibility of our approach.

II. Related Work

A survey of techniques published until 2004 is given in [1]. Early work on reconfigurable hardware SAT focuses on *instance specific* approaches (e.g., [2], [3], [4], [5]), in which a circuit is generated specific to a single SAT instance. This requires computationally expensive circuit resynthesis and reconfiguration of the hardware once a new instance is to be evaluated and severely limits the size of possible input instances. *Application specific* hardware solvers do not require reconfiguration between solving instances. A popular approach is to implement BCP, the most work intensive step of the popular DPLL procedure, on hardware, and handle more complex tasks such as conflict analysis and decision heuristics in software [6], [7], [8], [9]. Fully functional hardware solvers are presented in [10], [11], [12].

Capacity is an issue for all these solvers. Examples of more large-scale approaches include the BCP accelerator presented in [9], which can accommodate 64K variables and equally many clauses of length 9, or the solver in [12], which can accommodate 10K variables and 280K fixed-length clauses. Many SAT instances of practical interest are not representable within these restrictions.

A number of methods have been proposed to increase the capacity of hardware-based solvers: Examples include using a larger FPGA [4] or multiple FPGAs [7], [8], [9], splitting the problem into small subproblems [5], partitioning the instance into small-sized frames that are loaded on-demand [12], or combining a software solver with a hardware solver for small-size subproblems [10]. Our approach, in contrast, explores the feasibility of directly accessing off-chip memory resources.

III. Memory Access Patterns in SAT Solvers

In FPGA designs, very small amounts of data can be stored on arrays of state-holding flip-flops. Larger amounts can be stored in dedicated Block RAM (BRAM) modules on the FPGA chip, or off-chip on external RAM. On-chip memory is very limited, with typical sizes smaller than 4MB, but access is fast and can be performed in parallel. Access to DRAM

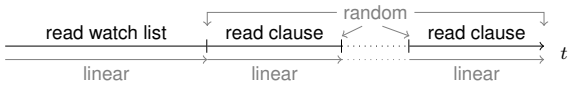


Fig. 1. Watch list and clause read operations in Algorithm 1

memory is performed via an external memory controller using an asynchronous protocol. We have used a freely available controller presented in [13] for our implementation. The time required for a single read and write command depends on a number of factors including access locality, memory clock speed and the implementation of the memory controller. Random accesses are, on average, significantly slower than linear streaming access.

Most modern SAT solvers are based on the Conflict Driven Clause Learning (CDCL) framework, which utilizes clause learning and backjumping ([14], [15]), and spend most of the runtime in Boolean Constraint Propagation (BCP). BCP can be efficiently implemented using a watched-literal scheme [16] where two literals in each clause are observed for changes. Literal watching can be implemented by associating each literal with a list, which records the clauses that have to be visited when the literal is contradicted during search.

Algorithm 1 The BCP step

```
1: procedure Propagate(l : literal)
2:     wl ← readWatchList(l)                          ▷ R
3:     while HasNextClause(wl) do
4:         c ← readNextClause(wl)                     ▷ R
5:         v ← readVariableValues(c)                  ▷ R
6:         s ← analyse(v, c)
7:         if s = Conflict then return Conflict
8:         else if s = Deduction then writeDeducedValue()   ▷ W
9:         changeWatchLits()                          ▷ W
10:    return Unknown
```

In Algorithm 1, the inner core of the BCP algorithm is presented in a way that emphasizes memory accesses. ReadVariableValues fetches the values assigned to variables occurring in a clause. The Analyse function determines the status of a clause and returns a result that indicates if an action needs to be taken. Finally, ChangeWatchLits modifies watch lists in accordance with the two-watched literal scheme.

The memory access pattern for reads is illustrated in Figure 1. Reads are issued in a linear fashion on successive addresses, with intermittent single random accesses. We will refer to this access pattern as *quasi-linear*.

In order to estimate the viability of using DRAM in a reconfigurable-hardware SAT solver, we compared the efficiency of quasi-linear memory accesses on the BEE3 platform with the same access pattern implemented in software and run on an average PC. Since modern CPUs have large multi-level caches which allow fast access to recently used data, it is not a priori clear that the performance is similar, even when similar types of main memory are used.

For our experiments and implementation of the BCP core, we use a pre-production version of the BEE3 FPGA board which has four XC5VLX110T FPGAs. Each FPGA has two independent memory channels connected to dual channel DDR2-533 RDIMMs with each channel populated with 8GB of memory (giving a total of 64GB for the whole system). The FPGAs are connected in a ring and a cross-over board also provides direct connections between the other two FPGAs. Although this platform was primarily developed for the emulation of multi-core processors we believe it is an interesting platform for hardware SAT-solving because of the large amount of off-chip memory and the ability to use eight independent memory channels to experiment with hardware parallelisation techniques.

	PC (400MHz RAM)	Virtex5 (250MHz RAM)
rnd./lin.	93.7 / 1066.7	44.4 / 1066.7
quasi-lin.	691.9	984.6

TABLE I
AVERAGE MEMORY ACCESS SPEED (MB/S)

The BEE3 system also provides a variety of I/O ports including for each FPGA an RS232 serial port, dual 10GBase-CX4 Ethernet interfaces, a single PCI-Express x8 end-point slot and a Gigabit Ethernet port. We use the Gigabit Ethernet port to communicate with a host PC running Windows 7.

We compared DDR2 memory access speed on a 3GHz CPU with memory clocked at 400MHz and a Virtex-5 FPGA with memory clocked at 250MHz. Random address values were precomputed and read (linearly) from an array in the software case, and generated on-the-fly in the hardware case. The memory controller has a granularity of 256 bit per memory access (288-bits including error correction bits). Hence, for the linear and quasi-linear access cases, a single read and write operation can manipulate 8 integers of width 32 at once.

The test setup consisted of reading and incrementing 256MB of 32 bit integers. The results are shown in Table I, which shows read/write speeds for a completely random, linear and quasi-linear access patterns. The quasi-linear access pattern reads 64 words linearly before performing a random address jump. As can be seen from the table, the access speed for quasi-linear access is comparable on the two platforms, despite the lower memory clock speed of the FPGA. Since quasi-linear accesses are characteristic for the DPLL algorithm, this result gives some preliminary indication that a hardware-based implementation of DPLL with direct DRAM access is feasible.

IV. BUILDING A DRAM-BASED BCP-CORE

Since straight-forward highly parallel approaches are not practicable when accessing DRAM directly, we base our implementation on modern software CDCL solvers and enhance it with fine-grained parallelism where possible. The only data which we store on-chip are the current value of variables, all other data is kept in off-chip DRAM. In this situation, it becomes necessary to explore parallelisation techniques that are still viable in the context of the memory bottleneck that is created by off-chip data storage.

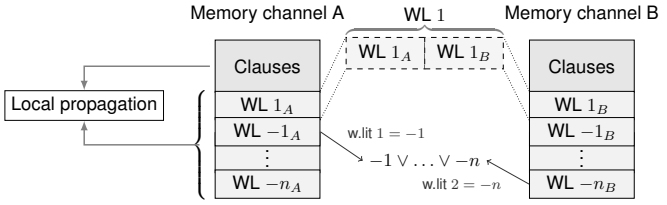


Fig. 2. Parallel watched literal scheme

A. The Parallel Watched-Literal Scheme

While DRAM access is inherently sequential, the BEE3 board retains some options for concurrent off-chip memory operations by offering multiple RAM channels, each of which is connected to its own RAM chip and can be controlled independently.

A key aspect of our approach is the ability to exploit two independent memory channels on each of the FPGAs, since it maps naturally to the two-watched literal scheme. In the watched literal scheme, two literals of each clause are designated and watched for changes. This is implemented by keeping a list for each literal, and appending all clauses to it in which it is being watched.

In our BCP implementation, we parallelise the two-watched-literal scheme by watching each of the two literals of a single clause on a separate memory channel (see Figure 2). Each literal is associated with two watch lists that are stored on separate channels A and B. Clause data is stored redundantly on the two memory chips. This allows to localise the inner core of BCP to require only memory accesses on a single memory channel. When the routine in Algorithm 1 is executed, the two partial watch lists are fetched independently on the two memory chips. After this step, the while-loop at line 3 of the algorithm can be executed completely in parallel by performing propagation local to data stored on each of the memory channels.

Redundant storage of clause data creates a memory overhead that is not significant in view of the large amount of available off-chip memory, but can speed up the processing of a watch list by up to 100%. By dividing the watched literals between the two memory channels, the average length of watch lists on each channel will be equal.

B. Parallel Inference

When relying on off-chip memory resources, clauses need to be read sequentially after the watch list is retrieved. The amount of possible parallelisation in analysing clauses is directly limited by the rate at which clauses can be streamed from memory.

After a clause has been retrieved, its variables' values have to be read (line 5 in Algorithm 1). The actual analysis step (line 6) can then be performed in a single clock cycle by a dedicated analysis circuit. We store variable values on on-chip memory resources. A value can therefore be accessed in a single cycle. Large clauses might still require a number of cycles to fill up all variable values of interest. Depending on how fast a clause can be streamed from memory and how many variable values need to be fetched before the status of a clause can be determined, there can be an overlap with new clauses arriving while a previous clause is still fetching variable values, as illustrated in Figure 3.

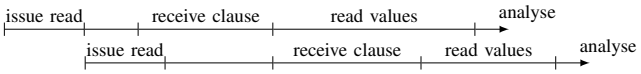


Fig. 3. Timing of the clause analysis step

To speed up these cases, we have implemented a limited form of inference parallelism. The clause analysis step is performed by propagator cores, which are assigned clauses that arrive from memory. Once a core has received a clause, it starts issuing requests for variable values to a common bus, and listening for useful variable values on another bus. Once enough variable values have been received to determine clause status, the core sets a ready flag and waits for the next clause assignment. In most cases, a core does not need to fetch all variable values in order to determine the status of a clause. In case a clause is either satisfied or is neither conflicting nor leads to a deduction, the result can be determined early.

The number of propagator cores is a parameter in our design. Once the analysis speed outpaces clause throughput no further efficiency gains can be obtained by adding cores. In our implementation, we have therefore instantiated the design with two propagation cores per memory channel.

C. Algorithmic Description of the BCP step

We will now give an algorithmic description of our solver, before discussing the implementation architecture. We present an overview in Algorithm 2. The procedures BCP and BCPCore correspond to the hardware modules of the same name that are discussed in the next section.

Algorithm 2 Algorithmic description of hardware BCP

```
1:  procedure BCP(l : literal)
2:      q ← {l}
3:      while |q| > 0 do
4:          p ← pop(q)                                   ▷ pop queue
5:          BCPCore(p, A), BCPCore(p, B)                 ▷ execute in parallel
6:          if conflict(A) ∨ conflict(B) then return Conflict
7:          append(q, deductions(A) ∪ deductions(B))
8:
9:  procedure BCPCore(l : literal, X : memory channel)
10:     wl ← issueReadWatchList(l, X)                    ▷ watch list fetch
11:     while ¬watchListReceived() do
12:         addr ← waitForClauseAddress();
13:         issueClauseRead(addr,X)                      ▷ clause fetch
14:     while ¬allClausesReceived(wl) do                 ▷ clause propagation
15:         c ← waitForClause(); assignToFreePropagationCore(c)
16:     writeBackWL(l,X)                                 ▷ write new watch list for l
17:     appendWatches(X)                                 ▷ append changed watched literals
```

In the BCP step, propagation literals are incrementally taken from a queue, after which BCPCore is executed in parallel on memory channels A and B. Each BCPCore reads its (partial) watch list from memory and issues "clause read" commands as soon as clause addresses are received. Arriving clause data

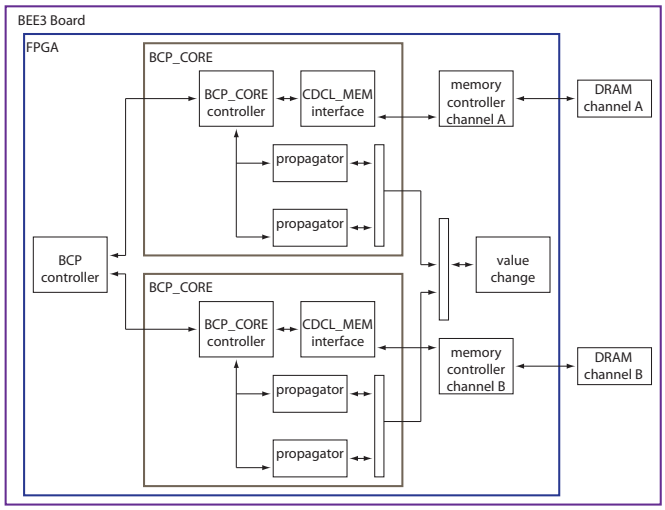


Fig. 4. Implementation Architecture

is distributed to the propagator cores which record their results for later evaluation. The addresses of those clauses which remain in the watch list (e.g., already satisfied clauses) are written back to memory in the call to WriteBackWL. The addresses of all other clauses are appended to their new watch lists in the AppendWatches step. Deduction results are recorded and processed in the main BCP procedure. If no conflict is found, the procedure appends new deduction results to the BCP queue.

D. Implementation Description

In our implementation, we use external DRAM to store clause and watch list information, while we use on-chip BRAM to store variable values. We use an openly available DDR2 memory controller [13].

The architecture of our BCP module is presented in Figure 4. The "BCP controller" block corresponds to the BCP procedure in Algorithm 2. It manages a BCP queue, issues propagate commands to the two "BCP_CORE" modules, and controls the modification of watch lists. The two BCP cores receive propagation literals from the BCP controller, issue memory requests to the "CDCL_MEM interface" module and distribute clauses on their partial watch list between free "propagator" cores. The propagator units access a common bus to read literal values.

In our BEE3 implementation, we limit the clause size to 24 literals to enable efficient propagation, and impose a limit on total size of watch lists to 256 clause addresses (128 per memory channel). This allows storage of instances with up to 1 million variables and 70 million clauses. We have validated our approach in simulation and synthesized our circuit with a memory clock frequency of 200 MHz and control logic clocked at 100 MHz. Obtaining benchmark results is work in progress.

V. Conclusion

In this paper we have presented an implementation of a Boolean constraint propagation core that does not rely on limited on-chip memory resources to store instance data, but instead directly accesses off-chip DRAM. Based on the memory access behaviour of CDCL solvers and the characteristics of DRAM, we have proposed techniques that introduce parallelism in spite of the memory bottleneck created by using off-chip resources. The evaluation of our implementation is work in progress. Our initial exploration is encouraging and we conclude that there is a good potential for implementing high performance parallel hardware SAT solvers by carefully designing and tuning the circuits that make up the memory hierarchy.

Future work includes the completion of the system which drives the parallel hardware BCP core by adapting an existing SAT-solver like MiniSAT and executing it on an embedded soft processor on the Virtex-5 FPGA or on an embedded hard core processor like a PowerPC or ARM core. Currently we use just one of the four FPGAs on the BEE3 system and in future work we hope to exploit all four FPGAs.

References

[1] I. Skliarova and A. B. Ferrari, "Reconfigurable hardware SAT solvers: A survey of systems," *IEEE Trans. Computers*, vol. 53, no. 11, pp. 1449–1461, 2004.

[2] P. Zhong, M. Martonosi, P. Ashar, and S. Malik, "Using configurable computing to accelerate Boolean satisfiability," *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 18, no. 6, pp. 861–868, 1999.

[3] T. Suyama, M. Yokoo, H. Sawada, and A. Nagoya, "Solving satisfiability problems using reconfigurable computing," *IEEE Trans. VLSI Syst.*, vol. 9, no. 1, pp. 109–116, 2001.

[4] M. Platzner and G. D. Micheli, "Acceleration of satisfiability algorithms by reconfigurable hardware," in *FPL*, ser. Lecture Notes in Computer Science, R. W. Hartenstein and A. Keevallik, Eds., vol. 1482. Springer, 1998, pp. 69–78.

[5] M. Abramovici and J. T. de Sousa, "A SAT solver using reconfigurable hardware and virtual logic," *J. Autom. Reasoning*, vol. 24, no. 1/2, pp. 5–36, 2000.

[6] J. de Sousa, J. P. Marques-Silva, and M. Abramovici, "A configware/software approach to SAT solving," in *FCCM*, 2001.

[7] A. Dandalis and V. K. Prasanna, "Run-time performance optimization of an fpga-based deduction engine for SAT solvers," *ACM Trans. Design Automation of Electronic Systems*, vol. 7, no. 4, pp. 547–562, Oct. 2002.

[8] J. D. Davis, Z. Tan, F. Yu, and L. Zhang, "Designing an efficient hardware implication accelerator for SAT solving," in *SAT*, ser. Lecture Notes in Computer Science, H. K. Büning and X. Zhao, Eds., vol. 4996. Springer, 2008, pp. 48–62.

[9] ——, "A practical reconfigurable hardware accelerator for boolean satisfiability solvers," in *DAC*, L. Fix, Ed. ACM, 2008, pp. 780–785.

[10] I. Skliarova and A. B. Ferrari, "A software/reconfigurable hardware sat solver," *IEEE Trans. VLSI Syst.*, vol. 12, no. 4, pp. 408–419, Apr. 2004.

[11] M. Waghmode, K. Gulati, S. P. Khatri, and W. Shi, "An efficient, scalable hardware engine for Boolean satisfiability," in *ICCD*. IEEE, 2006.

[12] K. Gulati, S. Paul, S. P. Khatri, S. Patil, and A. Jas, "Fpga-based hardware acceleration for boolean satisfiability," *ACM Trans. Design Autom. Electr. Syst.*, vol. 14, no. 2, 2009.

[13] C. Thacker, "DDR2 controller for the BEE3," http://research.microsoft.com/en-us/downloads/12e67e9a-f130-4fd3-9bbd-f9e448cd6775.

[14] R. J. Bayardo and R. Schrag, "Using CSP look-back techniques to solve real-world SAT instances," in *AAAI/IAAI*, 1997, pp. 203–208.

[15] J. P. Marques-Silva and K. A. Sakallah, "GRASP - a new search algorithm for satisfiability," in *ICCAD*, 1996, pp. 220–227.

[16] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: Engineering an efficient sat solver," in *DAC*. ACM, 2001, pp. 530–535.

Boosting Minimal Unsatisfiable Core Extraction

Alexander Nadel
Intel Corporation, P.O. Box 1659, Haifa 31015 Israel
Email: alexander.nadel@intel.com

***Abstract*—A variety of tasks in formal verification require finding small or minimal unsatisfiable cores (subsets) of an unsatisfiable set of constraints. This paper proposes two algorithms for finding a minimal unsatisfiable core or, if a time-out occurs, a small non-minimal unsatisfiable core. Our algorithms can be applied to either standard clause-level unsatisfiable core extraction or high-level unsatisfiable core extraction, that is, an extraction of an unsatisfiable core in terms of "interesting" propositional constraints supplied by the user application. We demonstrate that one of our algorithms outperforms existing algorithms for clause-level minimal unsatisfiable core extraction on large well-known industrial benchmarks. We also show that our algorithms are highly scalable for the problem of high-level minimal unsatisfiable core extraction on huge benchmarks generated by Intel's proof-based abstraction refinement flow. In addition, we provide a comparative analysis of the impact of various algorithms on unsatisfiable core extraction.**

I. Introduction

Given an unsatisfiable formula in Conjunctive Normal Form (CNF), a *(clause-level) unsatisfiable core (UC)* is an unsatisfiable subset of its clauses. A *(clause-level) minimal unsatisfiable core (MUC)* is a clause-level UC that becomes satisfiable when any one of its clauses is removed. The problem for finding a small, a minimal, the smallest minimal, or all the minimal unsatisfiable cores has been addressed frequently in recent years [1]–[19], mainly due to the increasing importance of this problem in formal verification.

While clause-level UC extraction is widely used, the formulation of the problem of extracting a clause-level core implicitly assumes that a "good" core should contain as few clauses as possible, whereas many real-world applications require minimizing the number of high-level propositional *interesting constraints* in the core. A *high-level small/minimal unsatisfiable core* is a small/minimal subset of the interesting constraints, whose conjunction with the other constraints in the system is unsatisfiable.

In [13] an algorithm for finding all the high-level MUCs is proposed and applied during the refinement stage of the datapath abstraction refinement-based approach to formal equivalence verification (FEV) described in [20]. Specifically, an abstract counterexample is written as a set of interesting constraints. The abstract counterexample is encoded into CNF in order to find corresponding concrete bit-level counterexamples. If the CNF instance is unsatisfiable, then no such concretization exists, and the abstract counterexample is spurious. In this case, high-level MUCs are used to locate the source of infeasibility and refine the abstraction. The algorithm of [13] is reviewed in Section II.

High-level MUC extraction is used for compositional FEV [21], [22] in [23]. In compositional FEV, the design and the implementation are decomposed into pairs of corresponding slices. By proving the equivalence of all the pairs one can infer the equivalence of the models. It is essential for fast and correct FEV to allow the user (the designer) to specify assumptions that mimic the environment for each pair of slices. These assumptions can be used for the proof of equivalence, but the correctness of each assumption that impacts the proof must be proved separately afterwards. High-level MUC extraction, where the assumptions serve as the interesting constraints, is used to identify the assumptions that were relevant for the equivalence proof. The algorithm of finding a high-level MUC is only briefly sketched in [23] (in fact, a preliminary version of our Alg. 2 is used).

Another example where high-level UC extraction can be applicable is proof-based abstraction refinement for SAT-based hardware model checking, proposed independently in [24] and [25]. This algorithm uses bounded model checking (BMC) for increasing depths on the concrete design. When there is no counterexample up to a given depth, an UC is identified for this depth and an abstraction based on latches and/or gates is used to generate an abstract model which is then proved using complete model checking techniques. While the existing literature uses clause-level UC extraction for finding the abstraction, it would be more appropriate to use high-level UC extraction for this purpose, since the algorithm clearly needs UCs in terms of latches and/or gates, rather than clauses.

Finding one non-minimal core is the cheapest alternative in terms of run-time, but the least precise in terms of the size and accuracy of the core. Extracting all the minimal cores is the most precise, albeit the most costly, option. Finding one minimal core is a reasonable compromise between accuracy and run-time. In this paper we introduce two new algorithms applicable for both high-level and clause-level single MUC extraction. They can also return a small non-minimal core if a time-out occurs after the initial approximation stage, where the larger the time-out the smaller the core will be. One of the algorithms generalizes and improves the resolution-based approach to clause-level MUC extraction [6]–[8], while the other uses the selector variable-based approach to clause-level non-minimal UC extraction of [4], [11] as the starting point. We show that one of our algorithms, given large industrial benchmarks, yields empirically better results than previous approaches to clause-level MUC extraction. We demonstrate the scalability of our algorithms for high-level MUC core extraction using huge benchmarks generated by Intel's imple-

mentation of the proof-based abstraction refinement flow [24], [25]. Also, our work provides an extensive comparison between our new resolution-based and selector variable-based approaches to MUC extraction. Furthermore, we analyze the impact of the following on resolution-based MUC extraction: (1) different approaches to incremental SAT solving (pervasive clause reuse [26] versus reusing a single SAT instance [27]); (2) RRP (Resolution Refutation-based Pruning) [6]–[8]; (3) in-memory data structures with reference counters [9]–[11] versus on-disk data structures [1], [2].

The rest of the paper is organized as follows. Section II provides the necessary background and surveys the related work. Sections III and IV introduce our approaches (resolution-based and selector variable-based, respectively) to extracting a MUC. Section V presents and analyzes the experimental results. Section VI concludes our work.

II. Background and Related Work

We start this section with an overview of algorithms for incremental SAT solving, whose relevance to UC extraction will be explained shortly.

A. Incremental SAT Solving

Incremental SAT solving is intended to boost the solving of closely related SAT instances, which share clauses. It was noted in [26] that *pervasive clause reuse* (that is the reuse of learned clauses derived from shared input clauses in consecutive SAT invocations) provides a significant performance boost in SAT-based Automatic Test Pattern Generation. Another *single SAT instance-based* approach to incremental SAT solving was proposed in [27] in the context of incremental model checking and implemented in the Minisat SAT solver [28]. Minisat re-uses a single SAT instance for all the related invocations. After the solving is completed, one can add new clauses to Minisat and re-invoke the solver on the incremented instance. The single SAT instance-based approach is preferable to the pervasive clause reuse approach, since it reuses not only the relevant conflict clauses, but also all the information necessary for the decision and conflict clause deletion heuristics. However, it suffers from the drawback that it is not *decremental*, that is, it does not allow removing clauses between consecutive SAT invocations. *SAT solving under assumptions* [27] (also implemented in Minisat) provides a solution to this problem by allowing the user to supply a set of *assumptions* $Y = \{y_1, y_2, \ldots, y_m\}$ (where each assumption y_i is a literal) along with the input formula F. The solver returns "satisfiable" iff $F \wedge Y$ is satisfiable. The user application can augment related clauses that are expected to be removed with the negation of a literal l and assert these clauses when required by adding l to Y. An additional useful feature is that when $F \wedge Y$ is unsatisfiable, Minisat can return a small subset of the assumptions $Y' \subseteq Y$, called the *relevant assumptions*, such that $F \wedge Y'$ is still unsatisfiable [28]. The algorithm for returning the set of relevant assumptions is very cheap and requires only minimal changes to the solver. All the Y literals are picked as decision variables before all the other variables and are assigned true. Then standard SAT solving is used. The algorithm terminates when one of the assumptions y is forced to be false in clause C by Boolean Constraint Propagation (BCP). In this case the assumptions cannot hold together. Minisat resolves the C with all its predecessors in the implication graph until a clause containing the negations of Y's literals only is generated. The negation of this clause is returned as the set (conjunction) of relevant assumptions.

B. Unsatisfiable Core Extraction

The most scalable approach to extracting a small clause-level UC is the *resolution-based approach*. It is based on the ability of modern SAT solvers to store a resolution derivation during the process of solving and to generate a resolution refutation of a given unsatisfiable formula at the end. The basic resolution-based approach, discovered independently in [1] and [2], returns all the initial clauses connected to the empty clause $\square$ as the UC. This approach imposes little overhead on the SAT solver, hence it can handle huge instances having millions of clauses. Two methods for trimming the size of the core were proposed in [1] and [5], based on invoking the basic resolution-based approach until a fixed point is reached and manipulating the resolution refutation, respectively. Neither of these methods guarantees minimality.

A resolution-based algorithm for extracting a minimal UC, called Complete Resolution Refutation (CRR), was proposed in [6]–[8]. CRR first finds a resolution refutation π of the input formula and removes clauses that are not connected to the empty clause $\square$. Then, for each remaining input clause C, CRR removes the cone of C from π and invokes a SAT solver on the rest of the remaining clauses, including the conflict clauses. If the formula is satisfiable, then C belongs to a MUC; otherwise CRR removes all the clauses not connected to $\square$ from π and continues the loop until all the input clauses are either removed or are proved to belong to the MUC. CRR uses the pervasive clause reuse approach to incremental SAT solving: it invokes the SAT solver many times on related instances, re-using all the relevant conflict clauses. CRR's performance can be enhanced by applying a technique known as Resolution Refutation-based Pruning (RRP) [6]–[8], which is briefly described in Section III. CRR with RRP scales well for difficult industrial instances having up to one or two hundred clauses [6]–[8].

The early implementations of resolution-based algorithms for UC extraction stored the resolution derivation on disk [1], [2]. Several independent researches realized that the performance of UC extraction could be improved by storing the resolution derivation in memory. In [6] it was suggested as a direction for future work that storing the resolution derivation in memory could boost CRR. BooleForce [29] was the first solver to store the resolution derivation in memory (for extracting non-minimal UCs). An efficient implementation of the in-memory algorithm, based on reference counters, was proposed independently in [9]–[11]. The key observation is that if there are no references to the clause from either the

instance or the resolution derivation, it can safely be removed from the resolution derivation.

Now we describe another prominent approach to UC extraction–the *selector variable-based approach*–introduced in the AMUSE tool for non-minimal clause-level UC extraction [4]. This approach adds the negation of a fresh *selector variable* from a subset Y to each input clause. The SAT solver is then guided to assert the clauses by setting the selector variable to true whenever possible. In the end, the algorithm derives a Y-conflict clause containing a subset of the selector variables. The core consists of clauses the negation of whose corresponding selector variables belongs to the Y-conflict clause. AMUSE implementation requires changing the internals of the SAT solver. A very similar algorithm for non-minimal clause-level UC extraction which does not require changing the SAT solver was proposed in [11]. It provides the selector variables as assumptions, along with the formula augmented by selector variables, to Minisat. The UC consists of clauses whose selector variables are returned by Minisat as the relevant assumptions. Unlike the resolution-based approach, the selector variable-based approach does not need to store a resolution derivation. However, its major drawback is that adding selector variables causes the SAT solver to generate very long learned clauses, making it so that the algorithm does not scale well even to medium-size instances for clause-level MUC extraction. The selector variable-based approach to non-minimal clause-level UC extraction was shown to be inferior to the basic resolution-based approach in [11]. Moreover, AMUSE was shown to be much slower than the CRR algorithm in [6]–[8], even though AMUSE, unlike CRR, does not guarantee the minimality of the core. The selector variable-based approach can be extended for generating a number of clause-based UCs [4], the smallest clause-based MUC [12], and all the clause-based MUCs [13].

An algorithm for generating all the clause-level or all the high-level MUCs, called CAMUS, is proposed in [13]. First CAMUS computes the set of all the minimal correction subsets (MCSs) of a given unsatisfiable problem, where a correction subset is a subset of the constraints whose removal results in a satisfiable set of constraints. Then it finds the set of all the irreducible hitting sets of the MCSs, which is exactly the set of all the MUCs. The first stage of this algorithm is very costly, since it has to find *all* the MCSs. Yet, a version of CAMUS for finding all the high-level MUCs was successfully applied to formulas from the datapath abstraction domain [20] having more than one hundred clauses. The efficiency of the high-level MUCs extraction is achieved using Minisat's feature of SAT solving under assumptions with relevant assumption extraction as an underlying reasoning engine. In our context, it is important to note that the high-level MUCs extraction mode of CAMUS marks all the clauses that correspond to a particular interesting constraint with a particular selector variable. This operation allows CAMUS to use Minisat's features for reasoning about interesting constraints. Our Alg. 3 for finding a single MUC uses this operation as well, but, unlike CAMUS, we apply it to the problem of UC extraction

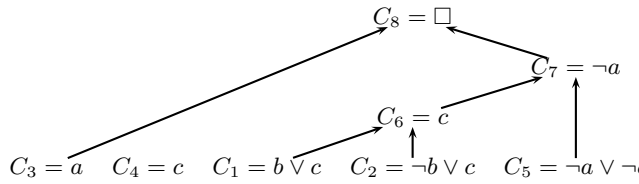


Fig. 1: An example. Assume $\Psi = \{R_1 = \{C_1, C_2\}, R_2 = \{C_3, C_4\}\}$; $\Omega = \{C_5\}$. Note that the only high-level MUC is $\{R_2\}$. A resolution refutation of $\mathit{Clss}(\Psi \wedge \Omega)$, addressed in the text, is shown.

.

in a straightforward manner which makes our high-level MUC extraction algorithm scalable to instances having millions of clauses. Note that although the second stage of CAMUS can easily be modified to return only one high-level MUC, this option does not seem to be practical, since CAMUS's first stage is clear overkill when only one MUC is required to be found.

III. Resolution-based Minimal Unsatisfiable Core Extraction

In this section we introduce a new resolution-based algorithm for high-level and clause-level minimal UC extraction. It may also return a non-minimal core if a time-out occurs after the initial approximation stage.

A. Definitions

We need to provide a number of well-known notions related to resolution. The *resolution rule* states that given clauses $D_1 = A \vee v$ and $D_2 = B \vee \neg v$, where A and B are also clauses, we can derive the clause $C = A \vee B$. The resolution rule application is denoted by $C = D_1 \otimes^v D_2$. A *resolution derivation* of a *target clause* C from a CNF formula F is a sequence $\pi = (C_1, C_2, \ldots, C_p \equiv C)$, where each clause C_i is either a clause of F (an initial clause) or derived by applying the resolution rule to C_j and C_k, where $j, k < i$ (a derived clause). A *resolution refutation* is a resolution derivation of the empty clause $\square$. A resolution derivation $\pi = (C_1, C_2, \ldots, C_p)$ can naturally be considered as a directed acyclic graph (dag) whose vertices correspond to all the clauses of π and in which there is an edge from a clause C_j to a clause C_i iff $C_i = C_j \otimes C_k$ (an example of such a dag appears in Fig. 1). Let π be a resolution derivation. A clause $D \in \pi$ is *reachable* from $C \in \pi$ if there is a path (of 0 or more edges) from C to D. The set of all vertices reachable from $C \in \pi$ (or from $\rho \subseteq \pi$), called the *cone* of C (or ρ), is denoted $Re(\pi, C)$ (or $Re(\pi, \rho)$). For the example in Fig. 1, $Re(\pi, \rho = \{C_1, C_2\}) = \{C_1, C_2, C_6, C_7, C_8\}$.

Now we provide definitions related to high-level UC extraction. Given a conjunction (set) of propositional formulas $\Psi = \{R_1, R_2, \ldots, R_m\}$ and a propositional formula Ω, such that $\Psi \wedge \Omega$ is unsatisfiable, $\mathit{UC}(\Psi, \Omega) \subseteq \Psi$ is a *high-level unsatisfiable core*, if $\mathit{UC}(\Psi, \Omega) \wedge \Omega$ is unsatisfiable. Each $R_i \in \Psi$ is an *interesting constraint (IC)* and the set Ω is the *remainder*. A high-level UC is *minimal* if removing any of its ICs makes its conjunction with the remainder satisfiable.

A *clause projection* $Clss(F)$ of a propositional formula F is a set of clauses equisatisfiable to F, generated by applying Tseitin encoding [30]. We sometimes refer to a formula F, meaning the associated clause projection $Clss(F)$.

Next we introduce our resolution-based algorithm for high-level MUC extraction. For clarity of presentation we start with a simple (yet novel) Alg. 1, which serves as the basis for the eventual Alg. 2.

B. The Basic Algorithm

Alg. 1 receives a set of ICs and the remainder. Its initial *approximation stage* (the first two lines) approximates a high-level MUC *muc* by placing in *muc* ICs whose intersection with a clause-level non-minimal UC is non-empty[1]. The clause-level non-minimal core is found using the basic resolution-based approach [1], [2]. The approximation stage of Alg. 1 corresponds to the "folk" algorithm for finding a high-level non-minimal UC. Note that even if the clause-level core is minimal, the high-level core is not necessarily minimal (this observation also holds for Alg. 2). Consider the example in Fig. 1. The set of clauses $\{C_1, C_2, C_3, C_5\}$ is a clause-level MUC of $Clss(\Psi \wedge \Omega)$. However, the corresponding set of interesting high-level constraints $\{R_1, R_2\}$ is not a high-level MUC.

Assume now that the algorithm enters the *minimization loop* (the "for all" loop). It simply goes over all the ICs remaining in *muc* and checks if a particular *removal candidate* R_i can be removed by invoking a SAT solver over the clause projection of the remainder and $muc \setminus \{R_i\}$. In the end, *muc* is a high-level MUC.

Note that if a time-out occurs during the minimization stage, the algorithm can still return a reduced, but not necessarily minimal, core. This property also holds for Alg. 2. We describe another property holding for both algorithms. This property is essential for guaranteeing that the algorithms indeed return a minimal core. Consider an IC R_j, such that $j \in muc$, but R_j is not the removal candidate for a certain minimization loop iteration. Note that all the clauses $Clss(R_j)$ are sent to the SAT solver, even if some of the clauses of $Clss(R_j)$ did not participate in the clause-level core returned by the SAT solver during the approximation stage. For example, suppose that the resolution refutation in Fig. 1 corresponds to the situation just after completion of the approximation stage. Assume that the removal candidate for the first iteration of the minimization loop is R_1. The clause $C_4 \in Clss(R_2)$ is not connected to $\square$. However, it must be sent to the SAT solver, otherwise the algorithm will erroneously conclude that R_1 must belong to the minimal core. Likewise, all the clauses in the clause projection of the remainder are sent to the SAT solver.

The main drawback of Alg. 1 is the lack of incrementality. The SAT solver is invoked each time on a new formula, while the learned conflict clauses and heuristical information are lost.

[1]We assume here and elsewhere in the paper that the clause projection of each constraint (either an interesting constraint or the remainder) is created by appying Tseitin encoding which generates *new* auxiliary variables for each translated entity.

Algorithm 1 Basic high-level MUC extraction

Require: $\Psi = \{R_1, R_2, \ldots, R_m\} \wedge \Omega$ is unsatisfiable
1: Extract a clause-level non-minimal unsatisfiable core $\overline{F}$ using the basic resolution-based approach
2: $muc := \{i \mid Clss(R_i) \cap \overline{F} \neq \emptyset\}$
3: **for all** $i \in muc$ **do**
4: Invoke a SAT solver on $Clss(\Omega \wedge \{R_j \mid j \in muc \setminus \{i\}\})$
5: **if** the result is "unsatisfiable" **then**
6: $muc := muc \setminus \{i\}$
7: **return** $\{R_i \mid i \in muc\}$

We would like to extend Alg. 1 so that it would reuse the same SAT instance. To be able to check whether a removal candidate belongs to the core, we need to have the ability to *conditionally remove* the cone of the removal candidate (that is, to remove the cone while maintaining the possibility of returning it efficiently), since this cone corresponds exactly to the removal candidate and all its logical consequences. In addition, we need to support both the efficient *return* of conditionally removed clauses to the SAT instance for cases where the removal candidate belongs to the minimal core, and the efficient *unconditional removal* of clauses to support the operation of removing ICs from the core. We will describe how we implemented these operations after presenting the flow of Alg. 2.

C. The Final Algorithm

Alg. 2 uses an incremental SAT solver (which also maintains a resolution derivation) and assumes that it returns a triplet that contains the result (which can either be "satisfiable" or "unsatisfiable"), an updated SAT instance, and an updated resolution derivation. The approximation stage of Alg. 2 (from the beginning until line 7) invokes the SAT solver over the set of ICs and the remainder. The cones (of ICs) that do not include $\square$ are removed from the instance forever. The algorithm maintains a set of *minimal core candidates*, *muc_cands*, initialized with the indexes of ICs whose cone includes $\square$. It is not known whether the ICs in *muc_cands* belong to the minimal core. The algorithm also maintains a set of *minimal core habitants*, *muc*, which contains ICs that belong to the minimal UC. Consider the minimization loop (the "while" loop). Each iteration picks a removal candidate from the minimal core candidates. It conditionally removes the cone of the removal candidate from the SAT instance and invokes the SAT solver. If the instance is satisfiable, the removal candidate is guaranteed to belong to the minimal core, and hence it is moved from *muc_cands* to *muc* and its cone is returned to the instance. If the instance is unsatisfiable, the algorithm refines the minimal core candidates by keeping there only those ICs whose cone includes $\square$. Cones of other ICs are removed forever. Hence one iteration of the loop may remove more than one IC from the set of minimal core candidates. In the end, the algorithm returns the set of minimal core habitants as the high-level MUC.

