Correct Hardware Design and Verification Methods

12th IFIP WG 10.5 Advanced Research Working Conference, CHARME 2003
L’Aquila, Italy, October 21-24, 2003
Proceedings
Preface

This volume contains the proceedings of CHARME 2003, the 12th Advanced Research Working Conference on Correct Hardware Design and Verification Methods. CHARME 2003 continues the series of working conferences devoted to the development and use of leading-edge formal techniques and tools for the design and verification of hardware and hardware-like systems.

Previous events in the ‘CHARME’ series were held in Edinburgh (2001), Bad Herrenalb (1999), Montreal (1997), Frankfurt (1995), Arles (1993) and Turin (1991). This series of meetings were organized in cooperation with IFIP WG 10.5 and 10.2. Prior meetings, stretching back to the earliest days of formal hardware verification were held under various names in Miami (1990), Leuven (1989), Glasgow (1988), Grenoble (1986), Edinburgh (1985) and Darmstadt (1984). We now have a well-established convention whereby the European CHARME conference alternates with its biennial counterpart, the International Conference on Formal Methods in Computer-Aided Design (FMCAD), which is held in even-numbered years in the USA.

CHARME 2003 took place during 21–24 October 2003 at the Computer Science Department of the University of L’Aquila, Italy. It was cosponsored by the IFIP TC10/WG10 Working Group on Design and Engineering of Electronic Systems.

The CHARME 2003 scientific program was comprised of:

- A morning Tutorial by Daniel Geist aimed at industrial and academic interchange.
- Two Invited Lectures by Wolfgang Roesner and Fabio Somenzi.
- Regular Sessions, featuring 24 papers selected out of 65 submissions, ranging from foundational contributions to tool presentations.
- Short Presentations, featuring 8 short contributions accompanied by a short presentation.

The conference, of course, also included informal tool demonstrations, not announced in the official program.

The topics in 2003 represented a change in the traditional conference repertoire. The motivation for this change was the general feeling that the tools and methodologies of the last decade have outrun their course. Specifically, hardware design today is driven to be specified in higher level of abstraction, with the advent of design languages such as SystemC and SystemVerilog. This stems from the fact that there is a definite crisis in our ability to harness the silicon that can today be manufactured on a single chip. The distinction between software and hardware is also getting blurry, since the architectures of systems-on-chips (SOCs) do not always determine up front what part of the chip’s functionality should be implemented in hardware and what part should be implemented in software as embedded code (firmware).
This situation of large silicon real estate raises many questions, and there are currently very few answers. It is up to the CHARME community to pioneer new directions in which the silicon industry should head in order to sustain the great success it has had in recent times. Our choice was to emphasize modelling and software in this conference. We hope that these will turn out to be the right choices, but only time will tell if we were right.

We are very grateful to the program committee and to all the referees for their assistance in selecting the conference papers.

Warm recognition is due to Giuseppe Della Penna, Benedetto Intrigila and Igor Melatti for taking care of the CHARME 2003 organization.

Special thanks are due to Giuseppe Della Penna for the CHARME 2003 Web, flier and poster design, as well as for taking care of too many aspects of the CHARME 2003 organization to mention them all.

IBM Labs in Haifa took care of printing and mailing CHARME 2003 fliers. We are grateful to Ms. Tamar Yogev for assisting us in this effort.

The organizers are very grateful to IBM, INTEL, the University of L’Aquila, and Regione Abruzzo, whose sponsorship made a significant contribution to financing the event.

Warm recognition is due to the technical support team. Markus Bajohr at the University of Dortmund together with Martin Karusseit of METAFrame Technologies who provided invaluable assistance to all the people using the online service during the crucial months preceding the conference.

Finally, we are grateful to Ms. Anna Kramer and to all the Springer LNCS editorial team for their first-class support during the preparation of this volume.

October 2003

Daniel Geist and Enrico Tronci
CHARME 2003 was organized by the Department of Computer Science, University of L’Aquila.

Executive Committee

Conference Chair: Enrico Tronci (University of Rome, Italy)
Program Chair: Daniel Geist (IBM, Israel)
Organizing Chair: Benedetto Intrigila (University of L’Aquila, Italy)
Publicity Chairs: Giuseppe Della Penna (University of L’Aquila, Italy)
Igor Melatti (University of L’Aquila, Italy)

Program Committee

Alan Hu (British Columbia) Ken McMillan (Cadence)
Alan Mycroft (Cambridge) Laurence Pierre (Marseille)
Anna Slobodova (Intel) Limor Fix (Intel)
Armin Biere (Swiss F.I. of Tech.) Mark Aagaard (Waterloo)
Byron Cook (Microsoft) Mary Sheeran (Chalmers)
Carl Pixley (Synopsys) Moshe Vardi (Rice)
Daniel Geist (IBM) Ofer Strichman (Carnegie-Mellon)
Dominique Borrione (Grenoble) Steve Johnson (Indiana)
Eli Singerman (Intel) Thomas Kropf (Bosch)
Enrico Tronci (Rome) Tiziana Margaria (Dortmund)
Ganesh Gopalakrishnan (Utah) Tom Melham (Oxford)
Hans Eveking (Darmstadt) Warren Hunt (Texas at Austin)
John O’Leary (Intel)

Referees

M. Aagaard N. Een A. Gupta
G. Al Sammane H. Eveking J. Harrison
S. Ben-David D. Fisman A. Hu
A. Biere L. Fix W. Hunt
D. Borrione R. Fraer S. Johnson
M. Boubekeur D. Geist R. Jones
K. Claessen R. Gertch G. Kamhi
B. Cook L. Gluhovsky S. Keidar
E. Dumitrescu G. Gopalakrishnan D. Kroening
VIII Organization

T. Kropf
T. Margaria
K. McMillan
T. Melham
M. Müller-Olm
M. Moulin
A. Mycroft
Z. Nevo
O. Niese
N. Nnarasimhan

J. O’Leary
J. Ouaknine
L. Pierre
C. Pixley
I. Rabinovitz
O. Rüthing
J. Schmaltz
O. Shacham
M. Sheeran
E. Singerman

A. Slobodova
B. Steffen
O. Strichman
M. Theobald
E. Tronci
M. Vardi
J. Yang
K. Yorav
E. Zarpas

Sponsoring Institutions

University of L’Aquila
Regione Abruzzo
IBM Corporation
Intel Corporation
Table of Contents

Invited Talks

What Is beyond the RTL Horizon for Microprocessor and System Design? ......................................................... 1
Wolfgang Roesner

The Charme of Abstract Entities ........................................ 2
Fabio Somenzi

Tutorial

The PSL/Sugar Specification Language
A Language for all Seasons ........................................ 3
Daniel Geist

Software Verification

Finding Regularity: Describing and Analysing Circuits That Are Not Quite Regular ........................................ 4
Mary Sheeran

Predicate Abstraction with Minimum Predicates .................... 19
Sagar Chaki, Edmund Clarke, Alex Groce, Ofer Strichman

Efficient Symbolic Model Checking of Software Using Partial Disjunctive Partitioning ........................................ 35
Sharon Barner, Ishai Rabinovitz

Processor Verification

Instantiating Uninterpreted Functional Units and Memory System:
Functional Verification of the VAMP .................................. 51
Sven Beyer, Chris Jacobi, Daniel Kröning, Dirk Leinenbach,
Wolfgang J. Paul

A Hazards-Based Correctness Statement for Pipelined Circuits ................................................................. 66
Mark D. Aagaard

Analyzing the Intel Itanium Memory Ordering Rules Using Logic Programming and SAT ........................................ 81
Yue Yang, Ganesh Gopalakrishnan, Gary Lindstrom, Konrad Slind
Automata Based Methods

On Complementing Nondeterministic Büchi Automata ...................... 96
  Sankar Gurumurthy, Orna Kupferman, Fabio Somenzi, Moshe Y. Vardi

Coverage Metrics for Formal Verification ............................... 111
  Hana Chockler, Orna Kupferman, Moshe Y. Vardi

“More Deterministic” vs. “Smaller” Büchi Automata for Efficient LTL Model Checking ........................................ 126
  Roberto Sebastiani, Stefano Tonetta

Short Papers 1

An Optimized Symbolic Bounded Model Checking Engine .................. 141
  Rachel Tzoref, Mark Matusevich, Eli Berger, Ilan Beer

Constrained Symbolic Simulation with Mathematica and ACL2 .......... 150
  Ghiath Al Sammane, Diana Toma, Julien Schmaltz, Pierre Ostier,
  Dominique Borrione

Semi-formal Verification of Memory Systems by Symbolic Simulation .................. 158
  Husam Abu-Haimed, Sergey Berezin, David L. Dill

CTL May Be Ambiguous When Model Checking Moore Machines .......... 164
  Cédric Roux, Emmanuelle Encrenaz

Specification Methods

Reasoning about GSTE Assertion Graphs ......................... 170
  Alan J. Hu, Jeremy Casas, Jin Yang

Towards Diagrammability and Efficiency in Event Sequence Languages .................. 185
  Kathi Fisler

Executing the Formal Semantics of the Accellera Property Specification Language by Mechanised Theorem Proving .................. 200
  Mike Gordon, Joe Hurd, Konrad Slind

Protocol Verification

On Combining Symmetry Reduction and Symbolic Representation for Efficient Model Checking .................. 216
  E. Allen Emerson, Thomas Wahl
On the Correctness of an Intrusion-Tolerant Group Communication Protocol ........................................... 231
Mohamed Layouni, Jozef Hooman, Sofiène Tahar

Exact and Efficient Verification of Parameterized Cache Coherence Protocols ............................................. 247
E. Allen Emerson, Vineet Kahlon

**Short Papers 2**

Design and Implementation of an Abstract Interpreter for VHDL ........ 263
Charles Hymans

A Programming Language Based Analysis of Operand Forwarding ........ 270
Lennart Beringer

Integrating RAM and Disk Based Verification within the Murϕ Verifier ......................... 277
Giuseppe Della Penna, Benedetto Intrigila, Igor Melatti,
Enrico Tronci, Marisa Venturini Zilli

Design and Verification of CoreConnectTM IP Using Esterel ............. 283
Satnam Singh

**Theorem Proving**

Inductive Assertions and Operational Semantics ....................... 289
J Strother Moore

A Compositional Theory of Refinement for Branching Time ............. 304
Panagiotis Manolios

Linear and Nonlinear Arithmetic in ACL2 ................................ 319
Warren A. Hunt, Jr., Robert Bellarmine Krug, J Moore

**Bounded Model Checking**

Efficient Distributed SAT and SAT-Based Distributed Bounded Model Checking ........................................... 334
Malay K Ganai, Aarti Gupta, Zijiang Yang, Pranav Ashar

Convergence Testing in Term-Level Bounded Model Checking .......... 348
Randal E. Bryant, Shuvendu K. Lahiri, Sanjit A. Seshia

The ROBDD Size of Simple CNF Formulas .............................. 363
Michael Langberg, Amir Pnueli, Yoav Rodeh
Model Checking and Application

Efficient Hybrid Reachability Analysis for Asynchronous Concurrent Systems ............................................ 378
    Enric Pastor, Marco A. Peña

Finite Horizon Analysis of Markov Chains with the Murϕ Verifier ................................................ 394
    Giuseppe Della Penna, Benedetto Intrigila, Igor Melatti,
    Enrico Tronci, Marisa Venturini Zili

Improved Symbolic Verification Using Partitioning Techniques ............ 410
    Subramanian Iyer, Debashis Sahoo, Christian Stangier, Amit Narayan,
    Jawahar Jain

Author Index ................................................ 425
Abstract. The current state of hardware logic design and verification is discussed based on the project flow used for IBM’s Power4 and Power5 projects.

The frequency and power requirements for these high-end chips constrain the logic design to a detailed RT-level in order to control physical effects. On the other hand, the complexity of the designs which embrace many speculative mechanisms to push functional performance to higher levels force an early specification of the microarchitecture with a high-level model.

A review how high-level modeling has advanced is based on the discussion which mechanisms of abstraction raise the specification above the RT-level. A critique of specification language design leads to the appeal to the formal verification community to focus efforts on the front-end of the high-level design process to help shape modeling languages with formally defined semantics that avoid the mistakes made in the past with ad-hoc language designs.
The Charme of Abstract Entities

Fabio Somenzi*

University of Colorado at Boulder
Fabio@Colorado.EDU

Abstract. Abstraction is fundamental in combating the state explosion problem in model checking. Automatic techniques have been developed that eliminate presumed irrelevant detail from a model and then refine the abstraction until it is accurate enough to prove the given property. This abstraction refinement approach, initially proposed by Kurshan, has received great impulse from the use of efficient satisfiability solvers in the check for the existence of error traces in the concrete model. Today it is widely applied to the verification of both hardware and software. For complex proofs, the challenge is to keep the abstract model small while carrying out most of the work on it. We review and contrast several refinement techniques that have been developed with this objective. These techniques differ in aspects that range from the choice of decision procedures for the various tasks, to the recourse to syntactic or semantic approaches (e.g., “moving fence” vs. predicate abstraction), and to the analysis of bundles of error traces rather than individual ones.

* Supported in part by SRC contract 2002-TJ-920.
Abstract. The Accellera EDA standards body has recently approved the PSL a standard property specification language for use in assertion-based verification via simulation and formal verification tools. This language, which is based on the Sugar language from IBM, is now supported by many EDA vendors. More than 40 individuals representing over 20 companies participated in the efforts to form the PSL standard from its Sugar basis.

The tutorial comprises 2 parts. In the first part, we describe the basic principles of PSL/Sugar, focusing on the ease with which complex design behaviors may be described with concise, readable PSL/Sugar assertions that crisply capture design intent. We summarize the temporal constructs of the language, including parameterized sequences and properties, directives, and modeling capabilities. We cover the general timing model of PSL/Sugar, which transparently supports both (single- or multi-clock) synchronous and asynchronous design, and, time permitting, we explain how PSL/Sugar has been defined to ensure consistent semantics for both simulation and formal verification applications.

In the second part of the tutorial, we present several applications of PSL/Sugar, ranging from simple to advanced assertion-based verification solutions. These include use of PSL/Sugar for dynamic assertion checking and formal model checking, including support for environment modeling and assume/guarantee reasoning. Examples of commercial verification tools which support the PSL/Sugar languages will also be presented.