Now we will discuss implementation details which are critical for performance. Conditionally removed clauses are

not deleted from the clause database, since this would make returning them cumbersome and costly. Rather, we make sure that these clauses are ignored by the solver's major algorithms, including Boolean Constraint Propagation (BCP) and clause-based heuristics (if such a heuristic, e.g., CBH [31], is used). This is done as follows. We remove the clauses from the WL data structure [32], which is used for BCP, then we mark the clauses and guide the heuristic to ignore the marked clauses. To return the conditionally removed clauses, it is sufficient to reinsert them into the WL data structure and unmark them for the clause-based heuristics. In addition, our implementation groups the following two operations into one pass over the clauses carried out just after executing line 10: (1) finding and conditionally removing the cone of the current removal candidate; (2) finding and either returning or unconditionally removing the cone of the previous removal candidate (or, for the first iteration only, unconditionally removing the cones of ICs found to be irrelevant during the approximation stage).

Algorithm 2 Resolution-based high-level MUC extraction

Require: $\Psi = \{R_1, R_2, \ldots, R_m\} \wedge \Omega$ is unsatisfiable
1: Initialize the SAT instance *SI* with *Clss*$(\Psi \wedge \Omega)$ and associate a resolution derivation π with *SI*
2: $\langle$*res*, *SI*, $\pi\rangle$:= SAT(*SI*)
3: **for** $i \in 1 \ldots m$ **do**
4: **if** $\square \notin$ *Re*$(\pi,$ *Clss*$(R_i))$ **then**
5: Remove *Re*$(\pi,$ *Clss*$(R_i))$ from *SI* forever
6: *muc_cands* := $\{i \mid \square \in$ *Re*$(\pi,$ *Clss*$(R_i))\}$
7: *muc* := $\{\}$
8: **while** *muc_cands* is non-empty **do**
9: k := a member of *muc_cands* $\setminus$ *muc*
10: Conditionally remove *Re*$(\pi,$ *Clss*$(R_k))$ from *SI*
11: *muc_cands* := *muc_cands* $\setminus \{k\}$
12: $\langle$*res*, *SI*, $\pi\rangle$:= SAT(*SI*)
13: **if** *res* = satisfiable **then**
14: Return *Re*$(\pi,$ *Clss*$(R_k))$ to *SI*
15: *muc* := *muc* $\cup \{k\}$
16: **else**
17: Remove *Re*$(\pi,$ *Clss*$(R_k))$ from *SI* forever
18: **for** $i \in$ *muc_cands* **do**
19: **if** $\square \notin$ *Re*$(\pi,$ *Clss*$(R_i))$ **then**
20: Remove *Re*$(\pi,$ *Clss*$(R_i))$ from *SI* forever
21: *muc_cands* := *muc_cands* $\setminus \{R_i\}$
22: **return** $\{R_i \mid i \in$ *muc*$\}$

Standard clause-level MUC extraction is a particular case of high-level MUC extraction where each IC consists of a single clause and the remainder is empty. Consider Alg. 2 as an algorithm for clause-level MUC extraction and compare it to the CRR algorithm [6]–[8] described in Section II. The algorithms have a similar structure. Both try to reuse all the relevant conflict clauses between different iterations of the minimization loop. The main difference between them is that while CRR creates a new SAT instance for each minimization loop iteration, Alg. 2 reuses a single SAT instance. There is an additional difference between the implementation of CRR and the currently fastest implementation of Alg. 2. Alg. 2's fastest implementation uses the latest in-memory data structures with reference counters for storing the resolution derivation [9]–[11], while the CRR implementation of [6]–[8] uses the on-disk approach. Section V demonstrates that Alg. 2 empirically outperforms CRR for clause-level MUC extraction.

Our current implementation of resolution-based algorithms uses reference counters for efficiently removing unreferenced nodes in the in-memory resolution derivation. However, we noticed that using reference counters for this purpose is redundant, since the same effect can be achieved by removing unreferenced nodes during the standard interprocessing required for the clause deletion heuristic as follows. The solver stores the list L of all the clauses deleted by the clause deletion heuristic. Note that only clauses that appear in L should be considered for removal from the resolution derivation. When L becomes larger than some threshold, the algorithm removes from the resolution derivation all the clauses in L whose predecessors in the resolution derivation also appear in L. The exact implementation details are solver-specific. We have been working on implementing this idea and experimenting with it in the hope that it will result in further memory footprint reduction.

RRP [6]–[8], used to enhance CRR, is directly applicable to Alg. 2. The underlying idea is that a model for *SI* during any minimization loop iteration can only be found under such a partial assignment that falsifies every clause in some path in *Re*$(\pi,$ *Clss*$(R_k))$ from a clause in *Clss*(R_k) to $\square$. The claim is correct, since finding a model for *SI* that satisfies every path in *Re*$(\pi,$ *Clss*$(R_k))$ would mean that there is a satisfiable vertex cut in π, contradicting the assumption that π is a resolution refutation. For example, consider again Fig. 1 and suppose that R_1 is picked as the first removal candidate by the minimization loop. A model for *SI* can be found either when $b, c = 0; a = 1$ for the path C_1, C_6, C_7, C_8 or $c = 0; b, a = 1$ for the path C_2, C_6, C_7, C_8. RRP takes advantage of the described property during the minimization loop by guiding the decision heuristic and the backtracking engine of the SAT solver to falsify paths in *Re*$(\pi,$ *Clss*$(R_k))$ in a systematic manner. We analyze the impact of RRP on Alg. 2's performance in Section V.

IV. Selector Variable-based Minimal Unsatisfiable Core Extraction

This section proposes a new selector variable-based algorithm for extracting a single high-level or clause-level MUC or, if a time-out occurs after the initial approximation stage, a non-minimal core. Our algorithm takes advantage of the ability of modern SAT solvers to solve the problem under assumptions and to return a set of relevant assumptions for proving the unsatisfiability [27], [28], explained in Section II.

Consider Alg. 3. It is composed of the approximation stage and the minimization loop, exactly like Algs. 1 and 2. First, the algorithm allocates a fresh selector variable s_i for each IC R_i and augments each clause in the clause projection of R_i with $\neg s_i$. At every stage of the algorithm every conflict clause in the cone of R_i will contain the literal $\neg s_i$. Hence, assigning some s_i to true means asserting the IC R_j, while assigning s_j to false means removing the associated IC by satisfying all the relevant clauses. Our algorithm takes advantage of

these properties to support the conditional and unconditional removal of the ICs from the instance, as well as their return to it, without explicitly maintaining a resolution derivation.

The approximation stage of the algorithm launches a SAT solver, providing it the input formula (updated with the selector variables) and the set of selector variables as the assumptions. Suppose that the solver returns: (1) the satisfiability status; (2) the updated incremental CNF instance *SI*; and (3) the set of relevant assumptions *rel_asm*. After invoking the solver, the set of ICs which corresponds to the selector variables in *rel_asm* is a (not necessarily minimal) high-level UC. The cones of the other ICs are removed from the instance. This is done by adding unit clauses to the instance, each of which contains the negation of a selector variable corresponding to one particular IC. One could physically remove the clauses, but this would require the expensive operation of going over the clauses explicitly and rebuilding the clause database. In our implementation, the SAT solver identifies and removes the satisfied clauses, at no additional cost, as part of the standard interprocessing required for the clause deletion heuristic.

The minimization loop of our algorithm maintains the sets of minimal core candidates and minimal core habitants like Alg. 2. Each iteration of the minimization loop removes a particular removal candidate from the set of minimal core candidates and launches the SAT solver on the formula, supplying it a set of assumptions that does not include the removal candidate. If the result is satisfiable, the removal candidate belongs to the MUC, otherwise it does not belong to it. In the latter case, based on the set of relevant assumptions returned by the SAT solver, the algorithm refines the set of minimal core candidates and removes the cones of unnecessary ICs. At the end, the set of minimal core habitants is returned as the high-level MUC.

Compare selector variable-based Alg. 3 to resolution-based Alg. 2. The selector variable-based approach saves the overhead of maintaining a resolution derivation and making additional passes over the clause database. Alg. 3 is also much simpler to implement. However, the selector variable-based approach has the drawback that adding new variables to the formula makes the conflict clauses larger and the solver slower. An empirical comparison of Alg. 3 and Alg. 2 is provided in Section V.

Algorithm 3 Selector variable-based high-level MUC extraction

Require: $\Psi = \{R_1, R_2, \ldots, R_m\} \wedge \Omega$ is unsatisfiable
1: For each $R_i \in \Psi$: $Sel(R_i) := \{\neg s_i \vee C | C \in Clss(R_i)\}$, where s_i is a new variable
2: $SI := (\wedge_{i:=1\ldots m} Sel(R_i)) \wedge Clss(\Omega)$
3: $\langle res, rel\_asm, SI\rangle := SATAsm(SI; \{s_1, s_2, \ldots s_m\})$
4: For each $j \notin rel\_asm$: Add a unit clause $\neg s_j$ to *SI*
5: $muc\_cands := \{i \mid s_i \in rel\_asm\}$
6: $muc := \{\}$
7: **while** *muc_cands* is non-empty **do**
8: $k :=$ a member of $muc\_cands \setminus muc$
9: $muc\_cands := muc\_cands \setminus \{k\}$
10: $\langle res, rel\_asm, SI\rangle := SATAsm(SI; \{s_i \mid i \in muc\_cands \cup muc\})$
11: **if** *res* = satisfiable **then**
12: $muc := muc \cup \{k\}$
13: **else**
14: $muc\_cands := \{i \mid s_i \in rel\_asm\}$
15: For each $j \notin rel\_asm$: Add a unit clause $\neg s_j$ to *SI*
16: **return** $\{R_i \mid i \in muc\}$

V. Experimental Results

In this section we empirically compare algorithms for clause-level MUC extraction and high-level MUC extraction.

A. Clause-Level Minimal Unsatisfiable Core Extraction

We used the same benchmarks that were used in [6]–[8]. The instances were taken from well-known unsatisfiable families from bounded model checking (*barrel, longmult*) [33] and microprocessor verification (*fvp-unsat.2.0, pipe_unsat_1.0*) [34]. The size of the instances ranged from 6,069 to 189,109 clauses, the average size being 49,986 clauses. Detailed information regarding these instances appears in [6]–[8]. All the algorithms were implemented in the Eureka SAT solver [35]. All experiments were carried out on a machine with 4Gb memory and two Intel Xeon CPU 3.06 processors.

Note that CRR with RRP is the best existing algorithm for extracting a clause-level MUC, given large difficult formal verification benchmarks. It was shown in [6]–[8] that CRR with RRP convincingly outperforms AMUSE [4] and MUP [36]. There exist a number of other approaches to UC extraction, such as those based on adaptive core search [3], Brouwer's fixed-point approximation algorithm [14], local search [15], a combination of local search and complete search [16], [19], a branch and bound algorithm [17], and genetic algorithms [18]. However, none of these approaches has been shown to scale well to large-size or even medium-size benchmarks. The instances considered in the experimental results sections of the papers mentioned above rarely exceed 10,000 clauses; in most cases the instances considered have at most a few thousand or even a few hundred clauses. This is not surprising, since these algorithms do not utilize the power of DPLL-based SAT solvers, currently the only approach that can solve large and difficult CNFs. Note that the problem of finding a MUC is DP-complete [37], hence in general extracting a MUC is at least as difficult as SAT solving; moreover, unless NP=co-NP, it is more difficult.

Consider Table I. Columns 1MR, 1MN, and 1DN (the names are explained in the caption of Table I) correspond to various implementations of Alg. 2. Columns PDR and PDN correspond to the CRR+RRP and CRR algorithms. Column PMN can be thought of as an in-memory implementation of CRR or as a modification of Alg. 2 that creates multiple SAT instances with pervasive clause reuse between them. Column SV corresponds to the selector variable-based approach of Alg. 3. Compare the best implementation of our Alg. 2 (1MN) to the best previous approach CRR with RRP (PDR). 1MN is clearly preferable, as it is faster overall and faster for thirteen out of sixteen instances. In addition, it manages to find the

TABLE I: Comparing algorithms for clause-level MUC extraction. The first column contains instance names (where, p/b/l stand for pipe/barrel/longmult). Each cell in the next seven columns contains the execution time in seconds on the top, the core size on the bottom-left, and the number of clauses whose status was not determined within the time-out of 2 hours on the bottom-right (the core is minimal iff 0 appears on the bottom-right of the corresponding cell). The first letter of the abbreviated algorithm names corresponds to the approach to incremental SAT solving: "P"/"1" stand for pervasive clause reuse/single SAT instance-based, respectively. The second letter corresponds to the data structures: "D"/"M" stand for on-disk/in-memory with reference counters, respectively. The third letter corresponds to RRP invocation: "R"/"N" stand for turning RRP on/turning RRP off, respectively. Bold times are the best times.

	Resolution-based												SV	
Inst	**1MR**		**1MN**		**PMN**		**1DN**		**PDR**		**PDN**			
4p	7200		**1417**		2326		7200		3791		4055		2021	
	25399	25341	18164	0	17928	0	18622	4050	18897	0	18609	0	17472	0
4p_1_ooo	7200		**1528**		2593		7200		2928		4579		4323	
	20445	20443	12213	0	12211	0	12444	8417	12246	0	12226	0	12887	0
4p_2_ooo	4718		**2383**		3428		7200		4566		7062		4999	
	14456	0	14438	0	14572	0	16360	11047	14553	0	14569	0	14560	0
4p_3_ooo	5053		**2560**		3694		7200		4465		6285		5357	
	15844	0	15850	0	15811	0	16141	7168	15892	0	15899	0	16177	0
4p_4_ooo	4768		**2432**		4706		7200		5865		7200		6354	
	17625	0	17558	0	17633	0	19215	13680	17872	0	17916	370	17793	0
3p_k	343		**167**		310		810		469		540		239	
	6784	0	6784	0	6787	0	6786	0	6783	0	6783	0	7074	0
4p_k	7200		**1426**		2243		7200		2938		3261		3097	
	21459	21459	17045	0	17039	0	17218	3403	17055	0	17075	0	18786	0
5p_k	7200		7200		7200		7200		7200		7200		7200	
	45406	45404	36423	8946	37479	22827	37523	36363	39336	19888	38800	28221	49134	47179
b5	286		68		71		869		115		128		**48**	
	2653	0	2653	0	2653	0	2653	0	2653	0	2653	0	2653	0
b6	1514		**348**		433		7200		436		552		402	
	4437	0	4437	0	4437	0	4498	659	4437	0	4437	0	4437	0
b7	1802		849		800		7200		1081		1108		**700**	
	6877	0	6877	0	6877	0	7324	2927	6877	0	6877	0	6877	0
b8	7200		4115		4479		7200		**4110**		4923		5758	
	10260	1390	10077	0	10076	0	12452	11382	10075	0	10075	0	10076	0
l4	23		14		14		25		**12**		**12**		78	
	972	0	972	0	972	0	972	0	972	0	972	0	972	0
l5	191		143		130		321		100		**97**		642	
	1518	0	1518	0	1518	0	1518	0	1518	0	1518	0	1520	0
l6	1121		**968**		1072		3129		1760		1615		5705	
	2189	0	2189	0	2190	0	2189	0	2189	0	2189	0	2194	0
l7	7200		**5099**		7200		7200		7200		7200		7200	
	2982	36	2982	0	2994	649	3203	1814	3454	1993	3071	973	3494	2895
Total	*63019*		***30717***		*40699*		*84354*		*47036*		*55817*		*54123*	
	199306	*114073*	*170180*	*8946*	*171177*	*23476*	*179118*	*100910*	*174809*	*21881*	*173669*	*29564*	*186106*	*50074*

minimal UC within the time-out for one more instance.

Comparing 1MN and SV clearly shows that our resolution-based approach is preferable to our selector variable-based approach for clause-level MUC extraction. The overhead of adding a variable per clause is too high.

The single SAT instance-based approach to incrementality results in better performance for the in-memory version of the resolution-based approach to UC extraction (compare 1MN to PMN). This is not surprising, as it takes advantage of the information gathered by the decision variable and clause deletion heuristics. However, it deteriorates the performance of the on-disk algorithm (compare 1DN to PDN). The problematic aspect of such a combination is that the single SAT instance-based approach stores the entire resolution derivation in a single file whose growing size does not allow it to be processed efficiently.

While RRP is helpful for CRR (compare PDR and PDN), it deteriorates the performance of Alg. 2 (compare 1MR and 1MN). We can also report that PMN outperforms PMR (PMR's results are not reported in the table due to space limitations). Hence RRP does not work well when the resolution derivation is stored in-memory. The reasons for this could be related to higher memory consumption, since RRP requires more memory due to additional bookkeeping. We plan to investigate and optimize the performance of RRP for Alg. 2 in the future.

B. High-Level Minimal Unsatisfiable Core Extraction

Compare the algorithms for high-level MUC extraction. As far as we know, this paper is the first to address the problem of extracting a single minimal (or small, if a time-out occurs) high-level UC. A much more expensive algorithm, called CAMUS, for the much more difficult problem of extracting all the high-level MUCs, is proposed in [13]. We provided a description of CAMUS in Section II-B. We used 49 instances generated from the abstraction stage of Intel's implementation of the proof-based abstraction refinement flow for model checking [24], [25]. All the instances are available from the author. The abstraction is in terms of latches. Each instance consists of two files: the standard CNF file in DIMACS format, and a file that maps latches to their clause projection. The goal was to minimize the number of latches in the core in order to create a more accurate abstraction. Consider Table II. Note that our instances have on average more than 1,850,000 clauses, while the largest benchmark has more than 5,000,000 clauses. Such instances are beyond the reach of modern algorithms

TABLE II: Statistics for instances used for testing high-level MUC extraction algorithms. The remainder fraction is the fraction of the remainder out of all the clauses.

	Clauses	Remainder Fraction	ICs Num.	Mean IC size
Min.	136878	0.955	584	3.89
Average	1853500	0.968	3367	17.27
Max.	5136873	0.977	4030	48.6

TABLE III: Comparing algorithms for high-level UC extraction.

	1MR	**1MN**	**PMN**	**1DN**	**PDR**	**PDN**	**SV**
UC time	**416**	**416**	**416**	766	766	766	588
UC size	28.9	28.9	28.9	28.9	28.9	28.9	27.7
MUC time	3843	3797	4731	9238	44699	44345	**3278**
MUC size	9.6	9.6	9.6	9.6	9.6	9.6	9.5

for clause-level MUC extraction. Note also that the fraction of the remainder among the clauses is very high, while the number of ICs (latches) is 3367 on average. We compared the same 7 algorithms that were compared for clause-level UC extraction. Consider Table III. The table summarizes the performance of our algorithms for both high-level non-minimal UC extraction (which corresponds to the approximation stage only) and high-level MUC extraction. The overall run-time and the average core sizes are displayed. Column PMN can be considered either as a modification of Alg. 2 that creates a new SAT instance for each minimization loop iteration or as a generalization of CRR for high-level minimal UC extraction.

While the best resolution-based approach 1MN is preferable to the selector variable-based approach for non-minimal UC extraction, the selector variable-based approach is preferable for minimal UC extraction. Hence, while adding selector variables does not pay off for non-minimal UC extraction, even when the number of ICs is relatively low, it turns out to be useful for the minimization loop. This result hints that the overhead of additional passes over the clauses is greater than the overhead of maintaining additional variables for high-level UC extraction (at least for our benchmarks). Also note that: (1) RRP is not helpful for the high-level UC extraction. The reason is, apparently, that $Re(\pi, Clss(R_k))$ is too large to be efficiently explored, since it has too many source clauses. (2) Not surprisingly, the in-memory data structure is clearly preferable to the on-disk one. (3) The single SAT instance-based approach to incrementality pays off even if the on-disk data structure is used (as opposed to the situation in clause-level UC extraction). The apparent reason is that considerably fewer operations of extracting the core using the file are required for high-level UC extraction.

VI. Conclusion and Future Work

We introduced two new algorithms (resolution-based and selector variable-based) for finding a minimal unsatisfiable core (or a small non-minimal core, if a time-out occurs). Our algorithms can be applied to either standard clause-level minimal unsatisfiable core extraction or high-level unsatisfiable core extraction, that is, extraction of a minimal unsatisfiable subset in terms of "interesting" propositional constraints provided by the user application.

We demonstrated that our resolution-based algorithm outperforms existing algorithms for standard clause-level minimal unsatisfiable core extraction on large, difficult industrial benchmarks. We also demonstrated the empirical usefulness and scalability of both our algorithms for high-level minimal unsatisfiable core extraction on huge benchmarks generated by Intel's proof-based abstraction refinement flow.

In addition, we provided a detailed comparison of various algorithms and heuristics for minimal unsatisfiable core extraction. An important conclusion is that while our resolution-based approach is clearly preferable to our selector variable-based approach for standard clause-level minimal unsatisfiable core extraction, the latter approach is faster for high-level minimal unsatisfiable core extraction. We found that the single SAT instance-based approach to incremental SAT solving results in better performance than the pervasive clause reuse approach. Furthermore, the in-memory data structure with reference counters for storing the resolution derivation is preferable to the on-disk data structure. Finally, RRP was not found to be helpful for the newly proposed algorithms.

We plan to investigate how to efficiently integrate RRP into our algorithms in the future. Furthermore, we have been working on improving the in-memory data structures for storing the resolution derivation by removing the redundant usage of reference counters (as described in Section III-C). In addition, we plan to study the impact of our algorithms within various applications, such as proof-based abstraction refinement and compositional FEV. Finally, we plan to study how to enhance our algorithms to extract more than one MUC.

Acknowledgments

The author would like to thank Amit Palti for supporting this work, Paul Inbar for editing the paper, and Vadim Ryvchin and Iddo Tzameret for providing useful comments.

References

[1] L. Zhang and S. Malik, "Extracting small unsatisfiable cores from unsatisfiable Boolean formula." in *Preliminary Proceedings of Sixth International Conference on Theory and Applications of Satisfiability Testing (SAT'03)*, 2003.

[2] E. Goldberg and Y. Novikov, "Verification of proofs of unsatisfiability for CNF formulas," in *Proceedings of Design, Automation and Test in Europe Conference and Exhibition (DATE'03)*, 2003, pp. 886–891.

[3] R. Bruni, "Approximating minimal unsatisfiable subformulae by means of adaptive core search," *Discrete Applied Mathematics*, vol. 130, no. 2, pp. 85–100, 2003.

[4] Y. Oh, M. N. Mneimneh, Z. S. Andraus, K. A. Sakallah, and I. L. Markov, "AMUSE: a minimally-unsatisfiable subformula extractor," in *Proceedings of the 41th Design Automation Conference (DAC'04)*, 2004, pp. 518–523.

[5] R. Gershman, M. Koifman, and O. Strichman, "Deriving small unsatisfiable cores with dominators," in *CAV*, ser. Lecture Notes in Computer Science, T. Ball and R. B. Jones, Eds., vol. 4144. Springer, 2006, pp. 109–122.

[6] N. Dershowitz, Z. Hanna, and A. Nadel, "A scalable algorithm for minimal unsatisfiable core extraction," *CoRR*, vol. abs/cs/0605085, 2006.

[7] ——, "A scalable algorithm for minimal unsatisfiable core extraction." in *SAT*, ser. Lecture Notes in Computer Science, A. Biere and C. P. Gomes, Eds., vol. 4121. Springer, 2006, pp. 36–41. [Online]. Available: http://dblp.uni-trier.de/db/conf/sat/sat2006.html#DershowitzHN06

[8] A. Nadel, "Understanding and improving a modern SAT solver," Ph.D. dissertation, Tel Aviv University, Tel Aviv, Israel, August 2009.

[9] A. Biere, "PicoSAT essentials," *JSAT*, vol. 4, no. 2-4, pp. 75–97, 2008.

[10] O. Shacham and K. Yorav, "On-the-fly resolve trace minimization," in *DAC*. IEEE, 2007, pp. 594–599.
[11] R. Asín, R. Nieuwenhuis, A. Oliveras, and E. Rodríguez-Carbonell, "Efficient generation of unsatisfiability proofs and cores in SAT," in *LPAR*, ser. Lecture Notes in Computer Science, I. Cervesato, H. Veith, and A. Voronkov, Eds., vol. 5330. Springer, 2008, pp. 16–30.
[12] I. Lynce and J. P. M. Silva, "On computing minimum unsatisfiable cores," in *SAT*, 2004.
[13] M. H. Liffiton and K. A. Sakallah, "Algorithms for computing minimal unsatisfiable subsets of constraints," *J. Autom. Reasoning*, vol. 40, no. 1, pp. 1–33, 2008.
[14] H. van Maaren and S. Wieringa, "Finding guaranteed MUSes fast," in *SAT*, ser. Lecture Notes in Computer Science, H. K. Büning and X. Zhao, Eds., vol. 4996. Springer, 2008, pp. 291–304.
[15] É. Grégoire, B. Mazure, and C. Piette, "Using local search to find MSSes and MUSes," *European Journal of Operational Research*, vol. 199, no. 3, pp. 640–646, 2009.
[16] C. Piette, Y. Hamadi, and L. Sais, "Efficient combination of decision procedures for MUS computation," in *FroCos*, ser. Lecture Notes in Computer Science, S. Ghilardi and R. Sebastiani, Eds., vol. 5749. Springer, 2009, pp. 335–349.
[17] M. H. Liffiton, M. N. Mneimneh, I. Lynce, Z. S. Andraus, J. Marques-Silva, and K. A. Sakallah, "A branch and bound algorithm for extracting smallest minimal unsatisfiable subformulas," *Constraints*, vol. 14, no. 4, pp. 415–442, 2009.
[18] J. Zhang, S. Li, and S. Shen, "Extracting minimum unsatisfiable cores with a greedy genetic algorithm," in *Australian Conference on Artificial Intelligence*, ser. Lecture Notes in Computer Science, A. Sattar and B. H. Kang, Eds., vol. 4304. Springer, 2006, pp. 847–856.
[19] J. Zhang, S. Shen, and S. Li, "Tracking unsatisfiable subformulas from reduced refutation proof," *JSW*, vol. 4, no. 1, pp. 42–49, 2009.
[20] Z. S. Andraus, M. H. Liffiton, and K. A. Sakallah, "Refinement strategies for verification methods based on datapath abstraction," in *ASP-DAC*, F. Hirose, Ed. IEEE, 2006, pp. 19–24.
[21] S.-Y. Huang and K.-T. Cheng, *Formal equivalence checking and design debugging*. Norwell, MA, USA: Kluwer Academic Publishers, 1998.
[22] Z. Khasidashvili, D. Kaiss, and D. Bustan, "A compositional theory for post-reboot observational equivalence checking of hardware," in *FMCAD*. IEEE, 2009, pp. 136–143.
[23] O. Cohen, M. Gordon, M. Lifshits, A. Nadel, and V. Ryvchin, "Designers work less with quality formal equivalence checking," in *Design and Verification Conference (DVCon)*, 2010.
[24] K. L. McMillan and N. Amla, "Automatic abstraction without counterexamples." in *Proceedings of the Ninth International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'03)*, 2003, pp. 2–17.
[25] A. Gupta, M. K. Ganai, Z. Yang, and P. Ashar, "Iterative abstraction using SAT-based BMC with proof analysis," in *ICCAD*. IEEE Computer Society / ACM, 2003, pp. 416–423.
[26] J. P. M. Silva and K. A. Sakallah, "Robust search algorithms for test pattern generation," in *FTCS*, 1997, pp. 152–161.
[27] N. Eén and N. Sörensson, "Temporal induction by incremental SAT solving," *Electr. Notes Theor. Comput. Sci.*, vol. 89, no. 4, 2003.
[28] ——, "The MiniSat page." [Online]. Available: http://www.minisat.se/
[29] A. Biere, "Booleforce," http://fmv.jku.at/booleforce. [Online]. Available: http://fmv.jku.at/booleforce
[30] G. S. Tseitin, "On the complexity of derivations in the propositional calculus," *Studies in Mathematics and Mathematical Logic*, vol. Part II, pp. 115–125, 1968.
[31] N. Dershowitz, Z. Hanna, and A. Nadel, "A clause-based heuristic for SAT solvers." in *SAT*, ser. Lecture Notes in Computer Science, F. Bacchus and T. Walsh, Eds., vol. 3569. Springer, 2005, pp. 46–60. [Online]. Available: http://dblp.uni-trier.de/db/conf/sat/sat2005.html#DershowitzHN05
[32] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: Engineering an efficient SAT solver," in *DAC*. ACM, 2001, pp. 530–535.
[33] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu, "Symbolic model checking without BDDs," in *Proceedings of the Fifth International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'99)*, 1999, pp. 193–207. [Online]. Available: http://www.inf.ethz.ch/personal/biere/papers/BiereCimattiClarkeZhu-TACAS99.ps.gz
[34] M. Velev and R. Bryant, "Effective use of Boolean satisfiability procedures in the formal verification of superscalar and VLIW microprocessors," in *Proceedings of the 38th Design Automation Conference (DAC'01)*, 2001, pp. 226–231. [Online]. Available: http://www.ece.cmu.edu/~mvelev/DAC01.ps.gz
[35] A. Nadel, M. Gordon, A. Palti, and Z. Hanna, "Eureka-2006 SAT solver," http://fmv.jku.at/sat-race-2006/descriptions/4-Eureka.pdf. [Online]. Available: http://fmv.jku.at/sat-race-2006/descriptions/4-Eureka.pdf
[36] J. Huang, "MUP: A minimal unsatisfiability prover," in *Proceedings of the Tenth Asia and South Pacific Design Automation Conference (ASP-DAC'05)*, 2005, pp. 432–437.
[37] C. H. Papadimitriou and M. Yannakakis, "The complexity of facets (and some facets of complexity)," in *Proceedings of the Fourteenth Annual ACM Symposium on the Theory of Computing (STOC'82)*, 1982, pp. 255–260.

Propelling SAT and SAT-based BMC using Careset

Malay K. Ganai
NEC Laboratories America, Princeton, NJ, USA

***Abstract*—We introduce the notion of careset, a subset of variables in a Boolean formula that must be assigned in any satisfying assignment. We propose a restricted branching technique in a CDCL solver (i.e., DPLL-based SAT solver with clause learning) such that every decision path is prefixed with decisions on such a careset. Although finding a non-trivial careset may not be tractable in general, we demonstrate that for a SAT-based bounded model checking (BMC) application we can derive it automatically from the sequential behaviors of programs. Our proposed branching technique significantly reduces the search effort of a CDCL solver, and leads to a performance improvement of 1-2 orders of magnitude over well-known heuristics, and over top-ranked solvers of SAT2009 competition, that do not exploit system-level information. We also discuss the proof complexity of such a restricted CDCL solver.**

I. Introduction

In application domains such as bounded model checking (BMC) of software and hardware [1], the analysis engine has to explore paths of bounded length to validate the reachability property. The problem instances are typically derived from transition relation capturing the sequential behaviors of an underlying system using suitable transformation. These problem instances are typically encoded into Boolean formulas (e.g., CNF DIMACS format). The core of the analysis engine uses a DPLL-based [2] SAT solver to search through a Boolean formula. As paths get longer, the number of possible paths, and hence the search space, increases exponentially.

The state-of-the-art SAT solvers use various techniques to prune the search space faster. Some of the important ones are frequent restarts [3], [4], intelligent branching heuristics [5], [6], and learning conflict-driven resolution clauses [7] and binary clauses [8]. These solvers are also well-engineered using techniques such as two-literal watch scheme [6], efficient preprocessing [9], hybrid representation [10], and many others (e.g. [11], [12]). In spite of these improvements, the "loss" of high-level information during encoding can significantly degrade their performance. By loss, we imply that system-level structure and behavior cannot be inferred from a Boolean formula without knowing the actual transformation steps.

- *Structure of the transition relation*: During logic synthesis (i.e., bit-blasting of the transition relation), there are substantial losses of structural information such as types of arithmetic and logical modules, connectivity among such modules (i.e., their dependencies), and independent (i.e., controlling) variables.
- *System level behavior*: The constraints and sequential behaviors get lost during behavioral-level synthesis (i.e., during modeling of a system).

Previous experimental studies [10], [13]–[18] have shown some success in exploiting structural information in a propositional formula to improve CDCL solvers (i.e., DPLL-based solver using Conflict-Driven Clause Learning). Some of these include: (I) branching restriction on dominating input variables [13]–[15], backdoors variables [19], justification gates [10], [18], fanout gates, and variables in dependency graphs [14], [20]; (II) learning non-trivial circuit clauses corresponding to symmetry [21], special gates such as XOR, XNOR, and ITE gates [16], [17]; and (III) re-coding CNF using circuit observability don't cares (Cir-ODC) [22]. However, these techniques do not exploit system-level information.

A. Overview of our approach

Although it has been proved [23] that CDCL is exponentially stronger (i.e., the search tree can be exponentially shorter) than DPLL [2], the size of the search tree of CDCL can still be very large as it is sensitive to a branching order. Choosing the right variables and their order to shorten the search tree are the primary focus of this paper.

It is a well known fact that not all variables need to be assigned while determining a satisfiable result. With that in mind, we formalize the notion of *careset*, a subset of variables that must be assigned in any satisfying assignment. We extend the definition to an unsatisfiable instance, by defining careset on maximal satisfiable subsets. We propose a restricted branching technique in a CDCL solver such that every decision path is prefixed with a sequence of decisions on such a careset. We refer to such a sequence as a *branching prefix sequence*. Even though finding such a non-trivial set and such a sequence may not be tractable in general, we demonstrate that for a software verification application we can derive them automatically from the sequential behaviors of programs.

We compare formally the proof complexity [24] of restricted CDCL vis-a-vis unrestricted CDCL in terms of the size of the shortest proofs, measured in the number of decisions, they can produce. For a given careset c, and its size $|c|$, we show that the shortest proof (π') (and its size $|\pi'|$) obtained in restricted CDCL cannot be greater than the shortest proof (π) (and its size $|\pi|$) obtained in unrestricted CDCL by more than a factor of $f(c)$ i.e., $|\pi'| \leq f(c) \cdot |\pi|$, where $f(c) = 2^{|c|}$ in general. However, for the software model checking application $f(c)$ can be much smaller than $2^{|c|}$.

For such an application, we demonstrate that our branching technique significantly reduces the search effort of our CDCL solver (based on [10]) by helping it learn shorter and useful clauses earlier during the search process. We observe that the length of clauses learnt are reduced by an order-of-magnitude on average. This leads to a performance gain of 1-2 orders of magnitude over the well known heuristics such as VSIDS [6] and circuit-based [10], [14], [18], [22]. Even though we have not yet included the latest and greatest improvements in our solver, we demonstrate an order of magnitude improved performance of such a restricted CDCL solver over the well-engineered top-ranked solvers of SAT2009 competition [25].

For generality reasons, these advanced solvers do not intend to exploit system-level information. However, without such

information, the performance penalty incurred by these solvers is in orders of magnitude as observed in our experiments. Our goal is to draw attention to the SAT community of substantial progresses that are still possible in branching techniques as they play decisive role in the SAT performance.

B. Related Work

In [13]–[15], [26], problem structure was exploited to restrict the branching only to a smaller set of variables, referred to as an independent variables set (IVS). These variables correspond to non-deterministic initial state variables and primary input variables for circuit applications [14], action variables in planning applications [13], and task variables in task sequencing problems [23]. By definition, these variables dominate others variables that are not in the set i.e., dependent variables. A total assignment on IVS uniquely determines the values of the dependent variables. While such restrictions help in specific applications, they can degrade the performance of CDCL exponentially worse when compared to DPLL on some other application [27].

In [19], a notion of backdoor variables was introduced, where the branching was restricted only to such variables. The idea is that once all of these variables have values, the reduced formula can be solved by a polynomial-time solver. For a constraint Boolean circuit, an IVS is a backdoor set. It was demonstrated [28] that there is a strong correlation between the size of a backdoor set and the hardness of the corresponding Boolean formula. In general, finding a backdoor set from a given Boolean formula, is intractable [28]. Researches have also studied both theoretically and empirically [29] with the notion of backbone set [30]. A backbone set of a satisfiable Boolean formula is a set of literals which are assigned unique common values in every satisfying assignment. It has been shown that finding such a set is also intractable [28].

Our proposed notion of careset is different from the notion of backdoor set or IVS. As we shall see later, a careset is a necessary set while a backdoor set (or IVS) is a sufficient set for a satisfiable formula. A careset is also different from a backbone set, as careset variables need not have a unique common assignment in every satisfying assignment.

In [10], [18], [22] circuit observability don't cares (Cir-ODC) were used to restrict the branching to justification gates only, and avoid branching on the unobservable gates. In general, such a branching is oblivious to system-level information. In [31], functional information such as arithmetic types were used to guide the decision engine. In our previous work [32], we bias the decision choice on variables corresponding to control state predicates, and thereby, use sequential behaviors to guide the search. In this work, we provide a formal justification for such biasing, and further improve the decision process using branching prefix sequences.

Outline: The rest of the paper is organized as follows: With some background in Section II, we formalize the notion of careset, and introduce our branching method in Section III. In Section IV, we give an overview of software model checking. For that application, we present a method to generate careset variables automatically, and describe our branching technique in Section V. This is followed by a formal exposition on proof complexity of the method in Section VI, and its detailed experimental evaluation in Section VII. We give our conclusions and future directions in Section VIII.

II. Preliminaries

`CNF`. A CNF formula F is defined as a conjunctive set, i.e., AND $(\cdot)$ of clauses where each *clause* is a disjunctive set, i.e., OR $(+)$ of literals. A *literal* is a variable v (positive) or its negation $\bar{v}$ (negative). Let $vars(F)$ and $clauses(F)$ represent the set of all variables and clauses in F, respectively. An *assignment* for F is a Boolean function $\alpha : V \mapsto \{0,1\}$, where $V \subseteq vars(F)$. We use $v \in \alpha$ to denote that v is assigned under α. We say an assignment α is *total* if $V = vars(F)$, otherwise, it is *partial*. A literal l is *false (true)* under α if $\alpha(l) = 0(1)$. A variable (and literal) is *free* if it is not assigned. A clause is *satisfied* if at least one of its literals is true. A clause is *conflicting* if all its literals are false. An assignment α is *satisfying* if all clauses in F are satisfied by α, and not necessarily all variables be assigned. We use $F|_\alpha$ to denote the simplified formula where the corresponding assigned variables $(\in \alpha)$ are replaced with their assigned values, and false literals and satisfied clauses are removed. A *maximal satisfiable subset* (MSS) of F corresponds to a subset of clauses of F that is maximally satisfiable, i.e., adding any remaining clause would make it unsatisfiable. For a set S, we use $|S|$ to denote its cardinality.