Participants in the tutorial will have an excellent opportunity to learn about both the language and its applications directly from the speaker, Dr. Danny Geist, who heads a research group in the IBM Haifa lab where Sugar was conceived.
Finding Regularity: Describing and Analysing Circuits That Are Not Quite Regular

Mary Sheeran
Chalmers University of Technology
ms@cs.chalmers.se

Abstract. We demonstrate some simple but powerful methods that ease the problem of describing and generating circuits that exhibit a degree of regularity, but are not as beautifully regular as the text-book examples. Our motivating example is not a circuit, but a piece of C code that is widely used in graphics applications. It is a sequence of compare-and-swap operations that computes the median of 25 inputs. We use the example to illustrate a set of circuit design methods that aid in the writing of sophisticated circuit generators.

1 Introduction

In arithmetic and digital signal processing, many algorithms are well understood, and result in efficient regular circuits. The functional approach to hardware design has proved particularly well-suited to the development of such circuits [3, 10]. Here, we continue to explore this theme; this paper is not about verification, but about design methods – a valid, if under-represented, topic of the Charme conference. We emphasise the description of circuits, as we feel that ease of describing the intended circuit is a key to design productivity. The methods presented here go beyond what can be done in VHDL or C, through the use of higher order functions and polymorphism, which are features of many functional programming languages. The examples shown use Lava, a hardware design system implemented as an embedded domain specific language in the functional programming language Haskell [2].

Batcher’s classic odd even merge sorting algorithm illustrates the power and elegance of the combinator-based approach to describing complex networks:

\[
\begin{align*}
\text{oemerge} :: \text{Int} & \rightarrow ([\text{a}] \rightarrow [\text{a}]) \rightarrow [\text{a}] \rightarrow [\text{a}] \\
oemerge 1 \text{ s2} &= \text{s2} \\
oemerge n \text{ s2} &= \text{ilv} (\text{oemerge} (n\text{-}1) \text{ s2}) \rightarrow - \odds \text{ s2} \\
\text{oesort} :: \text{Int} & \rightarrow ([\text{a}] \rightarrow [\text{a}]) \rightarrow [\text{a}] \rightarrow [\text{a}] \\
oesort 0 \text{ s2} &= \text{id} \quad \text{-- the identity function} \\
oesort n \text{ s2} &= \text{two} (\text{oesort} (n\text{-}1) \text{ s2}) \rightarrow - \text{oemerge} n \text{ s2}
\end{align*}
\]

Here, \text{ilv}, for \text{interleave}, is a combinator that applies the given function to the odd and even elements of a list of inputs, to produce the odd and even elements of the output list. So, the function \text{ilv} \ \text{reverse} applied to the list [1..8] gives
[7,8,5,6,3,4,1,2]. reverse is a Haskell function whose type is \([a] \to [a]\). It takes a list of elements of any type \(a\) to a list of elements of the same type. It is a *polymorphic* function and works at many types. Similarly, \(\text{intv}\) has type \(([a] \to [b]) \to [a] \to [b]\). It takes a function from list of \(a\) to list of \(b\) and returns a function of the same type. In functional programming parlance, it is a *higher order function*; it takes a function and returns a function. We use polymorphic higher order functions like \(\text{intv}\) to capture circuit interconnection patterns. A second such function is \(\text{two}\), which applies a function to the first \(n\) elements and to the second \(n\) elements of a \(2n\)-length input list, so that, for instance, \(\text{two reverse} [1..8] \) is \([4,3,2,1,8,7,6,5]\). Serial composition is written \(\to\), and \(\text{odds s2}\) applies \(s2\) to pairs of adjacent elements of the input, but starting with the second element rather than the first. The function \(\text{oesort}\) is parameterised both on an integer and on a two-input, two-output sorter component, \(s2\). The integer and the \(s2\) parameter determine the size and type of the resulting network. For instance, \(\text{oesort 3 intSort2}\) is a circuit that sorts lists of integers of length \(2^3\), built from a component that sorts a 2-list of integers, \(\text{intSort2}\).

\[
\text{intSort2} :: [\text{Signal Int}] \to [\text{Signal Int}]
\]

\[
\text{intSort2} [x,y] = [\text{imin} (x,y), \text{imax}(x,y)]
\]

To illustrate the combinators, \(\text{oesort 3 s2}\) is shown in figure 1. Values flow through the network from left to right, and the vertical lines are 2-sorters. The first (or leftmost) value of the input list is input along the top wire.

The \(\text{oesort}\) pattern can be instantiated with many different comparator components, depending on the context in which the sorter is to be placed. The same description can be used to give bit-parallel and bit-serial implementations, simply by plugging in new comparator components. The object of study is the connection pattern from which both combinational and sequential sorters can be built. To perform verification, we plug in a 2-sorter on \(\text{bitSort2}\) and, using the 0-1 principle [7], verify functional correctness by generating and checking a propositional formula that states that a fixed-size circuit obeys the required
sorting property. The 0-1 principle states that if a network with \( n \) input lines sorts all \( 2^n \) sequences of 0s and 1s into nondecreasing order, it will sort any arbitrary sequence of \( n \) numbers into nondecreasing order. We have studied the design and analysis of sorting networks in a previous paper [3], and we use the same verification methods in this paper.

The problem that we want to address here is the fact that not all circuits are beautiful. They don’t all have a number of inputs that is a power of two, and they don’t all have such an obvious recursive structure. For example, how would we describe any 7-sorter that contains the minimal number of comparators (which is known to be 16 [7])? More generally, how do we describe circuits that are somewhat regular?

Via a running example, a median circuit, we present a series of ideas for how to make more sophisticated circuit descriptions, using polymorphism and higher order functions. Shadow values and clever components are aids to writing circuit generators. Non Standard Interpretation is an old idea that we (and others) have used before. Here, we use ordinary polymorphism and components of different types, and do not rely on Haskell’s type classes (although type classes are used extensively in the Lava implementation). Finally, we needed to extend our range of combinator types in order to explore a variety of solutions to the median problem. We have deliberately not used the more esoteric parts of Haskell, in the hope of making the ideas usable in other contexts.

The median example was inspired not by a circuit, but by a piece of C-code, due to Paeth, which appears in Graphics Gems I, a book of classic graphics algorithms [11]. It is a sequence of 99 compare-swap operations that arranges an array of 25 inputs so that the median element is in the middle position, and all smaller elements are at lower indices (and hence all larger are at larger indices). I first came across a transliteration of this code in reference [5], where it is claimed (informally and without justification) that this function cannot be performed in fewer than 99 comparison-swaps without further information about the input. The application area of such programs (and circuits) is median filtering of digital images, in which \( n \) by \( n \) windows of the image have their middle pixel replaced by the median pixel, thus removing white noise. A 5 by 5 kernel (as it is called) is often used, so the algorithm is of practical interest. A common approach is to actually sort the 25 pixels, using Batcher’s odd even merge sort, but in a more general variant that allows the division of the input into two parts of unequal length. That would take 138 comparators.

2 Shadow Values I

The user of Lava describes circuits by writing circuit generators. For example, in the \texttt{osort} example above, the recursive description is instantiated at a particular size, and with a particular type of comparator, in order to produce a circuit. When we simulate, say, an 8-sorter on integers, what happens is that in the background a representation of the concrete circuit is created, and the \texttt{simulate} function walks over that representation:
Here, the values that flow through the circuit are of type Signal Int and are circuit level values (even though they look like integers). The component intSort2 sorts two such circuit level integers. However, the 3 that is a parameter to oesort is an ordinary Haskell integer. This is an important distinction, at least intuitively, as the Lava user must be able to tell what is a circuit description and what is a more general Haskell function. There are circuit level values (with Signal types), and there are ordinary Haskell values that are used in the generation of circuits. Once we have got to a concrete circuit in the internal netlist representation, all the ordinary Haskell values have disappeared. But in writing the Haskell code that is to be used to generate such a netlist, we can make use of ordinary Haskell values, and can make decisions about how the circuit should look, based upon them. A common pattern is to pair a Haskell value with each circuit level value. The shadow values can control the shape of the resulting circuit.

The simplest form of shadow value is just a boolean that indicates whether the corresponding wire should have any components attached to it. The Haskell function `tomarked f` applies `f` only to those inputs that are paired with `True`. It simply passes through those inputs that are paired with `False`.

Main> `tomarked (map (*2)) [(1,True),(3,False),(5,True)]` [(2,True),(3,False),(10,True)]

Here, only the first and third values are doubled. We can use this idea when generating circuits. If `f` is a connection pattern that places instances of the component `s` in a particular way on `n` inputs, to give `n` outputs, we might want to get a circuit with `n - i` inputs by deleting the top `i` wires and all components attached to them. The resulting circuit will take `n - i` inputs. We pair each of those `n - i` real inputs with `True`, and then add `i` dummy inputs paired with `False`. Then, we can apply `f` (`tomarked s`) to the resulting marked list, secure in the knowledge that the dummy wires will never be touched. Then, we can drop the dummy wires, and all the marks, to produce `n - i` circuit level outputs. This is what the function `cutTop i` does. Similarly, `cutTopBottom i j` cuts `i` wires at the top and `j` on the bottom. Note that a component that is an argument to `tomarked` must be flexible, in that it may be required to deal with a number of arguments that are smaller than usual, because of the presence of inputs marked with `False`. In our sorting example, this means that we need a component that is not just a 2-sorter, but that can also deal with one or even zero inputs. The function `smallSort` takes a two-input sorter and makes it flexible in this way. We will have reason to extend this function later.

```
smallSort s2 [] = []
smallSort s2 [a] = [a]
smallSort s2 [a,b] = s2 [a,b]
```

For example, we can make a 7-sorter from an 8-input odd even merge sorter by using `cutTop 1` and `(oesort 3)`. The resulting network is shown in figure 2.
It is derived from the network shown in figure 1 by omitting the top wire, and the three comparators connected to it.

In this instance, the resulting netlist has only 7 inputs and 7 outputs, and it no longer looks very regular. All history of how that netlist was generated using shadow booleans is forgotten at this stage. The reader might argue that one could just use padded inputs and leave the pruning of unnecessary gates and wires to the lower level design tools. However, we find this approach more convenient and less error prone. We have found that padding makes for unreadable circuit descriptions, and can lead to the introduction of bugs. Also, we often make designs in which we first develop abstract circuits (say with integers whose representation has not yet been chosen flowing on wires). We want to be able to prune these circuits at an early stage in the design, before we are ready to produce input to lower level design tools.

Formal verification using a SAT-solver is done in the usual way [3]. (Satzoo is a SAT-solver developed by Een here at Chalmers [6]. The function satzoo creates a file in DIMACS format that is passed to the solver, the output of which is then passed back to the Haskell interpreter.)

\[
\text{sortCheck } n \ cct = \\
\quad \text{satzoo (prop_doesSortsize (cct (smallSort bitSort2)) n)}
\]

Main> sortCheck 7 (cutTop 1 (oesort 3))
Satzoo: ... (t=0.0) Valid.

Because we consider only restricted forms of networks, we choose not to prove that the networks permute their inputs. Such proofs, if required, can also be done using a SAT-solver in Lava.

3 Non-standard Interpretation

We have already seen how to verify sorting networks by using a 2-sorter on bits and the 0-1 principle. This is an example of non-standard interpretation, in which we replace the circuit components with others that are intended to
gather information about the circuit. We then simulate the circuit with the new components, and suitable initialising inputs, to perform the required analysis.

To count the number of comparators in a circuit, we replace each comparator by a component that adds one to its left hand input and passes its right hand input through unchanged. Then, at the end, we sum all of the numbers appearing on the output. (This simple method works as long as all of the information-carrying wires eventually reach the output, but that is the case for all of our networks.) We simulate the resulting circuit on a list of zeros. Note that \texttt{csize2} is most definitely a circuit level component, whose inputs and outputs are lists of integer signals. It is included as a first step towards the use of such functions during circuit generation, rather than, as here, during simulation. A more general \texttt{count} function would be a recursive function over the internal data type representing circuits.

\begin{verbatim}
csize2 :: [Signal Int] -> [Signal Int]
csize2 [i,j] = [plus(1,i),j]

count n cct
   = simulate(cct (smallSort csize2) ->- sum) (replicate n 0)

Main> count 7 (cutTop (oesort 3))
16
\end{verbatim}

The 7-sorter has as few comparators as possible. Circuit depth is just as easy to calculate. Again, integers flow on the wires, and the depth of the output of a comparator is one more than the integer maximum of the inputs. The 7-sorter has optimal depth (which is 6)\cite{6}.

\begin{verbatim}
cdepth2 :: [Signal Int] -> [Signal Int]
cdepth2 [x,y] = [m,m]
   where m = plus(1,imax(x,y))

depth n cct
   = simulate(cct (smallSort cdepth2) ->- imaximum) (replicate n 0)
\end{verbatim}

Cutting 2 wires on the top of an 8-sorter also gives a size-optimal circuit with 12 comparators. We don’t do so well when the number of inputs to the sorter is just above a power of two, rather than just below. The smallest known 9-sorters have 25 comparators, but cutting 7 wires from a 16-input odd even merge sort gives a 28-comparator sorter.

Our next step is to generalise the combiners \texttt{ilv} and \texttt{two} to be multi-way rather than two-way. This leads us to a generalisation of odd even merge sort, and also broadens the range of sorters and other networks that can be described easily.

4 Generalised Combinators

Recall that \texttt{two f} applies \texttt{f} to each half of a list. Its generalisation, \texttt{parI i f} applies \texttt{f} to each \texttt{i}th part of the list, so that, for instance, \texttt{parI 5 f} applies \texttt{f}
to each fifth of the list. The function `concat` flattens a list of lists back into a list. The general version of `ilv` instead chops the list into i-length sublists and transposes, to give i sublists, before applying `map f` and then returning the list to its original order.

```haskell
parI i f = chopinto i ->- map f ->- concat
ilvI i f = chop i ->- transpose ->- map f ->- transpose ->- concat
```

Armed with these new combinators, we can generalise `oesort`, provided we can figure out what `odds` should become. Well, `odds s2` sorts an almost-sorted list. It is able to sort the list by comparing only adjacent elements, and it compares only those elements that have not already been compared. For \( i = 3 \), it turns out that the new pattern, which we will call `fmerge i`, should compare elements a distance two apart, and then adjacent elements, while refraining from comparing elements whose relation is already known. In general, `fmerge i` should first compare elements a distance \( i - 1 \) apart, then \( i - 2 \) and so on, down to 1. The function `dist i k ss` applies `ss` to elements a distance `k` apart, but avoids comparing elements in each i-length sublist.

```haskell
fmerge i ss = compose [dist i k ss | k <- reverse [1..(i-1)]]
oemergeI i 1 ss = ss
oemergeI i n ss = ilvI i (oemergeI i (n-1) ss) ->- fmerge i ss
oesortI i 0 ss = id
oesortI i n ss = parI i (oesortI i (n-1) ss) ->- oemergeI i n ss
```

Think of the second parameter to `oesortI` as the number of dimensions. The instance `oesort i j` sorts a list of length i to the power of j. The i parameter, the size of each dimension, must be odd, although 2 works as a special case (and gives Batcher’s odd even merge sort shown earlier). For larger even-length dimensions, some extra comparators are needed, but we will not pursue this topic here.

Now, if we are to use this general sorting algorithm for i greater than 2, we must be able to make sorting components (for use as the `ss` parameter) for more than two inputs. To do this, we extend the function `smallSort` that was introduced earlier. The 3-sorter is made from three comparators, and is completely standard. The 4- and 5-sorters are made from `oesort` (and are optimal in both size and depth). Larger sized sorters are easily included in a similar way, and it may then make sense to change the style of the definition to a case analysis on the length of the input.

```haskell
sort3l s2 [x,y,z] = [a,b,c]
  where
    [x1,y1] = s2 [x,y]
    [y2,c] = s2 [y1,z]
    [a,b] = s2 [x1,y2]
smallSort s2 [] = []
smallSort s2 [a] = [a]
smallSort s2 [a, b] = s2 [a, b]
smallSort s2 [a, b, c] = sort3l s2 [a, b, c]
smallSort s2 [a, b, c, d] = oesort 2 s2 [a, b, c, d]
smallSort s2 [a, b, c, d, e] = cutTop 3 (oesort 3) (smallSort s2) [a, b, c, d, e]

If we restrict oesortI to two dimensions, we get the sorting algorithm proposed by Kolte et al. [8] from Motorola. In that case, the rows and columns of the $i \times i$ grid are first sorted, and then the call of fmerge i sorts all the diagonal lines, starting with the main diagonals. What we add here is both a much more streamlined verification process and the generalisation to more than two dimensions. The paper by Kolte et al proposes an elaborate scheme for testing the proposed sorting network, but the use of a SAT-solver and the 0-1 principle is a much easier option. On the other hand, the Motorola paper develops software for a complete median filter that gives impressive performance on a particular architecture. It would be very interesting to develop an efficient median filter on an FPGA and compare its performance with more standard implementations. That is future work.

Using 3 dimensions, for example, we can quickly analyse a 27-sorter (made from 3- and 2-sorters) to find that it has depth 20 and size 154. This is one comparator smaller (though considerably deeper) than the general two-way odd even merge. We will make use of oesortI 3 3 later, when constructing the 25-median circuit.

Further discussion of the algorithm oesortI is beyond the scope of this paper. We believe that fmerge could be improved for larger dimension sizes, and Van Voorhis’ work shows how to deal with even-length dimensions [12]. Independent of the example, we are pleased with the simplicity of the generalised combinatorics. They give the user access to a broader range of connection patterns, without the need to learn many new combinatorics.

Now, we return to the 25-median problem. To solve it, we need to use more complicated shadow values than those that we have seen so far. We aim to keep only those parts of a sorter that contribute to arranging the outputs of the median circuit into an order that satisfies the specification.

5 Shadow Values II

We saw in section 3 that we can gather information about an instantiated circuit by simulating it using specially designed circuit level components like csize2. Here, we use similar ideas, but in the world of shadow values. Shadow values have so far been unchanging Boolean values. Now, we make them more dynamic and more complicated.

The idea is to use shadow values to record information about the circuit so far, allowing decisions to be made about how the rest of the circuit should look. For the median example, what we want to do is to figure out for each “wire” in the circuit whether or not it is still in the running to be the median, and so needs to be processed further. And we want to do this figuring out at circuit generation time. This is not straightforward, and requires some insights into
the mathematics of sorting. We cannot go into the details here, but the reader is referred to the work of Van Voorhis to see the kinds of arguments that are required [12]. Our approach is to rewrite our sorter so that the first steps are to sort the different dimensions of the input. So, for example, a two-dimensional sorter will start by sorting the rows and columns, and a three-dimensional sorter will sort along each of the three axes. This pattern is called a butterfly network. It is straightforward to rewrite oesortI into a butterfly network of sorters followed by the rest, which we call bafterI. boesortI 3 2 is shown in Figure 5. It is essentially the same as the optimal 25-comparator 9-sorter due to Floyd [7].

```
bflyI i 0 f = id
bflyI i n f = parI i (bflyI i (n-1) f) ->- (iter (n-1) (ilvI i) f)
```

```
boemergeI i 1 ss = id
boemergeI i n ss = ilvI i (boemergeI i (n-1) ss) ->- fmerge i ss
```

```
bafterI i 1 ss = id
bafterI i n ss = parI i (bafterI i (n-1) ss) ->- boemergeI i n ss
```