A *P-Solver* solves a Boolean formula F in polynomial time if it accepts F. For example, a 2SAT-Solver that solves 2SAT-CNF (i.e., a set of clauses with at most of 2 literals) but rejects all others, is a P-Solver. A non-empty set of variables S is a *backdoor* [19] in a satisfiable F if for some assignment $\alpha : S \mapsto \{0,1\}$, P-Solver can show $F|_\alpha$ to be satisfiable. Such a set is *strong* if for all such α, P-Solver can solve $F|_\alpha$, i.e., show it to be sat/unsat. A set of variables S is a *backbone* [30] of satisfiable F if there is a unique partial assignment $\alpha : S \mapsto \{0,1\}$ such that $F|_\alpha$ is satisfiable. Note, assigning opposite value to a backbone variable would make $F|_\alpha$ unsatisfiable.

`Circuit`. We consider a Boolean circuit G represented as a DAG where each node represents a circuit gate, i.e, `OR`, `AND`, `XOR`, or `NOT`, and each edge connects a gate to its fanout node. We define an assignment for G as a Boolean function $\alpha : W \mapsto \{0,1\}$, where W is the set of all gate outputs and primary inputs of G. We say a gate is *justified*, when its input values justify its output value. For example, for $g = \mathsf{AND}(a,b)$, $g = 0$ can be justified by either $a = 0$ or $b = 0$. Note, a primary input and a gate with no output value are always justified. We say a gate is *totally justified*, if its inputs are also justified transitively; otherwise, it is *partially justified*.

A constraint Boolean circuit is a pair $\langle G, \tau \rangle$ where some gates in G are constrained with an assignment τ. Note, without a constraint τ, a Boolean circuit is always satisfiable. We say $\langle G, \tau \rangle$ is satisfiable if there exists an assignment, referred as *justifying*, which (i) preserves the input/output relation of each gate, and (ii) each constraint gate is totally justified. One can encode a constraint Boolean circuit $\langle G, \tau \rangle$ into an equi-satisfiable CNF formula $cnf(\langle G, \tau \rangle)$ in linear-time using standard "Tseitin translation."

`CDCL`. The basic DPLL procedure [2] has three main steps applied repeatedly: branch on a literal, apply *unit propagation* (UP) rule, i.e., forcing a free literal true when all the other

literals in a clause are false, and backtrack chronologically when a conflict is observed. It stops when either all clauses are satisfied or all branches are explored. Conflict-driven Clause learning [7] (CDCL) improves the basic procedure by learning resolvent clauses after analyzing the causes of a conflict. In the sequel, we use "CDCL" to denote any implementation of the CDCL procedure, and use "a CDCL solver" to denote a specific implementation.

III. CARESET

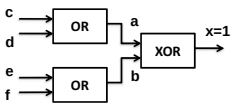
Before we delve into the formal definition of careset, we first define *minimally satisfying assignment* for a satisfiable Boolean formula F for a given P-Solver.

***Definition** 1 (Minimally Satisfying Assignment (MSA)):* We say an assignment α of Boolean formula F is minimally satisfying for a given P-Solver such that (i) UP rule cannot be applied on $F|_\alpha$ further, (ii) $F|_\alpha$ can be shown to be satisfiable by the P-Solver, and (iii) unassigning at least one variable in α would violate the condition (i) or (ii). We use $MSA(F, P)$ to denote the set of all MSAs of F for a given P-Solver.

Example 1: Let F be $(a+\bar{x}+d)(\bar{a}+x+c)(b+\bar{y}+\bar{d})(\bar{b}+y+\bar{c})$. Then, $\alpha = \{x = 0, y = 0\}$ is an MSA w.r.t. a 2SAT-Solver, as $F|_\alpha = (\bar{a} + c)(\bar{b} + \bar{c})$ is a 2SAT-CNF formula.

***Definition** 2 (Minimally Justifying Assignment (MJA)):* For a constraint Boolean circuit $\langle G, \tau\rangle$, we say an assignment β is minimally justifying if un-assigning any $v \in \beta$ would leave some constraint gate partially justified.

Example 2: All MJAs $\beta_1 - \beta_4$ for the constraint circuit $\langle G, \{(x = 1)\}\rangle$ are shown below: Consider a P-Solver



$\beta_1 = \{x = 1, a = 1, b = 0, e = 0, f = 0, c = 1\}$,
$\beta_2 = \{x = 1, a = 1, b = 0, e = 0, f = 0, d = 1\}$,
$\beta_3 = \{x = 1, a = 0, b = 1, c = 0, d = 0, e = 1\}$,
$\beta_4 = \{x = 1, a = 0, b = 1, c = 0, d = 0, f = 1\}$.

that applies arbitrary values to a set of unassigned primary input variables, and applies UP rule recursively on the circuit clauses. Such a solver, referred as `CktSim`, can always satisfy the gate clauses of an unconstraint Boolean circuit.

***Proposition** 1:* β is an MJA of $\langle G, \tau\rangle$ iff β is an MSA for $cnf(\langle G, \tau\rangle)$ w.r.t. a `CktSim` as P-solver.

One can verify that $\beta_1 - \beta_4$ are MSAs for $cnf(\langle G, \tau\rangle)$ w.r.t a `CktSim`. Note that $\alpha = \{x = 1, a = 1, b = 0, e = 0, f = 0\}$ is an MSA w.r.t a 2SAT-Solver, but not w.r.t. a `CktSim`.

In the sequel, we use `CktSim` as the given P-Solver, and use $MSA(F)$ to denote $MSA(F, \texttt{CktSim})$. We now formally introduce the notion of careset for a satisfiable formula F, given `CktSim` as a P-Solver. Let F_{red} denote a reduced formula F after applying the UP rule recursively on F.

***Definition** 3 (Careset):* A non-empty set S of variables ($\subseteq vars(F)$) is a *careset* for a given formula F, such that a S variable is assigned in every MSA of F, i.e., $v \in S \rightarrow \forall \alpha \in MSA(F).\ v \in \alpha$. Such a set S is *maximum* when it includes all such variables, i.e., $S = \{v \mid \forall \alpha \in MSA(F).\ v \in \alpha\}$. We say S is *non-trivial* if $\exists v \in S.\ v \in vars(F_{red})$; otherwise, it is *trivial*.

In the sequel, we use $careset(F)$ to denote a non-trivial careset of F, which may not be maximum unless noted otherwise. Intuitively, a careset is a set of variables that must be assigned to "witness" a satisfying assignment.

Using Proposition 1, we define careset for a Boolean constraint circuit $\langle G, \tau\rangle$ as $careset(F)$ where $F = cnf(\langle G, \tau\rangle)$. For Example 2, non-trivial caresets of $\langle G, (x = 1)\rangle$ are $\{x, a\}$,$\{x, b\}$, and $\{x, a, b\}$ as a, b, c are assigned in all MJAs, i.e., $\beta_1 - \beta_4$. The set $\{x, a, b\}$ is the maximum careset. These caresets are non-trivial as values on a, b cannot be obtained by unit propagation on $x = 1$, while $\{x\}$ is a trivial careset.

We extend the definition of careset to an unsatisfiable formula F by defining it on maximal satisfiable subsets of F. Let $MSS(F)$ denote a set of all MSS of F. Then, $careset(F) := \cup_{F' \in MSS(F)} careset(F')$. Such a careset is maximum, when careset for each F' is maximum. Note, a non-trivial $careset(F')$ for any MSS F' of F is also a non-trivial careset for F.

Comparing Careset, Backdoor, Backbone. In contrast to a backbone set, where variables are necessarily set to unique values, careset variables only need to be assigned, not necessarily to unique values, in any satisfying assignment. Compared to a backdoor set, which is a *sufficient* set, a careset is a *necessary* set for solving a problem satisfiable. Such a necessary set is arguably smaller than a backdoor set, and therefore can help the decision engine prioritize better.

For Example 2, a backbone set is $\{x = 1\}$, a backdoor set is $\{x, a, b, c, e, f\}$ (as `CktSim` returns satisfiable for assignment β_1), a strong backdoor set is $\{c, d, e, f\}$ (as `CktSim` returns SAT/UNSAT for a total assignment on the primary inputs), and a careset is $\{x, a, b\}$.

A. Branching Strategy using Careset

We observe that for a satisfiable instance, a complete assignment on careset variables is a "gateway" to a satisfying solution. Intuitively, for such instances we should branch on careset variables first, before branching on the other variables. Such a branching technique is also a good heuristic for unsatisfiable instances as argued below.

Assume F is unsatisfiable. Let $F' \in MSS(F)$, and $C = clauses(F) \setminus clauses(F')$. Let $vars(\alpha)$ denote the set of variables assigned under $\alpha \in MSA(F')$ and α_S denote values of S variables under assignment α. As F is unsatisfiable, $\exists S \subseteq vars(\alpha)$ such that α_S makes some clause $c \in C$ conflicting. We say α is blocked by c. Any $\beta \in MSA(F')$ is also blocked by $c \in C$, if $\alpha_S = \beta_S$. Since careset variables must be assigned in any MSA of F', branching on them first can lead to early blockage of MSAs, and faster resolution.

We refer to such a branching technique as *branching prefix sequence*. In contrast to a backdoor set where the (ideal) goal is to obtain the smallest set, our (ideal) goal is to obtain the maximum careset. However, obtaining such a set is as hard as finding all MSAs. For practical reasons, we would like to obtain a careset as large as possible, not necessarily maximum. We would like to answer three key questions:

- How can a non-trivial and useful careset be obtained?
- How can such a set be exploited in a CDCL solver?
- How can the strength of such a CDCL solver be accessed?

In Sections IV-V, we answer the first two questions by considering a software model checking application, and using the application-specific knowledge to derive a non-trivial careset and exploit it in CDCL that is restricted with branching prefix sequence. In Section VI, we compare the relative proof complexity of restricted CDCL w.r.t. unrestricted CDCL. In Section VII, we compare experimentally our restricted CDCL solver against the state-of-the-art CDCL solvers that do not exploit such application knowledge.

IV. Application: Model Checking of Software

We briefly discuss our model building step (similar to [32]) from a given C program. We first obtain a simplified control and data flow graph (CDFG) by flattening the structures and arrays into scalar variables of simple finite types (Boolean, 32-bit integer). We handle pointer accesses using direct memory access on a finite heap model, and apply standard slicing and constant propagation. We do not inline non-recursive procedures to avoid blow up, but bound and inline recursive procedures up to a user-defined depth. From the simplified CDFG, we build a deterministic extended FSM (EFSM) where each control state (or block) is identified with a unique *id*. We use a program counter PC to track the control state *id*. For the ease of explanation, we focus on simplified CDFGs that have a unique entry block (Src) and an error block (Err). We are interested in checking reachability properties such as array bounds violations, null pointer dereferencing, and assertion failures; that is, whether there is an execution trace from Src to Err block. We use EFSM and CDFG interchangeably to mean the same structure.

Example 3: Consider a low-level C program `foo` as shown in Figure 1, with its EFSM M. The control states, shown as boxes, correspond to control points in the program, as also indicated by the line numbers. Note, each control state is identified with a number in the attached small square box. For example, *Err* block 10 corresponds to the assertion in line 17. Update transitions of data path expressions are shown at each control state. A directed edge (a, b) between control states a, b corresponds to the control flow between the associated control points in the program. Each directed edge is associated with an enabling condition.

Based on such a CDFG, we encode the transition model T of an EFSM symbolically as $T := T_C \wedge T_D$, where T_C encodes (control) transition relation for PC, i.e., the guarded transitions between the control states, and T_D encodes (data) update transition relation for datapath variables based on the expressions assigned to the variables in various control states in the model. We illustrate the translation of T for Example 3. We use v, v' to denote current and next state variable, g_{ij} to denote the guarded transition predicate at a directed edge (i, j), and $B_r := (PC = r)$ to denote the control state predicate. For ease of readability, we use C syntax '?:' to denote *if-then-else* operator, and other standard relation operators. We obtain a Boolean encoding of the update and the guarded transition relations under the assumption of 32-bit integer variables (not shown separately).

Transition relation for PC $[T_C(PC', PC, a, b)]$

$$PC' := B_1 \wedge g_{12} \ ?\ 2 : B_1 \wedge g_{16} \ ?\ 6 : B_2 \wedge g_{23} \ ?\ 3 : B_2 \wedge g_{24} \ ?\ 4 : \cdots : 11$$

where $\forall r \in \{1, \cdots, 11\}\ B_r := (PC = r)$, and $g_{12} := (a \geq b)$, $g_{16} := (a < b)$, $g_{23} := (a < b)$, $g_{24} := (b \leq a)$, and so on.

Update transition relation $[T_D(a', a, b', b, PC)]$

$$a' := B_1 \ ?\ a_0 : B_4 \ ?\ (a - b) : B_7 \ ?\ (a - b) : a$$
$$b' := B_1 \ ?\ b_0 : B_3 \ ?\ (b - a) : B_8 \ ?\ (b - a) : b$$

where a_0, b_0 are initial symbolic state values of a, b, resp., i.e., $1 \leq a_0, b_0 \leq 10$.

Bounded Model Checking. Let s^i denote a state at i^{th} step from some initial state s^0, and $T(s^i, s^{i+1})$ denote the state transition relation. A BMC instance (denoted as BMC^k) comprises checking if an LTL (Linear Temporal Logic) property ϕ can be falsified in *exactly* k steps from s_0, i.e.,

$$BMC^k := I \wedge T^{0,k} \wedge \neg\phi(s^k) \tag{1}$$

where $\phi(s_k)$ denotes the predicate that ϕ holds in state s^k, and I denote the initial state predicate, and $T^{0,k}$ denote the unrolled transition relation $\bigwedge_{0 \leq i < k} T^{i,i+1}$ where $T^{i,i+1} := T(s^i, s^{i+1})$. Given a bound n, a ***BMC run*** comprises checking the satisfiability of BMC^k iteratively for $0 \leq k \leq n$ using a SAT solver. In the sequel, we focus only on the reachability of block Err from block Src, i.e., $\phi := F(PC = Err)$, where F is the eventually LTL operator, and $I := (PC^0 = Src) \wedge D^0$, where D^0 is the initial state predicate on datapath variables.

A. Control Flow Reachability

We use CFG to denote a CDFG without the enabling condition and update transitions. A *control path* is a sequence of successive control states, denoted as $\gamma^{0,k} = (c_0, \ldots, c_k)$, where (c_i, c_{i+1}) is a directed edge in the CFG. We use $c \in \gamma^{0,k}$ to denote that c belongs to the sequence. An *unrolled CFG* for depth d is a DAG that corresponds to an unfolded CFG where the transitions after depth d is removed, shown as an example in Figure 1 for $d = 7$. A *control state reachability* (CSR) analysis is a breadth-first traversal of the unrolled CFG where a control state b is one step reachable from a *iff* there is a directed edge (a, b). At a given sequential depth d, let $R(d)$ represent the set of control states that can be reached in CFG in one step from the states in $R(d-1)$, with $R(0) = c_0$.

Computing CSR for the unrolled CFG of M (Figure 1), we obtain the set $R(d)$ for $0 \leq d \leq 7$: $R(0) = \{1\}$, $R(1) = \{2, 6\}$, $R(2) = \{3, 4, 7, 8\}$, $R(3) = \{5, 9\}$, $R(4) = \{2, 10, 6, 11\}$, $R(5) = \{3, 4, 7, 8\}$, $R(6) = \{5, 9\}$, $R(7) = \{2, 10, 6, 11\}$.

We use $from(r)$ and $to(r)$ to denote set of blocks reachable from and to r, respectively. The unrolled transition relation $T^{0,k}$ capture the following control flow constraints *implicitly*. We use v^d to denote the unrolled variable v in $T^{d,d+1}$. B_r^d refers to the Boolean control state predicate $(PC^d = r)$, i.e., whether PC at depth d is at control state r. It has been shown that these constraints when added explicitly, improve the search [33].

- Reachable Block Constraint (RBC): At least one block is reachable at d i.e., $\exists r \in R(d)$. (B_r^d).
- Mutual Exclusion Constraint (MEC): At most one block is reachable at d, i.e, $\forall r \neq t$. $(B_r^d \rightarrow \neg B_t^d)$

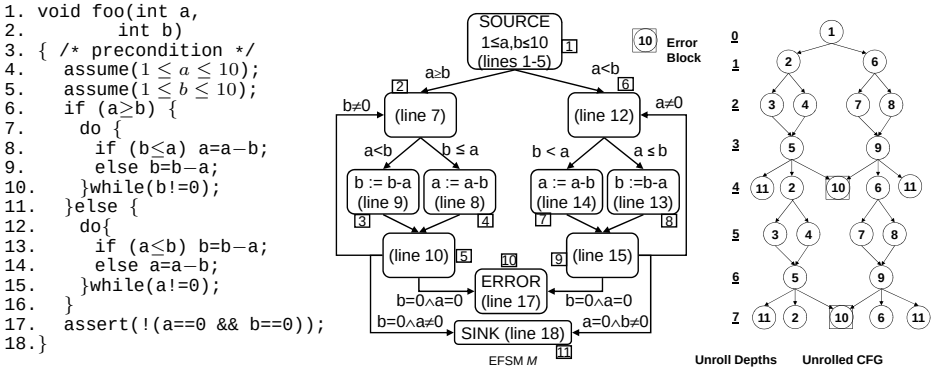


Fig. 1. A sample C code, its EFSM M, and an unrolled CFG for depth 7.

- Forward Reachable Block Constraint (FRBC): If r is reachable at $d < k$, then $t \in from(r)$ is reachable at $d+1$, i.e., $\exists t \in from(r).\ (B_r^d \rightarrow B_t^{d+1})$.
- Backward Reachable Block Constraint (BRBC): If r is reachable at $d > 0$, then $t \in to(r)$ is reachable at $d-1$, i.e., $\exists t \in to(r).\ (B_r^d \rightarrow B_t^{d-1})$.

V. Generating a Careset for BMC

As per Eqn 1, $BMC^k = B^0_{Src} \wedge D^0 \wedge T^{0,k} \wedge B^k_{Err}$, where D^0 is initial state predicate on datapath variables. Let $\Gamma^{a,b}$ denote a set of all control paths $\gamma^{0,k}$ between control states a and b, i.e., $\{\gamma^{0,k} \mid c_0 = a, c_k = b\}$. We say $c_d \in \Gamma^{0,k}$ iff $c_d \in \gamma^{0,k}$ for some $\gamma^{0,k} \in \Gamma^{0,k}$. The following theorem will provide a basis for generating a non-trivial careset for BMC^k.

***Theorem** 1:* A non-trivial careset for BMC^k is a set of control state predicate variables in all the control paths from Src to Err, i.e., $\{B^d_{c_d} \mid c_d \in \Gamma^{Src,Err}, 0 \le d \le k\}$.

Proof. We consider two cases based on whether BMC^k is satisfiable or not.

Case 1: BMC^k is satisfiable. Clearly, $\exists \gamma^{0,k}$ s.t. $\gamma^{0,k}$ witnesses the control reachability of *Err* block from *Src* block. Let α be the corresponding MJA of BMC^k. Then, $\forall c_d \in \gamma^{0,k}.\ \alpha(B^d_{c_d}) = 1$, as otherwise, $B^k_{Err} = true$ would not be totally justified. As per MEC flow constraint, $\forall c_d, c'_d \in R(d).\ B^d_{c_d} \rightarrow \neg B^d_{c'_d}$, i.e., control predicate variables in the control paths other than $\gamma^{0,k}$ are implied false. Thus, the claim holds.

Case 2: BMC^k is unsatisfiable. We construct an MSS F' of BMC^k as follows: Initially, $F' = \emptyset$. We include the constraints $(B^0_{Src} \wedge B^k_{Err})$, and add all the constraints corresponding to the unrolled expressions for PC without the guarded expressions, i.e, we treat g^d_{ij} as free input variables. Note, F' constructed so far captures only the control flow constraints, and is therefore satisfiable. We then add the remaining data path and guarded expressions until F' becomes an MSS of BMC^k. By definition of a careset and using the argument as in Case 1, the claim follows. □

Based on the above theorem, we obtain a $careset(BMC^k)$ by doing forward and backward traversal from Src and Err blocks, resp. on the CFG, and including B^d_r corresponding to a control state $r \in R(d)$ that is visited by traversal in both direction. For Example 3 (Figure 1), $careset(BMC^4)$ is $\{B^0_1, B^1_2, B^2_3, B^2_4, B^3_5, B^1_6, B^2_7, B^2_8, B^3_9, B^4_{10}\}$.

A. Branching Prefix Sequence

We introduce the notion of branching prefix sequence (BPS)[1], a kind of restrictive branching where every decision path (starting at decision level 0) is prefixed with a given ordered sequence of branching literals.

***Definition** 4:* A *branching prefix sequence* (BPS) for a formula F is an ordered sequence $\sigma = (l_1, \cdots l_m)$ of literals of F such that a CDCL solver always picks first free literal l_i in σ (skips the assigned literals), and branches with l_i set to true. If all the literals in σ are assigned, default branching heuristic is applied. During backtracking, some of the literals in σ can become free. At any decision level, the solver always branches on the first free literal in σ, if one exists. However, the literals of σ are neither removed nor reordered. CDCL using a BPS is referred to as CDCL$_{bps}$.

We use the $careset(BMC^k)$ variables to obtain a BPS. We first define an ordering relation based on control distance of a careset variable $B^d_r \in careset(BMC^k)$.

***Definition** 5:* A control distance of a careset variable $B^d_r \in careset(BMC^k)$ is a function $\delta : careset(BMC^k) \mapsto \{0, \cdots, k\}$ such that $\delta(B^d_r) = k - d$.

For example, $B^1_6 \in careset(BMC^4)$ has a control distance $\delta(B^1_6) = 4 - 1 = 3$.

***Definition** 6:* An increasing (decreasing) sequence of literals in $careset(BMC^k)$ is defined as a total order on the careset variables (i.e., positive literals) with respective to a non-decreasing (non-increasing) control distances. Variables with the same control distances are ordered using some heuristic such as literal count. We use IS^k (DS^k) to denote increasing (decreasing) sequence of $careset(BMC^k)$.

An increasing sequence IS^4 for $careset(BMC^4)$ is as follows: $\{B^4_{10}(0)$, $B^3_5(1)$, $B^3_9(1)$, $B^2_3(2)$, $B^2_4(2)$, $B^2_7(2)$, $B^2_8(2)$, $B^1_2(3)$, $B^1_6(3)$, $B^0_1(4)\}$, where the values in the brackets refer to the respective control distances of the variables. Here we broke the tie using the corresponding control state id. In actual implementation, we use the VSIDS [6] scores.

Intuitively, an IS^k used as a BPS helps a CDCL solver to prune the *infeasible local path segments that are closer* to Err block by learning useful clauses with fewer decisions. We observed in our experiments that such an approach reduces the average length of conflict clauses per conflict (denoted as

[1]The notion of BPS differs from branching sequence [23] where a literal is chosen once, and may not be assigned on every decision path.

AvgCL) and search tree size (i.e., number of decisions) by 1-2 orders of magnitude. In Section VII, we provide detailed experimental results supporting our intuition. Now we discuss the proof complexity of such an approach.

VI. Proof Complexity of CDCL$_{bps}$

We use the notion of proof complexity [24] to compare the relative power of proof inference systems P and P' based on the shortest proofs π and π' they can produce, resp. Let $|\pi|$ and $|\pi'|$ denote the respective proof sizes. We say P' *polynomially simulates* P when $|\pi'| \leq 2^{O(logn)}(= poly(n)) \cdot |\pi|$ for all families of formula over n variables; otherwise, P' cannot polynomially simulate P. For example, it was shown [23] that DPLL cannot polynomially simulate CDCL.

In CDCL and CDCL$_{bps}$ proof systems, we measure the size of their shortest proofs in terms of their search tree, i.e., the number of decisions. As CDCL is unrestricted, an identical proof can be obtained in CDCL as in CDCL$_{bps}$ by applying the same decision order in CDCL as applied in CDCL$_{bps}$. Thus, the following holds trivially.

***Proposition** 2:* CDCL polynomially simulates CDCL$_{bps}$.

Unfortunately, we cannot claim in the other direction due to branching restriction in CDCL$_{bps}$. However, we provide a worst case bound on the size of its shortest proof. For any unsatisfiable formula F over n variables, let S be a careset of F, with $|S|$ denoting its size. Let $\pi_{bps}(F)$ ($\pi(F)$) and $|\pi_{bps}(F)|$ ($|\pi(F)|$ denote the shortest proof and its size, resp., obtained in CDCL$_{bps}$ (CDCL).

***Theorem** 2:* The shortest proof obtained in CDCL$_{bps}$ cannot be greater than that obtained in CDCL by more than a factor of $f(S)$, i.e., $|\pi_{bps}(F)| \leq f(S) \cdot |\pi(F)|$, where $f(S) = 2^{|S|}$.

Proof. Consider the search tree of CDCL$_{bps}$. Let $\mathcal{U} = \{\sigma_1 \cdots \sigma_m\}$ represent a set of unique assignments made on careset variables before a proof is generated in CDCL$_{bps}$, i.e, $\forall(i \neq j).\exists v \in \sigma_i, v \in \sigma_j.\ \sigma_i(v) \neq \sigma_j(v)$. Clearly, as F is unsatisfiable, each (partial) assignment on $vars(F)$ that ends in conflict, includes exactly one σ_i for some i. We claim that $|\pi_{bps}(F)| = \Sigma_i^m |\pi_{bps}(F|_{\sigma_i})| = \Sigma_i^m |\pi(F|_{\sigma_i})| \leq \Sigma_i^m |\pi(F)| \leq 2^{|S|} \cdot |\pi(F)|$. The first equality holds as assignments on careset variables are made before the rest in CDCL$_{bps}$. The second equality holds as branching heuristics of CDCL$_{bps}$ and CDCL are the same on non careset variables. The following inequality holds as CDCL is a natural proof system. The last inequality holds as $m \leq 2^{|S|}$. □

In practice, we often see conflict before all the careset variables are assigned. Moreover, all variables in S may not be independent, i.e., some are implied by others in S, in which case $m \ll 2^{|S|}$. Especially, for $F = BMC^k$ (Eqn 1), $S = careset(BMC^k)$ (Theorem 1), and $\Gamma^{Src,Err}$ (denoting a set of control paths from Src to Err), upper bound on $\mathcal{U}$ is determined by the number of control paths, i.e., $|\Gamma^{Src,Err}|$.

***Corollary** 1:* $|\pi_{bps}(F)| \leq |\Gamma^{Src,Err}| \cdot |\pi(F)|$.

VII. Experiments

We experimented with eight sets of benchmarks `E1-E8`, each with 1 to 3 properties. These correspond to software models and properties generated using software verification platform `F-Soft` [32] from real-world C programs such as network protocol and mobile software. Our experiments were conducted on a Linux box with Intel Pentium 4 CPU 3.2GHz, 2GB of RAM. On these models, we used a SAT-based BMC [17], [33].

We used an incremental hybrid SAT solver [10], [17], where a BMC instance is represented in And-Inverter circuit graph (AIG) and the learnt clauses are represented in CNF. It implements Chaff algorithm [6] using 1UIP clause learning scheme [34], and VSIDS [6] for branching. Note, it does not include many recent improvements such as preprocessing (SATeLite [9]), learning binary clauses during BCP [8], smart frequent restarts [4], and others (e.g. [11], [12]). Unlike a pure CNF solver[2], it has direct access to circuit information. Optionally, it uses circuit-based branching heuristics [10], [14], [18] (such as branching on the inputs of currently unjustified gates only). We refer to this branching heuristic as `CKT`.

We also provide such a hybrid solver with a BPS. For each BMC^k instance, we automatically generate sequences IS^k and DS^k (ref. Section V-A). We use `iBPS` (`dBPS`) to denote the branching heuristic where IS^k (DS^k) is used as a BPS.

Combining the above heuristics, we consider following four solvers `B1-B4` for performance comparison.

- `B1:VSIDS` The CDCL solver with `VSIDS` heuristic.
- `B2:CKT+VSIDS` The CDCL solver with `CKT` heuristic. However, when there are many choices at a decision level, the tie is broken with `VSIDS` heursitic.
- `B3:iBPS+CKT+VSIDS` The CDCL solver with `iBPS` heuristic. When there is no free literal in the BPS, it branches as `B2` until the BPS has a free literal.
- `B4:dBPS+CKT+VSIDS` Similar to `B3` but uses `dBPS`.

Experiment Set I. We compare the performance of the solvers `B1-B4` on benchmarks `E1-E8` for each BMC run, comprising solving BMC^k for each $k \geq 0$ until time out. We gave a timeout of 1200s for each BMC run. In each run, we generate and solve BMC instances incrementally at depth k. Other than branching, all other heuristics were kept the same. We show the results in Figure 2.

In `X-axis` we show BMC depths analyzed before timeout occurs, and in `Y-axis` we show the cumulative solve time (in sec) after each unrolled depth. We also labeled a few selected graphs for better readability.

Clearly, `B3` outperforms `B1,B2,B4` by several orders of magnitude. `B4` outperforms `B1` only in 3 cases, i.e., `E1`, `E3`, `E7`. This shows that branching order is equally important in a CDCL solver. We do not see much improvement of `B2` over `B1`. `B3` finds two witnesses (at depths 43 and 63, resp.) in `E7` while `B4` finds only the shorter witness (i.e., at depth 43). `B1` or `B2` finds neither of them. Clearly, circuit information does not provide much of guidance compared to system-level information. We do not show separately the comparison data with the solver used in [32], wherein the control state predicate variables are given higher VSIDS scores initially. We observed that the performance of such a solver is marginally better or comparable to that of `B1` on these BMC runs.

We provide detailed comparison results between `B2` and `B3` in Table I. For each benchmark, we obtained a list of BMC^k

[2]In a CNF solver, one can use Cir-ODC CNF encoding [22] to exploit circuit information, albeit with additional overhead compared to [10].

instances that were solved by B2 and B3 in more than 5s but less than 1200s. Note, all these instances are unsatisfiable. After sorting them on k, we selected *min*imum, *med*ian and *max*imum instances from the list as shown in Columns 1-2. The number of variables $(\#V)$ in these instances are about 200k-1.3M, as shown in Column 3. In Columns 4-6, we present the results of B2: the number of decisions (#D), the average conflict-clause length (AvgCL), and the time taken (in sec) (T). In Columns 7-11, we present the results of B3: the size of careset as percentage of #V (#CV), the number of decisions (#D), the number of decisions on careset variables as the percentage of #D (#DCV), the average conflict-clause length AvgCL, and the time taken T(s), resp. We observe that baring a few cases, AvgCL is about an order of magnitude smaller in B3, compared to B2. Clearly, iBPS guides the solver B3 better in learning useful clauses earlier, thereby, reducing the overall solve time significantly. We also find that the number of careset variables is about 1-3% of the number of variables, and in the most unsatisfiable instances, decisions on them are sufficient to solve the instance.

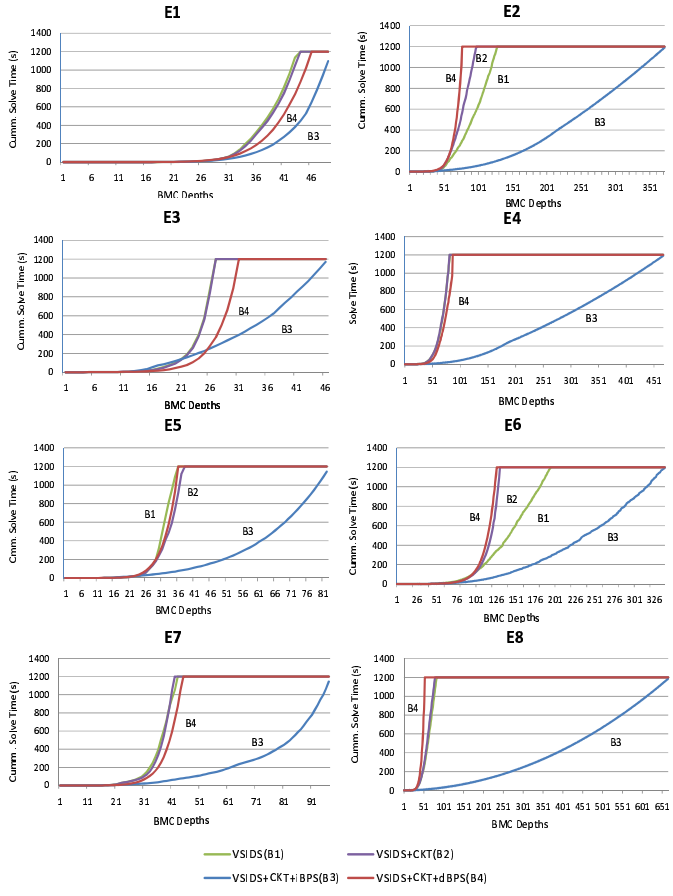


Fig. 2. BMC runs on E1-E8

Experiment Set II. We obtained DIMACS CNF format from the BMC instances at different depths from Experiment Set I. To keep our focus on hard instances, we included those on which B3 solver takes more than 5 sec to solve. We obtained a total of 130 instances; out of which 128 are

TABLE I
COMPARING B2 AND B3 ON (UNSAT) BMC^k.

BMC^k Instance			B2: CKT+VSIDS			B3: iBPS+CKT+VSIDS				
Ex	Depth k	#V ('000)	#D	AvgCL (#lit)	T (s)	#CV (%#V)	#D	#DCV (%#D)	AvgCL (#lit)	T (s)
E1	min: 28	262	381	247	5	1.1	710	100	27	3
	med: 38	445	1533	691	56	1.1	1977	100	99	21
	max: 43	522	1400	1081	62	1.1	595	100	251	25
E2	min: 46	250	669	275	5	2.3	137	100	27	0
	med: 47	257	421	145	5	2.3	137	100	27	0
	max: 49	271	909	223	9	2.3	124	100	30	0
E3	min: 16	582	242	350	8	0.7	435	100	28	13
	med: 22	1121	1107	2076	61	0.7	557	100	35	17
	max: 24	1323	413	5984	50	0.7	25	100	201	3
E4	min: 42	194	855	75	5	1.3	46	100	30	0
	med: 49	239	1293	85	8	1.3	47	100	63	0
	max: 80	440	2296	682	59	1.2	80	100	158	0
E5	min: 23	442	249	1547	5	0.7	275	100	32	1
	med: 33	721	1844	1262	66	0.7	386	100	286	2
	max: 36	804	1752	1773	65	0.7	198	100	246	1
E6	min: 93	223	397	3980	5	1.4	280	25	499	0
	med: 100	241	550	4062	9	1.4	204	68	537	1
	max: 130	320	1240	8773	88	1.4	320	77	675	2
E7	min: 28	223	243	610	5	1.0	248	100	70	0
	med: 31	257	245	550	8	1.0	141	92	49	0
	max: 40	359	973	1189	56	1.0	348	50	226	1
E8	min: 33	231	766	76	5	1.0	121	100	9	0
	med: 35	250	807	124	6	1.0	109	100	8	0
	max: 38	278	1021	67	8	1.0	109	100	10	0

#V: number of variables in thousands, #D: number of decisions
AvgCL: average conflict-clause length
#CV: careset size as % of #V, #DCV: % of #D on careset variabes)

TABLE II
NECLA SAT VS SAT2009 WINNERS [25].

Solver	SAT Time(s)	UNSAT Time(s)	Total Time(s)
NECLA SAT	24 (2)	2,042 (128)	2,066 (130)
Preco-SAT	468 (2)	18,367 (128)	18,835 (130)
mini-SAT	372 (1)	19,475 (91)	19,847 (92)
			1,601† (NECLA) 12,131† (Preco)
glucose	663 (1)	39,411 (111)	40,074 (112)
			1,789‡ (NECLA) 17,381‡ (Preco)

(.) # solved cases (out of 130) within 1800 sec.
† Time taken on 92 solved cases of miniSAT
‡ Time taken on 112 solved cases of glucose

unsatisfiable and 2 are satisfiable. The number of variables in these instances ranges between 120K-2.2M, and the number of clauses ranges between 350K-6.6M. These benchmarks are also made publicly available [35].

We used B3 solver without CKT heuristic, and refer it as NECLA SAT solver[3] (For the sake of fair comparison, and to show the benefits of iBPS exclusively, we disabled circuit heuristics.) We compared this solver with PrecoSAT, miniSAT, and glucose, the top-ranked solvers in SAT2009 competition under application category [25]. These solvers use explicit or in-built preprocessor (e.g., SATeLite [9]), and smart frequent restarts [4], while NECLA SAT does not include any of these techniques. We gave a time limit of 1800s per instance. We provide a summary of the results in Table II.

In Column 1, we list the solvers we compared. In Column 2(3), we present the solve time (in sec) for SAT(UNSAT) instances. In Column 4 we present the total time taken (in sec) for the solved instances only. In Columns 2-4, we show the number of instances solved by each solver in brackets. Where the solved instances are less than 130, i.e., for glucose and

[3]NECLA SAT w/o iBPS corresponds to B1 solver.

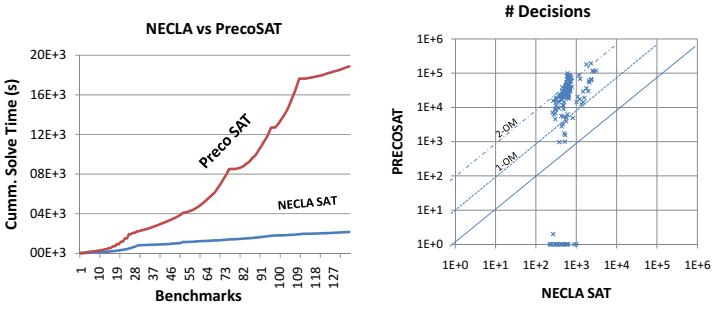


Fig. 3. NECLA SAT vs PrecoSAT: Cumm times, and # decisions

`miniSAT`, we also provide the total time taken by `NECLA SAT` and `PrecoSAT` on those solved instances.

`NECLA SAT` and `PrecoSAT` solve all 130 instances, while `glucose` and `miniSAT` solve 112 and 92 instances, respectively. Clearly, `NECLA SAT` solver outperforms the rest solvers by about an order of magnitude.

We also compare `NECLA SAT` and `PrecoSAT` in more details as shown in Figure 3. In the left figure, we show the instances solved (along `X-axis`), and the cumulative time taken in sec (along `Y-axis`). We observe that `NECLA SAT` outperforms `PrecoSAT` consistently by about an order of magnitude. In the right figure, we present a scatter plot comparing the number of decisions between `NECLA SAT` (along `X-axis`) and `PrecoSAT` (along `Y-axis`) in *logarithmic scale*, where each 'x' mark corresponds to an instance solved. We observe that the number of decisions in `NECLA SAT` are 1-2 orders of magnitude smaller than that in `PrecoSAT`, as indicated by clustering of 'x' between dotted lines namely, `1-OM` and `2-OM`. All the marks on `X-axis` (i.e., with 0 decision in `PrecoSAT`) are instances solved by the in-built preprocessor [9] of `PrecoSAT`. For these instances, `NECLA SAT` takes about 5-10s, about the same as the preprocessing time. `NECLA SAT` also requires an order of magnitude fewer backtracks comparatively (not shown separately). Clearly, benefit from using `iBPS` outweighs many heuristics in `PrecoSAT`.