```
boesortI i n ss = bflyI i n ss ->- bafterI i n ss
```

The reason why we do this is that the sortedness of the different dimensions, which is the result of the initial butterfly network, remains unaffected throughout the rest of the network. Also, inside the butterfly, sorting each new dimension leaves the previously sorted dimensions still sorted. So, after the butterfly, it is easy to figure out, for a given wire, how many other wires are greater than or smaller than it. We give each wire an address that records what happened in the butterfly. So, for example, the address [2,1,2] is given to a wire that has “passed through” the top, bottom, and top of three 2-way comparators. After the butterfly, this wire is greater than or equal to the following set of wires:
Given one, one needs to separate the remaining wires. On each wire, we place a single (shadow) inverter, which required involving only one and under it. The shadow component for the 2-sorter manipulates and updates these lists, which represent sets of addresses, and so do not contain duplicates. The standard function `nub` removes duplicates from a list.

\[
\text{combs2} :: \mathbb{[[\text{Address}],[\text{Address}]}} \rightarrow \mathbb{[[\text{Address}],[\text{Address}]}} \n\]

\[
\text{combs2 } \{(11, g1), (12, g2)\} = \{(\text{nub} (11+12), g1), (12, \text{nub}(g1++g2))\} \n\]

So, the wire that “passes through” the lower part of the comparator gets a new \((\text{over}, \text{under})\) pair containing the union of the two input \text{over} lists, but only the lower \text{under} list. For the upper wire, the situation is dual. Then, the \text{lengths of these lists} give good information about the status of a wire, and its relation to the remaining wires. On the input to the circuit, we provide information about the target for each wire. In our case, we place a single (shadow) integer on each wire, and the wire should be taken out of the running (in the same way as with the simple shadow Booleans that we saw earlier) once it is known to be either greater than or less than that number of other wires. The target remains unchanged, while the address lists grow longer as one moves through the network. (One could choose to use two integers for the target, which could be different for the \text{over} and \text{under} lists, but that is not necessary in the median examples shown here.) The new shadow component is `combine id combs2` where

\[
\text{combine } f \ g \ [(a,x),(b,y)] = [(fa,gx),(fb,gy)] \n\]

where

\[
[fa,fb] = f \ [a,b] \n\]

\[
[gx,gy] = g \ [x,y] \n\]

Each wire has a shadow value of type \((\text{Int}, ([\text{Address}],[\text{Address}]))\), that is a pair of an integer and a pair of lists of addresses.

A wire is certain \text{not} to be the median if the number of distinct addresses that are either smaller than or greater than it is large enough. The target is set to \(1 + \lfloor n/2 \rfloor\), where \(n\) is the number of inputs to the median circuit. Just after the butterfly, the address lists are all singletons containing the address of the wire to which they are attached. The function \text{placeTargetAddressI} introduces the required initial shadow values.

To be able to make use of these shadow values, we must generalise \text{tomarked}. The function \text{onPredicate p f} causes \text{f} to be applied only to those inputs for which the predicate \(p\) is true of the shadow value.

Recall that the version of \text{oesortI} with the butterfly in the first columns was

\[
\text{boesortI i n ss = bflyI i n ss -> bafterI i n ss} \n\]

Following this definition, we define

\[
\text{medI i j ss = bflyI i j ss ->}\n\]

\[
\text{placeTargetAddressI i j ->} \n\]
We leave the butterfly alone, but transform \texttt{bafterI} so that it performs the calculations described above when deciding whether or not to include a comparator. The result is promising:

Main> medCheck 27 (medI 3 3)
Satzoo: ... (t=0.3) Valid.

Main> count 27 (medI 3 3)
114

Main> count 27 (boesortI 3 3)
154

We have a circuit that correctly places the median input in the middle output, and all of the smaller values to the left of it in the output list. This property is checked by the observer \texttt{medCheck}, whose key function is \texttt{reallyMedian}, which checks that a given value is larger than all of the elements of a given list, and smaller than all of the elements of another. Logical implication (written \(==>)\) is the ordering on bits, and \texttt{and1} is a multi-input \texttt{and} gate.

\[
\text{reallyMedian} \quad \text{a smaller bigger} = \quad \text{and1} \quad ([s ==a \mid s \leftarrow \text{smaller}] \quad ++ \quad [a ==b \mid b \leftarrow \text{bigger}])
\]

Again, we use the 0-1 principle, which applies also in the context of median networks; for a proof of this, see [9]. (It should be noted that the \(O(\log n)\) depth selection networks developed in reference [9] are far from being practical.)

We have saved 40 comparators in making a 27-median circuit from a 27-sorter. And the step to a 25-median circuit is now an easy one. We simply cut off the top and bottom wires, and attached comparators. Note that for making smaller median circuits from larger ones, it is necessary to crop the network symmetrically. To illustrate the step from a sorter to a median circuit, Figure 4 shows a 7-median circuit made from the 9-sorter shown in figure 3.

The 7-median circuit is optimal, but, sadly, that for 25 inputs has 102 comparators. And making the 25-median circuit directly (out of 2, 3, 4 and 5-sorters), using \texttt{medI 5 2} takes 112 comparators, although 12 of them could be pruned from the butterfly, which has so far been left untouched. Since we have pored over the Paeth code and discovered that it starts with a butterfly of 3-sorters that is missing its top and bottom wires, we choose to make a final change to the 102-comparator network. We use one last idea, \textit{clever components} that adapt themselves to the context in which they find themselves in the final circuit.

6 Clever Components

When a component is applied to inputs that have shadow values, then the definition of the component can decide what is to happen by looking at those shadow
values. We have seen this several times. More interestingly, we can, in the definition, look to see what a particular arrangement of the basic components does to those shadow values. This is done simply by applying the proposed circuit to the inputs (which are mixed concrete and shadow values) and then looking only at the resulting shadow values. Then, the decision about what circuit to actually apply to the inputs can depend on those computed shadow values. This is a kind of “try it and see” approach, used during circuit generation.

To make the idea more concrete, let us return to the median circuit. Consider the case of a flexible 3-sorter, and our predicate (notmedianI) that indicates when an output is now done. If the 3-sorter is applied to 3 inputs (none of which is done), then it might be the case that two of the outputs become done. In that case, we don’t need to know the order between the two done outputs, and we might as well use a 2-comparator min or max circuit of three inputs, as appropriate. So, think of a component that applies circuit A to the inputs, has a look at the shadow part of the result, and decides whether or not to be circuit B or to remain as circuit A, when producing the actual outputs of the component.

The definition of the clever version of smallSort starts off looking very much like that of smallSort, but with the addition of the predicate to the parameters of the function. When smallSortV p ca is applied to three inputs, [in1,in2,in3], it computes sort3l ca [in1,in2,in3], and names the resulting shadow values a1, a2 and a3. By applying the predicate to those values, it can decide which of the max3l, min3l or sort3l patterns to actually use. Note that this is not just about calculating the cone of influence of the wires that are not done. Removing the comparator closest to the output of a 3-sorter can give either the maximum or the minimum circuit, but not both.

\[
\text{smallSortV p ca [in1,in2,in3]} = \begin{cases} 
\text{max3l ca [in1,in2,in3]} & \text{if (p a1) && (p a2)} \\
\text{min3l ca [in1,in2,in3]} & \text{else if (p a2) && (p a3)} \\
\text{sort3l ca [in1,in2,in3]} & \text{else}
\end{cases}
\]

where
\[
[(_,a1),(_,a2),(_,a3)] = \text{sort3l ca [in1,in2,in3]}
\]

If needed, smallSortV should be extended to longer inputs in a similar manner.
Our final median circuit generator, medVI, is identical to the previous medI, except that smallSort is replaced by smallSortV (notmedianI i). And this does the trick! cutTopBottom 1 1 (medVI 3 3) has 98 comparators. We can use the same descriptions to generate median circuits of other sizes.

The modification of the sorter is most definitely a hack, though a rather effective one. Using another sorter, hand-crafted for the purpose, we have, in fact, been able to get the number of comparators down to 96, but we are unable to generalise that sorter to other sizes, and so chose not to present it here. The circuit development shown here allowed us to exemplify shadow values and clever components. However, for a circuit, as distinct from a C program, one should really aim for small depth rather than small number of comparators, so we have many more median circuits to explore. It would have been more pleasing to develop a recursive median circuit meeting our specification from scratch. But even when we have designed specific median circuits, we have found them difficult to express. They have an annoying lack of regularity, in that they tend to have fewer comparators in each phase as one approaches the outputs, but not according to any simple pattern. This is what led us to use clever components.

There are other ways to make median circuits, for example by looking at the inputs bit by bit [4], and indeed they may well be better. We have restricted our attention to comparator-based networks so far.

7 Related Work

Ideas similar to shadow values and clever components were used in the generation of the FM9001 netlist, as part of a large microprocessor verification effort [1]. Circuit generators (for example for the ALU) not only used precursors of both shadow values and clever components, but were also verified to meet their specifications for all sizes. This was done by a deep embedding of the DUAL-EVAL netlist language produced by the generators within the Boyer-Moore logic. The proofs required that the interpretation of the resulting netlists did indeed work correctly for all possible sets of inputs. This work, which is so closely in line with our aims, is barely mentioned in the published paper, which concentrates on the overall verification goal. Indeed, it is barely mentioned (in the English text) even in the very long technical report on the verification effort, so it will be necessary to delve into the code. I did not know about this work when writing the first version of this paper. My disappointment at discovering that my ideas are not as new as I thought has been overshadowed by the realisation that my tentative ideas for mixing clever components and verification have already been shown to work.

In the current version of Lava, we perform formal verification only of fixed size circuit instances. A first step towards making use of the FM9001 generator work would be to generate DUAL-EVAL code and to perform proofs about that code. However, we have long considered a move to using first order provers and inductive proofs of recursive circuits. Our emphasis on the use of higher order functions gives our circuit descriptions (and our use of shadow variables and
clever components) a different style from those in the FM9001 work. The next step will be to find good ways to combine the best of both approaches.

8 Conclusion

We have presented a collection of methods that together allow us to describe and analyse circuits that are not quite regular. We distinguish circuit generation time from circuit analysis time, and there is a clear analogy with compile time and run time, and with static and dynamic semantics in VHDL. The aim of circuit generation is to produce a representation (in terms of a suitable recursive data type) of a complete fixed-size circuit, something very close to a netlist. Circuit analysis is what happens when we turn this representation into various other notations, in order to scrutinise it further, often with the help of external tools such as SAT-solvers and model checkers. Simulation is one such analysis.

During circuit generation, we use the power of Haskell to control the process of generating the required netlist. Special values called shadow values are associated with the circuit level values, and can be used to control the generation process. They can be static, like the shadow Booleans that we use when omitting unwanted parts of networks, or dynamic, like the address lists that we used to track progress towards a target in the median circuit example. The shadow values can also encode information about the circuit that feeds a component, allowing the component itself to decide what circuit would best be introduced into the network at that point. These clever components are likely to have many applications. For example, the “try it and see” could extend to calling external tools like, say, automatic place and route tools with possible circuits that might be included in the final design, and then picking the one that gives the best result according to some criterion such as timing, testability or power consumption. We have not yet incorporated the notion of layout into this work but that will be the next step. At Chalmers, we are developing a language that captures wires and layout explicitly, but uses Lava functions for describing circuit function. It can be seen as a generalisation of layout combinators [3], and we have had to move from 2-dimensional tiles to 3-dimensional blocks. We aim to be able to capture the ways in which regular circuits become irregular during the design process, for example when they are designed to fit under a particular interconnect fabric. Our intention is to combine the design methods illustrated here with circuit analyses that capture wire length and related non-functional properties. Thus, it is the problem of how to do interconnect-aware design that is the main motivation for this research.

However, there is a second motivation, the need to push formal verification earlier in the design process. We had speculated that clever components would allow sub-parts of circuits to be verified during circuit generation, but had not yet performed any experiments in this area. The FM9001 work shows, very convincingly, that these ideas enable both hierarchical proofs and the generation of circuits that are built for verifiability. That the FM9001 proof can simply be rerun for any size is an extremely important property of the verification effort.
The use of verified circuit generators in the FM9001 work goes beyond what we had envisaged. We feel spurred on to investigate ways to support inductive proofs of recursive circuit generators based on Lava combinators, while still aiming for as much proof automation as possible.

Finally, we would like to investigate whether or not our methods, and in particular clever components, could be applied to the description and analysis of reconfigurable circuits.

Acknowledgements. Thanks to Satnam Singh, who suggested the median example. This research is funded by the Swedish funding agency Vetenskapsrådet. Thanks to the anonymous reviewers for their thoughtful reports. It was especially useful to learn of related work in the generation and verification of the FM9001 microprocessor.

References

Predicate Abstraction with Minimum Predicates*

Sagar Chaki, Edmund Clarke, Alex Groce, and Ofer Strichman

Computer Science Department, Carnegie Mellon University, Pittsburgh, PA

chaki,emc,agroce,ofers@cs.cmu.edu

Abstract. Predicate abstraction is a popular abstraction technique employed in formal software verification. A crucial requirement to make predicate abstraction effective is to use as few predicates as possible, since the abstraction process is in the worst case exponential (in both time and memory requirements) in the number of predicates involved. If a property can be proven to hold or not hold based on a given finite set of predicates $\mathcal{P}$, the procedure we propose in this paper finds automatically a minimal subset of $\mathcal{P}$ that is sufficient for the proof. We explain how our technique can be used for more efficient verification of C programs. Our experiments show that predicate minimization can result in a significant reduction of both verification time and memory usage compared to earlier methods.

1 Introduction

Predicate abstraction [13] is a commonly used abstraction technique in formal verification of both software and hardware. Like other abstractions, when successful it can be used to prove the correctness (or incorrectness) of a property with only partial information about the reachable states of the system. This facilitates the verification of systems larger than would otherwise be possible. Predicate abstraction has been used widely both for hardware [5] and software [2, 9] verification. In this article we focus on its application to the verification of C programs.

Verification of programs typically concentrates on the control flow of the program (e.g. checking if a particular control point is reachable), rather than on the data manipulated by it (e.g. checking functional correctness). Predicate abstraction is a common abstraction technique used in this context. Given a program $\Pi$ and a set of predicates $\mathcal{P}$, verification with predicate abstraction consists of constructing and analyzing an automaton $A(\Pi, \mathcal{P})$, an abstraction of $\Pi$ relative to $\mathcal{P}$.

* This research was sponsored by the Semiconductor Research Corporation (SRC) under contract no. 99-TJ-684, the National Science Foundation (NSF) under grant no. CCR-9803774, the Office of Naval Research (ONR), and the Naval Research Laboratory (NRL) under contract no. N00014-01-1-0796.

© Springer-Verlag Berlin Heidelberg 2003
We will describe in more detail predicate abstraction for verification of C programs in section 2. For now let us just mention that the process of constructing $A(\Pi, \mathcal{P})$ is in the worst case exponential, both in time and space, in $|\mathcal{P}|$. Therefore a crucial point in deriving efficient algorithms based on predicate abstraction is the choice of a small set of predicates. In other words, one of the main challenges in making predicate abstraction effective is distinguishing a small set of predicates that are sufficient for determining whether a property holds or not. In this article we present an automated technique for finding the minimal such set from a given set of candidate predicates.

In the original article describing predicate abstraction [13] the process of selecting predicates is done manually. An automatic method for choosing predicates was suggested by Ball and Rajamani [2]. They follow a CounterExample Guided Abstraction Refinement (CEGAR) loop, which we now describe. Let $\phi$ be the property that we wish to verify over the program $\Pi$. We denote by $MC$ a model checking algorithm that takes both $A(\Pi, \mathcal{P})$ and $\phi$ as inputs and outputs TRUE if $A(\Pi, \mathcal{P}) \models \phi$ and a counterexample $\tau$ otherwise. We assume $\phi$ is a safety property, so that $\tau$ is a finite acyclic trace of $A(\Pi, \mathcal{P})$. Since $\tau$ is a trace of $A(\Pi, \mathcal{P})$, it is often called an abstract trace. Let $\gamma$ be a trace concretization function that maps every abstract trace to a sequence of instructions of $\Pi$ consistent with the control flow graph. In order to check whether this sequence is a valid trace of $\Pi$, we define a Trace Checking algorithm $TC$ that takes $\Pi$ and $\tau$ as inputs and returns TRUE if $\gamma(\tau)$ is a valid trace of $\Pi$ and FALSE otherwise. In the latter case $\tau$ is called a spurious counterexample. Finally, if $\tau$ is spurious, we need to eliminate it from the abstract model. We say that a set of predicates $\mathcal{P}'$ eliminates $\tau$ iff for every trace $\tau'$ of $A(\Pi, \mathcal{P}')$, $\gamma(\tau) \neq \gamma(\tau')$; i.e., the concretization of all traces in $A(\Pi, \mathcal{P}')$ are different from $\gamma(\tau)$. Given these definitions, we now describe the four steps of the CEGAR loop (usually $\mathcal{P} = \emptyset$ initially):

1. **Abstract.** Construct $A(\Pi, \mathcal{P})$.
2. **Verify.** If $MC(A(\Pi, \mathcal{P}), \phi) = \text{TRUE}$, return property holds.  
   Otherwise let $\tau$ be the counterexample.
3. **Check.** If $TC(\Pi, \tau) = \text{TRUE}$ return property does not hold.
4. **Refine.** Update $\mathcal{P}$ so as to eliminate $\tau$. Go to step 1.

Step 4 is the crucial one, and also the subject of this article. In previous work [2, 9] the refinement is done by adding predicates that eliminate the new spurious counterexample while maintaining the predicates that were found in previous iterations. This guarantees that no spurious counterexample will be repeated. However, this accumulative approach cannot guarantee a minimal set of predicates, because it depends on the order in which the counterexamples are identified and the choice of predicates at each step.

For example, consider a scenario where the first counterexample, $\tau_1$, can be eliminated by either $p_1$ or $p_2$, and the process chooses $p_1$. Now it finds another counterexample, $\tau_2$, which can only be eliminated by the predicate $p_2$. The process now proceeds with both $p_1$ and $p_2$, although $p_2$ by itself is sufficient to
eliminate both \( \tau_1 \) and \( \tau_2 \). The framework that we present in this article, on the other hand, finds a minimal set of predicates that eliminate all the spurious counterexamples discovered so far. This guarantees a minimal set of predicates throughout the process, which is expected to reduce the overall verification time and required space. Our experimental results show that indeed the number of predicates and consequently the amount of memory required are significantly reduced.

**Related Work.** Predicate abstraction was introduced by Graf and Saidi in [13]. It was subsequently used with considerable success in both hardware and software verification [2,8,9]. The notion of CEGAR was originally introduced by Kurshan [10] (originally termed localization) for model checking of finite state models. Both the abstraction and refinement techniques for such systems, as applied in his and consequent works, are essentially different than the predicate abstraction approach we follow. For example, abstraction in localization reduction is done by assigning non-deterministic values to selected sets of variables, while refinement corresponds to gradually returning to the original definition of these variables. More recently the CEGAR framework has also been successfully adapted for verifying infinite state systems [12], and in particular software [3,9]. The problem of finding small sets of predicates (yet not minimal) is also being investigated in the context of hardware designs in [5].

The rest of this article is structured as follows. In the next section we discuss in more detail the CEGAR loop for predicate abstraction and how it is used for verifying C programs. In section 3 we describe in detail the procedure for selecting a minimal set of predicates. In section 4 we present the results of applying our technique to several realistic examples and detail our conclusions.

## 2 Predicate Abstraction/Refinement for C Programs

In the introduction we discussed the overall structure of a CEGAR loop. In this section we explain how this framework can be applied for verifying C programs. We do so by describing how the various basic blocks of the CEGAR loop are implemented. In particular, we discuss the construction of \( A(\Pi, \mathcal{P}) \) in section 2.1, the notion of trace concretization (\( \gamma \)) in section 2.2, the trace checking algorithm \( TC \) in section 2.3, and a method for checking whether a set of predicates eliminates a spurious counterexample in section 2.4.

### 2.1 Constructing the Abstract Model

We begin with the process of constructing \( A(\Pi, \mathcal{P}) \) given a C program \( \Pi \) and an initial set of predicates \( \mathcal{P} \). For the sake of simplicity, we assume that \( \Pi \) consists of a single monolithic C \emph{main} procedure obtained via inlining (we disallow function pointers and recursion in order to make inlining effective). Without loss of generality, we can assume that there are only four kinds of statements in \( \Pi \): assignments, \texttt{if-then-else} branches, \texttt{goto} and \texttt{return}. We denote by \emph{Stmt} the
set of statements of $\Pi$ and by $\text{Exp}$ the set of all pure (side-effect free) C expressions over the variables of $\Pi$. As a running example we use the following simple C program and the property that label L4 is unreachable.

\[
\begin{align*}
\text{int } & x,y; \\
L0: & \quad x = 1; \\
L1: & \quad y = 1; \\
L2: & \quad \text{if } (x == y) \\
L3: & \quad y = 1; \\
L4: & \quad \text{else } y = 2;
\end{align*}
\]

**Initial Abstraction with Control Flow Automata.** The construction of $\mathcal{A}(\Pi,\mathcal{P})$ begins with the construction of the control flow automaton (CFA) of $\Pi$. The states of a CFA correspond to control points in the program. The transitions between states in the CFA correspond to possible transitions between their associated control points in the program, assuming that every branch in the program can be taken. Thus, a CFA of a program is a conservative abstraction of the program’s control flow, i.e. it allows a superset of the possible traces of the program.

Formally the CFA is a 4-tuple $\langle S_{CF}, I_{CF}, T_{CF}, \mathcal{L} \rangle$ where:

- $S_{CF}$ is a set of states.
- $I_{CF} \in S_{CF}$ is an initial state.
- $T_{CF} \subseteq S_{CF} \times S_{CF}$ is a set of transitions.
- $\mathcal{L} : S_{CF} \setminus \{\text{final}\} \rightarrow Stmt$ is a labeling function.

$S_{CF}$ contains a distinguished final state which does not belong to the domain of $\mathcal{L}$. The transitions between states reflect the flow of control between their labeling statements: $\mathcal{L}(I_{CF})$ is the initial statement of $\Pi$ and $(s_1, s_2) \in T_{CF}$ iff one of the following conditions hold:

- $\mathcal{L}(s_1)$ is an assignment or goto with $\mathcal{L}(s_2)$ as its unique successor.
- $\mathcal{L}(s_1)$ is a branch with $\mathcal{L}(s_2)$ as its then or else successor.
- $\mathcal{L}(s_1)$ is a return statement and $s_2 = \text{final}$.

The CFA is equivalent, as we will shortly see, to $\mathcal{A}(\Pi, \emptyset)$.

**Example 1.** The CFA of our example program is shown in Figure 1(a), where every state $s$ is labeled with $\mathcal{L}(s)$. Henceforth we will refer to each CFA state by the corresponding statement label. We will use final for the final state. Therefore the states of the CFA in Figure 1(a) are L0 \ldots L4 and final with L0 being the initial state. \hfill \square

**Predicate Inference.** The main challenge in predicate abstraction is to identify the predicates that are necessary for proving the given property. In our framework we require $\mathcal{P}$ to be a subset of the branch statements in $\Pi$. Therefore we sometimes refer to $\mathcal{P}$ or subsets of $\mathcal{P}$ simply as a set of branches, where the
Fig. 1. (a) The CFA for our example program, (b) The CFA labeled with inferred predicates if \( P = \{ x = y \} \), i.e., it contains the only branch in the program, and (c) The abstract automaton \( \mathcal{A}(\Pi, P) \), which proves that L4 is not reachable.

The algorithm uses a procedure for computing the weakest precondition \( WP \) of a predicate \( p \) relative to a given statement. We define \( WP \) in the same way as Ball and Rajamani [2]. First, consider a C assignment statement \( a \) of the form \( v = e \). Let \( \varphi \) be a pure C expression (\( \varphi \in Exp \)). Then the weakest precondition of \( \varphi \) with respect to \( a \), denoted by \( WP(\varphi, a) \), is obtained from \( \varphi \) by replacing every occurrence of \( v \) in \( \varphi \) with \( e \). A second case considers a C assignment statement \( a \) in which \( e \) is assigned to a variable whose address is stored in \( v \), i.e. \( a \) is of the form \( *v = e \). Let \( \{v_1, \ldots, v_n\} \) be the set of variables appearing in \( \varphi \) and for \( 1 \leq i \leq n \) let \( a_i \) be the assignment statement \( v_i = e \); \( WP(\varphi, a) \) is then:

\[
(\bigwedge_{i=1}^{n}((v == &v_i) \land WP(\varphi, a_i))) \land \bigwedge_{i=1}^{n}((v! = &v_i)) \land \varphi
\]

The weakest precondition is clearly an element of \( Exp \) as well. The purpose of predicate inference is to create \( P_s \)'s that lead to a very precise abstraction of the program relative to the predicates in \( P \). Intuitively, this is how it works. Let \( s, t \in S_{CP} \) such that \( L(s) \) is an assignment statement and \( (s, t) \in T_{CP} \). Suppose a predicate \( p_t \) gets inserted in \( P_t \) at some point during the execution of \( PredInfer \) and suppose \( p_s = WP(p_t, L(s)) \). Now consider any execution state of \( \Pi \) where the control has reached \( L(t) \) after the execution of \( L(s) \). It is obvious that \( p_t \) will be true in this state iff \( p_s \) was true before the execution of \( L(s) \). In terms of the CFA, this means that the value of \( p_t \) after a transition from \( s \) to \( t \) can be determined precisely on the basis of the value of \( p_s \) before the transition. This motivates the inclusion of \( p_s \) in \( P_s \). The cases in which \( L(s) \) is not an assignment statement can be explained analogously.
Input: Set of branch statements $\mathcal{P}$  
Output: Set of $\mathcal{P}_s$'s associated with each CFA state  
Initialize: $\forall s \in S_{CF}, \mathcal{P}_s := \emptyset$  
Forever do  
For each $s \in S_{CF}$ do  
  If $\mathcal{L}(s)$ is an assignment statement and $\mathcal{L}(s')$ is its successor  
    For each $p' \in \mathcal{P}_s'$ add $WP(p', \mathcal{L}(s))$ to $\mathcal{P}_s$  
  Else if $\mathcal{L}(s)$ is a branch statement with condition $c$  
    If $\mathcal{L}(s) \in \mathcal{P}$ add $c$ to $\mathcal{P}_s$  
    If $\mathcal{L}(s')$ is a 'then' or 'else' successor of $\mathcal{L}(s)$, $\mathcal{P}_s := \mathcal{P}_s \cup \mathcal{P}_s'$  
    Else if $\mathcal{L}(s)$ is a 'goto' statement with successor $\mathcal{L}(s')$, $\mathcal{P}_s := \mathcal{P}_s \cup \mathcal{P}_s'$  
  If no $\mathcal{P}_s$ was modified in the 'for' loop, exit  

Fig. 2. Algorithm $PredInfer$ for predicate inference.

Note that $PredInfer$ may not terminate in the presence of loops in the CFA. However, this does not mean that our approach is incapable of handling C programs containing loops. In practice, we force termination of $PredInfer$ by limiting the maximum size of any $\mathcal{P}_s$. Using the resulting $\mathcal{P}_s$'s, we can compute the states and transitions of the abstract model as described in the next section. Irrespective of whether $PredInfer$ was terminated forcefully or not, the resulting model is guaranteed to be a sound abstraction of $\Pi$. We have found this approach to be very effective in practice. A similar algorithm was proposed by Dams and Namjoshi [7].

Example 2. Consider the CFA described in Example 1. Suppose $\mathcal{P}$ contains the only branch (L2) in our example program. Then $PredInfer$ begins with $\mathcal{P}_{L2} = \{(x == y)\}$. From this it obtains $\mathcal{P}_{L1} = \{WP((x == y), y = 1; )\} = \{(x == 1)\}$ and then $\mathcal{P}_{L0} = \{WP((x == 1), x = 1; )\} = \{(1 == 1)\}$. As $1 == 1$ is trivially true, we do not include it in $\mathcal{P}_{L0}$. Thus $\mathcal{P}_{L0} = \emptyset$. Finally $\mathcal{P}_{L3} = \mathcal{P}_{L4} = \mathcal{P}_{final} = \emptyset$. Figure 1(b) shows the CFA with each state $s$ labeled on the outside by $\mathcal{P}_s$. □

The States and Transitions of the Abstract Model. So far we have described a method for computing the initial abstraction (the CFA) and a set of predicates associated with each location in the program. The states of the abstract system $A(\Pi, \mathcal{P})$ correspond to the various possible valuations of the predicates in each location (this is the reason why the abstract graph is exponential in the number of predicates). Formally, for a CFA node $s$ suppose $\mathcal{P}_s = \{p_1, \ldots, p_k\}$. Then a valuation of $\mathcal{P}_s$ is a boolean vector $v_1, \ldots, v_k$. Let $\mathcal{V}_s$ be the set of all predicate valuations of $\mathcal{P}_s$. Then the predicate concretization function $\Gamma_s : \mathcal{V}_s \rightarrow Exp$ is defined as follows. Given a valuation $V = \{v_1, \ldots, v_k\} \in \mathcal{V}_s$, $\Gamma_s(V) = \bigwedge_{i=1}^{k} p_i^{v_i}$ where $p_i^{TRUE} = p_i$ and $p_i^{FALSE} = \neg p_i$. As a special case, if $\mathcal{P}_s = \emptyset$, then $\mathcal{V}_s = \{\bot\}$ and $\Gamma_s(\bot) = \text{TRUE}$.

Example 3. Suppose $\mathcal{P}_s = \{(a == 0), (b > 5), (c < d)\}$, $V_1 = \{0, 1, 1\}$ and $V_2 = \{1, 0, 1\}$. Then $\Gamma_s(V_1) = (\neg(a == 0)) \land (b > 5) \land (c < d)$ and $\Gamma_s(V_2) = (a == 0) \land (\neg(b > 5)) \land (c < d)$. □
Computing the transitions between the states in $\mathcal{A}(\Pi, \mathcal{P})$ requires a theorem prover. We add a transition between two abstract states unless we can prove that there is no transition between their corresponding concrete states. If we cannot prove this, we say that the two states (or the two formulas representing them) are admissible. This problem can be reduced to the problem of deciding whether $\neg(\psi_1 \land \psi_2)$ is valid, where $\psi_1$ and $\psi_2$ are arbitrary quantifier free first order logic formulas. In general this problem is known to be undecidable. However for our purposes it is sufficient that the theorem prover be sound and always terminate. Several publicly available theorem provers (such as Simplify [11]) have this characteristic.

Given arbitrary formulas $\psi_1$ and $\psi_2$, we say that the formulas are admissible if the theorem prover returns \texttt{false} or \texttt{unknown} on $\neg(\psi_1 \land \psi_2)$. We denote this by $\text{Adm}(\psi_1, \psi_2)$. Otherwise the formulas are inadmissible, denoted by $\neg \text{Adm}(\psi_1, \psi_2)$.