VIII. Conclusion and Future Work

Branching plays a crucial role in a CDCL solver. We introduce the notion of careset variables and branching prefix sequence to guide the decision engine. We derive such a careset from software model checking application, and use it to improve the performance of a CDCL solver by an order of magnitude compared to the latest best SAT solvers that do not exploit system-level information. We also compared formally the resolution power of restricted CDCL vis-a-vis unrestricted CDCL. Overall, our results serve as a proof of concept that the analysis of system behaviors can be used to improve a SAT solver performance dramatically.

On the one hand, the current advanced SAT solvers do not intend to take advantage of system-level information for generality reasons, but on the other hand, the performance penalty of not using such information could be in the orders of magnitude, as observed in our software model checking experiments. For a better trade-off, we believe that there is a further scope in improving SAT-formulation where one can generate SAT problems conducive for the state-of-the-art solvers. In future, we would also like to detect careset during runtime, in contrast to its static determination as presented.

References

[1] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *Proc. of TACAS*, 1999.

[2] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem proving. *Communications of the ACM*, 1962.

[3] C. P. Gomes, B. Selman, and H. Kautz. Boosting combinatorial search through randomization. In *Proc. of AAAI*, 1998.

[4] J. Huang. The effect of restarts on the efficiency of clause learning. In *Proc. of IJCAI*, 2007.

[5] J. N. Hooker and V. Vinay. Branching rules for satisfiability. *In Proc. of JAR*, 1995.

[6] M. W. Moskewicz and C. F. Madigan and Y. Zhao and L. Zhang and S. Malik. Chaff: Engineering an efficient sat solver. In *Proc. of DAC*, 2001.

[7] J. Silva and K. Sakallah. GRASP–A New Search Algorithm For Satisfiability. In *Proc. of ICCAD*, Santa Clara, CA, November 1996.

[8] F. Bacchus and J. Winter. Effective preprocessing with hyper-resolution and equality reduction. In *SAT*, 2003.

[9] N. Eén and A. Biere. Effective preprocessing in sat through variable and clause elimination. In *Proc. of SAT*, 2005.

[10] M. K. Ganai, P. Ashar, A. Gupta, L. Zhang, and S. Malik. Combining strengths of circuit-based and CNF-based algorithms for a high-performance SAT solver. In *Proc. of DAC*, 2002.

[11] A. Biere. PicoSAT essentials. In *Proc. of JSAT*, 2008.

[12] A. Biere. P{re,i}coSAT@SC'09. In *SAT Competition*, 2009.

[13] E. Giunchiglia, A. Massarotto, and R. Sebastiani. Act, and the rest will follow: Exploiting determinism in planning as satisfiability. In *Proc. of AAAI*, pages 948–953, 1998.

[14] O. Strichman. Tuning SAT checkers for bounded model checking. In *Proc. of CAV*, 2000.

[15] E. Giunchiglia, M. Maratea, and A. Tacchella. Dependent and independent variables in propositional satisfiability. In *Proc. of JELIA*, 2002.

[16] R. Ostrowski, É. Grégoire, B. Mazure, and L. Sais. Recovering and exploiting structural knowledge from cnf formulas. In *CP*, 2002.

[17] M. K. Ganai and A. Gupta. *SAT-based Scalable Formal Verification Solutions*. Springer Science and Business Media, 2007.

[18] M. Järvisalo, T. A. Junttila, and I. Niemelä. Justification-based non-clausal local search for sat. In *ECAI*, 2008.

[19] R. Williams, C. P. Gomes, and B. Selman. Backdoors to typical case complexity. In *Proc. of IJCAI*, 2003.

[20] H. Kautz, D. Mcallester, and B. Selman. Exploiting variable dependency in local search. In *In Abstracts of the Poster Sessions of IJCAI-97*, 1997.

[21] F. A. Aloul, A. Ramani, I. L. Markov, and K. A. Sakallah. Solving difficult sat instances in the presence of symmetry. In *Proc. of DAC*, 2002.

[22] Z. Fu, Y. Yu, and S. Malik. Considering circuit observability don't cares in cnf satisfiability. In *Proc. of DATE*, 2005.

[23] P. Beame, H. A. Kautz, and A. Sabharwal. Towards understanding and harnessing the potential of clause learning. *In Proc. of JAIR*, 2004.

[24] S. A. Cook and R. A. Reckhow. The relative efficiency of propositional proof systems. In *Journal of Symbolic Logic*, 1977.

[25] SAT competiton 2009. `http://www.satcompetition.org/2009/`.

[26] J. M. Crawford and A. B. Baker. Experimental results on the application of satisfiability algorithms to scheduling problems. *Proc. of AAAI*, 1994.

[27] M. Järvisalo and T. A. Junttila. Limitations of restricted branching in clause learning. *Constraints*, 14(3), 2009.

[28] P. Kilby, J. K. Slaney, S. Thiébaux, and T. Walsh. Backbones and backdoors in satisfiability. In *AAAI*, 2005.

[29] A. J. Parkes. Clustering at the phase transition. In *AAAI/IAAI*, 1997.

[30] R. Monasson, R. Zecchina, S. Kirkpatrick, B. Selman, and L. Troyansky. Determining computational complexity from characteristic 'phase transitions'. In *Nature 400:*, 1999.

[31] R. Wille, G. Fey, D. Große, S. Eggersglüß, and R. Drechsler. SWORD: A SAT like prover using word level information. In *VLSI-SoC*, 2007.

[32] F. Ivančić, Z. Yang, M. K. Ganai, A. Gupta, and P. Ashar. Efficient SAT-based Bounded Model Checking for Software Verification. In *Proceedings of ISOLA*, 2004.

[33] M. K. Ganai and A. Gupta. Accelerating high-level bounded model checking. In *Proc. of ICCAD*, 2006.

[34] L. Zhang and C. F. Madigan and M. H. Moskewicz and S. Malik. Efficient conflict driven learning in a boolean satisfiability solver. In *Proc. of ICCAD*, 2001.

[35] System Analysis and Verification Team. NECLA SAV Benchmarks. *http://www.nec-labs.com/research/system/systems_SAV-website/benchmarks.php*.

Efficiently Solving Quantified Bit-Vector Formulas

Christoph M. Wintersteiger
ETH Zurich
Zurich, Switzerland
christoph.wintersteiger@inf.ethz.ch

Youssef Hamadi
Microsoft Research, Cambridge, UK
LIX École Polytechnique, Palaiseau, France
youssefh@microsoft.com

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

***Abstract*—In recent years, bit-precise reasoning has gained importance in hardware and software verification. Of renewed interest is the use of symbolic reasoning for synthesising loop invariants, ranking functions, or whole program fragments and hardware circuits. Solvers for the quantifier-free fragment of bit-vector logic exist and often rely on SAT solvers for efficiency. However, many techniques require quantifiers in bit-vector formulas to avoid an exponential blow-up during construction. Solvers for quantified formulas usually flatten the input to obtain a quantified Boolean formula, losing much of the word-level information in the formula. We present a new approach based on a set of effective word-level simplifications that are traditionally employed in automated theorem proving, heuristic quantifier instantiation methods used in SMT solvers, and model finding techniques based on skeletons/templates. Experimental results on two different types of benchmarks indicate that our method outperforms the traditional flattening approach by multiple orders of magnitude of runtime.**

I. Introduction

The complexity of integrated circuits continues to grow at an exponential rate and so does the size of the verification and synthesis problems arising from the hardware design process. To tackle these problems, bit-precise decision procedures are a requirement and oftentimes the crucial ingredient that defines the efficency of the verification process.

Recent years also saw an increase in the utility of bit-precise reasoning in the area of software verification where low-level languages like C or C++ are concerned. In both areas, hardware and software design, methods of automated synthesis (e.g., LTL synthesis [23]) become more and more tangible with the advent of powerful and efficient decision procedures for various logics, most notably SAT and SMT solvers. In practice, however, synthesis methods are often incomplete, bound to very specific application domains, or simply inefficient.

In the case of hardware, synthesis usually amounts to constructing a module that implements a specification [23], [20], while for software this can take different shapes: inferring program invariants [16], finding ranking functions for termination analysis [28], [24], [8], program fragment synthesis [26], or constructing bugfixes following an error-description [27] are all instances of the general synthesis problem.

In this paper, we present a new approach to solving quantified bit-vector logic. This logic allows for a direct mapping of hardware and (finite-state) software verification problems and is thus ideally suited as an interface between the verification or synthesis tool and the decision procedure.

In many practically relevant applications, support for uninterpreted functions is not required and if this is the case, quantified bit-vector formulas can be reduced to quantified Boolean formulas (QBF). In practice however, QBF solvers face performance problems and they are usually not able to produce models for satisfiable formulas, which is crucial in synthesis applications. The same holds true for many automated theorem provers. SMT solvers on the other hand are efficient and produce models, but usually lack complete support for quantifiers.

The ideas in this paper combine techniques from automated theorem proving, SMT solving and synthesis algorithms. We propose a set of simplifications and rewriting techniques that transform the input into a set of equations that an SMT solver is able to solve efficiently. A model finding algorithm is then employed to refine a candidate model iteratively, while we use function or circuit templates to reduce the number of iterations required by the algorithm. Finally, we evalutate a prototype implementation of our algorithm on a set of hardware and software benchmarks, which indicate speedups of up to five orders of magnitude compared to flattening the input to QBF.

II. Background

We will assume the usual notions and terminology of first order logic and model theory. We are mainly interested in many-sorted languages, and bit-vectors of different sizes correspond to different sorts. We assume that, for each bit-vector sort of size n, the equality $=_n$ is interpreted as the identity relation over bit-vectors of size n. The if-then-else (multiplexer) bit-vector term ite_n is interpreted as usual as $ite(true, t, e) = t$ and $ite(false, t, e) = e$. As a notational convention, we will always omit the subscript. We call 0-arity function symbols *constant* symbols, and 0-arity predicate symbols *propositions*. *Atoms*, *literals*, *clauses*, and *formulas* are defined in the usual way. Terms, literals, clauses and formulas are called *ground* when no variable appears in them. A *sentence* is a formula in which free variables do not occur. A *CNF formula* is a conjunction $C_1 \wedge \ldots \wedge C_n$ of clauses. We will write CNF formulas as sets of clauses. We use a, b and c for constants, f and g for function symbols, p and q for predicate symbols, x and y for variables, C for clauses, φ for formulas, and t for terms. We use $x{:}\, n$ to denote that variable x is a bit-vector of size n. When the bit-vector size is not specified, it is implicitly assumed to be 32. We use $f{:}\, n_1, \ldots, n_k \to n_r$ to denote to denote that function symbol

f has arity k, argument bit-vectors have sizes $n_1, \ldots, n_k$, and the result bit-vector has size n_r.

We use $\varphi[x_1, \ldots, x_n]$ to denote a formula that may contain variables $x_1, \ldots, x_n$, and similarily $t[x_1, \ldots, x_n]$ is defined for a term t. Where there is no confusion, we denote $\varphi[x_1, \ldots, x_n]$ by $\varphi[\overline{x}]$ and $t[x_1, \ldots, x_n]$ by $t[\overline{x}]$. In the rest of this paper, the difference between functions and predicates is trivial, and we will thus only discuss functions except at a few places.

We use the standard notion of a structure (interpretation). A structure that satisfies a formula F is said to be a model for F. A theory is a collection of first-order sentences. Interpreted symbols are those symbols whose interpretation is restricted to the models of a certain theory. We say a symbol is free or uninterpreted if its interpretation is not restricted by a theory. We use $BitVec$ to denote the bit-vector theory. In this paper we assume the usual interpreted symbols for bit-vector theory: $+_n$, $*_n$, $concat_{m,n}$, $\leq_n$, 0_n, 1_n, ... Where there is no confusion, we omit the subscript specifying the actual size of the bit-vector.

A formula is *satisfiable* if and only if it has a model. A formula F is satisfiable modulo the theory $BitVec$ if there is a model for $\{F\} \cup BitVec$.

III. Quantified Bit-Vector Formulas

A *Quantified Bit-Vector Formula* (QBVF) is a many-sorted first-order logic formula where the sort of every variable is a bit-vector sort. The QBVF-satisfiability problem is the problem of deciding whether a QBVF is satisfiable modulo the theory of bit-vectors. This problem is decidable because every universal (existental) quantifier can be expanded into a conjunction (disjunction) of potentially exponential, but finite size. A distinguishing feature in QBVF is the support for uninterpreted function and predicate symbols.

Example 1: Arrays can be easily encoded in QBVF using quantifiers and uninterpreted function symbols. In the following formula, the uninterpreted functions f and f' are used to represent arrays from bit-vectors of size 8 to bit-vectors of the same size, and f' is essentially the array f updated at position $a + 1$ with value 0:

$$f'(a+1) = 0 \ \wedge \ (\forall x : 8.\ x = a + 1 \vee f'(x) = f(x)) \ .$$

Quantified *Boolean* formulas (QBF) are a generalization of Boolean formulas, where quantifiers can be applied to each variable. Deciding a QBF is a PSPACE-complete problem. Note that any QBF problem can be easily encoded in QBVF by using bit-vectors of size 1. The converse is not true, QBVF is more expressive than QBF. For instance, uninterpreted function symbols can be used to simulate non-linear quantifier prefixes. The EPR fragment of first-order logic comprises formulas of the form $\exists^*\forall^*\varphi$, where φ is a quantifier-free formula with predicates but without function symbols. EPR is a decidable fragment because the Herbrand universe of a EPR formula is always finite. The satisfiability problem for EPR is NEXPTIME-complete.

Theorem 1: The satisfiability problem for QBVF is NEXPTIME-complete.

QBVF can be used to compactly encode many practically relevant verification and synthesis problems. In hardware verification, a fixpoint check consists in deciding whether k unwindings of a circuit are enough to reach all states of the system. To check this, two copies of the k unwindings are used: Let $T[x, x']$ be a formula encoding the transition relation and $I[x]$ a formula encoding the initial states of a circuit. Furthermore, we define

$$T^k[x, x'] \equiv T[x, x_0] \wedge \left(\bigwedge_{i=1}^{k-1} T[x_{i-1}, x_i] \right) \wedge T[x_{k-1}, x'] \ .$$

Then a fixpoint check for k unwindings corresponds to the QBV formula

$$\forall x, x' \ .\ I[x] \wedge T^k[x, x'] \rightarrow \exists y, y' \ .I[y] \wedge T^{k-1}[y, y'] \ ,$$

where x, x', y, and y' are (usually large) bit-vectors.

Of renewed interest is the use of symbolic reasoning for synthesing code [26], loop invariants [7], [16] and ranking functions [8] for finite-state programs. All these applications can be easily encoded in QBVF. To illustrate these ideas, consider the following abstract program:

pre
while (c) { T }
$post$

In the loop invariant synthesis problem, we want to synthesise a predicate I that can be used to show that $post$ holds after execution of the *while-loop*. Let $pre[x]$ be a formula encoding the set of states reachable before the beginning of the loop, $c[x]$ be the encoding of the entry condition, $T[x, x']$ be the transition relation, and $post[x]$ be the encoding of the property we want to prove. Then, a suitable loop invariant exists if the following QBV formula is satisfiable.

$$\begin{array}{l} \forall x.\ pre[x] \rightarrow I(s) \ \wedge \\ \forall x, x'.\ I(x) \wedge c[x] \wedge T[x, x'] \rightarrow I(x') \ \wedge \\ \forall x.\ I(x) \wedge \neg c[x] \rightarrow post[x] \end{array}$$

An actual invariant can be extracted from any model that satisfies this formula.

Similarly, in the ranking function synthesis problem, we want to synthesise a function $rank$ that decreases after each loop iteration and that is bounded from below. The idea is to use this function to show that a particular loop in the program always terminates. This problem can be encoded as the following QBVF satisfiability problem.

$$\begin{array}{l} \forall x.\ rank(x) \geq 0 \ \wedge \\ \forall x, x'.\ c[x] \wedge T[x, x'] \rightarrow rank(x') < rank(x) \end{array}$$

Note that the general case of this encoding requires uninterpreted functions. The call to $rank$ can not be replaced with an existentially quantified variable, as it is impossible to express the correct variable dependencies in a linear quantifier prefix.

IV. SOLVING QBVF

In this section, we describe a QBVF solver based on ideas from first-order theorem proving, SMT solving and synthesis tools. First, we present a set of simplifications and rewriting rules that help to greatly reduce the size and complexity of typical QBV formulas. Then, we describe how to check whether a given model satisfies a QBVF and how to use this to construct new models, using templates to speed up the process (sometimes exponentially).

A. Simplifications & Rewriting

Modern first-order theorem provers spend a great part of their time in simplifying/contracting operations. These operations are inferences that remove or modify existing formulas. Our QBVF solver implements several simplification/contraction rules found in first-order provers. We also propose new rules that are particularly useful in our application domain.

1) Miniscoping: Miniscoping is a well-known technique for minimizing the scope of quantifiers [17]. We apply it after converting the formula to negation normal form. The basic idea is to distribute universal (existential) quantifiers over conjunctions (disjunctions). This transformation is particularly important in our context because it increases the applicability of rules based on rewriting and macros. We may also limit the scope of a quantifier if a sub-formula does not contain the quantified variable. That is,

$$(\forall \overline{x}.F[\overline{x}] \vee G) \Longrightarrow (\forall \overline{x}.F[\overline{x}]) \vee G$$

when G does not contain x. We use a similar rule for existential quantifiers over disjunctions.

2) Skolemization: Similarly to first-order theorem provers, in our solver, existentially quantified variables are eliminated using *Skolemization*. A formula $\forall x.\ \exists y.\ \neg p(x) \vee q(x,y)$ is converted into the equisatisfiable formula $\forall x.\ \neg p(x) \vee q(x, f_y(x))$, where f_y is a fresh function symbol.

3) A conjunction of universally quantified formulas: After conversion to Negation Normal Form, miniscoping, and skolemization, the QBV formula is written as a conjunction of universally quantified formulas: $(\forall \overline{x}.\ \varphi_1[\overline{x}]) \wedge \ldots \wedge (\forall \overline{x}.\ \varphi_n[\overline{x}])$. This form is very similar to that used in first-order theorem provers. However, we do not require each $\varphi_i[\overline{x}]$ to be a clause.

4) Destructive Equality Resolution (DER): DER allows us to solve a negative equality literal by simply applying the following transformation:

$$(\forall x, \overline{y}.\ x \neq t \vee \varphi[x, \overline{y}]) \Longrightarrow (\forall \overline{y}.\ \varphi[t, \overline{y}])\ ,$$

where t does not contain x. For example, using DER, the formula $\forall x, y.\ x \neq f(y) \vee g(x,y) \leq 0$ is simplified to $\forall y.\ g(f(y), y) \leq 0$. DER is essentially an equality substitution rule. This becomes clear when we write the clause on the left-hand-side using an implication: $\forall x, \overline{y}.\ x = t \rightarrow \varphi[x, \overline{y}]$. It is straightforward to implement DER; a naive implementation eliminates a single variable at a time. In our experiments, we observed this naive implementation was a bottleneck in benchmarks where hundreds of variables could be eliminated. The natural solution is to eliminate as many variables simultaneously as possible. The only complication in this approach is that some of the variables being eliminated may *depend* on each other. We say a variable x *directly depends* on y in DER, when there is a literal $x \neq t[y]$. In general we are presented with a formula of the following form:

$$\forall x_1, \ldots, x_n, \overline{y}.\ x_1 \neq t_1 \vee \ldots \vee x_n \neq t_n \vee \varphi[x_1, \ldots, x_n, \overline{y}]\ ,$$

where each x_i may depend on variables x_j, $j \neq i$. First, we build a dependency graph G where the nodes are the variables x_i, and G contains an edge from x_i to x_j whenever x_j depends on x_i. Next, we perform a topological sort on G, and whenever a cycle is detected when visiting node x_i, we remove x_i from G and move $x_i \neq t_i$ to $\varphi[x_1, \ldots, x_n, \overline{y}]$. Finally, we use the variable order $x_{k_1}, \ldots, x_{k_m}$ $(m \leq n)$ produced by the topological sort to apply DER simultaneously. Let θ be a *substitution*, i.e., a mapping from variables to terms. Initially, θ is empty. For each variable x_{k_i} we first apply θ to t_{k_i} producing t'_{k_i}, and then update $\theta := \theta \cup \{x_{k_i} \mapsto t'_{k_i}\}$. After all variables x_{k_i} were processed, we apply the resulting substitution θ to $\varphi[x_1, \ldots, x_n, \overline{y}]$.

As a final remark, the applicability of DER can be increased using theory solvers. The idea is to rewrite inequalities of the form $t_1[x, \overline{y}] \neq t_2[x, \overline{y}]$, containing a universal variable x, into $x \neq t'[\overline{y}]$. This rewriting step is essentially equivalent to a theory solving step, where $t_1[x, \overline{y}] = t_2[x, \overline{y}]$ is solved for x. In the case of linear bit-vector equations, this can be achieved when the coefficient of x is odd [12].

5) Rewriting: The idea of using rewriting for performing equational reasoning is not new. It traces back to the work developed in the context of Knuth-Bendix completion [21]. The basic idea is to use unit clauses of the form $\forall \overline{x}.\ t[\overline{x}] = r[\overline{x}]$ as rewrite rules $t[\overline{x}] \rightsquigarrow r[\overline{x}]$, when $t[\overline{x}]$ is "bigger than" $r[\overline{x}]$. Any instance $t[\overline{s}]$ of $t[\overline{x}]$ is then replaced by $r[\overline{s}]$. For example, in the formula

$$(\forall x.\ f(x,a) = x)\ \wedge\ f(h(b),a) \geq 0,$$

the left conjunct can be used as the rewrite rule $f(x,a) \rightsquigarrow x$. Thus, the term $f(h(b),a) \geq 0$ can be simplified to $h(b) \geq 0$, producing the new formula

$$(\forall x.\ f(x,a) = x)\ \wedge\ h(b) \geq 0\ .$$

We observed that rewriting is quite effective in many QBVF benchmarks, in particular, in hardware fixpoint check problems. Our goal is to use rewriting as an incomplete simplification technique. So, we are not interested in computing critical pairs and generating a confluent rewrite system. First-order theorem provers use sophisticated *term orderings* to orient the equations $t[\overline{x}] = r[\overline{x}]$ (see, e.g., [17]). We found that any term ordering, where interpreted symbols (e.g., +, *) are considered "small", works for our purposes. This can be realised, for instance, using a Knuth-Bendix Ordering where the weight of interpreted symbols is set to zero. The basic idea of this heuristic is to replace uninterpreted symbols with interpreted ones. For example, using $f(x) \rightsquigarrow 2x + 1$, we can simplify

$f(a) - a$ to $2a + 1 - a$, and then apply a bit-vector rewriting rule and reduce it to $a + 1$.

6) Macros & Quasi-Macros: A *macro* is a unit clause of the form $\forall \overline{x}.\ f(\overline{x}) = t[\overline{x}]$, where f does not occur in t. Macros can be eliminated from QBV formulas by simply replacing any term of the form $f(\overline{r})$ with $t[\overline{r}]$. Any model for the resultant formula can be extended to a model that also satisfies $\forall \overline{x}.\ f(\overline{x}) = t[\overline{x}]$. For example, consider the formula

$$(\forall x.\ f(x) = x + a) \wedge f(b) > b\ .$$

After macro expansion, this formula is reduced to the equisatisfiable formula $b + a > b$. The interpretation $a \mapsto 1,\ b \mapsto 0$ is a model for this formula. This interpretation can be extended to

$$f(x) \mapsto x + 1,\ a \mapsto 1,\ b \mapsto 0\ ,$$

which is a model for the original formula. This particular way to represent models is described in more detail in section IV-B.

A *quasi-macro* is a unit clause of the form

$$\forall \overline{x}.f(t_1[\overline{x}], \ldots, t_m[\overline{x}]) = r[\overline{x}]\ ,$$

where f does not occur in $r[\overline{x}]$, $f(t_1[\overline{x}], \ldots, t_m[\overline{x}])$ contains all $\overline{x}$ variables, and the following system of equations can be solved for $x_1, \ldots, x_n$

$$y_1 = t_1[\overline{x}],\ \ldots,\ y_m = t_m[\overline{x}]\ ,$$

where $y_1, \ldots, y_m$ are new variables. A solution of this system is a substitution

$$\theta : x_1 \mapsto s_1[\overline{y}],\ \ldots,\ x_n \mapsto s_n[\overline{y}]\ .$$

We use the notation $\varphi \downarrow \theta$ to represent the application of the substitution θ to the formula φ. Then, the *quasi-macro* can be replaced with the *macro*

$$\forall \overline{y}.f(\overline{y}) = ite(\bigwedge_i y_i = t_i[\overline{x}], r[\overline{x}], f'(\overline{y})) \downarrow \theta$$

where f' is a fresh function symbol. Intuitively, the new formula is saying that when the arguments of f are of the form $t_i[\overline{x}]$, then the result should be $r[\overline{x}]$, otherwise the value is not specified. Now, the quasi-macro was transformed into a macro, the quantifier can be eliminated using macro expansion.

Example 2 (Quasi-Macro): $\forall x.f(x + 1, x - 1) = x$ is a quasi-macro, because the system $y_1 = x + 1,\ y_2 = x - 1$ can be solved for x. A possible solution is the substitution $\theta = \{x \mapsto y_1 - 1\}$. Thus, we can transform this quasi-macro into the macro:

$$\forall y_1, y_2.\ f(y_1, y_2) = ite(y_1 = x + 1 \wedge y_2 = x - 1,\ x,\ f'(y_1, y_2)) \downarrow \theta$$

After applying the substitution θ and simplifying the formula, we obtain

$$\forall y_1, y_2.\ f(y_1, y_2) = ite(y_2 = y_1 - 2,\ y_1 - 1,\ f'(y_1, y_2))\ .$$

In our experiments, we observed that the solvability condition is trivially satisfied in many instances, because all variables $\overline{x}$ are actual arguments of f. Assume that variable x_i is the k_i-th argument of f. Then, the substitution θ is of the form $\{x_1 \mapsto y_{k_1}, \ldots, x_n \mapsto y_{k_n}\}$. For example, in many benchmarks we found quasi-macros that are bigger versions of

$$\forall x_1, x_2.\ f(x_1,\ x_1 + x_2,\ x_2) = r[x_1, x_2]\ .$$

7) Function Argument Discrimination (FAD): We have observed that after applying DER the i-th argument of many function applications is always a bit-vector value such as: 0, 1, 2, etc. For any function symbol f and QBV formula φ, the following macro can be conjoined with φ while preserving satisfiability:

$$\forall x, \overline{y}.\ f(x, \overline{y}) = ite(x = v, f_v(\overline{y}), f'(x, \overline{y}))\ ,$$

where f_v and f' are fresh function symbols, and v is a bit-vector value. Now, suppose that the first argument of all f-applications are bit-vector values. The macro above will reduce $f(v', \overline{t})$ to $f_v(\overline{t})$ when $v = v'$, and $f'(v', \overline{t})$ otherwise. The transformation can be applied again to the f' applications if their first argument is again a bit-vector value.

Example 3 (FAD): Let φ be the formula

$$(\forall x.\ f(1, x, 0) \geq x) \wedge$$
$$f(0, a, 1) < f(1, b, 0)\ \wedge\ f(0, c, 1) = 0\ \wedge\ c = a.$$

Applying FAD twice (for the values 0 and 1) on the first argument of f, we obtain

$$(\forall x.\ f_1(x, 0) \geq x) \wedge$$
$$f_0(a, 1) < f_1(b, 0)\ \wedge\ f_0(c, 1) = 0\ \wedge\ c = a.$$

Applying FAD for the third argument of f_1 and f_0 results in

$$(\forall x.\ f_{1,0}(x) \geq x) \wedge$$
$$f_{0,1}(a) < f_{1,0}(b)\ \wedge\ f_{0,1}(c) = 0\ \wedge\ c = a.$$

Since FAD is based on macro definitions, the infrastructure used for constructing interpretations for macros may be used to build an interpretation for f based on the interpretations of $f_{1,0}$ and $f_{0,1}$.

8) Other simplifications: As many other SMT solvers for bit-vector theory ([6], [5], [2]), our QBVF solver implements several bit-vector specific rewriting/simplification rules such as: $a - a \Longrightarrow 0$. These rules have been proved to be very effective in solving quantifier-free bit-vector benchmarks, and this is also the case for the quantified case.

From now on, we assume that there is a procedure `Simplify` that, given a QBV formula φ, converts it into negation normal form, then applies miniscoping, skolemization, and the other simplifications described in this section up to saturation.

B. Model Checking Quantifiers

Given a structure M, it is useful to have a procedure `MC` that checks whether M satisfies a universally quantified formula φ or not. We say `MC` is a *model checking procedure*. Before we describe how `MC` can be constructed, let us take a look at how structures are encoded in our approach. We use BV to denote the structure that assigns the usual interpretation to the (interpreted) symbols of the bit-vector theory (e.g., $+$, $*$,

concat, etc). In our approach, the structures M are based on BV. We use $|BV|_n$ to denote the interpretation of the sort of bit-vectors of size n. With a small abuse of notation, the elements of $|BV|_n$ are $\{0_n, 1_n, \ldots, 2_n^{n-1}\}$. Again, where there is no confusion, we omit the subscript. The interpretation of an arbitrary term t in a structure M is denoted by $M[\![t]\!]$, and is defined in the standard way. We use $M\{x \mapsto v\}$ to denote a structure where the variable x is interpreted as the value v, and all other variables, function and predicate symbols have the same interpretation as in M. That is, $M\{x \mapsto v\}(x) = v$. For example, $BV\{x \mapsto 1\}[\![2 * x + 1]\!] = 3$. As usual, $M\{\overline{x} \mapsto \overline{v}\}$ denotes $M\{x_1 \mapsto v_1\}\{x_2 \mapsto v_2\} \ldots \{x_n \mapsto v_n\}$.

For each uninterpreted constant c that is a bit-vector of size n, the interpretation $M(c)$ is an element of $|BV|_n$. For each uninterpreted function (predicate) $f\colon n_1, \ldots, n_k \to n_r$ of arity k, the interpretation $M(f)$ is a term $t_f[x_1, \ldots, x_k]$, which contains only interpreted symbols and the free variables $x_1 : n_1, \ldots, x_k : n_k$. The interpretation $M(f)$ can be viewed as a *function definition*, where for all $\overline{v}$ in $|BV|_{n_1} \times \ldots \times |BV|_{n_k}$, $M(f)(\overline{v}) = BV\{\overline{x} \mapsto \overline{v}\}[\![t_f[\overline{x}]]\!]$.

Example 4 (Model representation): Let φ_a be the following formula:

$$(\forall x.\ \neg(x \geq 0) \vee f(x) < x) \wedge$$
$$(\forall x.\ \neg(x < 0) \vee f(x) > x + 1) \wedge$$
$$f(a) > b \ \wedge \ b > a + 1\ .$$

Then the interpretation

$$M_a := \{f(x) \mapsto \mathit{ite}(x \geq 0, x - 1, x + 3),\ a \mapsto -1,\ b \mapsto 1\}$$

is a model for φ_a. For instance, we have $M[\![f(a)]\!] = 2$.
Usually, SMT solvers represent the interpretation of unintepreted function symbols as finite *function graphs* (i.e., lookup tables). A function graph is an explicit representation that shows the value of the function for a finite (and relatively small) number of points. For example, let the function graph $\{0 \mapsto 1,\ 2 \mapsto 3,\ \mathit{else} \mapsto 4\}$ be the interpretation of the function symbol g. It states that the value of the function g at 0 is 1, at 2 it is 3, and for all other values it is 4. Any function graph can be encoded using ite terms. For example, the function graph above can be encoded as $g(x) \mapsto \mathit{ite}(x = 0, 1, \mathit{ite}(x = 2, 3, 4))$. Our approach for enconding interpretations is *symbolic* and potentially allows for an exponentially more succinct representation. For example, assuming f is a function from bit-vectors of size 32, the interpretation $f(x) \mapsto \mathit{ite}(x \geq 0, x - 1, x + 3)$ would correspond to a very large function graph.

When models are encoded in this fashion, it is straightforward to check whether a universally quantified formula $\forall \overline{x}.\ \varphi[\overline{x}]$ is satisfied by a structure M [13]. Let $\varphi^M[\overline{x}]$ be the formula obtained from $\varphi[\overline{x}]$ by replacing any term $f(\overline{r})$ with $M[\![f(\overline{r})]\!]$, for every uninterpreted function symbol f. A structure M satisfies $\forall \overline{x}.\ \varphi[\overline{x}]$ if and only if $\neg\varphi^M[\overline{s}]$ is unsatisfiable, where $\overline{s}$ is a tuple of fresh constant symbols.

Example 5: For instance, in Example 4, the structure M_a satisfies $\forall x.\ \neg(x \geq 0) \vee f(x) < x$ because

$$s \geq 0 \wedge \neg(\mathit{ite}(s \geq 0, s - 1, s + 3) < s)$$

is unsatisfiable. Let M_b be a structure identical to M_a in Example 4, but where the interpretation $M_b(f)$ of f is $x + 2$. M_b does not satisfy $\forall x.\ \neg(x \geq 0) \vee f(x) < x$ in φ_a because the formula $s \geq 0 \wedge \neg(s + 2 < s)$ is satisfiable, e.g., by $s \mapsto 0$. The assignment $s \mapsto 0$ is a *counter-example* for M_b being a model for φ_a.

The model-checking procedure `MC` expects two arguments: a universally quantified formula $\forall \overline{x}.\ \varphi[\overline{x}]$ and a structure M. It returns $\top$ if the structure satisfies $\forall \overline{x}.\ \varphi[\overline{x}]$, and a non-empty finite set V of counter-examples otherwise. Each counter-example is a tuple of bit-vector values $\overline{v}$ such that $M\{\overline{x} \mapsto \overline{v}\}[\![\varphi[\overline{x}]]\!]$ evaluates to *false*.

C. Template Based Model Finding

In principle, the verification and synthesis problems described in section III can be attacked by any SMT solver that supports universally quantified formulas, and that is capable of producing models. Unfortunately, to the best of our knowledge, no SMT solver supports complete treatment of universally quantified formulas, even if the variables range over finite domains such as bit-vectors. On satisfiable instances, they will often not terminate or give up. On some unsatisfiable instances, SMT solvers may terminate using techniques based on *heuristic-quantifier instantiation* [9].

It is not surprising that standard SMT solvers cannot handle these problems; the search space is simply too large. Synthesis tools based on automated reasoning try to constrain the search space using *templates*. For example, when searching for a ranking function, the synthesis tool may limit the search to functions that are linear combinations of the input. This simple idea immediately transfers to QBVF solvers. In the context of a QBVF solver, a template is just an expression $t[\overline{x}, \overline{c}]$ containing free variables $\overline{x}$, interpreted symbols, and fresh constants $\overline{c}$. Given a tuple of bit-vector values $\overline{v}$, we say $t[\overline{x}, \overline{v}]$ is an *instance* of the template $t[\overline{x}, \overline{c}]$. A template can also viewed as a *parametric* function definition. For example, the template $ax+b$, where a and b are fresh constants, may be used to guide the search for an interpretation for unary function symbols. The expressions $x + 1$ ($a \mapsto 1, b \mapsto 1$) and $2x$ ($a \mapsto 2, b \mapsto 0$) are instances of this template.

We say a *template binding* for a formula φ is a mapping from uninterpreted function (predicate) symbols f_i, occurring in φ, to templates $t_i[\overline{x}, \overline{c}]$. Conceptually, one template per uninterpreted symbol is enough. If we want to consider two different templates $t_1[\overline{x}, \overline{c_1}]$ and $t_2[\overline{x}, \overline{c_2}]$ for an uninterpreted symbol f, we can just combine them in a single template $t'[\overline{x}, (\overline{c_1}, \overline{c_2}, c)] \equiv \mathit{ite}(c = 1, t_1[\overline{x}, \overline{c_1}], t_2[\overline{x}, \overline{c_2}])$, where c is a new fresh constant. This approach can be extended to construct templates that are combinations of smaller "instructions" that can be combined to construct a template for the desired class of functions.

Without loss of generality, let us assume that φ contains only one uninterpreted function symbol f. So, a template based model finder is a procedure `TMF` that given a ground formula φ and a template binding $\mathtt{TB} = \{f \mapsto t[\overline{x}, \overline{c}]\}$, returns a structure M for φ s.t. the interpretation of f is $t[\overline{x}, \overline{v}]$ for

solver(φ, TB)
  φ := Simplify(φ)
  w.l.o.g. assume φ is of the form $\forall \overline{x}.\ \phi[\overline{x}]$
  ρ := HeuristicInst($\phi[\overline{x}]$)
  loop
    if SMT(ρ) = *unsat* **return** *unsat*
    M := TMF(ρ, TB)
    if $M = \bot$ **return** *unsat modulo* TB
    V := MC(φ, M)
    if $V = \top$ **return** (*sat*, M)
    $\rho := \rho \wedge \bigwedge_{\overline{v} \in V} \phi[\overline{v}]$

Fig. 1. QBVF solving algorithm.

some bit-vector tuple $\overline{v}$ if such a structure exists. TMF returns $\bot$ otherwise. Since we assume φ is a ground formula, a standard SMT solver can be used to implement TMF. We just need to check whether

$$\varphi \wedge \bigwedge_{f(\overline{r}) \in \varphi} f(\overline{r}) = t[\overline{r}, \overline{c}]$$

is satisfiable. If this is the case, the model produced by the SMT solver will assign values to the fresh constants $\overline{c}$ in the template $t[\overline{x}, \overline{c}]$. When TMF($\varphi$, TB) succeeds we say φ is satisfiable *modulo* TB.

Example 6 (Template Based Model Finding): Let φ be the formula

$$f(a_1) \geq 10 \wedge f(a_2) \geq 100 \wedge f(a_3) \geq 1000 \wedge$$
$$a_1 = 0 \wedge a_2 = 1 \wedge a_3 = 2$$

and the template binding TB be $\{f \mapsto c_1 x + c_2\}$. Then, the corresponding satisfiability query is:

$$f(a_1) \geq 10 \wedge f(a_2) \geq 100 \wedge f(a_3) \geq 1000 \wedge$$
$$a_1 = 0 \wedge a_2 = 1 \wedge a_3 = 2 \wedge$$
$$f(a_1) = c_1 a_1 + c_2 \wedge f(a_2) = c_1 a_2 + c_2 \wedge$$
$$f(a_3) = c_1 a_3 + c_2$$

The formula above is satisfiable, e.g., by the assignment $c_1 \mapsto 1$ and $c_2 \mapsto 1000$. Therefore, φ is satisfiable modulo TB.

D. Solver Architecture

The techniques described in this section can be combined to produce a simple and effective solver for non-trivial benchmarks. Figure 1 shows the algorithm used in our prototype. The solver implements a form of *counter-example guided refinement* where a failed *model-checking* step suggests new instances for the universally quantified formula. This method is also a variation of *model-based quantifier instantiation* [13] based on templates. The procedure SMT is an SMT solver for the quantifier-free bit-vector and uninterpreted function theory (QF_UFBV in SMT-LIB [1]). The procedure HeuristicInst($\phi[\overline{x}]$) creates an initial set of ground instances of $\phi[\overline{x}]$ using heuristic instantiation. Note that the formula ρ is monotonically increasing in size, so the procedures SMT and TMF can exploit incremental solving features available in state-of-the-art SMT solvers.