**A Procedure for Constructing $\mathcal{A}(\Pi, \mathcal{P})$.** We now define $\mathcal{A}(\Pi, \mathcal{P})$. Formally, it is a triple $(\mathcal{S}_A, I_A, T_A)$ where:

- $\mathcal{S}_A = \cup s \in \mathcal{S}_{CF} \{s\} \times \mathcal{V}_s$ is the set of states.
- $I_A = \{I_{CF}\} \times \mathcal{V}_{CF}$ is the initial set of states.
- $T_A \subseteq \mathcal{S}_A \times \mathcal{S}_A$ is the transition relation, defined as follows: $((s_1, V_1), (s_2, V_2)) \in T_A$ iff $(s_1, s_2) \in T_{CF}$ and one of the following conditions hold:
  1. $L(s_1)$ is an assignment statement and $\text{Adm}(\Gamma_{s_1}(V_1), \text{WP}(\Gamma_{s_2}(V_2), L(s_1)))$.
  2. $L(s_1)$ is a branch statement with a branch condition $c$, $L(s_2)$ is its then successor, $\text{Adm}(\Gamma_{s_1}(V_1), \Gamma_{s_2}(V_2))$ and $\text{Adm}(\Gamma_{s_1}(V_1), c)$.
  3. $L(s_1)$ is a branch statement with a branch condition $c$, $L(s_2)$ is its else successor, $\text{Adm}(\Gamma_{s_1}(V_1), \Gamma_{s_2}(V_2))$ and $\text{Adm}(\Gamma_{s_1}(V_1), \neg c)$.
  4. $L(s_1)$ is a goto statement and $\text{Adm}(\Gamma_{s_1}(V_1), \Gamma_{s_2}(V_2))$.
  5. $L(s_1)$ is a return statement and $s_2$ is the final state.

**Example 4.** Recall the CFA from Example 1 and the predicates corresponding to CFA nodes discussed in Example 2. The $\mathcal{A}(\Pi, \mathcal{P})$ obtained in this case appears in Figure 1(c). Let us see why there is a transition from $(L_0, \bot)$ to $(L_1, \text{true})$. Since $L(L_0)$ is an assignment statement, by rule 1 above we compute the following expressions:

- $I_{L_0}(\bot) = \text{true}$
- $I_{L_1}(\text{true}) = (x == 1)$
- $\mathcal{L}(L_0) = (x = 1)$
- $\text{WP}(I_{L_1}(\text{true}), L(L_0)) = \text{WP}((x == 1), x = 1; ) = (1 == 1) = \text{true}$
- $\text{Adm}(\text{true}, \text{true})$.

Thus, we add a transition from $(L_0, \bot)$ to $(L_1, \text{true})$. Examining a possible transition from $(L_0, \bot)$ to $(L_1, \text{false})$, we similarly compute $I_{L_1}(\text{false}) = (x ==
Input: A trace $\tau$ of $\mathcal{A}(\Pi, \mathcal{P})$ s.t. $\gamma(\tau) = \langle s_1, \ldots, s_n \rangle$
Output: TRUE iff $\tau$ is valid (can be simulated on the concrete system)
Variable: $X$ of type formula
Initialize: $X :=$ TRUE
For $i = n$ to 1
  If $s_i$ is an assignment
    $X := \mathcal{WP}(X, s_i)$
  Else if $s_i$ is a branch with condition $c$
    If ($i < n$)
      If $s_{i+1}$ is the ‘then’ successor of $s_i$, $X := X \land c$
      else $X := X \land \neg c$
    If ($X \equiv$ FALSE) return FALSE
Return TRUE

Fig. 3. Algorithm $\mathcal{TC}$ to check the validity of a trace of $\Pi$.

1)) and $\mathcal{WP}((\neg(x == 1)), x = 1;) = (\neg(1 == 1))$. Since $\neg\text{Adm}(\text{TRUE}, (\neg(1 == 1)))$, there is no transition between these two abstract states. The presence or absence of other transitions can be explained in a similar manner. As no state labeled by L4 is reachable, we have proven that our example property holds. □

Clearly, if we do not limit the size of $\mathcal{P}_s$, $|S_A|$ is exponential in $|\mathcal{P}|$. Hence so are the worst case space and time complexities of constructing $\mathcal{A}(\Pi, \mathcal{P})$.

2.2 Trace Concretization

A trace of $\mathcal{A}(\Pi, \mathcal{P})$ is a finite sequence $\langle(s_1, V_1), \ldots, (s_n, V_n)\rangle$ such that (i) for $1 \leq i \leq n$, $(s_i, V_i) \in S_A$, (ii) $(s_1, V_1) \in I_A$ and (iii) for $1 \leq i < n$, $((s_i, V_i), (s_{i+1}, V_{i+1})) \in T_A$. Given such a trace $\tau = \langle(s_1, V_1), \ldots, (s_n, V_n)\rangle$ of $\mathcal{A}(\Pi, \mathcal{P})$, the concretization of $\tau$ is defined as $\gamma(\tau) = \langle L(s_1), \ldots, L(s_n)\rangle$. Thus, the concretization of an abstract trace is a trace of $\Pi$: a sequence of statements that correspond to some trace in the control flow graph of $\Pi$.

2.3 Trace Checking

The $\mathcal{TC}$ algorithm, described in Figure 3, takes $\Pi$ and a counterexample $\tau$ as inputs and returns TRUE if $\gamma(\tau)$ is a valid trace of $\Pi$. This is a backward traversal based algorithm. There is an equivalent algorithm [3] that is forward traversal based and uses strongest postconditions instead of weakest preconditions.

2.4 Checking Trace Elimination

Given a spurious counterexample $\tau = \langle(s_1, V_1), \ldots, (s_n, V_n)\rangle$ and a set of branches $\mathcal{P}$, we will need to determine if $\mathcal{P}$ eliminates $\tau$. To do so we: (i) construct $\mathcal{A}(\Pi, \mathcal{P})$ and (ii) determine if there exists a trace $\tau'$ of $\mathcal{A}(\Pi, \mathcal{P})$ such that $\gamma(\tau) = \gamma(\tau')$. The algorithm, called $\text{TraceEliminate}$, is described in Figure 4.$\dagger$

$\dagger$ Note that in practice this step can be carried out in an on-the-fly manner without constructing the full $\mathcal{A}(\Pi, \mathcal{P})$. 
Predicate Abstraction with Minimum Predicates

Input: Spurious trace \( \tau \) s.t. \( \gamma(\tau) = \langle s_1, \ldots, s_n \rangle \) and a set of predicates \( \mathcal{P} \)
Output: \text{true} if \( \tau \) is eliminated by \( \mathcal{P} \) and \text{false} otherwise

Compute \( A(\Pi, \mathcal{P}) = \langle S_A, I_A, T_A \rangle \)

Variable: \( X, Y \) of type subset of \( S_A \)
Initialize: \( X := \{(s, V) \in S_A \mid s = s_1\} \)
If \((X = \emptyset)\) return \text{true}
For \(i = 2\) to \(n\) do
\(Y := \{(s', V') \in S_A \mid (s' = s_i) \land \exists (s, V) \in X. ((s, V), (s', V')) \in T_A\}\)
If \((Y = \emptyset)\) return \text{true}
\(X := Y\)
Return \text{false}

Fig. 4. Algorithm \textit{TraceEliminate} to check if a spurious trace can be eliminated.

3 Predicate Minimization

We now present the algorithm for discovering a \textit{minimal} set of branches \( \mathcal{P} \) of a program \( \pi \) that will help us prove or disprove a safety property \( \phi \).

3.1 The \textit{Sample-and-Eliminate} Algorithm

Algorithm \textit{Sample-and-Eliminate}, described in Figure 5, is based on an abstraction refinement loop that keeps the set of predicates minimal throughout the process. It is modeled after the \textit{Sample-and-Separate} algorithm [6], where it is used in a CEGAR framework for hardware verification. At each step it finds a counterexample if one exists and checks whether it corresponds to a concrete counterexample, as usual. Unlike previous approaches [3,9], however, it finds a minimal set of predicates that eliminate all the concrete spurious traces that were found so far (in the last line of the loop.) Our approach to solving this minimization problem is the subject of Section 3.2.

Input: Program \( \Pi \), safety property \( \phi \)
Output: \text{true} if proved that \( \Pi \models \phi \), \text{false} if proved \( \Pi \not\models \phi \), and \text{unknown} otherwise.

Variable: \( T \) set of spurious counterexamples, \( P \) set of predicates
Initialize: \( T := \emptyset \), \( P := \emptyset \)
Forever do
If \( \mathcal{MC}(A(\Pi, P), \phi) = \text{true} \) return \text{true}
Else let \( \tau \) be the abstract counterexample
If \( \mathcal{T C}(\tau) = \text{true} \) return \text{false}
If \( P \) is the set of all branches in \( \Pi \) then return \text{unknown}
\( T := T \cup \{\tau\} \)
\( P := \text{minimal set of branches of } \Pi \text{ that eliminates all elements of } T \)

Fig. 5. Algorithm \textit{Sample-and-Eliminate} uses a minimal set of predicates taken from a program’s branches to prove or disprove \( \Pi \models \phi \), if such a proof is possible.
3.2 Minimizing the Eliminating Set

The last line of Sample-and-Eliminate presents the following problem: given a set of spurious counterexamples \( T \) and a set of candidate predicates \( P \) (all the branches of \( \Pi \) in our case), find a minimal set \( p \subseteq P \) which eliminates all the traces in \( T \). We present a three step algorithm for solving this problem. First, find a mapping \( T \mapsto 2^P \) between each trace in \( T \) and the set of sets of predicates in \( P \) that eliminate it. This can be achieved by iterating through every \( p \subseteq P \) and \( \tau \in T \), using TraceEliminate to determine if \( p \) can eliminate \( \tau \). This approach is exponential in \( |P| \) but below we list several ways to reduce the number of attempted combinations:

- Limit the size or number of attempted combinations to a small constant, e.g. 5, assuming that most traces can be eliminated by a small set of predicates.
- Stop after reaching a certain size of combinations if any eliminating solutions have been found.
- Break up the control flow graph into blocks and only consider combinations of predicates within blocks (keeping combinations in other blocks fixed).
- Use data flow analysis to only consider combinations of related predicates.
- For any \( \tau \in T \), if a set \( p \) eliminates \( \tau \), ignore all supersets of \( p \) with respect to \( \tau \) (as we are seeking a minimal solution).

Second, encode each predicate \( p_i \in P \) with a new Boolean variable \( p_i^b \). We use the terms ‘predicate’ and ‘the Boolean encoding of the predicate’ interchangeably. Third, derive a Boolean formula \( \sigma \), based on the predicate encoding, that represents all the possible combinations of predicates that eliminate the elements of \( T \). We use the following notation in the description of \( \sigma \). Let \( \tau \in T \) be a trace:

- \( k_\tau \) denotes the number of sets of predicates that eliminate \( \tau \) (\( 1 \leq k_\tau \leq 2^{|P|} \)).
- \( s(\tau, i) \) denotes the \( i \)-th set (\( 1 \leq i \leq k_\tau \)) of predicates that eliminates \( \tau \). We use the same notation for the conjunction of the predicates in this set.

The formula \( \sigma \) is defined as follows:

\[
\sigma = \bigwedge_{\tau \in T} \bigvee_{i=1}^{k_\tau} s(\tau, i)
\]  

(1)

For any satisfying assignment to \( \sigma \), the predicates whose Boolean encodings are assigned \textsc{true} are sufficient for eliminating all elements of \( T \).

From the various possible satisfying assignments to \( \sigma \), we look for the one with the smallest number of positive assignments. This assignment represents the minimal number of predicates that are sufficient for eliminating \( T \). Since \( \sigma \) includes disjunctions, it cannot be solved directly with a 0-1 ILP solver. We therefore use PBS [1], a solver for Pseudo Boolean Formulas.

A pseudo-Boolean formula is of the form \( \sum_{i=1}^n c_i b_i \triangleright k \), where \( b_i \) is a Boolean variable and \( c_i \) is a rational constant for \( 1 \leq i \leq n \). \( k \) is a rational constant and \( \triangleright \) represents one of the inequality or equality relations (\( \{<,\leq,>,\geq,=\} \)). Each
such constraint can be expanded to a CNF formula (hence the name pseudo-Boolean), but this expansion can be exponential in $n$. PBS does not perform this expansion, but rather uses an algorithm designed in the spirit of the Davis-Putnam-Loveland algorithm that handles these constraints directly. PBS accepts as input standard CNF formulas augmented with pseudo-Boolean constraints. Given an objective function in the form of pseudo-Boolean formula, PBS finds an optimal solution by repeatedly tightening the constraint over the value of this function until it becomes unsatisfiable. That is, it first finds a satisfying solution and calculates the value of the objective function according to this solution. It then adds a constraint that the value of the objective function should be smaller by one. This process is repeated until the formula becomes unsatisfiable. The objective function in our case is to minimize the number of chosen predicates (by minimizing the number of variables that are assigned TRUE):

$$\min \sum_{i=1}^{n} p_i^b$$  \hspace{1cm} (2)

**Example 5.** Suppose that the trace $\tau_1$ is eliminated by either $\{p_1, p_3, p_5\}$ or $\{p_2, p_5\}$ and that the trace $\tau_2$ can be eliminated by either $\{p_2, p_3\}$ or $\{p_4\}$. The objective function is $\min \sum_{i=1}^{5} p_i^b$ and is subject to the constraint:

$$\sigma = ((p_2^b \land p_3^b \land p_5^b) \lor (p_2^b \land p_4^b)) \land ((p_2^b \land p_3^b) \lor (p_4^b))$$

The minimal satisfying assignment in this case is $p_2^b = p_3^b = p_4^b = \text{TRUE.}$ \hfill \Box

Other techniques for solving this optimization problem are possible, including minimal hitting sets and logic minimization. The PBS step, however, has not been a bottleneck in any of our experiments.

### 4 Experiments and Conclusions

We implemented our technique inside the MAGIC [4] tool. MAGIC was designed to check weak simulation of properties of labeled transition systems (LTSs) derived from C programs. We experimented with MAGIC with and without predicate optimization. We also performed experiments with a greedy predicate minimization strategy implemented on top of MAGIC. In each iteration, this strategy first adds predicates sufficient to eliminate the spurious counterexample to the predicate set $\mathcal{P}$. Then it attempts to reduce the size of the resulting $\mathcal{P}$ by using the algorithm described in Figure 6. The advantage of this approach is that it requires only a small overhead (polynomial) compared to Sample-and-Eliminate, but on the other hand it does not guarantee an optimal result. Further, we performed experiments with Berkeley’s BLAST [9] tool. BLAST also takes C programs as input, and uses a variation of the standard CEGAR loop based
Input: Set of predicates $\mathcal{P}$
Output: Subset of $\mathcal{P}$ that eliminates all spurious counterexamples so far
Variable: $X$ of type set of predicates

LOOP: Create a random ordering $\langle p_1, \ldots, p_k \rangle$ of $\mathcal{P}$
For $i = 1$ to $k$ do
    $X := \mathcal{P} \setminus \{p_i\}$
    If $X$ can eliminate every spurious counterexample seen so far
        $\mathcal{P} := X$
    Goto LOOP
Return $\mathcal{P}$

Fig. 6. Greedy predicate minimization algorithm.

on lazy abstraction, but without minimization. Lazy abstraction refines an abstract model while allowing different degrees of abstraction in different parts of a program, without requiring recomputation of the entire abstract model in each iteration. Laziness and predicate minimization are, for the most part, orthogonal techniques. In principle a combination of the two might produce better results than either in isolation.

Benchmarks. We used two kinds of benchmarks. A small set of relatively simple benchmarks were derived from the examples supplied with the BLAST distribution and regression tests for MAGIC. The difficult benchmarks were derived from the C source code of openssl-0.9.6c, several thousand lines of code implementing the SSL protocol used for secure transfer of information over the Internet. A critical component of this protocol is the initial handshake between a server and a client. We verified different properties of the main routines that implement the handshake. The names of benchmarks that are derived from the server routine and client routine begin with ssl-srvr and ssl-clnt respectively. In all our benchmarks, the properties are satisfied by the implementation. The server and client routines have roughly 350 lines each but, as our results indicate, are non-trivial to verify.

Results. Figure 7 summarizes our results. Time for all experiments is given in seconds. All experiments were performed on an AMD Athlon XP 1600 machine with 900 MB of RAM running RedHat 7.1. The column Iter reports the number of iterations through the CEGAR loop necessary to complete the proof. Predicates are listed differently for the two tools. For BLAST, the first number is the total number of predicates discovered and used and the second number is the number of predicates active at any one point in the program (due to lazy abstraction this may be smaller). In order to force termination we imposed a limit of three hours on the running time. We denote by ‘*’ in the Time column examples that could not be solved in this time limit. In these cases the other columns indicate relevant measurements made at the point of forceful termination.

For MAGIC, the first number is the total number of expressions used to prove the property, i.e. $|\cup_{s \in S_C} \mathcal{P}_s|$. The number of predicates (the second number)
may be smaller, as MAGIC combines multiple mutually exclusive expressions (e.g. $x = 1$, $x < 1$, and $x > 1$) into a single, possibly non-binary predicate, having a number of values equal to the number of expressions (plus one, if the expressions do not cover all possibilities.) The final number for MAGIC is the size of the final $P$. For experiments in which memory usage was large enough to be a measure of state space size rather than overhead, we also report memory usage (in megabytes).

The first MAGIC results are for the MAGIC tool operating in the standard refinement manner: in each iteration, predicates sufficient to eliminate the spurious counterexample are added to the predicate set. The second MAGIC results are for the greedy predicate minimization strategy. The last MAGIC results are for predicate minimization. Rather than solving the full optimization problem, we simplified the problem as described in section 3. In particular, for each trace we only considered the first 1,000 combinations and only generated 20 eliminating combinations. The combinations were considered in increasing order of size. After all combinations of a particular size had been tried, we checked whether at least one eliminating combination had been found. If so, no further combinations were tried. In the smaller examples we observed no loss of optimality due to these restrictions. We also studied the effect of altering these restrictions on the larger benchmarks and we report on our findings later.