Theorem 2: The algorithm in Figure 1 is complete modulo the given template TB.

The algorithm in Figure 1 is complete for QBVF if TMF never fails, that is, M is never $\bot$. This can be accomplished using a template that simply covers *all* relevant functions: Let us assume w.l.o.g. that every function in φ has only one argument and it is a bit-vector of size 2^n. Then, using the template

$$ite(x = c_1, a_1, \ldots, ite(x = c_{2^n-1}, a_{2^n-1}, a_{2^n}) \ldots)$$

guarantees that TMF will never fail, where $c_1, \ldots c_{2^n-1}$, $a_1, \ldots$, a_{2^n} are the template parameters. Of course, it is impractical to use this template in practice. Therefore, in our implementation, we consider templates of increasing complexity. We essentially use an outer-loop that automatically increases the size of the templates whenever the inner-loop returns *unsat modulo* TB.

In many cases, using actual tuples of bit-vector values is not the best strategy for instantiating quantifers. For example, assume f is a function from bit-vectors of size 32 to bit-vectors of the same size in

$$(\forall x.\ f(x) \geq 0), \quad f(a) < 0\ .$$

To prove this formula to be unsatisfiable, we should instantiate the quantifier with a instead of the 2^{32} possible bit-vector values. Therefore, we use an approach similar to the one used in [13]. Given a tuple $(v_1, \ldots, v_n)$ in V, if there is a term t in ρ s.t. $M[\![t]\!] = v_i$, we use t instead of v_i to instantiate the quantifier. Of course, in practice, we may have several different t's to chose from. In this case we select the syntactically smallest one, and break ties non-deterministically.

E. Additional Techniques for Solving QBVF

Templates may be used to eliminate uninterpreted function (predicate) symbols from any QBV formula. The idea is to replace any function application $f_i(\overline{r})$ (ground or not) in a QBV formula φ with the template definition $t_i[\overline{r}, \overline{c}]$. The resultant formula φ' contains only uninterpreted constants and interpreted bit-vector operators. Therefore, *bit-blasting* can be used to encode φ' into QBF. This observation also suggests that template model finding is essentially approximating a NEXPTIME-complete problem (QBVF satisfiability) as a PSPACE-complete one (QBF satisfiability). Of course, the reduction is effective iff the size of the templates are polynomially bounded by the input formula size.

If the QBV formula is a conjunction of many universally quantified formulas, a more attractive approach is quantifier elimination using BDDs [3] or resolution and expansion [4]. Each universally quantified clause can be independently processed and the resultant formulas/clauses are combined. Another possibility is to apply this approach only to a selected subset of the universally quantified sub-formulas, and rely on the approach described in section IV-D for the remaining ones.

Finally, first-order resolution and subsumption can also be used to derive new implied QBV universally quantified clauses and to delete redundant ones.

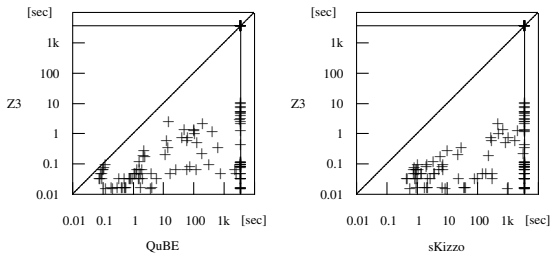


Fig. 2. Hardware fixpoint checks: QuBE & sKizzo vs. Z3

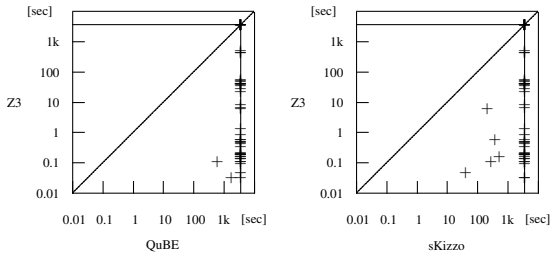


Fig. 3. Ranking function synthesis: QuBE & sKizzo vs. Z3

V. Experimental Results

To assess the efficacy of our method we present an evaluation of the performance of a preliminary QBVF solver based on the code-base of the Z3 SMT solver [10]. Our prototype first applies the simplifications described in section IV-A. It then iterates model checking and model finding as described in sections IV-B and IV-C. The benchmarks that we use for our performance comparison are derived from two sources: a) hardware fixpoint checks and b) software ranking function synthesis [8]. It is not trivial to compare our QBVF solver with other systems, since most SMT solvers do not perform well in benchmarks containing bit-vectors and quantifiers. In the past, QBF solvers have been used to attack these problems. We therefore compare to the state-of-the-art QBF solvers sKizzo [3] and QuBE [14].

Formulas in the first set exhibit the structure of fixpoint formulas described in section III. The circuits that we use as benchmarks are derived from a previous evaluation of VCEGAR [18][1] and were extracted using a customized version of the EBMC bounded Model Checker[2], which is able to produce fixpoint checks in QBVF and QBF form. In total, this benchmark set contains 131 files.

Our second set of benchmarks cannot be directly encoded in QBF because they contain uninterpreted function symbols. So, we decided to consider only ranking functions that are linear polynomials. By applying this template we can convert the problem to QBF as described in section IV-E. Thus, the problem here is to synthesise the coefficients for the polynomial. Further details, especially on the size of the coefficients, were described previously [8].

[1]These benchmarks are available at http://www.cprover.org/hardware/

[2]EBMC is available at http://www.cprover.org/ebmc/

All our benchmarks were extracted in two forms: in QBVF form (using SMT-LIB format) and in QBF form (using the QDIMACS format) and they were executed on a Windows HPC cluster of AMD Athlon 2 GHz machines with a time limit of 3600 seconds and a memory limit of 2 GB.

As indicated by Figure 2 our approach outperforms the QBF solvers on all instances, sometimes by up to five orders of magnitude and it solves almost all instances in the benchmark set (110 out of 131). Most of the benchmarks solved in this category (87 out of 110) are solved by our simplifications and rewriting rules only. In the remaining cases, the model refinement algorithm takes less then 10 iterations.

Figure 3 shows the results for the ranking function benchmark set. Again, our algorithm outperforms the QBF solvers by up to five orders of magnitude. The number of iterations required to find a model or prove non-existence of a model in these benchmarks is again very small: almost all instances require only one or two iterations and the maximum number of iterations is 9. Even though our algorithm exhibits similar speedups on both benchmark sets, the behaviour on the second set is quite different: None of the instances in this set is completely solved by the simplifications or rewriting rules. The model finding algorithm is required on each of them.

VI. Related Work

In practice it is often the case that uninterpreted functions are not strictly required. In this case, QBVFs can be flattened into either a propositional formula or a quantified Boolean formula (QBF). This is possible because bit-vector variables may be treated as a vector of Boolean variables. Operations on bit-vectors may be bit-blasted, but this approach increases the size of the formula considerably (e.g., quadratically for multipliers), and structural information is lost. In case of quantified formulas, universal quantifiers can be expanded since each is a quantification over a finite domain of values. This usually results in an exponential increase of the formula size and is therefore infeasible in practice. An alternative method is to flatten the QBV formula without expanding the quantifiers. This results in a QBF and off-the-shelf decision procedures (QBF solvers) like sKizzo [3], Quantor [4] or QuBE [14] may be employed to decide the formula. In practice, the performance of QBF solvers has proven to be problematic, however.

One of the potential issues resulting in bad performance may be the prenex clausal form of QBFs. It has thus been proposed to use non-prenex non-clausal form [11], [15]. This has been demonstrated to be beneficial on certain types of formulas, but all known decision procedures fail to exploit any form of word-level information.

A further problem with QBF solvers is that only few of them support certification, especially the construction of models for satisfiable instances. This is an absolute necessity for solvers employed in a synthesis context.

SMT QF_BV solvers. For some time now, SMT solvers for the quantifier-free fragment of bit-vector logic existed.

Usually, those solvers are based on a small set of word-level simplifications and subsequent flattening (bit-blasting) to propositional formulas. Some solvers (e.g., SWORD [29]), try to incorporate word-level information while solving the flattened formula. Some tools also have limited support for quantifiers (e.g. BAT [22]), but this is usually restricted to either a single quantifier or a single alternation of quantifiers which may be expanded at feasible cost. Most SMT QF_BV solvers support heuristic instantiation of quantifiers based on E-matching [9]. On some unsatisfiable instances, this may terminate with a conclusive result, but it is of course not a solution to the general problem. The method that we propose uses SMT solvers for the quantifier-free fragment to decide intermediate formulas and therefore represents an extension of SMT techniques to the more general QBV logic.

Synthesis tools. Finally, there is recent and active interest in using modern SMT solvers in the context of synthesis of inductive loop invariants [25] and synthesis of program fragments [19], such as sorting, matrix multiplication, decompression, graph, and bit-manipulating algorithms. These applications share a common trait in the way they use their underlying symbolic solver. They search a template *vocabulary* of instructions, that are composed as a model in a satisfying assignment. This approach was the main inspiration for the template based model finder described in section IV-C.

VII. Conclusion

Quantified bit-vector logic (QBV) is ideally suited as an interface between verification or synthesis tools and underlying decision procedures. Decision procedures for different fragments of this logic are required in virtually every verification or synthesis technique, making QBV one of the most practically relevant logics. We present a new approach to solving quantified bit-vector formulas based on a set of simplifications and rewrite rules, as well as a new model finding algorithm based on an iterative refinement scheme. Through an evaluation on benchmarks that stem from hardware and software applications, we are able to demonstrate that our approach is up to five orders of magnitude faster when compared to a popular approach of flattening the formula to QBF.

References

[1] C. Barrett, A. Stump, and C. Tinelli, "The Satisfiability Modulo Theories Library (SMT-LIB)," `www.SMT-LIB.org`, 2010.

[2] C. Barrett and C. Tinelli, "CVC3," in *Proc. of CAV*, ser. LNCS, no. 4590. Springer, 2007.

[3] M. Benedetti, "Evaluating QBFs via Symbolic Skolemization," in *Proc. of LPAR*, ser. LNCS, no. 3452. Springer, 2005.

[4] A. Biere, "Resolve and expand," in *Proc. of SAT'04 (Revised Selected Papers)*, ser. LNCS, no. 3542. Springer, 2005.

[5] R. Brummayer and A. Biere, "Boolector: An efficient SMT solver for bit-vectors and arrays," in *Proc. of TACAS*, ser. LNCS, no. 5505. Springer, 2009.

[6] R. Bruttomesso, A. Cimatti, A. Franzén, A. Griggio, and R. Sebastiani, "The MathSAT 4 SMT solver," in *CAV*, ser. LNCS, no. 5123. Springer, 2008.

[7] M. Colón, "Schema-guided synthesis of imperative programs by constraint solving," in *Proc. of Intl. Symp. on Logic Based Program Synthesis and Transformation*, ser. LNCS, no. 3573. Springer, 2005.

[8] B. Cook, D. Kroening, P. Rümmer, and C. M. Wintersteiger, "Ranking function synthesis for bit-vector relations," in *Proc. of TACAS*, ser. LNCS, no. 6015. Springer, 2010.

[9] L. de Moura and N. Bjørner, "Efficient e-matching for SMT solvers," in *Proc. of CADE*, ser. LNCS, no. 4603. Springer, 2007.

[10] ——, "Z3: An efficient SMT solver," in *Proc. of TACAS*, ser. LNCS, no. 4963. Springer, 2008.

[11] U. Egly, M. Seidl, and S. Woltran, "A solver for QBFs in negation normal form," *Constraints*, vol. 14, no. 1, 2009.

[12] V. Ganesh and D. L. Dill, "A decision procedure for bit-vectors and arrays," in *Proc. of CAV*, ser. LNCS, no. 4590. Springer, 2007.

[13] Y. Ge and L. de Moura, "Complete instantiation for quantified formulas in satisfiabiliby modulo theories," in *CAV*, ser. LNCS, no. 5643. Springer, 2009.

[14] E. Giunchiglia, M. Narizzano, and A. Tacchella, "QuBE++: An efficient QBF solver," in *Proc. of FMCAD*, ser. LNCS, no. 3312. Springer, 2004.

[15] A. Goultiaeva, V. Iverson, and F. Bacchus, "Beyond CNF: A circuit-based QBF solver," in *Proc. of SAT*, ser. LNCS, no. 5584. Springer, 2009.

[16] S. Gulwani, S. Srivastava, and R. Venkatesan, "Constraint-based invariant inference over predicate abstraction," in *Proc. of VMCAI*, ser. LNCS, no. 5403. Springer, 2009.

[17] J. Harrison, *Handbook of Practical Logic and Automated Reasoning*. Cambridge University Press, 2009.

[18] H. Jain, D. Kroening, N. Sharygina, and E. M. Clarke, "Word-level predicate-abstraction and refinement techniques for verifying RTL verilog," *IEEE Trans. on CAD of Int. Circuits and Systems*, vol. 27, no. 2, 2008.

[19] S. Jha, S. Gulwani, S. Seshia, and A. Tiwari, "Oracle-guided component-based program synthesis," in *Proc. of ICSE*. ACM, 2010.

[20] B. Jobstmann and R. Bloem, "Optimizations for LTL synthesis," in *Proc. of FMCAD*. IEEE, 2006.

[21] D. E. Knuth and P. B. Bendix, "Simple word problems in universal algebra." in *Proc. Conf. on Computational Problems in Abstract Algebra*. Pergamon Press, 1970.

[22] P. Manolios, S. K. Srinivasan, and D. Vroon, "BAT: The bit-level analysis tool," in *Proc. of CAV*, ser. LNCS, no. 4590. Springer, 2007.

[23] A. Pnueli and R. Rosner, "On the synthesis of a reactive module," in *Proc. of POPL*. ACM, 1989.

[24] A. Podelski and A. Rybalchenko, "A complete method for the synthesis of linear ranking functions," in *Proc. of VMCAI*, ser. LNCS, no. 2937. Springer, 2004.

[25] S. Srivastava and S. Gulwani, "Program verification using templates over predicate abstraction," in *Proc. of PLDI*. ACM, 2009.

[26] S. Srivastava, S. Gulwani, and J. S. Foster, "From program verification to program synthesis," in *Proc. of POPL*. ACM, 2010.

[27] S. Staber and R. Bloem, "Fault localization and correction with QBF," in *Proc. of SAT*, ser. LNCS, no. 4501. Springer, 2007.

[28] A. Turing, "Checking a large routine," in *Report of a Conference on High Speed Automatic Calculating Machines*, 1949.

[29] R. Wille, G. Fey, D. Große, S. Eggersglüß, and R. Drechsler, "SWORD: A SAT like prover using word level information," in *Proc. Intl. Conf. on Very Large Scale Integration of System-on-Chip*. IEEE, 2007.

Boosting Multi-Core Reachability Performance with Shared Hash Tables

Alfons Laarman, Jaco van de Pol, Michael Weber
{a.w.laarman,vdpol,michaelw}@cs.utwente.nl
Formal Methods and Tools, University of Twente, The Netherlands

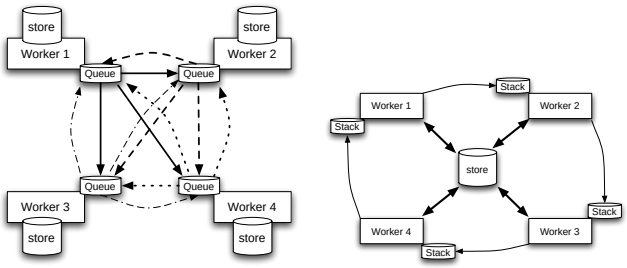
***Abstract*—This paper focuses on data structures for multi-core reachability, which is a key component in model checking algorithms and other verification methods. A cornerstone of an efficient solution is the storage of visited states. In related work, static partitioning of the state space was combined with thread-local storage. This solution leaves room for improvements. This paper presents a solution with a shared state storage. It is based on a lockless hash table implementation and scales better. The solution is specifically designed for the cache architecture of modern CPUs. Because model checking algorithms impose loose requirements on the hash table operations, their design can be streamlined substantially compared to related work on lockless hash tables. The resulting speedups are analyzed and compared with related tools. Our implementation outperforms two state-of-the-art multi-core model checkers, SPIN (presented at FMCAD 2006) and DiVinE, by a large margin, while placing fewer constraints on the load balancing and search algorithms.**

I. Introduction

Many verification problems are highly computational intensive tasks that can benefit from extra speedups. Considering the recent hardware trends, these speedups can only be delivered by exploiting the parallelism of the new multi-core CPUs.

Reachability, or full exploration of the state space, is a subtask of many verification problems [6], [8]. In model checking, reachability has in the past been parallelized using distributed systems [6]. With shared-memory systems, these algorithms can benefit from the low communication costs as has been demonstrated already [1]. In this paper, we show how the performance of state-of-the-art multi-core model checkers, like SPIN [13] and DiVinE [1], can be greatly improved using a carefully designed concurrent hash table as shared state storage.

Motivation: Holzmann and Bošnacki used a shared hash table with fine-grained locking in combination with the stack-slicing algorithm in their multi-core extension of the SPIN model checker [12], [13]. This shared storage enabled the parallelization of many of the model checking algorithms in SPIN: safety properties, partial order reduction and reachability. Barnat et al. implemented the same method in the DiVinE model checker [1]. They chose to implement the classic method of static state space partitioning, as used in distributed model checking [3]. They found the static partitioning method to scale better on the basis of experiments. The authors also mention that they were not able to develop a potentially better solution for shared state storage, namely the use of a lockless hash table. Thus it remains unknown whether reachability, based on shared state storage, can scale.



(a) Static partitioning (b) Stack slicing

Fig. 1. Different architectures for model checkers

TABLE I
Differences between architectures

Arch.	Sync. points	Pros / Cons
Fig. 1(a)	Queue	local (*cache efficient*) storage / *static* load balancing, *high* comm. costs, *limited* to BFS
Fig. 1(b)	Shared store, stack	low comm. costs / *specific* load balancing, limited to (pseudo) DFS
Shared store	Shared store, (queue)	low comm. costs, *flexible* load balancing, *flexible* exploration algorithm / scalability?

Using a shared state storage has further benefits. Fig. 1 shows the different architectures discussed thus far. Their differences are summarized in Table I and have been extensively discussed by Barnat et al. [3]. They also investigate a more general architecture with a shared storage and arbitrary load-balancing strategy (not necessarily stack-slicing). Such a solution is both simpler and more flexible, in the sense that it allows for more freedom in the choice of the exploration algorithm, including (pseudo) DFS, which enables fast searches for deadlocks and error states [20]. Holzmann already demonstrates this [12], but could not show desirable scalability of SPIN (as we will demonstrate). The *stack-slicing algorithm* [12], is a specific case of load balancing that requires DFS. In fact, any well-investigated load-balancing solution [21] can be used and tuned to the specific environment, for example, to support heterogeneous systems or BFS exploration. Inggs and Barringer use a *lossy* shared hash table [14], resulting in reasonable speedups at the cost of precision (states can potentially be revisited), but give little details on the implementation.

Contribution: We present a data structure for efficient concurrent storage of states. This enables scaling parallel implementations of reachability for many desirable exploration algorithms. The precise needs which parallel model checking algorithms impose on shared state storage are evaluated and a

fitting solution is proposed given the identified requirements. Experiments show that our implementation of the shared storage scales significantly better than an implementation using static partitioning, but also beats state-of-the-art model checkers. By analysis, we show that our design will scale beyond current state-of-the-art multi-core processors. The experiments also contribute to a better understanding of the performance of the latest versions of SPIN and DiVinE.

Overview: Section II presents background on reachability, load balancing, hashing, parallel algorithms and multi-core systems. Section III presents the lockless hash table, which we designed for shared state storage. But only after we evaluated the requirements that fast parallel algorithms impose on such a shared storage. In Section IV, the performance is evaluated against that of DiVinE 2 [2] and SPIN. A fair comparison can be made between the three model checkers on the basis of a set of models from the BEEM database which report the same number of states for both SPIN and DiVinE. We end this paper by putting the results we obtained into context, and an outlook on future work (Section V).

II. Preliminaries

Reachability in Model Checking: In model checking, a computational model of the system under verification (hardware or software) is constructed, which is then be used to compute all possible states of the system. The exploration of all states can be done symbolically, e.g., using *binary decision diagrams* (BDDs) to represent sets of states, or by enumerating and explicitly storing all states. While symbolic methods are attractive for a certain set of models, they are not a silver bullet: due to *BDD explosion*, sometimes plain enumerative methods are faster. In this paper, we focus on enumerative model checking.

Enumerative reachability analysis can be used to check for deadlocks and invariants and also to store the whole state space and verify multiple properties of the system at once. Reachability is an exhaustive search through the state space. The algorithm calls for each state the *next-state* function to obtain its successors until no new states are found (Alg. 1). We use an *open set* T, which can be implemented as a stack or queue, depending on the preferred exploration order: depth or breadth-first. The initial state s_0 is obtained from the model and added to T. In the loop starting on Line 1, a state is taken from T, its successors are computed using the model (Line 3) and each new successor state is put into T again for later exploration. To determine which state is new, a *closed set* V is used. V can be implemented with a hash table.

Possible ways to parallelize Alg. 1 have been discussed in the introduction. A common denominator of all these approaches is that the strict BFS or DFS order of the sequential algorithm is sacrificed in favor of *thread-local* open sets (fewer contention points). When using a shared state storage (in a general setup or with stack-slicing), a thread-safe set V is required, which will be discussed in the following section.

Load Balancing: A naive parallelization of reachability can be realized as follows: perform a depth-limited sequential BFS exploration and hand off the found states to several threads

```
   Data: Sequence T = {s_0}, Set V = ∅
1  while state ← T.get() do
2      count ← 0;
3      for succ in next-state(state) do
4          count ← count + 1;
5          if V.find-or-put(succ) then
6              T.put(succ);
7      if 0 == count ... report deadlock ...
```

Algorithm 1: Reachability analysis

that start executing Alg. 1 ($T = \{$*part of BFS exploration*$\}$ and V is shared). This is called *static load balancing*. For many models this will work due to common diamond-shaped state spaces. However, models with synchronization points or strict phase structure sometimes exhibit helix-shaped state spaces. Hence, threads run out of work when they reach a converge point in their exploration. A well-known problem that behaves like this is the *Towers of Hanoi* puzzle; when the smallest disk is on top of the tower only one move is possible.

Sanders [21] describes *dynamic load balancing* in terms of a problem P, a (reduction) operation $work$ and a $split$ operation. P_{root} is the initial problem. Sequential execution takes $T_{\text{seq}} = T(P_{\text{root}})$ time units. A problem is (partly) solved when calling $work(P, t)$, which takes $min(t, T(P))$ units of time. For parallel reachability, $work(P, t)$ is Alg. 1, where t has to be added as an extra input that limits the number of iterations of the *while* loop on Line 1 (and $P_{\text{root}} = T = \{s_0\}$). When threads become idle, they can poll others for work. The receiver will then split its own problem instance ($split(P) = \{P_1, P_2\}$, $T(P) = T(P_1) + T(P_2)$) and send one of the results to the polling thread.

Parallel architectures: We consider multi-core x86 server and desktop systems. These systems can process a large number of instructions per second, but have a relatively low memory bandwidth. Multiple levels of cache are used to continuously feed the cores with data. Some of these caches are shared among multiple cores (often L2) and others are local (L1), depending on the architecture of the CPU and number of CPUs. The cache coherence protocol ensures that each core in each CPU has a global view of the memory. It transfers blocks of memory to local caches and synchronizes them if a local block is modified by other cores. Therefore, if independent writes are performed on subsequent memory locations (on the same cache line), a problem known as *cache line sharing* (*false sharing*) occurs, causing gratuitous synchronization and overhead.

The cache coherence protocol cannot be preempted. To efficiently program these machines, few options are left. One way is to completely partition the input [3], thus ensuring per-core memory locality at the cost of increased inter-die communication. An improvement of this approach is to pipeline the communication using ring buffers, this allows prefetching (explicit or hardwired). This scheme was explored, e.g., by Monagan and Pearce [17]. The last alternative is to minimize the *memory working set* of the algorithm [19]. We define the memory working set as the number of different memory

locations that the algorithm updates in the time window that these usually stay in local cache. A small working set minimizes coherence overhead.

Locks: It is common to ensure mutual exclusion for a critical section of code by locks. However, for resources with high contention, locks become infeasible. *Lock proliferation* improves on this by creating more locks on smaller resources. *Region locking* is an example of this, where a data structure is split into separately locked regions based on memory locations. However, this method is still infeasible for computational tasks with very high throughput. This is caused by the fact that the lock itself introduces another synchronization point; and synchronization between processor cores takes time.

Lockless Algorithms: For high-throughput systems, *lock-free algorithms* (without mutual exclusion) are preferred. Lock-free algorithms guarantee system-wide progress, i.e., always *some* thread can continue. If an algorithm does not strictly provide progress guarantees (only statistically), but otherwise avoids explicit locks by the same techniques as used in lock-free solutions, it is called *lockless*. Lockless algorithms often have considerably simpler implementations, at no performance penalty. Lastly, *Wait-free algorithms* guarantee per-thread progress, i.e., *all* threads can continue.

Many modern CPUs implement a *Compare & Swap* operation (CAS) which ensures atomic memory modification while at the same time preserving data consistency if used in the correct manner. This can be done by reading the value from memory, performing the desired computation on it and writing the result back using CAS (Alg. 2). If the latter returns true, the modification succeeded, if not, the computation needs to be redone with the new value, or some other form of collision resolution should be applied.

Pre: $word \neq$ **null**
Post:$(*word_{pre} = testval \Rightarrow *word_{post} = newval) \wedge$
$(*word_{pre} \neq testval \Rightarrow *word_{post} = *word_{pre})$
atomic bool CAS(int *word, int testval, int newval)

Algorithm 2: "Compare&Swap" specification

Lockless algorithms can achieve a high level of concurrency. However, an instruction like CAS easily costs 100–1000 instruction cycles depending on the CPU architecture. Thus, abundant use defies its purpose.

Quantifying Parallelism: Parallelism is usually quantified by normalizing the performance gain with regard to a sequential run (*speedup*): $S = T_{seq}/T_{par}$. Linear speedups grow proportional to the number of cores and indicate that an algorithm scales well. Ideal speedup is achieved when $S \geq N$. For a fair comparison of scalability, it is important to use the fastest tool for T_{seq}, or speedups will not be comparable, since better optimized code is harder to scale (e.g., [13]).

Hashing: A well-studied method for storing and retrieving data with amortized time complexity $O(1)$ is *hashing* [16]. A hash function h is applied to the data, yielding an index in an array of *buckets* that contain the data or a pointer to the data. Since the domain of data values is usually unknown and much larger than the image of h, hash collisions occur when $h(D_1) = h(D_2)$, with $D_1 \neq D_2$. Structurally, collisions can be resolved either by inserting lists in the buckets (*chaining*) or by probing subsequent buckets (*open addressing*). Algorithmically, there is a wealth of options to maintain the "chains" and calculate subsequent buckets [9]. The right choice depends entirely on the requirements dictated by the algorithms that use the hash table.

III. A Lockless Hash Table

In principle, Alg. 1 seems easy to parallelize, in practice it is difficult to do this efficiently because of its memory intensive behavior, which becomes more obvious when looking at the implementation of set V. In this section, we present an overview of the options in hash table design. There is no silver bullet design and individual design options should be chosen carefully considering the requirements stipulated by the use of the hash table. Therefore, we evaluate the demands that the parallel model checking algorithms place on the state storage solution. We also mention additional requirements stemming from the targeted hardware and software systems. Finally, we present a specific hash table design.

A. Requirements on the State Storage

Our goal is to realize an efficient shared state storage for parallel model checking algorithms. Traditional hash tables associate a piece of data to a unique key in the table. In model checking, we only need to store and retrieve *state vectors*, therefore the key is the state vector itself. Henceforth, we will simply refer to it as *data*. Our specific model checker implementation introduces additional requirements, discussed later. First, we list the definite requirements on the state storage:

- The storage needs only one operation: `find-or-put`. This operation inserts the state vector if it is not found or yields a positive answer without side effects. We require `find-or-put` to be concurrently executable to allow sharing the storage among the different threads. Other operations are not necessary for reachability algorithms, since the state space is growing monotonically. By exploiting this feature we can simplify the algorithms, thus lowering the strain on memory, and avoiding cache line sharing. Our choice is in sharp contrast to standard literature on concurrent hash tables, which often favors a complete solution, which is optimized for more general access patterns [7], [19].
- The storage should not require continual memory allocation, for the obvious reasons that this behavior would increase the *memory working set*.
- The use of pointers on a per-state basis should be avoided. Pointers take a considerable amount of memory when large state spaces are explored (more than 10^8 states are easily reachable with today's model checkers), especially on 64-bit machines. In addition, pointers increase the memory working set.
- The time efficiency of `find-or-put` should scale with the number of processes executing it in parallel. Ideally,

the individual operations should — on average — not be slowed down by other operations executing at the same time, thus ensuring close-to linear speedup. Many hash table algorithms have a large memory working set due to their probing behavior or reordering behavior upon insertions. They suffer performance degradation in high throughput situations as is the case for us.

Specifically, we do not require the state storage to be resizable. The available memory on a system can safely be claimed for the table, because the largest part will be used for it eventually anyway. In sequential operation and especially in presence of a *delete* operation (shrinking tables), one would consider resizing for the obvious reason that it improves locality and thus cache hits. In a concurrent setting, however, these cache hits have the opposite effect of causing the earlier described cache line sharing among CPUs. We experimented with lockless and concurrent resizing mechanisms and observed large decreases in performance.

Furthermore, the design of the LTSmin tool [5], which we extended with a multi-core reachability, also introduces some specific requirements:

- The storage data consists only of integer arrays or vectors of known and fixed length. This is the encoding format for state vectors employed by our language front-ends.
- The storage is targeted at common x86 architectures, using only the available (atomic) instructions.

While the compatibility with the x86 architecture allows for concrete analysis, the applicability of our design is not limited to it. Lessons learned here are transferable to other architectures with similar memory hierarchy and atomic operations.

B. Hash Table Design

We determined that a low memory working set is one of the key factors to achieve maximum scalability. Also, we opt for simplicity whenever the requirements allow for it. From experience we know that complexity of a solution arises automatically when introducing concurrency. These considerations led us to the following design choices:

- **Open addressing**, since the alternative *chaining* hash table design would incur in-operation memory allocation or pre-allocation at different addresses, both leading to a larger memory working set.
- **Walking-the-line** is the name we gave to *linear probing* on a cache line, followed by *double hashing* (also employed elsewhere [7], [11]). Linear probing allows a core to benefit fully from a loaded cache line, while double hashing realizes better distribution.
- **Separated data** (vectors) in an indexed *data array* (of size $buckets \times |vector|$) ensures that the *bucket array* stays short[1] and subsequent probes can be cached.
- **Hash memoization** speeds up probing, by storing the hash (or part of it) in a bucket. This avoids expensive lookups in the data array as much as possible [7].
- **Lockless** operation on the bucket array using a dedicated value to indicate unused buckets. One bit of the hash can be used to indicate whether the vector was already written to the data array or whether writing is still in progress [7].
- **Compare-and-swap** is used as an atomic primitive on the buckets, which are precisely in either of the following distinguishable states: *empty*, *being written* and *complete*.

[1]E.g., 1 GB for a 32-bit memoized hash and 2^{28} buckets

C. Hash Table Operations

Alg. 3 shows the `find-or-put` operation. Buckets are represented by the $Bucket$ array, the separate data by the $Data$ array and hash functions used for double hashing by $\texttt{hash}_i$. Probing continues (Line 4) until either a free bucket is found for insertion (Line 8–10), or the data is found to be in the hash table (Line 14). Too many probes indicate a table size mismatch, which simply causes the application to abort. The *for* loop on Line 5 handles the walking-the-line probing behavior (Alg. 4). The other code inside this loop handles the synchronization among threads. We explain this part of the algorithm now in detail.

Buckets store memoized hashes and the *write status* bit of the data in the $Data$ array. The possible values of the buckets are thus: EMPTY, $\langle h, \text{WRITE}\rangle$ and $\langle h, \text{DONE}\rangle$, where h is the memoized hash. If an empty bucket is encountered on a probe sequence, the algorithm tries to claim it by atomically writing $\langle h, \text{WRITE}\rangle$ to it (Line 7). After finishing the writing of the data, $\langle h, \text{DONE}\rangle$ is written to the bucket (Line 9). Non-empty buckets prompt the algorithm to compare the memoized hashes (Line 11). Only if they match, the value in the data array is compared with the vector (Line 13).

```
   Data: size, Bucket[size], Data[size]
   input : vector
   output: seen
 1 count ← 1;
 2 h ← hash_count(vector);
 3 index ← h mod size;
 4 while count < THRESHOLD do
 5     for i in walkTheLineFrom(index) do
 6         if EMPTY = Bucket[i] then
 7             if CAS(Bucket[i], EMPTY, ⟨h, WRITE⟩) then
 8                 Data[i] ← vector;
 9                 Bucket[i] ← ⟨h, DONE⟩;
10                 return false;
11         if ⟨h, −⟩ = Bucket[i] then
12             while ⟨−, WRITE⟩ = Bucket[i] do ..wait.. done
13             if Data[i] = vector then
14                 return true;
15     count ← count + 1;
16     index ← hash_count(vector) mod size;
```

Algorithm 3: The `find-or-put` algorithm

Several aspects of the algorithm guarantee correct lockless operation:

Data: *cache_line_size*, *Walk*[*cache_line_size*]
input : *index*
output: *Walk*[*cache_line_size*]
1 *start* ← ⌊*index*/*cache_line_size*⌋ × *cache_line_size*;
2 **for** *i* ← *0* **to** *cache_line_size* − 1 **do**
3 ⌊ *Walk*[*i*] ← *start* + (*index* + *i*) mod *cache_line_size*;

Algorithm 4: Walking the (cache) line

- Whenever a write started for a hash value, the state of the bucket can never become empty again, nor can it be used for any other hash value. This ensures that the probe sequence remains deterministic and cannot be interrupted.
- The CAS operation on Line 7 ensures that only one thread can claim an empty bucket, marking it as non-empty with the hash value to memoize and with state WRITE.
- The *while* loop on Line 12 waits until the write to the data array has been completed.

Critical synchronization between threads occurs when multiple threads try to write to an empty bucket. The CAS operation ensures that only one will succeed. The others carry on in their probing sequence, either finding another empty bucket or finding the state vector in another bucket. This design can be seen as a lock on the lowest possible level of granularity (individual buckets), but without a true locking structure and associated additional costs. The algorithm implements the "lock" as *while* loop, which resembles a spinlock (Line 12). Although it could be argued that this algorithm is therefore not lock-free, it is possible to ensure local progress in the case that the "blocking" thread dies or hangs (making the algorithm wait-free). Wait-freeness is commonly achieved by making each thread fulfil local invariants, whenever they are not (yet) met by other threads [10]. Our measurements show, however, that under normal operation the loop on Line 12 is rarely hit due to the preceding hash memoization check (Line 11). Thus, we took the pragmatic choice of keeping the implementation as simple as possible.

Our implementation of the described algorithm requires exact guarantees from the underlying memory model. Reordering of operations by compilers and processors needs to be avoided across the synchronization points, otherwise the implementation becomes incorrect. It is, for example, a common optimization to execute the body of an *if* statement before the actual branching instruction. Such a speculative execution would keep the processor pipeline busy, but would be a disastrous reordering when applied to Line 7 and Line 8: the actual writing of the data would happen before the bucket is marked as full, allowing other threads to write to the same bucket. Likewise, reordering Line 8 and Line 9 would prematurely indicate that writing the data has completed.

Unfortunately, the ANSI C99 standard does not state any requirements on the memory model. The implementation would depend on the combination of CPU architecture and compiler. Our implementation uses the GNU gcc compiler for 64-bit x86 target platforms. A gcc built-in is used for the CAS operation and reads and writes from and to buckets are marked volatile.

Alg. 3 was modeled in PROMELA and checked for deadlocks with SPIN. One bug concerning the combination of write bit and memoized hash was found and corrected.

IV. Experiments

A. Methodology

We implemented the hash table of the previous section in our own model checking toolset LTSmin, which we discuss further in the following section. For our experiments, we reuse not only the input models, but also the `next-state` *implementation* of DiVinE 2.2. Therefore, a fair comparison with DiVinE 2.2 can be made. Furthermore, we performed experiments with the latest multi-core capable version of the model checker SPIN 5.2.4 [13] (DiVinE models were mechanically translated to SPIN's PROMELA input language). For our experiments, we chose full state space exploration via reachability as load generator for our state storage. Reachability exhibits similar access patterns as more complex verification algorithms, but reduces the code footprint and therefore potential pollution of our measurements with noise.

All model checkers were configured for maximum performance. For all tools, we compiled models to C with high optimization settings (`-O3`) (DiVinE also contains a model interpreter). SPIN's models were compiled with the following flags: `-O3 -DNOCOMP -DNOFAIR -DNOREDUCE -DNOBOUNDCHECK -DNOCOLLAPSE -DNCORE=`N `-DSAFETY -DMEMLIM=100000`; To run the models we used the options: `-m10000000 -c0 -n -w28`.

We performed our experiments on AMD Opteron 8356 16-core servers with 64 GB RAM, running a patched Linux 2.6.32 kernel.[2] All tools were compiled using gcc 4.4 in 64-bit mode with maximal compiler optimizations (`-O3`).

A total of 31 models from the BEEM database [18] have been used in the experiments (we filtered out models which were too small to be interesting, or too big to fit into the available memory). Every run was repeated at least four times, to exclude any accidental fluctuation in the measurements. Special care has been taken to keep all the parameters across the different model checkers the same. Especially SPIN provides a rich set of options with which models can be tuned to perform optimal. Using these parameters on a per-model basis could give faster results than presented here. It would, however, say little about the scalability of the core algorithms. Therefore, we decided to leave all parameters the same for all the models. We avoid resizing of the state storage in all cases by increasing the initial hash table size to accommodate 2^{28} states (enough for all benchmarked input models).

B. Results

Figure 2 shows the run times of only three models for all model checkers. We observe that DiVinE is the fastest model

[2]Experiments showed large regressions in scalability on newer 64-bit Linux kernels (degrading runtimes with 10+ cores). Despite being undetected since at least version 2.6.20 (released in 2007!), they were easily exhibited by our model checker. With a repeatable test case, the Linux developers quickly provided a patch: https://bugzilla.kernel.org/show_bug.cgi?id=15618

checker for sequential reachability. Since the last published comparison between DiVinE and SPIN [1], DiVinE has been improved with a model compiler. SPIN is only slightly slower than DiVinE and shows the same linear curve but with a gentler slope. We suspect that the gradual performance gains are caused by the cost of the inter-thread communication (see Table I).