Fig. 7. Results for BLAST and MAGIC with different refinement strategies. ‘*’ indicate run-time longer than 3 hours. ‘x’ indicate negligible values. Best results are emphasized.
For the smaller benchmarks, the various abstraction refinement strategies do not differ markedly. However, for our larger examples, taken from the SSL source code, the refinement strategy is of considerable importance. Predicate minimization, in general, reduced verification time (though there were a few exceptions to this rule, the average running time was considerably lower than for the other techniques, even with the cutoff on the running time). Moreover, predicate minimization reduced the memory needed for verification, which is an even more important bottleneck. Given that the memory was cutoff in some cases for other techniques before verification was complete, the results are even more compelling.

The greedy approach kept memory use fairly low, but almost always failed to find near-optimal predicate sets and converged much slower than the usual monotonic refinement or predicate minimization approaches. Further, it is not clear how much final memory usage would be improved by the greedy strategy if it were allowed to run to completion. Another major drawback of the greedy approach is its unpredictability. We observed that on any particular example, the greedy strategy might or might not complete within the time limit in different executions. Clearly, the order in which this strategy tries to eliminate predicates in each iteration is very critical to its success. Given that the strategy performs poorly on most of our benchmarks using a random ordering, more sophisticated ordering techniques may perform better. We leave this issue for future research.

**Optimality.** We experimented with two of the parameters that affect the optimality of our predicate minimization algorithm: (i) the maximum number of examined subsets (MAXSUB) and (ii) the maximum number of eliminating subsets generated (MAXELM) (that is, the procedure stops the search if MAXELM eliminating subsets were found, even if less than MAXSUB combinations were tried). We first kept MAXSUB fixed and took measurements for different values of MAXELM on a subset of our benchmarks viz. ssl-srvr-4, ssl-srvr-15 and ssl-clnt-1. Our results, shown in Figure 8, clearly indicate that the optimality is practically unaffected by the value of MAXELM.

Next we experimented with different values of MAXSUB (the value of MAXELM was set equal to MAXSUB). The results we obtained are summarized in Figure 9. It appears that, at least for our benchmarks, increasing MAXSUB
leads only to increased execution time without reduced memory consumption or number of predicates. The additional number of combinations attempted or constraints allowed does not lead to improved optimality. The most probable reason is that, as shown by our results, even though we are trying more combinations, the actual number or maximum size of eliminating combinations generated does not increase significantly. It would be interesting to investigate whether this is a feature of most real-life programs. If so, it would allow us, in most cases, to achieve near optimality by trying out only a small number of combinations or only combinations of small size.

Acknowledgments. We thank Rupak Majumdar and Ranjit Jhala for their help with BLAST.

References


Efficient Symbolic Model Checking of Software Using Partial Disjunctive Partitioning

Sharon Barner and Ishai Rabinovitz
IBM Haifa Research Laboratory, Haifa, Israel

Abstract. This paper presents a method for taking advantage of the efficiency of symbolic model checking using disjunctive partitions, while keeping the number and the size of the partitions small. We define a restricted form of a Kripke structure, called an or-structure, for which it is possible to generate small disjunctive partitions. By changing the image and pre-image procedures, we keep even smaller partial disjunctive partitions in memory. In addition, we show how to translate a (software) program to an or-structure, in order to enable efficient symbolic model checking of the program using its disjunctive partitions. We build one disjunctive partition for each state variable in the model directly from the conjunctive partition of the same variable and independently of all other partitions. This method can be integrated easily into existing model checkers, without changing their input language, and while still taking advantage of reduction algorithms which prefer conjunctive partitions.

1 Introduction

Symbolic model checking suffers from the known problem of state explosion. This explosion usually happens while performing the image or pre-image computation. In order to cope with this problem, symbolic model checkers use partitioned transition relations [8]. Using ordered conjunctive partitioning [7] is quite simple and sometimes allows early quantification while computing the image or pre-image; this serves to decrease the needed memory.

The RuleBase model checker [1] uses ordered conjunctive partitioning, and previous work showed its application to general purpose software [5,6]. In this paper, we show how disjunctive partitioning can be used to increase the efficiency of symbolic model checking for software.

Disjunctive partitioning, first introduced in [8], has several advantages over conjunctive partitioning. First, both image and pre-image computations are more efficient using disjunctive partitions, since quantification distributes over disjunction but not over conjunction [9,8]. For the same reason, distributed model checking using disjunctive partitions is also more scalable than using conjunctive partitioning, since each process can do the quantification on its own. As a result, the “heavy” computation is divided by the number of processes.

Despite the advantages of disjunctive partitioning, use of the technique is generally hindered by the difficulty in building the partitions. The method presented in [8] is efficient only for asynchronous circuits. It builds the disjunctive
partitions using an interleaving model, which allows only one wire to change its value at a time.

Both [2] and [4] suggested how to build disjunctive partitions for synchronous circuits. In [2], we see how to decompose an FSM into smaller FSMs, and then use this decomposition to split the conjunctive partitioned transition relation into a disjunction of conjunctive partitioned transition relations. In [4], a set of mutually exclusive events is used to decompose the behavior of the circuit to disjunctive partitions. Large disjunctive partitions are split into conjunctive partitions, which results in a DNF partitioning as in [2]. Both methods need additional information on the circuit in order to get a good decomposition.

Disjunctive partitioning is also used in [10], where each transition is a separate disjunctive partition. The contribution of [10] is in presenting the order in which the transitions should be executed in order to achieve improved performance.

While all the above works are applicable to models generated for software, applying them to software is problematic. The method of [8] is applicable to parallel software, but does not decompose each process to disjunctive partitions. On the other hand, [10] creates a large number of disjunctive partitions. The methods of [2] and [4] are not automated and require additional information from the user. We introduce a new method applicable to software models in which the decomposition is generated automatically, without additional information from the user. The number of disjunctive partitions created is similar to that of the conjunctive partitions for the same model, and the BDD size of the disjunctive partitions is comparable to that of the conjunctive partitions.

Software has the feature that in each step there is little change in the program variables. It is quite easy to build a model for software where each step changes only the pc (program counter), and at most, one additional state variable. We present a modeling language called ODL, which is natural for defining such models. We also present a method for translating from conjunctive partitions to disjunctive partitions and vice versa. These translations can be easily adapted by any symbolic model checker that uses conjunctive partitioning and by doing so, may benefit from the advantages of disjunctive partitioning.

In the traditional image computation algorithm, each disjunctive partition must represent the next value for all variables, so the disjunctive partition of state variable \( x \) should indicate the change of \( x \) and \( pc \), and the fact that all other variables keep their value. The latter information might severely impact the BDD size of the partition. In this work, we change the image and pre-image computation in such a way that they can work on the partial disjunctive partition of \( x \), which represents only the changes of \( x \) and \( pc \), and not the fact that all other variables keep their value. Using this algorithm decreases the BDD size needed to represent the disjunctive partitions and improves the image computation. This method is applicable not only for software models, but also to some other methods ([8], [2] and [4]) based on the fact that only a subset of the variables in the model can change their value in each disjunctive partition.

Finally we suggest two schemes for distributed model checking that use the disjunctive partitioning.
In our work we implemented the translation from conjunctive partitioned transition relation to disjunctive partitioned transition relation. We show that the size of the partial disjunctive partitions is equal to, or even smaller than, the size of the conjunctive partitions. In addition, we show that calculating reachability analysis using disjunctive partitions significantly outperforms calculation using conjunctive partitions.

The remainder of this paper is structured as follows: Section 2 states the preliminaries. Section 3 presents the generation of the model from the software and the ODL modeling language. Section 4 presents the translation between conjunctive and disjunctive partitions, and vice versa. Section 5 introduces partial disjunctive partitions and their advantages, and Section 6 presents the distributed version. In Section 7 we present some experimental results. We conclude and suggest some directions for future work in Section 8.

2 Preliminaries

A finite program can be modeled by a Kripke structure $M$ over a set of atomic propositions $AP$. $M = (S, S_0, R, L)$, where $S$ is a finite set of states, $S_0$ is a set of initial states, $R \subseteq S \times S$ is a total transition relation, and $L : S \rightarrow 2^{AP}$ is a labeling function that labels each state with the set of atomic propositions that are true in that state. The states of the Kripke structure are coded by a set of state variables $\bar{v}$. Each valuation to $\bar{v}$ is a state in the structure. Model checking is a technique for verifying finite state systems represented as Kripke structures. The basic operations in model checking are the image computation and the pre-image computation. Given a set of states $S$ and a transition relation $R$, represented in symbolic model checking by the BDDs $S(\bar{v})$ and $R(\bar{v}, \bar{v}')$ respectively, the image computation finds the set of all states related by $R$ to some state in $S$ and the pre-image computation finds the set of all states such that some state in $S$ is related to them by $R$. More precisely, $image(S(\bar{v}), R(\bar{v}, \bar{v}')) = \exists \bar{v}'(S(\bar{v}) \land R(\bar{v}, \bar{v}'))$ and $pre\_image(S(\bar{v}'), R(\bar{v}, \bar{v}')) = \exists \bar{v}'(S(\bar{v}') \land R(\bar{v}, \bar{v}'))$. The result of $image(S(\bar{v}), R(\bar{v}, \bar{v}'))$ is over $\bar{v}'$. In order to get the result over $\bar{v}$, all BDD variables are “unprimed”.

A conjunctive partitioned transition relation is composed of a set of partitions $and\_R_i$ such that $R(\bar{v}, \bar{v}') = \bigwedge_i and\_R_i(\bar{v}, \bar{v}')$. In case each state variable can be described by a single conjunctive partition (as in this work), we have that $and\_R_{v_i} = (v'_i = f_{v_i}(\bar{v}))$ and thus each partition is a function of $\bar{v}$ and $v'_i$ rather than $v_i$ and $\bar{v}'$. The image computation in this case is $image(S(\bar{v})) = \exists \bar{v}'(S(\bar{v}) \land (\bigwedge_i and\_R_{v_i}(\bar{v}, v'_i)))$.

Computing $\exists x A(\bar{v})$ is referred to as quantifying $x$ out of $A$. Early quantification [8] can make image and pre_image computations even more efficient. Early quantification is done by quantifying a variable $x$ out of the intermediate BDD result, after conjunction the last conjunctive partition that is dependent on $x$. Quantifying a variable out of the intermediate BDD may reduce the size of the BDD and as a result make the image computation easier.
A disjunctive partitioned transition relation is composed of a set of disjunctive partitions or \( R_i \) such that \( R(\bar{v}, \bar{v}') = \bigvee_i \text{or } R_i(\bar{v}, \bar{v}') \). In the case where each state variable can be changed only in a single disjunctive partition, we have that \( \text{or } R_{v_i} = (v'_i = f_{v_i}(\bar{v})) \land (\forall y \neq v_i : y = y') \). The image computation when using disjunctive partitions is done “early”, and thus \( \text{image}(S(\bar{v})) = \exists \bar{v}(S(\bar{v}) \land (\bigvee v_i \text{or } R_{v_i}(\bar{v}, \bar{v}'))) \). Because existential quantification distributes over disjunction, we have that every quantification is “early”, and thus \( \text{image}(S(\bar{v})) = \bigvee v_i \exists \bar{v}(S(\bar{v}) \land \text{or } R_{v_i}(\bar{v}, \bar{v}')) \). Because the quantification is done “early” for every \( v \) in the disjunctive partitioning, all intermediate BDD results depend only on \( \bar{v}' \), while when using conjunctive partitions the intermediate BDD results may depend both on \( \bar{v} \) and \( \bar{v}' \). Thus, using disjunctive partitions usually results in smaller intermediate BDDs than when using conjuncting partitions.

Note that as opposed to a conjunctive partition, the naive disjunctive partition is dependent on the entire vector \( \bar{v}' \), rather than just a single \( v'_i \). We return to this point later and show how to avoid it by modifying the image computation.

Let \( A \subseteq S \) be a set of states and let \( \bar{x} \) be a set of variables. We use the notation \( A|_{\bar{x}} \) to indicate the projection of the set \( A \) onto \( \bar{x} \). That is:

\[ A|_{\bar{x}} = \{ s \in S | \exists a \in A \text{ such that } s \text{ and } a \text{ agree on all values of the variables in } \bar{x} \} \]

### 3 Generating a Model from Software

Previous work showed the application of symbolic model checking to general purpose software [5,6] by translating C source code to EDL (Environment Description Language), a dialect of SMV [9], which is the input language to the RuleBase model checker. EDL, like SMV, is naturally suited for building of conjunctive partitions. That previous work was based on a specially-built parser and was limited to a small subset of C. In this work, we build a similar model using a full-blown compiler front-end. The most important thing about this model is that it has the following structure.

**Definition 1.** An or-structure is a Kripke structure in which for every two states \( s, s' \): if \( R(s, s') \) then \( s \) and \( s' \) are different from each other only in the values of the \( pc \) and no more than a single additional state variable \( x \).

The model we build has a state variable for each global variable in the C code and a state variable named \( pc \) (program counter) that holds the value of the next statement to be performed. The model also has stacks to support local variables, functions and recursion, and some special variables to support arrays and pointers (without pointer arithmetic). The basics of the generation process are explained here using a simple example. Afterward, we will discuss the special treatment for pointers and arrays.

The translation process first translates the C code to intermediate code. There are two reasons for using intermediate code: 1. It will ease the support of other input languages in the future. 2. It generates the \( pc \) in a way such that for each value of \( pc \), a maximum of one memory location changes its value. One
may object to using intermediate code because it increases the number of values $pc$ can get, and therefore increases the number of states in the model. While this is true, the number of $pc$ values is only multiplied by a small factor and therefore adds to the state variables only 2 or 3 bits, which are negligible.