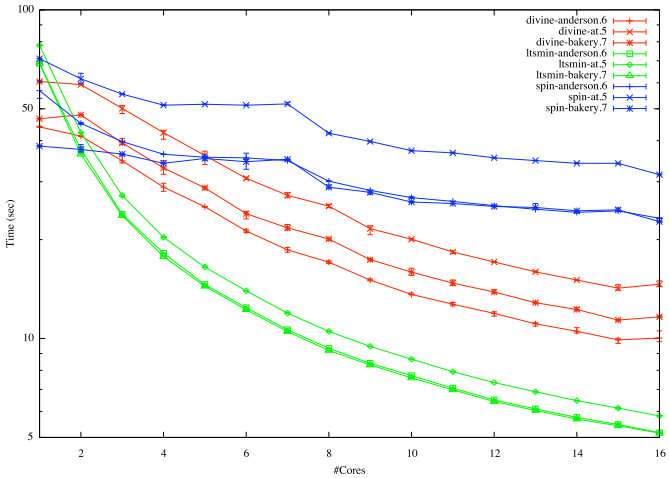


Fig. 2. (Log-scale) Runtimes in SPIN, LTSmin and DiVinE 2 (3 models)

LTSmin is slower in the sequential cases. We verified that the allocation-less hash table design causes this behavior; with smaller hash table sizes the sequential runtimes match those of DiVinE. We did not bother optimizing these results, because with two cores, LTSmin is already at least as fast as DiVinE.

Fig. 7, 8 and 9 show the speedups measured with LTSmin[3], DiVinE and SPIN (note that we normalize with T_{seq} of DiVinE, the fastest sequential tool). On 16 cores, LTSmin shows a two-fold improvement over DiVinE and a four-fold improvement over SPIN. We attribute the difference in scalability for DiVinE to the extra synchronization points needed for the inter-process communication by DiVinE. Recall that the model checker uses static state space partitioning, hence most successor states are enqueued at other cores than the one which generated them. Another disadvantage of DiVinE is its use of a management thread, which causes the regression at 8 and 16 cores.

SPIN shows inferior scalability even though it uses (like LTSmin) a shared hash table. SPIN also balances load based on *stack slicing*. We can only guess that the locking mechanism used in SPIN's hash table (region locking) are not as efficient as our lockless hash table. However, in LTSmin we obtained far better results even with the slower `pthread` locks. It might also be that stack slicing does not have a consistent granularity, because it uses the (irregular) search depth as a time unit (using the terms from Sec. II: $T(work(P_0, depth)) \gg T(work(P_1, depth))$).

Remark. A potential reason for the limited scalability of SPIN could be a memory bandwidth bottleneck. We tested this hypothesis by enabling SPIN's smaller, *collapsed* state vectors (`-DCOLLAPSE`). We carried out a full SPIN benchmark run

[3]Additional figures and more detail can be found in extended report [15].

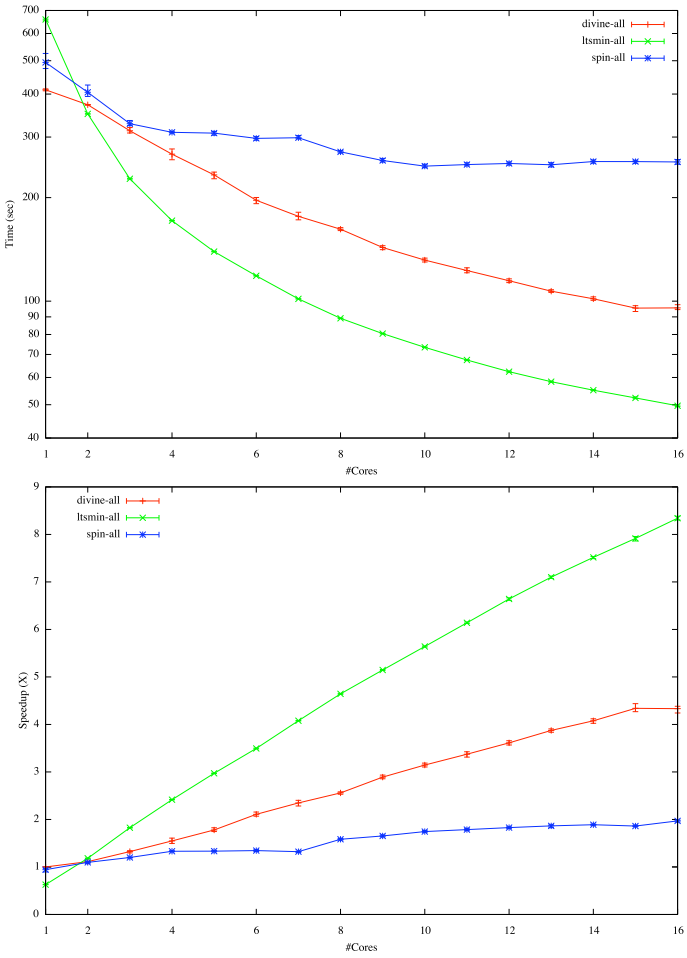


Fig. 3. Total runtime/speedup of SPIN, DiVinE 2.2 and LTSmin

with collapsing enabled and saw little improvement compared to the speedup results without `COLLAPSE`.[4] These results are consistent with the observation that LTSmin is faster, despite generally producing larger state vectors than both, SPIN and DiVinE (Table II): in LTSmin, each state variable gets 32-bit aligned (for API reasons, not performance).

Fig. 3 shows the total times and average speedups over all models and for all model checkers. Nineteen models could only be used because only for those, all tools report similar state counts (less than 20% difference; recall that for SPIN, models are translated from DVE to PROMELA).

C. Shared Storage Parameters

To verify our claims about the hash table design, we collected internal measurements and performed synthetic benchmarks for stress testing. First, we measured how often the write "lock" was hit. Fig. 4 plots the lock hits against the number of cores for several different sized models. For readability, only the worst-performing, and thus most interesting, models were chosen. Even then, the number of lock hits is a very small fraction of the number of `find-or-put` calls (equal to the number of transitions, typically in the hundreds of millions).

[4]http://fmt.cs.utwente.nl/~laarman/spin/

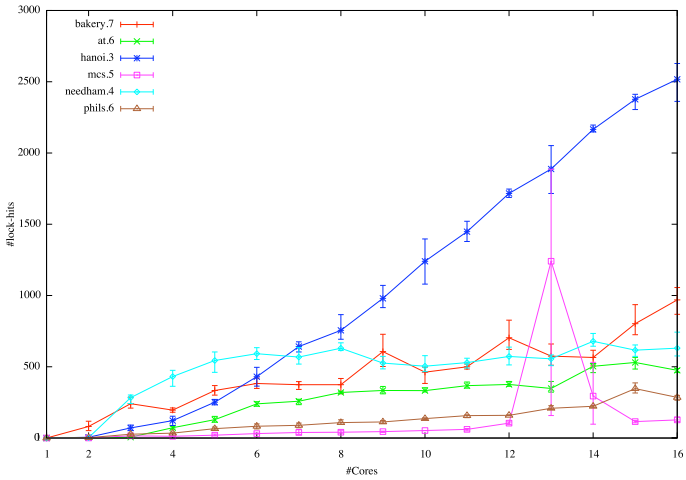


Fig. 4. Counting how often the algorithm "locks"

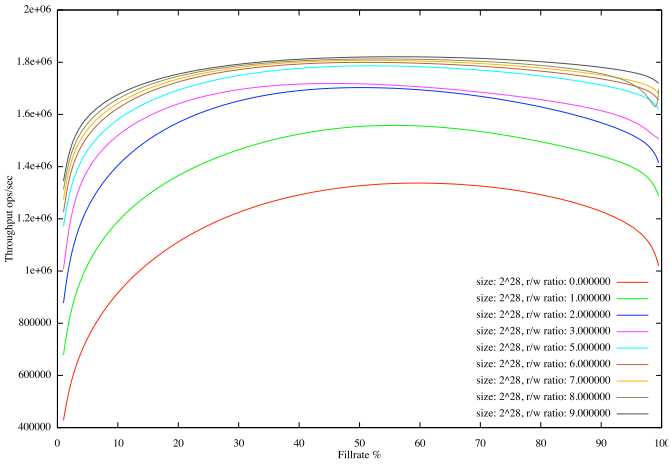


Fig. 5. Effect of fill rate and r/w-ratio on average throughput

We measured how the average throughput of Alg. 3 (number of `find-or-put` calls) is affected by the table *fill rate*, the table size and the read/write ratio. Fig. 5 illustrates the effects of different read/write ratios on the hash table using synthetic input data. The average throughput remains largely unaffected by a high fill rate, even up to 95 % (as for Fig. 6 in the Appendix, which plots the same lines for different table sizes). We conclude that the asymptotic time complexity of open-addressing hash tables poses little real problems in practice. However, an observable side effect of *oversized hash tables* is lower throughput for low fill rates due to increased cache misses. Our hash table design amplifies this effect because it uses a pre-allocated data array and no pointers. This explains the lower sequential performance of LTSmin.

We also measured the effect of varying the state vector size and did not find any noticeable change in the speedup behavior (except for the expected lower throughput due to higher data movement). This shows that hash memoization and a separate data array perform well. Walking-the-line probing shows better performance and scalability than double hashing alone, due to cache effects. Although slower on average, walking-the-line followed by double hashing beats simple linear probing at fill-rates above 95 % (in particular, on slower memory subsystems), because it leads to better distribution and thus fewer probes.

V. Discussion and Conclusions

We designed a hash table suitable for application in reachability analysis. We implemented it as part of a model checker together with different exploration algorithms (pseudo BFS and pseudo DFS) and explicit load-balancing. We demonstrated the efficiency of the complete solution by comparing the absolute speedups to SPIN 5.2.4 and DiVinE 2.2, both leading tools in this field. We claim two times better scalability than DiVinE and four times better than SPIN *on average* (Fig. 3), with individual results far exceeding these numbers. We also investigated the behavior of the hash table under different fill rates and found it to live up to the imposed requirements.

Limitations: Without the use of pointers the current design cannot easily cope with variably sized state vectors. In our model checker, this does not pose a problem because states are always represented by a vector of a static length. Our model checker LTSmin[5] can handle different front-ends. It connects to DiVinE-cluster, DiVinE 2.2, PROMELA (via NIPSVM [22]), mCRL, mCRL2 and ETF (internal symbolic representation of state spaces). Some of these input languages require variably sized vectors (NIPS). We solve this by an initial exploration which continues until a vector of stable size is found, and aborts when none can be found up to a fixed bound. So far, this limitation did not pose a problem.

For LTSmin, the results in the sequential case turn out to be around 20% slower than DiVinE 2.2. One of the culprits for this performance loss are the already mentioned suboptimal utilization of cache effects for small models (we verified that larger models suffer much less from this effect). Embracing pointers and allocation could be a potential remedy, however, it is unclear whether such a solution still scales when it actually matters (i.e., for large models).

Further performance is lost in an extra level of indirection (function calls) due to the design of LTSmin, which strictly separates the language front-end from the exploration algorithms. We are willing to pay this price in exchange for the increased modularity of our tool.

Discussion: We make several observations:

- We provide evidence that centralized state storage can be made to scale at least as well as static state space partitioning, contrary to prior belief [3].
- We also show that scalability is not as dependent on long state vectors and transition delays as earlier thought [12]. In fact, we argue that a scaling implementation performs better with smaller state vectors, because the number of operations performed per loaded byte is higher, thus closer to the strengths of modern multi-core systems.
- Shared state storage is also more flexible [3], for example allowing pseudo DFS (like the stack-slicing algorithm) and fast deadlock/invariant searches [20]. Moreover, it facilitates explicit load balancing algorithms, enabling the

[5]http://fmt.cs.utwente.nl/tools/ltsmin/

exploitation of *heterogeneous systems*. From preliminary experiments with load balancing we conjecture that overhead is negligible compared to static load balancing.

- Performance-critical parallel software needs adaptation to modern architectures (steep memory hierarchies). The performance difference between DiVinE, SPIN and LTSmin is an indication. DiVinE uses an architecture which is directly derived from distributed model checking and the goal of SPIN was for "these new algorithms [...] to interfere as little as possible with the existing algorithms for the verification of safety and liveness properties" [12]. With LTSmin, we had the opportunity to tune our design to the architecture of our target machines, with excellent pay-off. We noticed that avoiding cache line sharing and keeping a simple design was instrumental in the outcome.
- Holzmann conjectured that optimized sequential code does not scale well [13]. In contrast, our parallel implementation is faster in absolute numbers and also exhibits excellent scalability. We suspect that the (entirely commendable) design choice of SPIN's multi-core implementation to support most of SPIN's existing features *unchanged* is detrimental to scalability.

Applicability: The components of our reachability can be reused directly for other model checking applications. The hash table and the load balancing algorithms can be reused to realize scalable multi-core (weak) LTL model checking [1], [2], symbolic exploration and space-efficient enumerative exploration. We experimented with the latter using *tree compression* [4] based on our hash table. Results are very promising and we intend to follow up on that.

Future work: By exploring the possible solutions and gradually improving this work, we found a wealth of variables hiding in the algorithms and the models of the BEEM database. As can be seen from the figures, different models show different scalability. A valid question is how much this can be improved.

By now we have some ideas where these differences come from. For example, an initial version of the exploration algorithm employed static load balancing by means of an initial BFS exploration and handing off the states from the last level to all threads. Several models where insensitive to this static approach, others, like `hanoi` and `frogs`, are very sensitive due to the shape of their state spaces. Dynamic load balancing did not come with a noticeable performance penalty for the other models, but `hanoi` and `frogs` are still in the bottom of the figures. However, we have yet to see the options exhausted to improve these results by shared-memory load balancing techniques, at least to the point that the shape of the state space of these models allow.

We did not consider other types of hash tables, like *Cuckoo hashing* or *Hopscotch hashing* [11]. Cuckoo hashing is an unlikely candidate, since it requires updates on many locations upon inserts, easily resulting in extraneous cache coherence overhead. Hopscotch hashing could be considered because it combines a low memory working set with constant lookup times even under higher load factors. However, Hopscotch hashing increases the memory working set for insertions, potentially sacrificing some speedup. It would still be interesting to investigate its performance relative to our hash table.

VI. Acknowledgements

We thank Anton Starikov and the CMS group at UTwente for making their cluster available for our experiments. We thank Petr Ročkai and Jiří Barnat for their support on the DiVinE toolkits, and for reading a draft of this paper. Cliff Click and Gerard Holzmann also gave helpful comments on a draft version. Elwin Pater implemented the bridge between DiVinE and LTSmin. The Linux developers provided patches remedying performance regressions on newer kernels.

References

[1] Jiří Barnat, Luboš Brim, and P. Ročkai. Scalable multi-core LTL model-checking. In *Model Checking Software*, volume 4595 of *LNCS*, pages 187–203. Springer, 2007.

[2] Jiří Barnat, Luboš Brim, and Petr Ročkai. DiVinE 2.0: High-Performance Model Checking. In *2009 International Workshop on High Performance Computational Systems Biology (HiBi 2009)*, pages 31–32. IEEE Computer Society Press, 2009.

[3] Jiří Barnat and Petr Ročkai. Shared hash tables in parallel model checking. *Electronic Notes in Theoretical Computer Science*, 198(1):79 – 91, 2008. Proceedings of the 6th International Workshop on Parallel and Distributed Methods in verifiCation (PDMC 2007).

[4] Stefan Blom, Bert Lisser, Jaco van de Pol, and Michael Weber. A database approach to distributed state space generation. *Electron. Notes Theor. Comput. Sci.*, 198(1):17–32, 2008.

[5] Stefan Blom, Jaco van de Pol, and Michael Weber. LTSmin: Distributed and symbolic reachability. In *Proceedings of CAV 2010*, LNCS. Springer, 2010. (accepted for publication).

[6] Luboš Brim. Distributed verification: Exploring the power of raw computing power. In Luboš Brim, Boudewijn Haverkort, Martin Leucker, and Jaco van de Pol, editors, *Formal Methods: Applications and Technology*, volume 4346 of *LNCS*, pages 23–34. Springer, August 2006.

[7] Cliff Click. A lock-free hash table. Talk at JavaOne 2007, http://www.azulsystems.com/events/javaone_2007/2007_LockFreeHash.pdf, 2007.

[8] Byron Cook, Andreas Podelski, and Andrey Rybalchenko. Terminator: Beyond safety. In *Proceedings of CAV 2006*, pages 415–418, 2006.

[9] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3 edition, September 2009.

[10] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, March 2008.

[11] Maurice Herlihy, Nir Shavit, and Moran Tzafrir. Hopscotch hashing. *Distributed Computing*, pages 350–364, 2008.

[12] Gerard J. Holzmann. A stack-slicing algorithm for multi-core model checking. *Electronic Notes in Theoretical Computer Science*, 198(1):3 – 16, 2008. Proceedings of the 6th International Workshop on Parallel and Distributed Methods in verifiCation (PDMC 2007).

[13] Gerard J. Holzmann and Dragan Bošnacki. The design of a multicore extension of the SPIN model checker. *IEEE Trans. Softw. Eng.*, 33(10):659–674, 2007.

[14] Cornelia P. Inggs and Howard Barringer. Effective state exploration for model checking on a shared memory architecture. *Electr. Notes Theor. Comput. Sci.*, 68(4), 2002.

[15] Alfons Laarman, Jaco van de Pol, and Michael Weber. Boosting Multi-Core Reachability Performance with Shared Hash Tables. *ArXiv e-prints*, 1004.2772, April 2010.

[16] Witold Litwin. Linear hashing: a new tool for file and table addressing. In *VLDB '1980: Proceedings of the sixth international conference on Very Large Data Bases*, pages 212–223. VLDB Endowment, 1980.

[17] Michael Monagan and Roman Pearce. Parallel sparse polynomial multiplication using heaps. In *ISSAC '09: Proceedings of the 2009 international symposium on Symbolic and algebraic computation*, pages 263–270, New York, NY, USA, 2009. ACM.

[18] Radek Pelánek. BEEM: Benchmarks for explicit model checkers. In *Proc. of SPIN Workshop*, volume 4595 of *LNCS*, pages 263–267. Springer, 2007.

[19] Chris Purcell and Tim Harris. Non-blocking hashtables with open addressing. *Distributed Computing*, pages 108–121, 2005.
[20] Nageshwara V. Rao and Vipin Kumar. Superlinear speedup in parallel state-space search. *Foundations of Software Technology and Theoretical Computer Science*, pages 161–174, 1988.
[21] Peter Sanders. Lastverteilungsalgorithmen fur parallele tiefensuche. number 463. In *in Fortschrittsberichte, Reihe 10. VDI*. Verlag, 1997.
[22] Michael Weber. An embeddable virtual machine for state space generation. In D. Bosnacki and S. Edelkamp, editors, *Proceedings of the 14th International SPIN Workshop*, volume 2595 of *Lecture Notes in Computer Science*, pages 168–186, Berlin, 2007. Springer Verlag.

APPENDIX

Additional analysis can be found in an extended report [15].

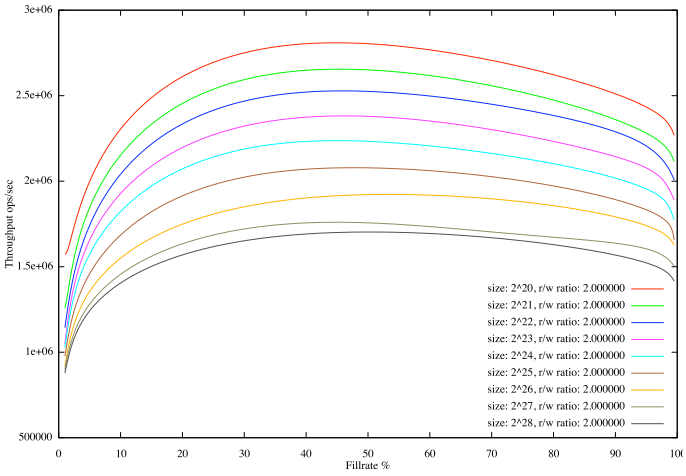


Fig. 6. Effect of size vs r/w-ratio on average throughput

TABLE II
MODEL DETAILS FOR DIVINE, LTSMIN AND SPIN

BEEM Model	Reachable States		State Vector Size [Byte]		
	DiVinE, LTSmin	SPIN	DiVinE	SPIN	LTSmin
anderson.6	18,206,917	18,206,919	25	68	76
at.5	31,999,440	31,999,442	20	68	56
at.6	160,589,600	—	20	—	56
bakery.6	11,845,035	11,845,035	24	48	80
bakery.7	29,047,471	27,531,713	24	48	80
blocks.4	104,906,622	88,987,772	23	44	88
brp.5	17,740,267	—	24	—	72
cambridge.7	11,465,015	—	60	—	208
elevator_planning.2	11,428,767	11,428,769	36	52	140
firewire_link.5	18,553,032	—	66	—	200
fischer.6	8,321,728	8,321,730	27	92	72
frogs.4	17,443,219	17,443,221	33	68	120
frogs.5	182,772,126	182,772,130	38	68	140
hanoi.3	14,348,907	14,321,541	63	116	228
iprotocol.6	41,387,484	—	43	—	148
iprotocol.7	59,794,192	—	47	—	164
lamport.8	62,669,317	62,669,317	22	52	68
lann.6	144,151,628	—	28	—	80
lann.7	160,025,986	—	35	—	100
leader_filters.7	26,302,351	26,302,351	36	68	120
loyd.3	239,500,800	214,579,860	18	44	64
mcs.5	60,556,519	53,779,475	26	68	84
needham.4	6,525,019	—	51	—	112
peterson.7	142,471,098	142,471,100	30	56	100
phils.6	14,348,906	13,956,555	45	140	120
phils.8	43,046,720	—	48	—	128
production_cell.6	14,520,700	—	42	—	104
szymanski.5	79,518,740	79,518,740	30	60	100
telephony.4	12,291,552	12,291,554	24	56	80
telephony.7	21,960,308	21,960,310	28	64	96
train-gate.7	50,199,556	—	43	—	128

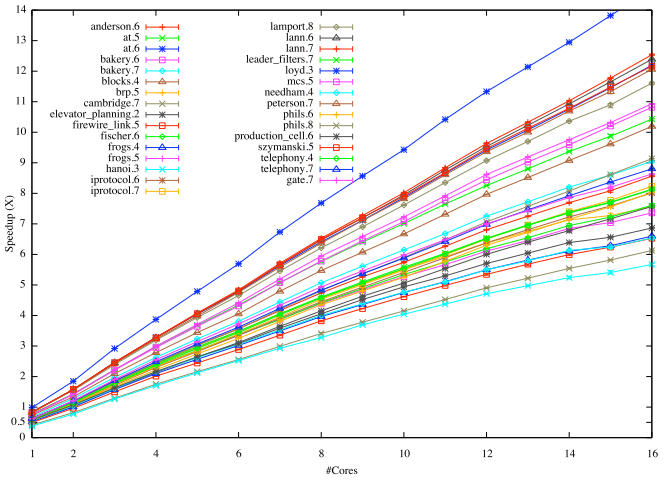


Fig. 7. Speedup of BEEM models with LTSmin (DiVinE as base case)

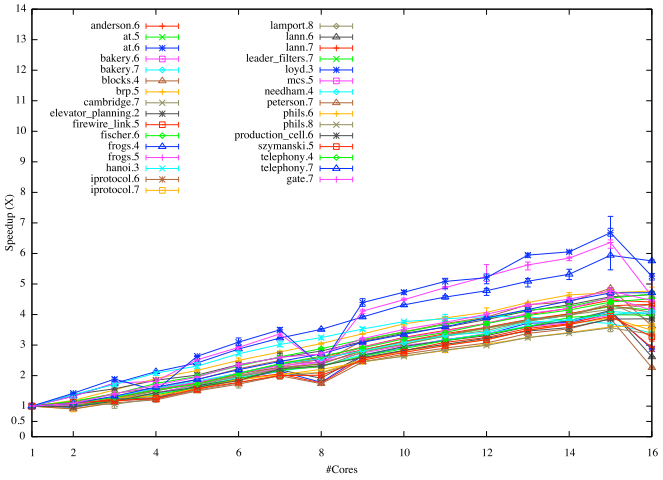


Fig. 8. Speedup of BEEM models with DiVinE 2.2 (DiVinE as base case)

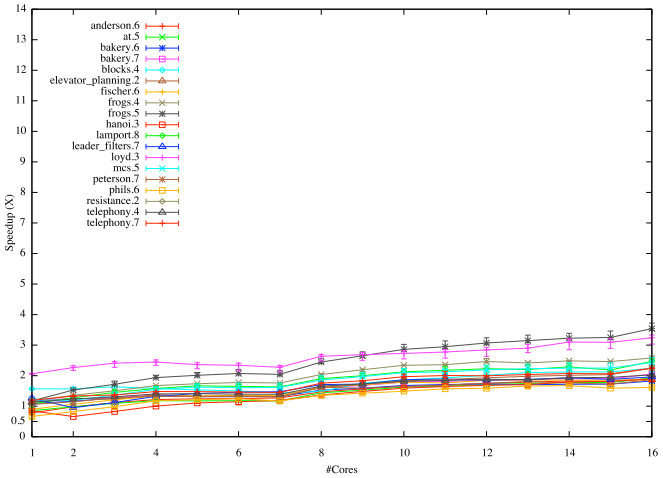


Fig. 9. Speedup of BEEM models with SPIN (DiVinE as base case)

Incremental Component-based Construction and Verification using Invariants

Saddek Bensalem[1] Marius Bozga[1] Axel Legay[2] Thanh-Hung Nguyen[1] Joseph Sifakis[1] Rongjie Yan[1]
[1] Verimag Laboratory, Université Joseph Fourier Grenoble, CNRS
[2]INRIA/IRISA, Rennes

***Abstract*—We propose invariant-based techniques for the efficient verification of safety and deadlock properties of concurrent systems. We assume that components and component interactions are described within the BIP framework, a tool for component-based design. We build on a compositional methodology in which the invariant is obtained by combining the invariants of the individual components with an interaction invariant that takes concurrency and interaction between components into account. In this paper, we propose new efficient techniques for computing interaction invariants. This is achieved in several steps. First, we propose a formalization of incremental component-based design. Then we suggest sufficient conditions that ensure the preservation of invariants through the introduction of new interactions. For cases in which these conditions are not satisfied, we propose methods for generation of new invariants in an incremental manner. The reuse of existing invariants reduces considerably the verification effort. Our techniques have been implemented in the D-Finder toolset. Among the experiments conducted, we have been capable of verifying properties and deadlock-freedom of DALA, an autonomous robot whose behaviors in the functional level are described with 500000 lines of C Code. This experiment, which is conducted with industrial partners, is far beyond the scope of existing academic tools such as NuSMV or SPIN.**

I. Introduction

Model Checking [10, 14] of concurrent systems is a challenging problem. Indeed, concurrency often requires computing the product of the individual systems by using both interleaving and synchronization. In general, the size of this structure is prohibitive and cannot be handled without manual interventions. In a series of recent works, it has been advocated that *compositional verification techniques* could be used to cope with state explosion in concurrent systems. Component-based design techniques confer numerous advantages, in particular, through reuse of existing components. A key issue is the existence of composition frameworks ensuring the correctness of composite components. We need frameworks allowing us not only reuse of components but also reuse of their properties for establishing global properties of composite components from properties of their constituent components. This should help cope with the complexity of global monolithic verification techniques.

Compositionality allows us to infer global properties of complex systems from properties of their components. The idea of compositional verification techniques is to apply divide-and-conquer approaches to infer global properties of complex systems from properties of their components. They are used to cope with state explosion in concurrent systems. Nonetheless, we also should consider the behavior and properties resulted from mutually interacting components. A compositional verification method based on invariant computation is presented in [3, 2]. This approach is based on the following rule:

$$\frac{\{B_i < \Phi_i >\}_i,\ \Psi \in II(\|_\gamma\{B_i\}_i, \{\Phi_i\}_i),\ (\bigwedge_i \Phi_i) \wedge \Psi \Rightarrow \Phi}{\|_\gamma\{B_i\}_i < \Phi >}$$

The rule allows to prove invariance of property Φ for systems obtained by using an n-ary composition operation $||$ parameterized by a set of interactions γ. It uses global invariants that are the conjunction of individual invariants Φ_i of individual components B_i and an *interaction invariant* Ψ. The latter expresses constraints on the global state space induced by interactions between components. In [3], we have shown that Ψ can be computed automatically from abstractions of the system to be verified. These are the composition of finite state abstractions B_i^α of the components B_i with respect to their invariants Φ_i. The approach has been implemented in the D-Finder toolset [2] and applied to check deadlock-freedom on several case studies described in the BIP (Behavior, Interaction, Priority) [1] language. The results of these experiments show that D-Finder is exponentially faster than well-established tools such as NuSMV [9].

Incremental system design methodologies often work by adding new interactions to existing sets of components. Each time an interaction is added, one may be interested to verify whether the resulting system satisfies some given property. Indeed, it is important to report an error as soon as it appears. However, each verification step may be time consuming, which means that intermediary verification steps are generally avoided. The situation could be improved if the result of the verification process could be reused when new interactions are added. Existing techniques, including the one in [3], do not focus on such aspects. In a very recent work [6], we have proposed a new fixed point based technique that takes incremental design into account. This technique is generally faster than the one in [3] for systems with an acyclic topology. For systems with a cyclic topology, the situation may however be reversed. There are also many case studies that are beyond the scope of these techniques.

In this paper, we continue the quest for efficient incremental

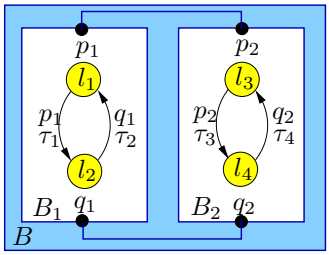


Fig. 1. A simple example

techniques for computing invariants of concurrent systems. We present a detailed methodology for incremental construction and verification of component-based systems. This is achieved in several steps. First, we propose a formalization of incremental component-based design. Then we suggest sufficient conditions that ensure the preservation of invariants through the introduction of new interactions. For cases in which these conditions are not satisfied, we propose methods for generation of new invariants in an incremental manner. The reuse of existing invariants reduces considerably the verification effort. Contrary to the technique in [6], our technique, which relies on a relation between behaviors of components and interactions, turns out to be efficient for both cyclic and acyclic topologies.

Our techniques have been implemented as extensions of the D-Finder toolset [2] and applied on several case studies. Our experiments show that our new methodology is generally much faster than the ones proposed in [3, 6]. In particular, we have been capable of verifying deadlock-freedom and safety properties of DALA, an autonomous robot whose behaviors in the functional level are described with 500000 lines of C Code. This experiment, which is conducted with industrial partners, is far beyond the scope of [3, 6] and of existing academic tools such as NuSMV or SPIN.

Structure of the paper. In section II, we recap the concepts that will be used through the paper as well as the incremental methodology introduced in [6]. Section III discusses sufficient conditions for invariant preservation while Section IV presents our incremental construction for invariants. Section V discusses the experiments. Finally, Section VI concludes the paper. Due to space limitation, some proofs and model descriptions are available from http://www-verimag.imag.fr/~yan/.

II. Preliminaries

In this section, we present concepts and definitions that will be used through the rest of the paper. We start with the concepts of *components*, *parallel composition of components*, *systems*, and *invariants*. In the second part of the section, we will recap a very recent methodology [6] we proposed for *incremental design* of composite systems.

A. Components, Interactions, and Invariants

In the paper, we will be working with a simplified model for component-based design. Roughly speaking, an atomic component is nothing more than a transition system whose transitions' labels are called *ports*. These ports are used to synchronize with other components. Formally, we have the following definition.

Definition 1 (Atomic Component)**.** *An atomic component is a transition system $B = (L, P, \mathcal{T})$, where:*

- *$L = \{l_1, l_2, \ldots, l_k\}$ is a set of locations,*
- *P is a set of ports, and*
- *$\mathcal{T} \subseteq L \times P \times L$ is a set of transitions.*

Given $\tau = (l, p, l') \in \mathcal{T}$, l and l' are the *source* and *destination* locations, respectively. In the rest of the paper, we use ${}^\bullet\tau$ and $\tau^\bullet$ to compute the source and destination of τ, respectively.

Example 1. *Figure 1 presents two atomic components. The ports of component B_1 are p_1 and q_1. B_1 has two locations: l_1 and l_2 and two transitions: $\tau_1 = (l_1, p_1, l_2)$ and $\tau_2 = (l_2, q_1, l_1)$.*

We are now ready to define parallel composition between atomic components. In the incremental design setting, the parallel composition operation allows to build bigger components starting from *atomic components*. Any composition operation requires to define a communication mode between components. In our context, components communicate via *interactions*, i.e., by synchronization on ports. Formally, we have the following definition.

Definition 2 (Interactions)**.** *Given a set of n components $B_1, B_2, \ldots, B_n$ with $B_i = (L_i, P_i, \mathcal{T}_i)$, an interaction a is a set of ports, i.e., a subset of $\bigcup_{i=1}^{n} P_i$, such that $\forall i = 1, \ldots, n.\ |a \cap P_i| \leq 1$.*

By definition, each interaction has at most one port per component. In the figures, we will represent interactions by link between ports. As an example, the set $\{p_1, p_2\}$ is an interaction between Components B_1 and B_2 of Figure 1. This interaction describes a synchronization between Components B_1 and B_2 by Ports p_1 and p_2. Another interaction is given by the set $\{q_1, q_2\}$. The idea being that a parallel composition is entirely defined by a set of interactions, which we call a *connector*. As an example the connector for B_1 and B_2 is the set $\{\{p_1, p_2\}, \{q_1, q_2\}\}$. In the rest of the paper, we simplify the notations and write $p_1 p_2 \ \ldots \ p_k$ instead of $\{p_1, \ldots, p_k\}$. We also write $a_1 + \ldots + a_m$ for the connector $\{a_1, \ldots, a_m\}$. As an example, notation for the connector $\{\{p_1, p_2\}, \{q_1, q_2\}\}$ is $p_1\ p_2 + q_1\ q_2$.

We now propose our definition for parallel composition. In what follows, we use I for a set of integers.

Definition 3 (Parallel Composition)**.** *Given n atomic components $B_i = (L_i,\ P_i,\ \mathcal{T}_i)$ and a connector γ, we define the parallel composition $B = \gamma(B_1, \ldots, B_n)$ as the transition system $(\mathcal{L}, \gamma, \mathcal{T})$, where:*

- *$\mathcal{L} = L_1 \times L_2 \times \ldots \times L_n$ is the set of* global *locations,*
- *γ is a set of interactions, and*
- *$\mathcal{T} \subseteq \mathcal{L} \times \gamma \times \mathcal{L}$ contains all transitions $\tau = ((l_1, \ldots, l_n), a, (l'_1, \ldots, l'_n))$ obtained by synchronization of sets of transitions $\{\tau_i = (l_i, p_i, l'_i) \in \mathcal{T}_i\}_{i \in I}$ such that $\{p_i\}_{i \in I} = a \in \gamma$ and $l'_j = l_j$ if $j \notin I$.*

The idea is that components communicate by synchronization with respect to interactions. Given an interaction a, only those

components that are involved in a can make a step. This is ensured by following a transition labelled by the corresponding port involved in a. If a component does not participate to the interaction, then it has to remain in the same state. In the rest of the paper, a component that is obtained by composing several components will be called a *composite component*. Consider the example given in Figure 1, we have a composite component $\gamma(B_1, B_2)$, where $\gamma = p_1\ p_2 + q_1\ q_2$. Observe that the component $\gamma_\perp(B_1, \ldots, B_n)$, which is obtained by applying the connector $\gamma_\perp = \sum_{i=1}^{n}(\sum_{p_j \in P_i} p_j)$, is the transition system obtained by interleaving the transitions of atomic components. Observe also that the parallel composition $\gamma(B_1, \ldots, B_n)$ of $B_1, \ldots, B_n$ can be seen as a *1-safe Petri net* (the number of tokens in all places is at most one) whose set of places is given by $L = \bigcup_{i=1}^{n} L_i$ and whose transitions relation is given by $\mathcal{T}$. In the rest of the paper, L will be called the *set of locations of* B, while $\mathcal{L}$ is the set of *global states*. We now define the concept of invariants, which can be used to verify properties of (parallel composition of) components. We first propose the definition of *system* that is a component with an initial set of states.

Definition 4 (System)**.** *A system $\mathcal{S}$ is a pair $\langle B, Init\rangle$ where B is a component and $Init$ is a state predicate characterizing the initial states of B.*

In a similar way, we distinguish invariants of a component from those of a system such that the invariants of a system $\mathcal{S} = \langle B, Init\rangle$ can be obtained from those of B according to the constraint $Init$. Therefore we define invariants for a component and for a system separately.

Definition 5 (Invariants)**.** *Given a component $B = (L, P, \mathcal{T})$, a predicate $\mathcal{I}$ on L is an invariant of B, denoted by $inv(B, \mathcal{I})$, if for any location $l \in L$ and any port $p \in P$, $\mathcal{I}(l)$ and $l \xrightarrow{p} l' \in \mathcal{T}$ imply $\mathcal{I}(l')$, where $\mathcal{I}(l)$ means that l satisfies $\mathcal{I}$. For a system $\mathcal{S} = \langle B, Init\rangle$, $\mathcal{I}$ is an invariant of $\mathcal{S}$, denoted by $inv(\mathcal{S}, \mathcal{I})$, if it is an invariant of B and if $Init \Rightarrow \mathcal{I}$.*

Clearly, if $\mathcal{I}_1,\ \mathcal{I}_2$ are invariants of B (respectively $\mathcal{S}$) then $\mathcal{I}_1 \wedge \mathcal{I}_2$ and $\mathcal{I}_1 \vee \mathcal{I}_2$ are also invariants of B (respectively $\mathcal{S}$).

Let $\gamma(B_1, \ldots, B_n)$ be the composition of n components with $B_i = (L_i,\ P_i,\ \mathcal{T}_i)$ for $i \in 1 \ldots n$. In the paper, an invariant on B_i is called a *component invariant* and an invariant on $\gamma(B_1, \ldots, B_n)$ is called an *interaction invariant*. To simplify the notations, we will assume that interaction invariants are predicates on $\bigcup_{i=1}^{n} L_i$.

B. Incremental Design

In component-based design, the construction of a composite system is both step-wise and hierarchical. This means that a system is obtained from a set of atomic components by successive additions of new interactions also called *increments*. In a very recent work [6], we have proposed a methodology to add new interactions to a composite component without breaking the synchronization. The techniques we will propose to compute and reuse invariants intensively build on this methodology, which is described hereafter.

First, when building a composite system in a bottom-up manner, it is essential that some already enforced synchronizations are not relaxed when increments are added. To guarantee this property, we propose the notion of *forbidden interactions*.

Definition 6 (Closure and Forbidden Interactions)**.** *Let γ be a connector.*

- *The closure γ^c of γ, is the set of the non empty interactions contained in some interaction of γ. That is $\gamma^c = \{a \neq \emptyset \mid \exists b \in \gamma.\ a \subseteq b\}$.*
- *The forbidden interactions γ^f of γ is the set of the interactions strictly contained in all the interactions of γ. That is $\gamma^f = \gamma^c - \gamma$.*

It is easy to see that for two connectors γ_1 and γ_2, we have $(\gamma_1 + \gamma_2)^c = \gamma_1^c + \gamma_2^c$ and $(\gamma_1 + \gamma_2)^f = (\gamma_1 + \gamma_2)^c - \gamma_1 - \gamma_2$.

In our theory, a connector describes a set of interactions and, by default, also those interactions in where only one component can make progress. This assumption allows us to define new increments in terms of existing interactions.

Definition 7 (Increments)**.** *Consider a connector γ over B and let $\delta \subseteq 2^\gamma$ be a set of interactions. We say δ is an increment over γ if for any interaction $a \in \delta$ we have interactions $b_1, \ldots, b_n \in \gamma$ such that $\bigcup_{i=1}^{n} b_i = a$.*

In practice, one has to make sure that existing interactions defined by γ will not break the synchronizations that are enforced by the increment δ. For doing so, we remove from the original connector γ all the interactions that are forbidden by δ. This is done with the operation of *Layering*, which describes how an increment can be added to an existing set of interactions without breaking synchronization enforced by the increment. Formally, we have the following definition.

Definition 8 (Layering)**.** *Given a connector γ and an increment δ over γ, the new set of interactions obtained by combining δ and γ, also called layering, is given by the following set $\delta\gamma = (\gamma - \delta^f) + \delta$ the incremental construction by layering, that is, the incremental modification of γ by δ.*

The above definition describes one-layer incremental construction. By successive applications of the rule, we can construct a system with multiple layers. Besides the fusion of interactions, incremental construction can also be obtained by first combining the increments and then apply the result to the existing system. This process is called *Superposition*. Formally, we have the following definition.

Definition 9 (Superposition)**.** *Given two increments δ_1, δ_2 over a connector γ, the operation of superposition between δ_1 and δ_2 is defined by $\delta_1 + \delta_2$.*

Superposition can be seen as a composition between increments. If we combine the superposition of increments with the layering proposed in Definition 8, then we obtain an incremental construction from a set of increments. Formally, we have the following proposition.

Proposition 1. *Let γ be a connector over B, the incremental*

construction by the superposition of n increments $\{\delta_i\}_{1\leq i\leq n}$ is given by

$$(\sum_{i=1}^{n}\delta_i)\gamma = (\gamma - (\sum_{i=1}^{n}\delta_i)^f) + \sum_{i=1}^{n}\delta_i \quad (1)$$

The above proposition provides a way to transform incremental construction by a set of increments into the separate constituents, where $\gamma - (\Sigma_{i=1}^{n}\delta_i)^f$ is the set of interactions that are allowed during the incremental construction process.

III. Invariant Preservation in Incremental Design

In Section II-B, we have presented a methodology for the incremental design of composite systems. In this section, we study the concept of *invariant preservation*. More precisely, we propose sufficient conditions to guarantee that already satisfied invariants are not violated when new interactions are added to the design.

We start by introducing the *looser synchronization preorder* on connectors, which we will use to characterize invariant preservation. As we have seen, interactions characterize the behavior of a composite component. We observe that if two interactions do not contain the same port, the execution of one interaction will not block the execution of the other interaction. Formally, we have the following definition.

Definition 10 (Conflict-free Interactions)**.** *Given a connector γ, let $a_1,\ a_2 \in \gamma$, if $a_1 \cap a_2 = \emptyset$, we say that there is no conflict between a_1 and a_2. If there is no conflict between any interactions of γ, we say that γ is conflict-free.*

We now propose a preorder relation that allows to guarantee the absence of conflicts when new interactions are added. Formally, we have the following definition.

Definition 11 (Looser Synchronization Preorder)**.** *We define the looser synchronization preorder $\preccurlyeq \subseteq 2^{2^P} \times 2^{2^P}$. For two connectors γ_1, γ_2, $\gamma_1 \preccurlyeq \gamma_2$ if for any interaction $a \in \gamma_2$, there exist interactions $b_1, \ldots, b_n \in \gamma_1$, such that $a = \bigcup_{i=1}^{n} b_i$ and there is no conflict between any b_i and b_j, where $1 \leq i, j \leq n$ and $i \neq j$. We simply say that γ_1 is looser than γ_2.*

The above definition requires that the stronger synchronization should be obtained by the fusion of conflict-free interactions. The reason is that the execution of interactions may be disturbed by two conflict interactions, i.e., the execution of one interaction could block the transitions issued from the other interaction. However, if we fuse them together, it means that the transitions of both interactions can be executed, which violates the constraints of the previous behavior. It is easy to see that if $\gamma_1,\ \gamma_2,\ \gamma_3,\ \gamma_4$ are connectors such that $\gamma_1 \preccurlyeq \gamma_2$, and $\gamma_3 \preccurlyeq \gamma_4$, then we have $\gamma_1 + \gamma_3 \preccurlyeq \gamma_2 + \gamma_4$.

We now propose the following proposition which establishes a link between the looser synchronization preorder and invariant preservation.

Proposition 2. *Let $\gamma_1,\ \gamma_2$ be two connectors over B. If $\gamma_1 \preccurlyeq \gamma_2$, we have $inv(\gamma_1(B), \mathcal{I}) \Rightarrow inv(\gamma_2(B), \mathcal{I})$.*

The above proposition, which will be used in the incremental design, simply says that if an invariant is satisfied, then it will remain when combinations of conflict-free interactions are added (following our incremental methodology) to the connector. This is not surprising as the tighter connector can only restrict the behaviors of the composite system.

We now switch to the more interesting problem of providing sufficient conditions to guarantee that invariants are preserved by the incremental construction.

Proposition 3. *Let γ be a connector over B and δ be an increment of γ such that $\gamma \preccurlyeq \delta$, then we have $\gamma \preccurlyeq \delta\gamma$.*

The above proposition, together with Proposition 2, says that the addition of an increment preserves the invariant if the initial connector is looser than the increment.

We continue our study and discuss the invariant preservation between the components obtained from superposition of increments and separately applying increments over the same set of components. We use the following definition.

Definition 12 (Interference-free Connectors)**.** *Given two connectors γ_1, γ_2, for any $a_1 \in \gamma_1$, $a_2 \in \gamma_2$, if either a_1 and a_2 are conflict-free or $a_1 = a_2$, we say that γ_1 and γ_2 are interference-free.*

This definition considers a relation between two connectors. We observe that two interference-free connectors will not break or block the synchronizations specified by each other. Though we require that the interactions between γ_1 and γ_2 are conflict-free, γ_1 or γ_2 respectively can contain conflict interactions. For example, consider two connectors $\gamma_1 = p_1\ p_2 + p_2\ p_3$, $\gamma_2 = p_4\ p_5$. γ_1 is not conflict-free, but γ_1 and γ_2 are interference-free.

We now present the main result of the section.

Proposition 4. *Consider two increments $\delta_1,\ \delta_2$ over γ such that $\gamma \preccurlyeq \delta_1$ and $\gamma \preccurlyeq \delta_2$, if δ_1 and δ_2 are interference-free, and $inv(\delta_1\gamma(B), \mathcal{I}_1)$, $inv(\delta_2\gamma(B), \mathcal{I}_2)$, we have $inv((\delta_1 + \delta_2)\gamma(B), \mathcal{I}_1 \wedge \mathcal{I}_2)$.*

The above proposition considers a set of increments $\{\delta_i\}_{1\leq i\leq n}$ over γ that are interference-free. The proposition says that if for any δ_i the separate application of increments over component $\delta_i\gamma(B)$ preserves the original invariants of $\gamma(B)$, then the system obtained from considering the superposition of increments over γ preserves the conjunction of the invariants of individual increments.

We now briefly study the relation between the looser synchronization preorder and *property preservation*. Figure 2 shows the three ingredients of the BIP language, that are (1) priorities, which we will not use here, (2) interactions, and (3) behaviors of components. We shall see that the looser synchronization preorder preserves invariants (Proposition 4). This means that the preorder preserves the so-called reachability properties. On the other hand, the preorder does not preserve deadlocks. Indeed, adding new interactions may lead to the addition of new deadlock conditions. Given two connectors γ_1 and γ_2 over component B such that γ_2 is tighter than γ_1,

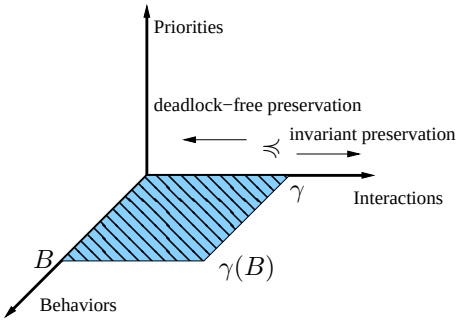


Fig. 2. Invariant preservation for looser synchronization relation

i.e., $\gamma_1 \preccurlyeq \gamma_2$, we can conclude that if $\gamma_2(B)$ is deadlock-free, then $\gamma_1(B)$ is deadlock-free. However, we can still reuse the invariant of $\gamma_1(B)$ as an over-approximation of the one of $\gamma_2(B)$.

Discussion. Though we can reuse invariants to save computation time, the invariants of the system with a looser connector may be too weak with respect to a new system obtained with a tighter connector. Consider the example given in Figure 1 and let $\gamma = p_1 + p_2 + q_1 + q_2$, $\delta_1 = p_1\ p_2$, and $\delta_2 = q_1\ q_2$. By using the technique presented in the next section, we shall see that the invariant for $\delta_1\gamma(B)$ and $\delta_2\gamma(B)$ is $(l_1 \vee l_2) \wedge (l_3 \vee l_4)$. By applying Proposition 4, we obtain that this invariant is preserved for $(\delta_1 + \delta_2)\gamma(B)$. This invariant is weaker than the invariant $(l_1 \vee l_2) \wedge (l_3 \vee l_4) \wedge (l_1 \vee l_4) \wedge (l_2 \vee l_3)$ that is directly computed on $(\delta_1 + \delta_2)\gamma(B)$. To overcome the above problem, we will now propose an approach that can be used to compute invariants in an incremental manner.

IV. Efficient Incremental computation of Invariants

In Section II-B, we have proposed a methodology to build a composite system by successive addition of increments. We now propose a methodology that allows to reuse existing interaction invariants when new interactions are added to the system. The section is divided in two subsections. In the first subsection, we recap the concept of *Boolean Behavioral Constraints* [3, 6], which can be used to characterize interaction invariants. In the second subsection, we propose our incremental methodology.

A. Boolean Behavioral Constraints (BBCs)

In [3], we have presented a verification method for component-based systems. The method uses a heuristic to symbolically compute invariants of a composite component. These invariants capture the interactions between components, which are the cause of global deadlocks. For this, it is sufficient to find an invariant that does not contain deadlock states. In this section, we improve the presentation of the result of [3] and prepare them for the incremental version that we will present in the next subsection.

Interactions describe the communication between different components, and transitions are the internal behavior of components. Here we unify these two types of behavioral description by introducing *Boolean Behavioral Constraints* (BBCs). We take $a_\tau = \{\{\tau_i\}_{i\in I} \mid (\forall i.\tau_i \in \mathcal{T}_i) \wedge (\{port(\tau_i)\}_{i\in I} = a)\}$. That is, a_τ consists of sets of component transitions involved in interaction a. As an example, consider the components given in Figure 1. Given $\gamma = p_1\ p_2 + q_1\ q_2$, we have $(p_1\ p_2)_\tau = \{\{\tau_1, \tau_3\}\}$, and $(q_1\ q_2)_\tau = \{\{\tau_2, \tau_4\}\}$.

Locations of components will be viewed as Boolean variables. We use $Bool[L]$ to denote the free Boolean algebra generated by the set of locations L. We also extend the notation ${}^\bullet\tau,\ \tau^\bullet$ to interactions, that is ${}^\bullet a = \{{}^\bullet\tau \mid \tau \in \mathcal{T}_i \wedge port(\tau) \in a\}$, and $a^\bullet = \{\tau^\bullet \mid \tau \in \mathcal{T}_i \wedge port(\tau) \in a\}$.

Definition 13 (Boolean Behavioral Constraints (BBCs))**.** *Let γ be a connector over a tuple of components $B = (B_1, \cdots, B_n)$ with $B_i = (L_i, P_i, \mathcal{T}_i)$ and $L = \bigcup_{i=1}^n L_i$. The Boolean behavioral constraints for component $\gamma(B)$ are given by the function $|\cdot| : \gamma(B) \to Bool[L]$ such that*

$$|\gamma(B)| = \bigwedge_{a\in\gamma} |a(B)|,$$
$$|a(B)| = \bigwedge_{\{\tau_i\}_{i\in I}\in a_\tau} \Big(\bigwedge_{l\in\{{}^\bullet\tau_i\}} (l \Rightarrow \bigvee_{l'\in\{\tau_i^\bullet\}} l')\Big)$$

If $\gamma = \emptyset$, then $|\gamma(B)| = true$, which means that no interactions between the components of B will be considered.

Roughly speaking, one implication $l \Rightarrow \bigvee_{l'\in\{\tau_i^\bullet\}} l'$ in $|\gamma(B)|$ describes a constraint on l that is restricted by an interaction of γ issued from l.

In what follows, we use $\bar{l}$ for the complement of l, i.e., $\neg l$.

Example 2. *Consider the components in Figure 1. Consider also the following connector $\gamma = p_1 + p_2 + q_1 + q_2$. Two increments over γ are $\delta_1 = p_1\ p_2$ and $\delta_2 = q_1\ q_2$. According to Definition 8, we have $\delta_1\gamma = p_1\ p_2 + q_1 + q_2$ when we only consider increment δ_1 over γ. For $\delta_1\gamma(B)$, the BBC $|p_1\ p_2(B)|$, $|q_1(B)|$ and $|q_2(B)|$ are respectively given by:*

$$|p_1p_2(B)| = (l_1 \Rightarrow l_2 \vee l_4) \wedge (l_3 \Rightarrow l_2 \vee l_4),$$
$$|q_1(B)| = (l_2 \Rightarrow l_1), \quad |q_2(B)| = (l_4 \Rightarrow l_3)$$

The BBC for $\delta_1\gamma(B)$ is $|\delta_1\gamma(B)| = |p_1p_2(B)| \wedge |q_1(B)| \wedge |q_2(B)| = (l_1 \Rightarrow l_2 \wedge l_4) \wedge (l_3 \Rightarrow l_2 \wedge l_4) \wedge (l_2 \Rightarrow l_1) \wedge (l_4 \Rightarrow l_3) = (\bar{l_1} \wedge \bar{l_2} \wedge \bar{l_3} \wedge \bar{l_4}) \vee (\bar{l_4} \wedge l_1 \wedge l_2) \vee (\bar{l_2} \wedge l_3 \wedge l_4) \vee (l_1 \wedge l_2 \wedge l_3) \vee (l_1 \wedge l_3 \wedge l_4)$.

When we consider two increments together, we have $(\delta_1 + \delta_2)\gamma(B) = p_1\ p_2 + q_1\ q_2$ by Definition 8 and 9. Because the BBC for interaction $q_1\ q_2$ over B is $(l_2 \Rightarrow l_1 \vee l_3) \wedge (l_4 \Rightarrow l_1 \vee l_3)$, we obtain that the BBC for $(\delta_1 + \delta_2)\gamma(B)$ is $|(\delta_1 + \delta_2)\gamma(B)| = |p_1p_2(B)| \wedge |q_1q_2(B)| = (l_1 \Rightarrow l_2 \vee l_4) \wedge (l_2 \Rightarrow l_1 \vee l_3) \wedge (l_3 \Rightarrow l_2 \vee l_4) \wedge (l_4 \Rightarrow l_1 \vee l_3) = (\bar{l_1} \wedge \bar{l_2} \wedge \bar{l_3} \wedge \bar{l_4}) \vee (l_1 \wedge l_2) \vee (l_2 \wedge l_3) \vee (l_1 \wedge l_4) \vee (l_3 \wedge l_4)$.

Example 2 shows that any BBC $|\gamma(B)|$ can be rewritten into a disjunctive normal form (DNF), where every conjunctive form is called a *monomial*. Any satisfiable monomial of $|\gamma(B)|$ is a solution of $|\gamma(B)|$. In fact, the enumeration of the clause of any monomial corresponds to an interaction invariant.

Theorem 1. *Let γ be a connector over a set of components $B = (B_1, \cdots, B_n)$ with $B_i = (L_i, P_i, \mathcal{T}_i)$ and $L = \bigcup_{i=1}^n L_i$, and $v : L \to \{true, false\}$ be a Boolean valuation different from false. If v is a solution of $|\gamma(B)|$, i.e., $|\gamma(B)|(v) = true$, then $\bigvee_{v(l)=true} l$ is an invariant of $\gamma(B)$.*

The above theorem gives a methodology to compute interaction invariants of $\gamma(B)$ directly from the solutions of $|\gamma(B)|$. In the rest of the paper, we will often use the term *BBC-invariant* to refer to the invariant that corresponds to a single solution of the BBC.

Since locations are viewed as Boolean variables, a location in a BBC is either a variable or the negation of a variable. As an example, l is a positive variable and $\neg l$ is a negative variable. However, as observed in Theorem 1, invariants are derived from positive variables of the solution of $|\gamma(B)|$. This suggests that all the negations should be removed. In general, due to incremental design and implementation (see Proposition 6 and Section V), these valuations can be removed gradually. We now propose a general mapping on removing variables with negations that do not belong to a given set of variables.

Definition 14 (Positive Mapping)**.** *Given two sets of variables L and L' such that $L' \subseteq L$, we define a mapping $p(L')$ over a disjunctive normal form formula that removes all the variables not in L' and with negations from the formula, such that*

$$(\bigwedge_{l_i \in L} l_i \wedge \bigwedge_{l_j \in L'} \bar{l_j} \wedge \bigwedge_{l_k \in L - L'} \bar{l_k})^{p(L')} = \bigwedge_{l_i \in L} l_i \wedge \bigwedge_{l_j \in L'} \bar{l_j}$$
$$(f_1 \vee f_2)^{p(L')} = f_1^{p(L')} \vee f_2^{p(L')}$$

where f_1 and f_2 are in disjunctive normal form.

If L' is empty, then the positive mapping will remove all the negations from a DNF formula f, which we will denote by f^p. Notice that $(\bigwedge_{i \in I} \bar{l_i})^p = false$.

We are now ready to propose an interaction invariant that takes all the solutions of the BBCs into account. We first introduce the notation $\widetilde{f}$ that stands for the dual of f, by replacing the AND operators with ORs (and vice versa) and the constant 0 with 1 (and vice versa). As we have seen, BBCs can be rewritten as a disjunction of monomials. By dualizing a monomial, one can obtain an interaction invariant. If one wants the strongest invariant that takes all the solution into account, one simply has to dualize the BBC. This is stated with the following theorem.

Theorem 2. *For any connector γ applied to a tuple of components $B = (B_1, \cdots, B_n)$, the interaction invariant of $\gamma(B)$ can be obtained as the dual of $|\gamma(B)|^p$, denoted by $\widetilde{|\gamma(B)|^p}$.*

Example 3. *We consider the components, connectors, and BBCs introduced in Example 2. The positive mapping removes variables with negations from $|\delta_1\gamma(B)|$ and $|(\delta_1 + \delta_2)\gamma(B)|$. We obtain that $\widetilde{|\delta_1\gamma(B)|^p} = (l_1 \vee l_2) \wedge (l_3 \vee l_4)$, and $\widetilde{|(\delta_1 + \delta_2)\gamma(B)|^p} = (l_1 \vee l_2) \wedge (l_3 \vee l_4) \wedge (l_1 \vee l_4) \wedge (l_2 \vee l_3)$. If we specify $Init = l_1 \wedge l_3$, every invariant of system $\langle \delta_1\gamma(B), Init \rangle$ and $\langle (\delta_1 + \delta_2)\gamma(B), Init \rangle$ should contain either l_1 or l_3. Therefore $(l_1 \vee l_2) \wedge (l_3 \vee l_4)$ is the interaction invariant of $\langle \delta_1\gamma(B), Init \rangle$, and $(l_1 \vee l_2) \wedge (l_3 \vee l_4) \wedge (l_1 \vee l_4) \wedge (l_2 \vee l_3)$ is the interaction invariant of $\langle (\delta_1 + \delta_2)\gamma(B), Init \rangle$.*

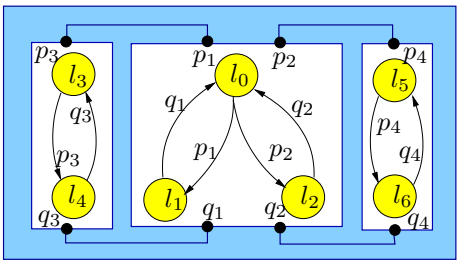


Fig. 3. An example for incremental computation of invariants

B. Incremental Computation of BBCs

In the previous section, we have shown that interaction invariants can be computed from the solutions of Boolean Behavioral Constraints. In this section, we show how to reuse existing invariants when new increments are added to the system. We first give a decomposition form for BBC and then show how this decomposition can be used to save computation time.

Proposition 5. *Let γ be a connector over B, the Boolean behavioral constraint for the composite component obtained by superposition of n increments $\{\delta_i\}_{1 \leq i \leq n}$ can be written as*

$$|(\sum_{i=1}^{n} \delta_i)\gamma(B)| = |(\gamma - (\sum_{i=1}^{n} \delta_i)^f)(B)| \wedge \bigwedge_{i=1}^{n} |\delta_i(B)| \quad (2)$$

Proposition 5 provides a way to decompose the computation of BBCs with respect to increments. The decomposition is based on the fact that different increments describe the interactions between different components. To simplify the notation, $\gamma - (\Sigma_{i=1}^{n} \delta_i)^f$ is represented by δ_0. We have the following example.

Example 4. *[Incremental BBC computation] In the example of Figure 3, let $\gamma = p_1 + p_2 + p_3 + p_4 + q_1 + q_2 + q_3 + q_4$. Two increments over γ are $\delta_1 = p_1\ p_3 + q_1\ q_3$ and $\delta_2 = p_2\ p_4 + q_2\ q_4$. The new connector obtained by applying δ_1 and δ_2 to γ is given by $(\delta_1 + \delta_2)\gamma = p_1\ p_3 + q_1\ q_3 + p_2\ p_4 + q_2\ q_4$. The BBC $|\delta_1(B)|$ and $|\delta_2(B)|$ are respectively given by:*

$$\begin{aligned} |\delta_1(B)| = \ & (l_0 \Rightarrow l_1 \vee l_4) \wedge (l_1 \Rightarrow l_0 \vee l_3) \wedge \\ & (l_3 \Rightarrow l_1 \vee l_4) \wedge (l_4 \Rightarrow l_0 \vee l_3), \\ |\delta_2(B)| = \ & (l_0 \Rightarrow l_2 \vee l_6) \wedge (l_2 \Rightarrow l_0 \vee l_5) \wedge \\ & (l_5 \Rightarrow l_2 \vee l_6) \wedge (l_6 \Rightarrow l_0 \vee l_5) \end{aligned}$$

Since $\gamma - (\delta_1 + \delta_2)^f = \emptyset$, we have $|(\delta_1 + \delta_2)\gamma(B)| = |\delta_1(B)| \wedge |\delta_2(B)|$.

We now switch to the problem of computing invariants while taking incremental design into account. We propose the following definition that will help in the process of reusing existing invariants.

Definition 15 (Common Location Variables L_c)**.** *The set of common location variables of a set of connectors $\{\gamma_i\}_{1 \leq i \leq n}$ is defined by $L_c = \bigcup_{i,j \in [1,n] \wedge i \neq j} support(\gamma_i) \cap support(\gamma_j)$, where $support(\gamma) = \bigcup_{a \in \gamma} {}^\bullet a \cup a^\bullet$, the set of locations involved in some interaction a of γ.*

Our incremental method assumes that an invariant has already been computed for a set of interactions (We use $\mathcal{I}_\delta$

to denote the BBC-invariant of $|\delta(B)|$). This information is exploited to improve the efficiency. The idea is as follows. According to Equation 1, the superposition of a set of increments $\{\delta_i\}_{1\leq i\leq n}$ over a connector γ can be regarded as separately applying increments over theirs constituents. We propose the following proposition, which builds on Equation 2.

Proposition 6. *Consider a composite component B. Let γ be a connector for B and assume a set of increments $\{\delta_i\}_{1\leq i\leq n}$ over $\gamma(B)$. Let $\delta_0 = \gamma - (\sum_{i=1}^{n}\delta_i)^f$, $\mathcal{I}_{\delta_i} = \bigwedge_{k\in I_i}\phi_k$, for $i = 0,\ldots,n$, be the BBC-invariants for each $|\delta_i(B)|$, $S_{\delta_i} = \bigvee_{k\in I_i} m_k$, for $i = 0,\ldots,n$, be the corresponding BBC-solutions, and let*

- *L_ϕ be the set of location variables in invariant ϕ,*
- *L_c be the common location variables between $\{\delta_0,\delta_1,\ \ldots,\delta_n\}$.*

Then the interaction invariant of $(\Sigma_{i=1}^{n}\delta_i)\gamma(B)$ is obtained as follows:

$$\mathcal{I} = \left(\bigwedge_{i=0}^{n}\ \bigwedge_{\substack{k\in I_i\wedge\\ L_c\cap L_{\phi_k}=\emptyset}}\phi_k\right)\wedge\left(\bigwedge_{(k_{i1},\ldots,k_{ir})\in\mathbb{D}}\ \bigvee_{j=1}^{r}\phi_{k_{ij}}\right)$$

where
$\mathbb{D} = \{(k_{i1},\ldots,k_{ir})|\ (\forall j = 1\ldots r\wedge k_{ij}\in I_{ij})\wedge(L_{\phi_{k_{ij}}}\cap L_c\neq\emptyset)\wedge(\bigwedge_{j=1}^{r} m_{k_{ij}}\neq false)\wedge((k_{i1},\ldots,k_{ir})\ \textit{is maximal})\}$.

The proposition simply says that one can take the conjunctions of BBC-invariants that do not share common variables, while one has to take the disjunction of the remaining invariants. This is to guarantee that common location variables will not change the satisfiability of the formulae. Observe that each non common variable occurs only in the solutions of one BBC. This allows deleting the non common variables with negations separately by using the positive mapping of common variables in every BBC-solutions, which reduces complexity of computation significantly.

Example 5. *[Incremental invariant computation] In Example 4, we have computed the BBCs for the two increments. Here we show how to compute the invariants from BBC-invariants of the increments. By Definition 15, we obtain that $L_c = \{l_0\}$. Let S_{δ_1}, S_{δ_2} be the BBC-solutions for $|\delta_1(B)|$ and $|\delta_2(B)|$ respectively, and $\mathcal{I}_{\delta_1},\mathcal{I}_{\delta_2}$ be their BBC-invariants, we have:*
$S_{\delta_1} = (\bar{l_0}\wedge\bar{l_1}\wedge\bar{l_3}\wedge\bar{l_4})\vee(l_0\wedge l_1)\vee(l_1\wedge l_3)\vee(l_0\wedge l_4)\vee(l_3\wedge l_4)$,
$S_{\delta_2} = (\bar{l_0}\wedge\bar{l_2}\wedge\bar{l_5}\wedge\bar{l_6})\vee(l_0\wedge l_2)\vee(l_2\wedge l_5)\vee(l_0\wedge l_6)\vee(l_5\wedge l_6)$,
$\mathcal{I}_{\delta_1} = (l_0\vee l_1)\wedge(l_0\vee l_4)\wedge(l_1\vee l_3)\wedge(l_3\vee l_4)$,
$\mathcal{I}_{\delta_2} = (l_0\vee l_2)\wedge(l_0\vee l_6)\wedge(l_2\vee l_5)\wedge(l_5\vee l_6)$
Because $\mathcal{I}_{(\delta_1+\delta_2)\gamma(B)} = \mathcal{I}_{((\gamma-(\delta_1+\delta_2)^f)+\delta_1+\delta_2)(B)}$ and $\gamma-(\delta_1+\delta_2)^f = \emptyset$, we have $\mathcal{I}_{(\delta_1+\delta_2)\gamma(B)} = \mathcal{I}_{(\delta_1+\delta_2)(B)}$.

Among the BBC-invariants, $(l_1\vee l_3),(l_3\vee l_4),(l_2\vee l_5),(l_5\vee l_6)$ do not contain any common location variables, so they will remain in the global computation. BBC-invariants $(l_0\vee l_1),(l_0\vee l_4)$, $(l_0\vee l_2)$ and $(l_0\vee l_6)$ contain l_0 as the common location variable, and the conjunction between every monomial from two groups of solutions are not false. So the final result is $(l_0\vee l_1\vee l_2)\wedge(l_0\vee l_4\vee l_6)\wedge(l_0\vee l_1\vee l_6)\wedge(l_0\vee l_2\vee l_4)\wedge(l_1\vee l_3)\wedge(l_3\vee l_4)\wedge(l_2\vee l_5)\wedge(l_5\vee l_6)$.

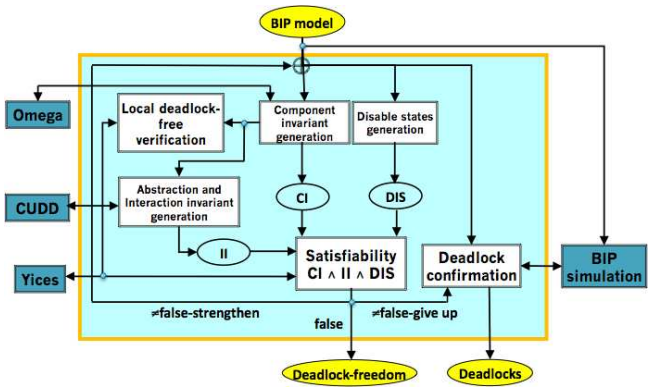


Fig. 4. D-Finder tool

V. Experiments

Our methodology for computing interaction invariants and deciding invariant preservation has been implemented in the D-Finder toolset [2].

In this section, we start with a brief introduction to the the D-Finder tool and explain what are the modifications that have. Then we show the experimental results obtained by implementing the methods discussed in this paper.

A. D-Finder Structure

D-Finder is an extension of the BIP toolset [7] – BIP can be used to define components and component interactions. D-Finder can verify both safety and deadlock-freedom properties of systems by using the techniques of this paper and of [3, 6].

We use $global$ to refer to the method of [3], FP for the incremental method of [6], and $Incr$ to refer to our new incremental technique.

The tool provides symbolic-representations-based methods for computing interaction invariants, namely the $Incr$ methods presented in this paper, the fixed point based method and its incremental method FP proposed in [6] as well as the $global$ method presented in [3] and discussed in Section II. D-Finder relies on the CUDD package [15] and represents sets of locations by BDDs. D-Finder also proposes techniques to compute component invariants. Those techniques, which are described in [3], relies on the Yices [11] and Omega [16] toolsets for the cases in where a component can manipulate data. A general overview of the structure of the tool is given in Figure 4.

D-Finder is mainly used to check safety properties of composite components. In this paper, we will be concerned with the verification of deadlock properites. We let DIS be the set of global states in where a deadlock can occur. The tool will progressively find and eliminate potential deadlocks as follows. D-Finder starts with an input a BIP model and computes component invariants CI by using the technique outlined in [3]. From the generated component invariants, it computes an abstraction of the BIP model and the corresponding interaction invariants II. Then, it checks satisfiability of the conjunction $II\wedge CI\wedge DIS$. If the conjunction is unsatisfiable, then there

is no deadlock else either it generates stronger component and interaction invariants or it tries to confirm the detected deadlocks by using reachability analysis techniques[1].

B. Implementation of the Incremental Method

We build on the symbolic implementation of the method in [3] that computes the interaction invariant of an entire system with all the interactions within the connector. The implementation relies on the CUDD package [15] and represents sets of locations by BDDs.

We have employed the following steps to integrate the incremental computation into the D-Finder tool. First we compute a set of common location variables from all the increments. Then we compute the BBC-solutions for every increment instead of computing the solutions for the connector in *global* method, and apply positive mapping to remove the location variables with negations that do not belong to the set of common location variables, to reduce the size of BDDs for BBC-solutions. We can either integrate existing solutions from the already computed BBCs progressively or integrate all the solutions when all the increments have been explored. Finally we apply positive mapping to remove all the remaining common location variables with negations and call the dual operation to obtain interaction invariant.

C. Experimental Results

We have compared the performance of the three methods on several case studies. All our experiments have been conducted with a 2.4GHz Duo CPU Mac laptop with 2GB of RAM.

We started by considering verification of deadlock properties. The case studies we consider are the Gas Station [12], the Smoker [13], the Automatic Teller Machine (ATM) [8] and the classical example of Producer/Consumer. Regarding the Gas Station example, we assume that every pump has 10 customers. Hence, if there are 50 pumps in a Gas Station, then we have 500 customers and the number of components including the operator is thus 551. In the ATM example, every ATM machine is associated to one user. Therefore, if we have 10 machines, then the number of components will be 22 (including the two components that describe the Bank). The computation times and memory usages for the application of the three methods on these case studies are given in Table I. Regarding the legend of the table, *scale* is the "size" of examples; *location* denotes the total number of control locations; *interaction* is for the total number of interactions. The computation time is given in minutes. The timeout, i.e., "-" is one hour. The memory usage is given in Megabyte (MB). Our technique is always faster than *global*. This means that we are also faster than tools such as NuSMV and SPIN that are known to be much slower than *global* on these case studies [3, 2]. Our *Incr* technique is faster than *FP* except for the Gas Station and it always consumes less memory.

[1] D-Finder is also connected to the state-space exploration tool of the BIP platform, for finer analysis when the heuristic fails to prove deadlock-freedom.

TABLE I
COMPARISON FOR ACYCLIC TOPOLOGIES.

Component information			Time (minutes)			Memory (MB)		
scale	location	interaction	*global*	*FP*	*Incr*	*global*	*FP*	*Incr*
Gas Station								
50 pumps	2152	2000	0:50	0:17	0:49	48	53	47
100 pumps	4302	4000	2:58	0:52	1:51	76	52	47
200 pumps	8602	8000	11:34	1:55	2:26	135	65	47
400 pumps	17202	16000	47:38	3:51	5:43	270	93	76
500 pumps	21502	20000	-	4:43	7:21	-	101	86
600 pumps	25802	24000	-	5:53	9:05	-	115	97
700 pumps	30102	28000	-	7:14	11:44	-	138	107
Smoker								
300 smokers	907	903	0:07	0:07	0:07	44	11	7
600 smokers	1807	1803	0:13	0:14	0:13	46	26	8
1500 smokers	4507	4503	1:38	0:44	0:34	65	54	18
3000 smokers	9007	9003	6:21	1:57	1:14	113	86	28
6000 smokers	18007	18003	27:03	5:57	3:24	222	172	55
7500 smokers	22507	22503	41:38	8:29	4:51	270	209	60
9000 smokers	27007	27003	-	11:36	6:34	319	247	96
ATM								
50 machines	1104	902	10:49	2:20	1:23	81	86	22
100 machines	2204	1802	43:00	6:00	1:57	142	271	44
250 machines	5504	4002	-	17:16	4:46	-	670	65
350 machines	7704	6302	-	27:54	8:18	-	938	77
600 machines	13204	10802	-	-	24:14	-	-	119
Producer/Consumer								
2000 consumers	4004	4003	0:27	0:33	0:31	57	16	11
4000 consumers	8004	8003	1:27	1:18	1:05	90	28	20
6000 consumers	12004	12003	3:01	2:32	2:03	126	37	31
8000 consumers	16004	16003	5:35	4:22	2:33	164	40	35
10000 consumers	20004	20003	8:44	6:12	3:15	218	66	56
12000 consumers	24004	24003	12:06	8:37	5:38	257	75	66

TABLE II
COMPARISON BETWEEN DIFFERENT METHODS ON DINING PHILOSOPHERS

Component information			Time (minutes)			Memory (MB)		
scale	location	interaction	*global*	*FP*	*Incr*	*global*	*FP*	*Incr*
500 philos	3000	2500	4:01	9:18	0:34	61	60	29
1000 philos	6000	5000	17:09	-	2:04	105	-	60
1500 philos	9000	7500	39:40	-	3:09	148	-	74
2000 philos	12000	10000	-	-	4:14	-	-	96
4000 philos	24000	20000	-	-	8:37	-	-	192
6000 philos	36000	30000	-	-	14:26	-	-	382
9000 philos	53000	45000	-	-	24:16	-	-	581

In Table II, we also provide results on checking deadlock-freedom for the dining philosopher algorithm. Contrary to the above examples, the dining philosopher algorithm has a cyclic topology, which cannot be efficiently managed with *FP* (this is the only case for which *global* was faster than *FP*.

Our results have also been applied on a complex case study that directly comes from an industrial application. More precisely, we have been capable of checking safety and deadlock-freedom properties on the modules in the functional level of the *DALA robot* [5]. DALA is an autonomous robot with modules described in the BIP language running at the functional level. Every module is in a hierarchy of composite components.

All together the embedded code of DALA in the functional level contains more than 500 000 lines of C code. The topology of the modules and the description of the behaviors of the components are complex. This is beyond the scope of tools such as NuSMV or SPIN. We first checked deadlock properties of individual modules. Both *global* and *FP* fail to check for deadlock-freedom (Antenna is the only module that can be checked by using *global*). However, by using *Incr*, we can always generate the invariants and check the deadlock-freedom of all the modules. Table III shows the time consumption in computing invariants for deadlock-freedom checking of seven modules by the incremental method; it also gives the number of states per module. In these modules we have successively detected (and corrected) two deadlocks within Antenna and

TABLE III
DEADLOCK-FREEDOM CHECKING ON DALA BY $Incr$ METHOD

module	component	location	interaction	states	time (minutes)
SICK	43	213	202	$2^{20} \times 3^{29} \times 34$	1:22
Aspect	29	160	117	$2^{17} \times 3^{23}$	0:39
NDD	27	152	117	$2^{22} \times 3^{14} \times 5$	8:16
RFLEX	56	308	227	$2^{34} \times 3^{35} \times 1045$	9:39
Battery	30	176	138	$2^{22} \times 3^{17} \times 5$	0:26
Heating	26	149	116	$2^{17} \times 3^{14} \times 145$	0:17
Platine	37	174	151	$2^{19} \times 3^{22} \times 35$	0:59

NDD, respectively.

Aside from the deadlock-freedom requirement, some modules also have safety property requirements such as causality (a service can be triggered only after a certain service has been running successfully, i.e., only if the variable corresponding to this service is set to true). In checking the causality requirement between different services, we need to compute invariants according to different causality requirement. Inspired from the invariant preservation properties introduced in Section III, we removed some tight synchronizations between some components[2] that would not synchronize directly with the components involved in the property and obtained a module with looser synchronized interactions. As the invariant of the module with looser synchronizations is preserved by the one with tighter synchronizations, if a property is satisfied in the former, then it is satisfied in the latter. Based on this fact, we could obtain the satisfied causality property in 17 seconds, while it took 1003 seconds before using the preorder. A more detailed description of DALA and other properties verified with our $Incr$ and invariant preservation methods can be found in [4].

VI. CONCLUSION

We present new incremental techniques for computing interaction invariants of composite systems defined in the BIP framework. In addition, we propose sufficient conditions that guarantee invariant preservation when new interactions are added to the system. Our techniques have been implemented in the D-Finder toolset and have been applied to complex case studies that are beyond the scope of existing tools.

As we have seen in Section V, our new techniques and the ones in [3, 6] are complementary. As a future work, we plan to set up a series of new experiments to give a deeper comparison between these techniques. This should help the user to select the technique to be used depending on the case study. Other future works include to extend our contribution to liveness properties and abstraction.

Acknowledgment. We are grateful to the reviewers for their careful work and their valuable and insightful comments and suggestions.

REFERENCES

[1] A. Basu, M. Bozga, and J. Sifakis. Modeling heterogeneous real-time components in BIP. In *Proceedings of the Fourth IEEE International Conference on Software Engineering and Formal Methods*, pages 3–12, Washington, DC, USA, 2006. IEEE Computer Society.

[2] S. Bensalem, M. Bozga, T.-H. Nguyen, and J. Sifakis. D-Finder: A tool for compositional deadlock detection and verification. In *Proceedings of the 21st International Conference on Computer Aided Verification*, pages 614–619, Berlin, Heidelberg, 2009. Springer-Verlag.

[3] S. Bensalem, M. Bozga, J. Sifakis, and T.-H. Nguyen. Compositional verification for component-based systems and application. In *Proceedings of the 6th International Symposium on Automated Technology for Verification and Analysis*, pages 64–79, Berlin, Heidelberg, 2008. Springer-Verlag.

[4] S. Bensalem, L. de Silva, M. Gallien, F. Ingrand, and R. Yan. "Rock solid" software: A verifiable and correct by construction controller for rover and spacecraft functional layers. In *Proceedings of the 10th International Symposium on Artificial Intelligence, Robotics and Automation in Space*, 2010.

[5] S. Bensalem, M. Gallien, F. Ingrand, I. Kahloul, and T.-H. Nguyen. Toward a more dependable software architecture for autonomous robots. *IEEE Robotics and Automation Magazine*, 16(1):1–11, 2009.

[6] S. Bensalem, A. Legay, T.-H. Nguyen, J. Sifakis, and R. Yan. Incremental invariant generation for compositional design. In *Proceedings of the 4th IEEE International Symposium on Theoretical Aspects of Software Engineering*, 2010.

[7] BIP. http://www-verimag.imag.fr/BIP,196.html?

[8] M. R. V. Chaudron, E. M. Eskenazi, A. V. Fioukov, and D. K. Hammer. A framework for formal component-based software architecting. In *Proceedings of Specification and Verification of Component-Based Systems Workshop*, pages 73–80, 2001.

[9] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri. NuSMV: a new symbolic model checker. *International Journal on Software Tools for Technology Transfer*, 2:410–425, 2000.

[10] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model checking*. The MIT Press, 1999.

[11] B. Dutertre and L. de Moura. A fast linear-arithmetic solver for DPLL(T). In *Proceedings of the 18th Computer-Aided Verification conference*, volume 4144 of *LNCS*, pages 81–94. Springer-Verlag, 2006.

[12] D. Heimbold and D. Luckham. Debugging Ada tasking programs. *IEEE Softw.*, 2(2):47–57, 1985.

[13] S. S. Patil. *Limitations and Capabilities of Dijkstra's Semaphore Primitives for Coordination among Processes*. Cambridge, Mass.: MIT, Project MAC, Computation Structures Group Memo 57, Feb, 1971.

[14] J.-P. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *Proceedings of the 5th Colloquium on International Symposium on Programming*, pages 337–351. Springer-Verlag, 1982.

[15] F. Somenzi. CUDD: CU decision diagram package.

[16] O. Team. The Omega library, 1996.

[2] The latter can be seen as an abstraction of the component in where some services have been removed.

Verifying Shadow Page Table Algorithms

Eyad Alkassar*, Ernie Cohen†, Mark Hillebrand†, Mikhail Kovalev*, and Wolfgang J. Paul*
*Saarland University, Saarbrücken, Germany
{eyad,kovalev,wjp}@wjpserver.cs.uni-saarland.de
†European Microsoft Innovation Center (EMIC GmbH), Aachen, Germany
{ecohen,mahilleb}@microsoft.com

***Abstract*—Efficient virtualization of translation lookaside buffers (TLBs), a core component of modern hypervisors, is complicated by the concurrent, speculative walking of page tables in hardware. We give a formal model of an x64-like TLB, criteria for its correct virtualization, and outline the verification of a virtualization algorithm using shadow page tables. The verification is being carried out in VCC, a verifier for concurrent C code.**

I. Introduction

Virtual addressing is the most common way for a *host* program (typically an OS or hypervisor), to virtualize the memory of a *guest* program. In a typical implementation, the translation from virtual addresses (VAs) to physical addresses (PAs) is controlled by page tables (PTs) in memory; the hardware concurrently walks these page tables, setting accessed (A) and dirty (D) bits in the page table entries (PTEs) as appropriate, and caching the translations in a translation lookaside buffer (TLB). When the guest addresses memory, the processor uses the TLB to translate virtual to physical addresses. If a suitable translation is not available, a hardware page fault (#PF) throws control to the host, giving it an opportunity to intercede. The processor automatically flushes the TLB in certain circumstances (e.g., on an address space switch), but it is generally up to the host to manage the coherency of the TLB.

Virtual addressing does not in itself provide correct virtualization for guests that edit their own page tables, such as operating systems. A standard solution[1] to this problem is to control guest address translation using a separate set of *shadow page tables* (SPTs), invisible to the guest, each of which "shadows" one of the guest page tables (GPTs). When a guest memory access results in a #PF, the host #PF handler walks the GPTs (simulating the TLB hardware), setting A and D bits in the GPT entries, and caching the translation (perhaps with an additional level of translation) in the SPTs. This allows the hardware to subsequently walk the SPTs to cache the SPT translations into the hardware TLB (HTLB). Thus, the SPTs, the #PF handler, and the HTLB act in concert to provide a virtual TLB (VTLB) to the guest.

Because walking the page tables slows down program execution, high-performance memory managers running in a guest are often very aggressive in their use of the TLB, flushing translations only when absolutely necessary, allowing the TLB to cache stale translations to some extent, with its correctness depending on fine details of the TLB semantics (e.g., exactly when A bits are set). High performance hypervisors are equally aggressive in flushing translations from the SPTs and HTLB only when necessary; for example, the SPT algorithm in the Hyper-V™ [3] hypervisor shares SPTs between different processors and address spaces, and selectively write-protects GPTs from guest edits to keep them in sync with their SPTs (so that they don't have to be flushed on a guest address-space switch) [4]. This combination makes SPT algorithms difficult to test (particularly since an error is likely to manifest in a guest failure long after the SPT entry leading to it has been flushed); for example, one bug in the aforementioned SPT algorithm required seven thread switches to manifest. This makes SPT algorithms an ideal target for formal verification.

We describe the verification of a simple shadow page table algorithm. We formulate the main invariants and present a verification pattern in VCC, an automatic verification environment for concurrent C code (available at http://vcc.codeplex.com/).

II. TLB Virtualization Problem

A. Hardware Model

The type of n-bit strings $\{0,1\}^n$ is denoted by $\mathbb{B}_n$. We interpret a string $a \in \mathbb{B}_{64}$ either as a 64-bit string, a natural number, or a PTE. We consider a word (64 bits) addressable memory, 45-bit long VAs, and PAs 49 bits long. We call the top-most 36 bits (for the VAs) or 40 bits (for the PAs) the page frame number (PFN). We decompose a virtual address $a \in \mathbb{B}_{45}$ into page table indices $a.px[i]$ for $i \in [1:4]$ of 9 bits each and a 9-bit physical page displacement $a.px[0]$.

An x64 multi-core/multi-processor machine is modeled with the record $h :: x64conf$, where $h.p[i]$ denotes the hardware configuration of the processor i, and $h.mm :: \mathbb{B}_{49} \mapsto \mathbb{B}_{64}$ represents the shared memory of the system. A processor configuration consists of a register $CR3$ giving the address of the root PT, a (processor local) TLB tlb, and an uninterpreted variable $state$ encapsulating the rest of the processor state. A PT consists of 512 PTEs, each being a struct with five fields: the SPT page frame number pfn at which the entry is pointing, accessed and dirty bits a and d, a present bit p, and the set of access rights r (e.g., writing access $r[rw]$). We define the less-or-equal operator on set of rights as $r1 \leq r2 = \forall j.r1[j] \leq r2[j]$.

The TLB state is modeled as a set of page table *walks*, each of which summarizes a partial or complete traversal of

[1]Recent Intel and AMD processors provide a hardware alternative, in the form of an extra address translation layer not visible to the guest OS [1], [2].

TABLE I: Semantics of the Abstract TLB

Transition name	**Guard on the configuration** h	**Resulting configuration** h'
$createw_{tlb}(i, va, r)$		$h/\{.p[i].tlb = h.p[i].tlb \cup \{winit(h.p[i].CR3, va, r)\}$
$deletew_{tlb}(i, w)$	$w \in h.p[i].tlb$	$h/\{.p[i].tlb = h.p[i].tlb \setminus \{w\}\}$
$extendw_{tlb}(i, w)$	$w \in h.p[i].tlb \wedge pte(h, w).a \wedge pte(h, w).p$ $\wedge (pte(h, w).d \vee w.l > 1 \vee \neg w.r[rw]) \wedge w.l > 0$	$h/\{.p[i].tlb = h.p[i].tlb \cup \{wext(h, w)\}\}$
$setaccess_{tlb}(i, w)$	$w \in h.p[i].tlb \wedge w.l > 0 \wedge pte(h, w).p$	$h/\{.mm[w.pfn][w.va.px[w.l]].a = 1\}$
$setdirty_{tlb}(i, w)$	$w \in h.p[i].tlb \wedge w.l = 1 \wedge \neg fault(h, w)$ $\wedge w.r[rw] \wedge pte(h, w).r[rw] \wedge pte(h, w).a$	$h/\{.mm[w.pfn][w.va.px[w.l]].d = 1\}$
$mov2cr3_{cpu}(i, pto)$	$h.p[i].tlb = \emptyset \wedge instr(h.p[i]) = mov2cr3$	$h/\{.p[i].state = step(h, i), .p[i].CR3 = pto\}$
$invlpg_{cpu}(i, va)$	$h.p[i].tlb \cap \{w \mid w.va = va \vee w.l > 0\} = \emptyset$ $\wedge instr(h.p[i].state) = invlpg$	$h/\{.p[i].state = step(h, i, va)\}$
$transl\_ok_{cpu}(i, va, r, pa)$	$w \in h.p[i].tlb \wedge w.va = va \wedge w.l = 0$ $\wedge r \leq w.r \wedge instr(h.p[i]) = mem\_instr$	$h/\{.p[i].state = step(h, i, pa)\} \wedge pa = w.pfn \circ va.px[0]$
$transl\_pf_{cpu}(i, va, r, f)$	$w \in h.p[i].tlb \wedge w.va = va \wedge fault(h, w)$ $\wedge r \leq w.r \wedge instr(h.p[i]) = mem\_instr$	$h/\{.p[i].state = step(h, i, f)\} \wedge f$

the page tables for a given VA. Each walk is given by a virtual address va, a level l giving the number of page table levels remaining to be walked,[2] the page frame number pfn of the next page table to be used for translation, and a set r of access rights giving all rights not denied by the walk gathered thus far. A walk is *complete* if its level is 0, and *partial* otherwise.

The function $winit(pfn, va, r)$ returns a walk with level 4 and the other components initialized according to the given parameters. The extension $wext(h, w)$ of a walk w in the hardware configuration h is defined as follows ($s/\{.c = v\}$ denotes update of field c of struct s to value v):

$$
\begin{aligned}
&pte(h, w) = h.mm[w.pfn \circ w.va.px[w.l]]\\
&fault(h, w) = \neg pte(h, w).p \vee \ \neg(w.r \leq pte(h, w).r)\\
&wext(h, w) =\\
&\quad w/\{\ .pfn \ \ = pte(h, w).pfn,\\
&\qquad .l \ \ \ \ \ = w.l - 1,\\
&\qquad .r \ \ \ \ \ = \lambda i.\ w.r[i] \wedge pte(h.w).r[i]\ \}
\end{aligned}
$$

A complete walk can be used to address memory iff the walk's VA matches the requested VA and the walk provides rights (write, execute, etc.) at least equal to those requested. A #PF can be generated only from a partial walk leading to a PTE that is non-present or provides insufficient rights.

Table I gives the behavior of our TLB model, expressed as a transition relation on the hardware configuration h. For some operations we introduce additional parameters, such that virtual address va, physical address pa, and #PF flag f in case of a CPU address translation. While the first five actions (indexed with tlb) model autonomous behavior of the TLB, the last four actions (indexed with cpu) abstractly model the CPU behavior, using the uninterpreted function $step()$ to update the CPU state. Note also that operations such as `INVLPG` that flush the TLB are modeled instead as blocking when the TLB contains offending entries; these models are equivalent because the TLB is allowed to delete walks at any time.

[2]To simplify the presentation, we do not consider large pages and legacy addressing modes here, so each complete walk goes through exactly four page tables. Also, we do not consider tagged TLBs or global page translations.

B. Correctness Criteria

A hypervisor provides to each guest the illusion of running on its own private memory, processors and TLBs. In the following we provide this illusion for a single guest,[3] modeled as a virtual machine $g :: x64conf$.This guest g is implemented on a single host machine running the hypervisor code, linked to this implementation by a coupling invariant. Hypervisor correctness is established by proving that execution of the host machine preserves this invariant, and that g behaves accordingly—in particular, that (i) the VTLBs $g.p[j].tlb$ of the guests satisfy the transition relation of Table I, and (ii) that any virtual memory access of this virtual processor is justified by a complete walk in its VTLB.

Given the transitive closure $\rightarrow^*_{tlb}$ of permissible TLB steps as defined in Table I, we can formulate the first property in form of forward simulation:

Invariant 1: Let h and h' be pre and post states of a host step, and g and g' be the abstracted guest machine states respectively. Then the changes to the TLB of any virtual processor (VP) j form a valid TLB transition:

$$g.p[j].tlb \rightarrow^*_{tlb} g'.p[j].tlb \tag{1}$$

To formulate the second invariant we need to introduce parts of the coupling invariant. The function $vp2hp(j)$ defines for a VP j on which host processors it is currently scheduled to run. The memory of the guest is mapped to some region of the shared memory of the hardware machine. Then, the memory mapping is defined by the injective function $gpa2hpa :: \mathbb{B}_{40} \mapsto \mathbb{B}_{40}$, which maps a guest physical PFN into the host physical PFN.

The second invariant establishes a relation between the VTLB and the implementation model. The walks contained in the HTLB should be present in the VTLB with respect to the guest memory projection. A function $hw2gw(w)$ translates a complete host walk w into a respective guest walk applying the $gpa2hpa^{-1}$ mapping to the field $w.pfn$ and leaving the other fields of the walk w unchanged.

[3]This can be easily generalized to multiple guests mapped to disjoint host memory portions.

The VTLB is given by the result of an abstraction function on the host configuration. In this case it will usually contain more complete walks than the HTLB. Nevertheless, every complete walk present in the HTLB should correspond to a complete walk in the VTLB.

Invariant 2: Let a complete walk w be present in the HTLB. Then a VTLB contains the walk $hw2gw(w)$.

$$w \in h.p[i].tlb \wedge w.l = 0 \implies \exists j.\ vp2hp(j) = i \wedge hw2gw(w) \in g.p[j].tlb \quad (2)$$

In order to infer the second invariant after a TLB step we also need an invariant dealing with partial walks in the HTLB. The statement of this invariant depends on the definition of the VTLB. Note, that the VTLB is not obliged to store partial walks, because their presence in the TLB is not mandatory.

III. SPT Algorithm

In this section we describe a basic implementation of a shadow page table algorithm, define the abstracted VTLBs, and state the required invariants to prove the correctness criteria from the previous section.

A. Overview on the Implementation

We assume that the $gpa2hpa$ map is static, and that appropriate data structure and functions are given to store and query it. The SPTs are located in an array $\texttt{SPT}[0:(n-1)]$. We define the functions $i2a :: \mathbb{N} \mapsto \mathbb{B}_{40}$ to return the host PFN of the SPT stored in each array element, and the function $a2i :: \mathbb{B}_{40} \mapsto \mathbb{N}$ as the inverse function on these PFNs. Every SPT may either be free or in use by a single VP j (more precisely the HTLB of the host processor it runs on). For each VP j we let $gwo(j)$ denote the current value of its CR3 register, and $hwo(j)$ denote the CR3 used on the host when the guest is actually running. The latter CR3 designates the top-level SPT used for VP j in the SPT array. We organize the SPTs for each VP as a tree of SPTs, and assign to each SPT a level ranging from 4 (top-level) to 1 (terminal) and a VA range for the addresses of the walks that might go through to this SPT (*prefix* of the SPT). The entries of non-terminal SPTs point to other SPTs, while the entries of terminal SPTs point to memory of the guest (under the $gpa2hpa$ map). The predicate $walks\_to(i, px, j)$ denotes that SPT with index i points to SPT j.

$$walks\_to(i, px, j) = (\texttt{SPT}[i][px].pfn = i2a(j))$$

Guest instructions and exceptions that operate on the TLBs are intercepted so that they can be virtualized in the SPTs.

Every SPT has an additional Page Table Info (PTI) data structure associated with it, which keeps auxiliary information about SPTs. The fields of $\texttt{PTI}[i]$ include *gpfn* (the guest physical PFN of the GPT) and l (the level of the SPT).

The algorithm maintains the SPTs by handling the following intercepts:

1) Flushing/Switching of CR3: Flushes of the guest (e.g., by executing `mov2cr3`) are intercepted by the hypervisor. The intercept is handled by freeing all the VP's SPTs, allocating a fresh top-level SPT (which has all its entries set to non-present), and executing an HTLB flush.

2) #PF intercept: When a host #PF is intercepted, the hypervisor walks the GPTs to determine the reason for the fault. If a GPTE is reached that has insufficient rights for the page-faulting operation or has the present bit not set, a page fault is injected into the guest (and the hypervisor returns). Simultaneously with walking the GPTs, the hypervisor also walks the associated SPTs down from the top-level SPT. If a non-present (non-terminal) SPTE is encountered during the walk, we update the SPTE to point to a newly allocated, zero-filled SPT; the new SPT shadows the GPT referenced by the corresponding GPTE. For a present SPTE we check whether rights and PFN still correspond to the GPTE. If not, the old SPT subtree is detached (and a hardware `INVLPG` executed on the faulty VA) before allocating, initializing, and pointing to a new SPT as before. A GPTE's A bit is set when the #PF handler walks it; A bits in SPTEs are always set. Note that all not dirty terminal SPT entries are kept write protected to propagate a D bit to the guest before it is set by the HTLB.

3) `INVLPG` intercept: The implementation walks down the SPTs for the `INVLPG` address and, when reaching a terminal SPTE, marks it non-present. Then it performs a hardware `INVLPG` on the faulty VA.

B. VTLB Abstraction

To define the virtual TLB abstraction and verify the invariants we introduce *ghost fields* in the PTI structure, which are used only for verification but are ignored by the implementation. The field $vpid$ stores the index of the VP using the associated SPT, the field $vpfn$ stores the SPT's prefix, the field r stores the accumulated rights from the top-level SPT to the given SPT, the reachability bit re distinguishes whether the HTLB can walk the SPT, and for terminal SPTs, the array $gp[0:511]$ in the PTI stores the *ghost present* bits of the terminal PTEs denoting whether the complete walks through the PTEs might be present in the HTLB.

Next, we define the coupling invariant. Every guest component is abstracted from the hardware machine h. General purpose registers of the VP j are either loaded into the hardware registers of the processor $vp2hp(j)$ or are stored in some implementation data structure.

We define the VTLB as the set of walks corresponding to the complete walks that might be cached by HTLB. Formally, we use the ghost fields of the PTI data structure to construct the set $walks(i)$, containing walks *sitting* on the SPT i.

$$walks(i) = \{w \mid w.r \leq \texttt{PTI}[i].r \wedge w.pfn = i2a(i) \wedge w.l = \texttt{PTI}[i].l \wedge \forall j \in [\texttt{PTI}[i].l + 1 : 4].\ w.va.px[j] = \texttt{PTI}[i].vpfn[9 \cdot j - 1 : 9 \cdot (j-1)]\}$$

We define the set of indices of the terminal SPTs belonging to the VP j by

$$tSPT(j) = \{i \mid \texttt{PTI}[i].l = 1 \wedge \texttt{PTI}[i].vpid = j\}.$$

The set of complete walks of the VP j is defined as follows:

$$cwalks(j) = \{wext(h, w) \mid w \in walks(i) \wedge i \in tSPT(j) \wedge \texttt{PTI}[i].gp[w.va.px[0]] \wedge w.r \leq \texttt{SPT}[i][w.va.px[0]].r\}$$

The VTLB is defined as a translation of complete shadow walks into the respective walks over the GPTs:

$$g.p[j].tlb = \{hw2gw(w) \mid w \in cwalks(j)\}$$

Note, that the VTLB definition does not use the implementation SPTE present bit, because some complete walks through present SPTEs may be flushed out of the HTLB by INVLPG.

C. Invariants

In this section we specify implementation dependent invariants used to prove Invariants 1 and 2.

We introduce the notion of *reachable* SPTs, to mark those SPTs which may have been walked by the HTLB since the last flush. Thus we store where partial HTLB walks may reside.

We maintain reachability using the ghost flag $\texttt{PTI}[i].re$ and the following invariants.

$$\forall j.\ \texttt{PTI}[a2i(gwo(j))].re \tag{3}$$

$$\texttt{PTI}[i].re \wedge walks\_to(i, px, i') \implies \texttt{PTI}[i'].re \tag{4}$$

$$w \in h.p[k].tlb \wedge w.l > 0 \implies w \in walks(a2i(w.pfn)) \wedge \texttt{PTI}[a2i(w.pfn)].re \tag{5}$$

$$\texttt{PTI}[i].re \wedge \texttt{SPT}[i][px].p \implies \texttt{PTI}[i].gp[px] \tag{6}$$

Invariant 3 states that the top level SPT is always reachable. Invariant 4 states that if a SPT is reachable then its descendants in the SPT tree are also reachable. Invariant 5 states about the partial walks in the HTLB, namely that all partial walks in the HTLB are sitting on the reachable SPTs. Invariant 6 states a connection between the reachable bit of the terminal SPT and the ghost present bits of the terminal SPTEs. It states that if a present bit in the SPTE of a reachable terminal SPT is set, then the ghost present bit for this entry is also set. Note, that for maintaining the invariants, whenever we detach a shadow subtree we have to perform a hardware INVLPG and reset the reachability bits for the SPTs in the subtree.

IV. Verification

To verify the C implementation of the algorithm we use VCC, a deductive verifier for concurrent C code.VCC extends C with ghost data (possibly of non-C types, such as mathematical integers and maps), ghost functions, function contracts, and (2-state) data invariants that constrain how fields of a "valid" object are allowed to change in a legal system step. The use of 2-state invariants both allows us to model abstract automata (like TLBs) and to prove forward simulations internally (via code annotation), rather than at the meta-level.

In VCC, the TLB state and its transition relation can be specified as a ghost type and a ghost predicate. These are used at two places: (i) we represent the HTLB as a C struct type with a field of this type storing its contents and the TLB transition relation used as a 2-state invariant on this field. (ii) we use the TLB transition relation on the abstracted VTLB state as a 2-state invariant of the SPT structures, obliging VCC to show that SPT updates satisfy TLB semantics.

Next, we (very briefly) sketch the correctness arguments for INVLPG handling and HTLB steps.

1) INVLPG intercept: The INVLPG intercept handler with annotations is shown below:

```
Walk ws[4]; Pte pte;
ws[4] = initwalk(gwo[j], va, r); y = 4;
while (ws[y].l && !ws[y].f) {
  atomic (SPT) { /* atomic read */
    pte = SPT[a2i(ws[y].pfn)][px(va,y)]; }
  ws[y−1] = wext(pte, ws[y]);
  y = y−1; }
if (l == 0)
  atomic (SPT) { /* atomic write */
    SPT[a2i(ws[1].pfn)][px(va,0)].p = 0; }
asm_invlpga(va);
spec( /* ghost code */
  for(k = 0; k < n; k++) {
    if (PTI[k].l == 1 && PTI[k].vpfn == pfn(va) && PTI[k].vpid == j)
      PTI[k].gp[px(va,0)] = 0; } )
```

The ghost construct ***atomic***(o) checks that the subsequent block of statements is being executed atomically by the implementation, with no updates other than on volatile fields of the designated object o. The ***spec***(...) wraps regular ghost code—in our case the code that resets the ghost present bits for the invalidated VA by iterating over all terminal SPTs. Since the VTLB abstraction depends on both ghost present bits and terminal SPT entries, the above ghost and implementation code implicitly alters the VTLB content.

The intercept handler emulates the following guest steps preserving Invariant 1: (i) for each $w \in g.p[j].tlb$ such that $w.va = va$ do $deletew_{tlb}(j, w)$, and (ii) $invlpg_{cpu}(j, va)$.

The other invariants hold due to the fact that after the hardware INVLPG is performed every walk belonging to HTLB is a walk that was sitting there before the intercept happened.

2) Hardware TLB steps: The HTLB can add walks, extend walks, remove walks, set A and D bits. The VTLB in this case remains unchanged. Invariant 5 follows from Invariants 3 and 4. Invariant 2 follows from Invariants 5 and 6, and the VTLB definition, specifically from the fact that VTLB contains all complete walks over the SPTs accessible by the VP.

The verification of the complete SPT algorithm in VCC is an ongoing effort.

Acknowledgment

This work was partially funded by the German Federal Ministry of Education and Research (BMBF) in the Verisoft XT project under grant 01 IS 07 008. Work of the third author done while at DFKI GmbH, Saarbrücken, Germany.

References

[1] *Intel 64 and IA-32 Architectures Software Developer's Manual – Volume 3B*, Intel Corporation, June 2009.

[2] *AMD64 Architecture Programmer's Manual Volume 2: System Programming*, 3rd ed., Advanced Micro Devices, Sep. 2007.

[3] Microsoft Corp., "Windows Server 2008 R2 – virtualization with Hyper-V," http://www.microsoft.com/windowsserver2008/en/us/hyperv-main.aspx, 2008.

[4] J. T.-J. Sheu, M. D. Hendel, L. Wang, E. S. Cohen, R. A. Vega, and S. A. Nanavati, "Reduction of operational costs of virtual TLBs," U.S. Patent 20 080 134 174, Jun. 5, 2008.

Impacting Verification Closure using Formal Analysis

Massimo Roselli
Cadence Design System
20089 Rozzano (Milan), Italy
mroselli@cadence.com

***Abstract*—Formal Analysis has been living in its own world. Its impact on the mainstream simulation was limited. In particular the results and metrics generated by Formal could not be factored into the simulation flow. In IEV we include a mechanism to translate formal result into simulation results and therefore enable contributions from the Formal Analysis effort to be accounted for in the simulation flow. In the demonstration we show the effects of this translation and how it can help to improve simulation coverage.**

Scalable and Precise Program Analysis at NEC

Gogul Balakrishnan Malay K. Ganai Aarti Gupta Franjo Ivančić Vineet Kahlon Weihong Li
Naoto Maeda Nadia Papakonstantinou Sriram Sankaranarayanan* Nishant Sinha Chao Wang

NEC Laboratories America, NJ, USA
http://www.nec-labs.com/research/system/systems_SAV-website

*University of Colorado Boulder, CO, USA

***Abstract*—The Systems Analysis & Verification Department at NEC Labs engages in foundational as well as applied research in the areas of verification and analysis of software and embedded systems. We have developed several tools and frameworks for scalable and precise analysis of programs, some of which are now used within the company on large software projects. This extended abstract highlights their main features and provides pointers to published papers with more details.**

I. F-SOFT

F-Soft is a platform for verifying source code programs [3], [4]. It can check C programs for runtime errors (such as pointer access violations, buffer overflow, memory leaks), standard library API usage, and other user-defined assertions. It can be used to check programs globally, or as an annotation checker to check user-written contracts. We have successfully analyzed large benchmark examples to find previously unknown bugs using F-Soft. An in-house product based on F-Soft, called VARVEL, is currently in use in NEC. We are now extending F-Soft to handle C++ programs.

Overall, F-Soft provides a cooperative, staged framework of various static analyses (including abstract interpretation) and model checking techniques, where the size and complexity of the model and the number of properties are successively reduced across the stages. This enables efficient use of high-precision analyses when needed, and is very effective for handling large programs with hundreds of (automatically instrumented) properties.

II. COBE (CONCURRENCY BENCH)

CoBe is a tool for finding concurrency-related bugs in multi-threaded C programs, such as data races, deadlocks, atomicity violations, and missed notifies. It leverages a combination of static analysis, dataflow analyses, and symbolic model checking.

CoBe starts by using statically computed lockset and lock acquisition history information [7], [5], as well as happens-before constraints induced by synchronization primitives and properties [9] to generate bug warnings. Each warning is then analyzed by employing a series of static analyses in a 'telescoping' fashion, i.e., in increasing order of precision but decreasing order of scalability to decide, as cheaply as possible, whether the warning is bogus. These static analyses exploit both static constraints, i.e., interleaving constraints arising from the use of synchronization primitives like Wait/Notify (rendezvous), Wait/NotifyAll (broadcasts), etc., and semantic constraints resulting from data flow. The semantic constraints are generated by deriving sound invariants, using abstract interpretation on domains with increasing precision (range, octagonal, and polyhedral analyses). These two classes of analyses, when used in conjunction, can weed out a large fraction of the bogus warnings [8]. Finally, symbolic model checking is leveraged to generate concrete error traces for the remaining warnings [6]. Telescoping greatly enhances the efficacy of the overall process by ensuring scalability without compromising on precision. We have successfully handled many large examples, including Linux device drivers.

III. FUSION

Fusion is a platform for combining dynamic and static verification techniques, for the purpose of finding bugs in concurrent C/C++ programs. Given a test case, it executes the program to derive a *Concurrent Trace Program (CTP)* that captures that trace. The CTP can be used for exploration of alternate thread schedules, which is difficult to do in standard testing where the thread schedule is controlled by the operating system. Specifically, we have used CTPs to combine an explicit search (in the style of dynamic partial order reduction) with SMT-based analysis to improve its performance and coverage [11]. We have also used CTPs for symbolic predictive analysis, where we can detect assertion violations or atomicity violations in alternate interleavings of the same events observed in the given trace [12], [13].

For long traces, exploring all interleavings of the events can be prohibitively expensive. We are investigating additional static analysis techniques that can quickly prune away some warnings [9]. We are also studying coverage-based metrics to guide systematic testing to explore only high-risk interleavings that are likely to lead to bugs.

IV. CONTESSA

Testing of multi-threaded programs poses enormous challenges. To improve the coverage of testing, we present a framework named CONTESSA [10] that augments conventional testing (concrete execution) with symbolic analysis in a scalable and efficient manner to explore both thread interleaving and input data space. It works on CTPs generated by Fusion. It utilizes partial-order reduction techniques [1] to generate verification conditions with reduced size and search

space [2]. These verification conditions are checked by an SMT-solver that can generate witness traces for bugs. The tool also provides visual support for debugging the witness traces.

ACKNOWLEDGMENT

We would like to gratefully acknowledge the support and efforts of the Varvel Development team and the Software Development Environment Engineering Division team in NEC Japan.

REFERENCES

[1] Malay K. Ganai and Sudipta Kundu. Reduction of verification conditions for concurrent system using mutually atomic transactions. In *SPIN*, pages 68–87, 2009.

[2] Malay K. Ganai and Chao Wang. Interval analysis for concurrent trace programs using transaction sequence graphs. In *International Conference on Runtime Verification (RV 10)*, 2010.

[3] Franjo Ivancic, Ilya Shlyakhter, Aarti Gupta, Malay K. Ganai, Vineet Kahlon, Chao Wang, and Zijiang Yang. Model checking C programs using F-SOFT. In *ICCD*, pages 297–308, 2005.

[4] 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.

[5] Vineet Kahlon. Boundedness vs. unboundedness of lock chains: Characterizing decidability of pairwise cfl-reachability for threads communicating via locks. In *LICS*, pages 27–36, 2009.

[6] Vineet Kahlon, Aarti Gupta, and Nishant Sinha. Symbolic model checking of concurrent programs using partial orders and on-the-fly transactions. In *CAV*, pages 286–299, 2006.

[7] Vineet Kahlon, Franjo Ivancic, and Aarti Gupta. Reasoning about threads communicating via locks. In *CAV*, pages 505–518, 2005.

[8] Vineet Kahlon, Sriram Sankaranarayanan, and Aarti Gupta. Semantic reduction of thread interleavings in concurrent programs. In *TACAS*, pages 124–138, 2009.

[9] Vineet Kahlon and Chao Wang. Universal causality graphs: A precise happens-before model for detecting bugs in concurrent programs. In *CAV*, pages 434–449, 2010.

[10] Sudipta Kundu, Malay K. Ganai, and Chao Wang. Contessa: Concurrency testing augmented with symbolic analysis. In *CAV*, pages 127–131, 2010.

[11] Chao Wang, Swarat Chaudhuri, Aarti Gupta, and Yu Yang. Symbolic pruning of concurrent program executions. In *ESEC/SIGSOFT FSE*, pages 23–32, 2009.

[12] Chao Wang, Sudipta Kundu, Malay K. Ganai, and Aarti Gupta. Symbolic predictive analysis for concurrent programs. In *FM*, pages 256–272, 2009.

[13] Chao Wang, Rhishikesh Limaye, Malay K. Ganai, and Aarti Gupta. Trace-based symbolic analysis for atomicity violations. In *TACAS*, pages 328–342, 2010.

Achieving Earlier Verification Closure Using Advanced Formal Verification

Michael Siegel
OneSpin Solutions
Munich, Germany
michael.siegel@onespin-solutions.com

Abstract— **OneSpin Solutions presents recent advances in formal assertion-based verification (ABV) that speed-up verification closure. We present a formal verification methodology called Operational ABV, which simplifies verification planning, eases assertion writing working from timing diagrams and enables an exhaustive formal coverage analysis ensuring that no design behavior is missed during verification. The formal coverage analysis automatically uncovers holes in the verification plan, detects unverified design functionality, and identifies errors and omissions in design specifications. The approach is demonstrated using an AHB2Wishbone bus bridge and OneSpin's 360 MV formal verification tool.**

We also demonstrate how to prevent X-related bugs through new exhaustive 4-state formal analysis techniques. The use of unknown or undefined values (X's) can improve RTL verification and synthesis. But unintended X-propagation in designs can cause data corruption and breaking of control paths.

We show recent enhancement of 360 MV for 4-state-logic formal analysis, where signals can explicitly become X and Z – extending the 0/1-models commonly used in formal verification tools. This X-aware formal analysis enables an exhaustive pre-synthesis X-analysis and X-verification, detecting, e.g., unintended X-propagation caused by uninitialized registers, and ensuring safe use of X's for verification and synthesis optimization in general.

Both, Operational ABV with formal coverage analysis and 4-state-logic formal X-analysis enable faster verification closure with significantly higher verification confidence.

For more information about OneSpin Solutions and 360 MV please visit http://www.onespin-solutions.com.

PINCETTE – Validating Changes and Upgrades in Networked Software

Hana Chockler
IBM Haifa Research Lab, Haifa, Israel. hanac@il.ibm.com

***Abstract*—PINCETTE is a STREP project under the European Community's 7th Framework Programme [FP7/2007-2013]. The project focuses on detecting failures resulting from software changes, thus improving the reliability of networked software systems. The goal of the project is to produce technology for efficient and scalable verification of complex evolving networked software systems, based on integration of static and dynamic analysis and verification algorithms, and the accompanying methodology. The resulting technology will also provide quality metrics to measure the thoroughness of verification. The PINCETTE consortium is composed of the following partners: IBM Israel, University of Oxford, Universita della Svizzera Italiana (USI), Universita degli Studi di Milano-Bicocca (UniMiB), Technical Research Center of Finland (VTT), ABB, and Israeli Aerospace Industries (IAI).**

I. Overview of PINCETTE Project

The PINCETTE project targets the problem of analyzing and validating complex systems upgrades. Currently, each change usually requires an expensive revalidation of the whole system or is simply not checked thoroughly, thus potentially introducing new errors into the system design. This problem stems from the fact that the state-of-the-art testing and formal verification tools are not optimized to validate system changes and upgrades, but instead focus on a single program version only.

In PINCETTE, we aim to extend the state-of-art in several ways, as we detail below. PINCETTE will introduce pioneering technologies to enable systematic component substitutability checks focusing on various classes of upgrades that are not solved by other approaches. One of the main innovations of the PINCETTE approach is the integration of technologies that work at different abstraction levels, addressing different classes of problems and sharing a common solution framework. Another advantage of our approach for upgrade analysis is that models of the software system will be automatically extracted from either the source code (written in C or C++) or by monitoring system execution at run-time. Driven by these models, the PINCETTE paradigm will unify and employ the various system upgrade checks throughout the system lifecycle from the design phase to the system deployment. It will thus ensure strong coverage of our analysis of safety and security issues of the evolving networked systems. The results of the analysis will be accompanied by quality metrics, allowing the user to measure the degree of confidence in these results, and to estimate the need for further analysis.

The project results will be validated in the following applications, which cover Europe's power grid operation, ITER fusion reactor maintenance, and aeronautics systems:

- ABB: development and maintenance of networked high reliability software. The software is written in C++ and focuses on the areas of high-voltage and medium-voltage substation automation, as well as various automation products, e.g., motor control or low-voltage switch-gear systems, or robot controllers. The smooth operation of the system is essential for ensuring power supply to several European countries. The system is widely distributed geographically; upgrades are made to every substation separately and in a different timeframe.
- VTT: Divertor Test Platform (DTP2) is a full-scale mockup facility to verify several of the remote maintenance operations of ITER fusion reactor. At DTP2, real-time and safety critical control system for remotely operated devices is developed and validated. The control system is implemented using C, LabVIEW and IEC 61131 programming languages and is distributed across the network. The control system is expected to go through constant upgrades during the entire lifecycle of the reactor.
- IAI: development of software for operating autonomous aircrafts for environmental monitoring, specifically, development of the Multi-sensors Stabilized Electro-Optic System (MSEOS) for forest fire detection, search for missing people on the ground and in the sea, snow, and airport runway monitoring in all weather conditions, written in C. The system has installations around the world, and due to the variety of customers, tailoring and upgrades are frequently required in order to meet customers' needs. The average time for upgrades to be deployed on the whole system is 6 months, during which time different versions of the software must co-exist in the same network.

This work is partially supported by the European Community's 7th Framework Programme [FP7/2007-2013] under grant agreement no. 257647 – project PINCETTE. The authors are solely responsible for the content. It does not represent the opinion of the European Community, and the European Community is not responsible for any use that might be made of data appearing therein.

Author Index