In Figure 1.a we can see a fragment of a C program. The code has two global variables called $x$ and $z$. This code is translated to an assembly-like intermediate code shown in Figure 1.b. In the intermediate code, there is a list of instructions, each with a unique $pc$ (program counter), listed at the beginning of each line. The $pc$ is updated to the $pc$ of the next line if not specified otherwise. The first two lines indicate the behavior for $pc = 18$ and $pc = 19$. This is the intermediate code generated for line 1 in the C code ($z = 0$). At $pc = 18$ the value 0 is inserted to $r1$, and $r1$ is inserted to $z$ in $pc = 19$. Lines like the one for $pc = 22$, which don’t have any code, are used as jump targets and only update the $pc$ to the $pc$ of the next line. Lines for $pc = 24$ through 27 perform the while condition: first in $pc = 24 x$ is inserted into $r2$ and then in $pc = 26$ it is checked if it is bigger than 0. A true answer sets the $pc$ to 29 (enter the loop), while a false answer sets it to the $pc$ of the next line, which in turn sets the $pc$ to 53 - after the loop.

Next we translate the intermediate code into a model. There are two possible translations: The first one is to translate the intermediate code to a language that has the style of a guarded transition system. Each transition is of the form: $pc = PC_1 \rightarrow (X \leftarrow f(X, Y, Z) \land pc \leftarrow PC_2)$. The guard is always a condition about the value of the $pc$ (each value of the $pc$ has exactly one transition) and the transition changes the value of the $pc$ and perhaps the value of one additional
%define main_r1 0
%define main_r2 x
%define main_r3 z
%define main_r4 5
%define main_r5 main_r3 + main_r4
%define main_r6 x
%define main_r7 main_r6 - 1

pc = 19 ⇒ (z ← main_r1 ∧ pc ← 22)
pc = 22 ⇒ (pc ← 26)
pc = 26 ⇒ (pc ← if (main_r2 > 0)
then 29 else 27)
pc = 27 ⇒ (pc ← 53)
pc = 29 ⇒ (pc ← 41)
pc = 41 ⇒ (z ← main_r5 ∧ pc ← 48)
pc = 48 ⇒ (x ← main_r7 ∧ pc ← 50)
pc = 50 ⇒ (pc ← 22)

%define main_r1 0
%define main_r2 x
%define main_r3 z
%define main_r4 5
%define main_r5 main_r3 + main_r4
%define main_r6 x
%define main_r7 main_r6 - 1

next(pc) ← case
pc = 19 : 22
pc = 22 : 26
pc = 26 : if (main_r2 > 0)
then 29 else 27
pc = 27 : 53
pc = 29 : 41
pc = 41 : 48
pc = 48 : 50
pc = 50 : 22
else : pc
esac;
next(x) ← case
pc = 48 : main_r7
else : x
esac;
next(z) ← case
pc = 19 : main_r1
pc = 41 : main_r5
else : z
esac;

(a) Model in ODL representation  (b) Model in EDL representation

Fig. 2. Example of div.c translation to EDL and ODL.

state variable ¹. We refer to this language as ODL. The translation to ODL is presented in Figure 2.a. The other possibility is to translate the intermediate code to EDL (Figure 2.b). For both possibilities we model the registers using a %define. In this way, the registers won’t use any bits in the model. This is possible because the intermediate code defines and uses each register only once.

The translation to ODL is very simple. Each line in the intermediate code is translated to a guarded expression representing the changes for this value of the pc. For example in pc = 19, z gets main_r1 (the %define that represents register r1), and pc is set to 22. In the EDL code, we need to gather all the assigns of a state variable to the same place. For instance, the code for next(z) includes assignments for the lines for pcs 19 and 41 of Figure 1.b. Another difference is that in ODL it is implicit that every state variable that is not mentioned, keeps its value, while the EDL explicitly codes it.

At first glance, it seems preferable to translate to ODL because it’s simpler to translate C code to ODL, and it is simpler to translate ODL to disjunctive

¹ Note that this transition may change a different variable depending on the value of other state variables. However, only one state variable will change its value at any one time. For instance, an assignment of the form a[i] = 5 will change a[0] or a[1], etc., depending on the value of i. But only one array location will change at any one time.
partitions. But translating the C code to EDL allows us to use RuleBase to read EDL, build the conjunctive partitions, and perform pre-model-checking reductions. A reduction is simply a conservative abstraction, that is, one that preserves both positive and negative truth values. Conjunctive partitions are more natural for performing simple reductions such as constant propagation as well as other more sophisticated reductions performed by RuleBase. Thus, even if we did not have conjunctive partitions, we would want to build them and translate the result of the reduction back to disjunctive partitions. Thus, we present methods for translating from conjunctive to disjunctive partitioning and vice versa in order to enable flexibility in our tool. In practice, using the reductions and translating the reduced conjunctive partitioning to disjunctive partitioning indeed proved to be useful. In addition, analyzing the translations enables us to bound the size of the disjunctive partitions, with respect to the conjunctive partitions.

3.1 Dealing with Pointers and Arrays

Modeling pointers and arrays creates a problem, because in general an assignment to a variable X from an array or a pointer causes the variable X to be dependent on more memory locations than an assignment from a scalar. In a naive approach, the BDD size of the partition for X will be quite large, because of the dependence on multiple variables. Furthermore, the large number of variables in a single partition results in many constraints on the BDD order for the entire model, which might result in a larger BDD size not just for the partition in question, but for the entire design.

We solve this problem by using cut-points [3]. Our translation adds four variables for each array. For array ar we add: \( l\_index\_ar \), \( l\_array\_ar \), \( r\_index\_ar \) and \( r\_array\_ar \) (the prefix \( l/r \) means that the array is in the left/right side of the assignment). We translate an assignment \( x = ar[i] \) to the three assignments described in Figure 3(a), and an assignment \( ar[i] = x \) to the three assignments described in Figure 3(b).

When using this translation on code containing assignments \( x = ar[i]; x = ar[j]; y = ar[i]; y = ar[j]; \), we get that \( r\_index\_ar \) is dependent on \( i \) and \( j \), \( r\_array\_ar \) is dependent on \( r\_index\_ar \) and all \( ar \) cells, and \( x \) and \( y \) are dependent only on \( r\_array\_ar \). Without cut-points, we would have had that both \( x \) and \( y \) are dependent on \( i, j \) and all cells of \( ar \).

In pointers, the problem is even more severe because there are generally more memory locations that can be affected by a pointer dereference than cells in an array. Still, the same idea is useful for pointers.

Note that using cut-points and \( \& \text{defines} \) for modeling registers causes a problem when translating statements like \( x = a[i] + a[j] \). We avoid this problem by splitting such statements into two: \( \text{temp} = a[i]; x = \text{temp} + a[j] \).

Our translation has another attribute. An assignment such as \( a[a[i]] = 5 \) is translated in the intermediate code into two different accesses to the array, one to get \( a[i] \) and the second to assign to \( a[a[i]] \), so that our translation creates the code in Figure 3(c).
3.2 Splitting of Self-Assignment Statements

Assignments statements in the code can be of two kinds:

1. **Self-assignment statement** - Assignment to a variable \( x \) in which the assigned value is a function of \( x \) (e.g., \( x+ = y \) or \( x = x + w + z \)). Such an assignment can be further divided into two kinds: constant self-assignment statement where we update the variable with a constant (e.g., \( x* = 4 \), \( x + + \)), and variable self-assignment statement (e.g. \( x+ = y \), \( x = x * b + c \)).

2. **Foreign-assignment statement** - Assignment to a variable \( x \) in which the assigned value is not dependent on the value of \( x \). (e.g. \( x = y \) or \( x = w + z \)).

In order to reduce BDDs size and achieve better performance we split variable self-assignment statements like \( x+ = y \) into two: \( \text{temp} = x \), \( x = \text{temp} + y \). This split increases the number of \( pc \) values and adds one variable (for all splits) but improves the overall performance. The reason will be explained in section 4.1. Constant self-assignment statements can remain as is.

4 Translating between Disjunctive and Conjunctive Partitions

In this section, we show how to build the disjunctive partition of a state variable \( x \), or \( R_x(\bar{v}, \bar{v}') \), from its conjunctive partition and \( R_x(\bar{v}, x') \) and vice versa. Our construction is applicable only to or-structures where each dereference, such as arrays and pointers, is broken by a cut-point. Let \( pc \) be the state variable that codes the program counter of the program and \( \bar{y} \) be the state variables which are different from \( pc \) and \( x \).

**Definition 2.** \( \text{dep.states}_x(\bar{v}) \) is a set of states such that for every \( s \in \text{dep.states}_x(\bar{v}) \) there exists \( s' \) such that \( R(s, s') \) and \( x \) has different values in \( s \) and \( s' \).

Intuitively, \( \text{dep.states}_x(\bar{v}) \) are all the states related to lines in the C program where \( x \) is assigned a value, except for the case where \( x \) is assigned the same value it had before the assignment.
Definition 3. \( \text{dep}_{pc}s_x(pc) \) is the set of pc values which are related to statements in which \( x \) may change \(^2\).

Definition 4. The partial disjunctive partition of a state variable \( x \), denoted by \( \text{por}_R x(pc, x, \bar{y}, x', pc') \), is the disjunctive partition \( \text{or}_R x(\bar{v}, \bar{v}') \) without the requirement that the variables in \( \bar{y} \) are left unchanged.

\[
(\text{or}_R x(\bar{v}, \bar{v}') = \text{por}_R x(pc, x, \bar{y}, x', pc') \land (\bar{y} = \bar{y}'))
\]

4.1 Building Disjunctive Partitions from Conjunctive Partitions

We now show how to build each disjunctive partition from the conjunctive partition of the same state variable and the conjunctive partition of \( pc \).

Translation for \( x \neq pc \): First we show how to build \( \text{or}_R x(\bar{v}, \bar{v}') \) for \( x \neq pc \).

1. Calculate \( \text{dep}_{states}_x(\bar{v}) \):

\[
\text{dep}_{states}_x(\bar{v}) = \exists x' (\text{and}_R x(\bar{v}, x') \land (x \neq x')).
\]

2. Intersect the quantification of \( x \) from \( \text{dep}_{states}_x(\bar{v}) \) with the conjunctive partitions of \( x \) and \( pc \):

\[
\text{por}_R x(pc, x, \bar{y}, x', pc') = (\exists x (\text{dep}_{states}_x(\bar{v}))) \land \text{and}_R x(\bar{v}, x') \land \text{and}_R pc(\bar{v}, pc')
\]

3. Intersect \( \text{por}_R x(\bar{v}, x', pc') \) with \( \bar{y} = \bar{y}' \) to indicate that the other variables do not change:

\[
\text{or}_R x(\bar{v}, \bar{v}') = \text{por}_R x(pc, x, \bar{y}, x', pc') \land (\bar{y} = \bar{y}')
\]

We use \( \text{dep}_{states}_x(\bar{v}) \) in our construction and not \( \text{dep}_{pcs}_x(pc) \) because two states in which the pc value is identical do not necessarily change the same state variable. For example, consider the C statement \( a[i] = 5 \) and assume that it is related to \( pc = 7 \). For each value of \( i \) this statement changes a different state variable. Thus, the value \( pc = 7 \), which is related to this statement, will be in more than one disjunctive partition. If we had used \( \text{dep}_{pcs}_x(pc) \) the state \( \{pc = 7; i = 2\} \) would have been both in the partition of \( a[2] \) and \( a[1] \). As a result, after conjuncting the disjunctive partition of \( a[1] \) with \( \bar{y} = \bar{y}' \) it would have contained another transition, that does not exist in the original model and changes only \( pc \) and not \( a[1] \) or \( a[2] \). This transition would have been entered to the disjunctive partition of \( a[1] \) because \( a[2] \) is in \( \bar{y} \). The quantification that appears in \( \text{por}_R x(pc, x, \bar{y}, x', pc') \) is discussed in detail later.

\(^2\) \( x \) may not always change its value in a certain \( pc \). For example, when \( x \) is a cell in an array, \( a[0] \), and the assignment is \( a[i] = 5 \), \( a[0] \) is assigned a value only if \( i = 0 \) and stays unchanged otherwise.
Translation for pc: Calculating $\text{por}_{\text{pc}}(pc, x, \bar{y}, pc')$ is a bit different.

1. Calculate $\text{dep}_{\text{pcs}_x}(pc)$ for each $x \neq pc$:
   \[
   \text{dep}_{\text{pcs}_x}(pc) = \text{dep}_{\text{states}_x}(\bar{v})|_{pc}
   \]

2. Calculate the set of $pc$ values $\text{jump}_{\text{pcs}}(pc)$ that are related to statements in which $pc$ is the only state variable that is changed. These $pc$ values are related to statements in which there is a control branch like an if statement.
   \[
   \text{jump}_{\text{pcs}}(pc) = \bigwedge_{x \neq pc} (\text{dep}_{\text{pcs}_x}(pc))
   \]

3. Intersect $\text{and}_{\text{pc}}(\bar{v}, pc')$ with $\text{jump}_{\text{pcs}}(pc)$ to get the value of $pc'$ for this $pc$ value.
   \[
   \text{por}_{\text{pc}}(pc, x, \bar{y}, pc') = \text{jump}_{\text{pcs}}(pc) \land \text{and}_{\text{pc}}(\bar{v}, pc')
   \]

4. Intersect $\text{or}_{\text{pc}}(pc, x, \bar{y}, pc')$ with $\bar{y} = \bar{y}'$, where $\bar{y}$ is all variables that are different from $pc$.
   \[
   \text{or}_{\text{pc}}(\bar{v}, \bar{v}') = \text{or}_{\text{pc}}(pc, x, \bar{y}, pc') \land (\bar{y} = \bar{y}')
   \]

Discussion: The general idea is that transitions in which only the $pc$ changes should be in the partition of the $pc$, and transitions in which both the $pc$ and some variable $x$ change should be in the partition of $x$. Naively, this means that a line with some assignment would appear in the partition of the variable being assigned, while a line without an assignment would appear in the partition of the $pc$. However, things are not so simple. Consider the assignment $x = 5$. If $x$ has the value 5 before the assignment, then a transition from this line changes only the $pc$. If $x$ has another value before the assignment, then this line changes both $x$ and the $pc$. A naive construction of the or-partitions from the and-partitions would put the transition from a state where $x$ has the value 5 into the partition of the $pc$, rather than into the partition of $x$. We would like to put this transition into the partition of $x$, because in this way the BDDs will be in some sense “cleaner” - that is, we hope that the BDD size will be smaller. Two other related problems are the case of assignments of the form $x+ = y$, where $y$ has the value 0, and the case of assignments to $a[i]$ for some array $a$, where $i$ is out of the array bounds. Our method deals with such cases as explained below.

In order to deal with assignments of the form $x+ = y$ (for which $y = 0$ may cause a problem in a naive construction) is the source of the splitting of variable self-assignment statements into two, as described in 3.2 above. This way, we avoid dealing with such assignments in the construction itself.
The problem with assignments such as \(a[i] = 5\) needs some explanation. Consider an array \(a[0..2]\) of size three and a statement \(a[i] = 5\), where \(i\) equals 7. Because \(a[7]\) is not a real variable in the program, there is no corresponding state variable in our model (otherwise the model would have been unbounded). Thus, in such a case, in our model only the \(pc\) is changed, and the conjunctive partitioned transition relation contains a transition which changes only the \(pc\). But this statement is related to transitions that do change variable values (for \(i < 3\), and thus does not “belong” in the partition for the \(pc\) (according to our notion of “cleanliness”). It is possible to overcome this problem by adding a new overflow variable to the model, the disjunctive partition of which will capture this behavior.

Finally, we note that in the general case, our translation does not work for statements such as \(a[i] = a[5]\) or \(a[a[i]] = a[i] + 1\). However, when the model is generated, as we suggested in section 3.1, such statements are always split up into several statements and therefore the problem is avoided.

4.2 Building Conjunctive Partitions from Partial Disjunctive Partitions

We previously discussed how to build a disjunctive partition from a conjunctive partition. In this subsection, we present the translation in the opposite direction.

1. We first calculate \(\operatorname{dep}_{x}(pc)\) simply by looking at the \(pcs\) that appear in \(\operatorname{por}_x(v, x', pc)\)

\[
\operatorname{dep}_{x}(pc) = \operatorname{por}_x(v, x', pc)|_{pc}
\]

2. Now we can calculate \(\operatorname{and}_x(v, x')\). It is formed from a union of two sets: the states in which \(x\) changes its value and the states in which \(x\) saves its value.

\[
\operatorname{and}_x(v, x') = (\exists pc' (\operatorname{por}_x(v, x', pc')) \lor (\operatorname{dep}_{x}(pc) \land x = x'))
\]

3. Now we can calculate \(\operatorname{and}_{pc}(v, pc')\). It is calculated by gathering the transition \(pc\) to \(pc'\) in all the partial disjunctive partitions of the variables and conjuncting it with \(\operatorname{por}_{pc}(v, pc')\).

\[
\operatorname{and}_{pc}(v, pc') = \operatorname{por}_{pc}(v, pc') \lor (\bigvee_{x \neq pc} \operatorname{por}_{pc}(v, pc', x')|_{pc, pc'})
\]

5 Using Partial Disjunctive Partitions

In the previous section, we showed how to calculate disjunctive partitions. Using this, we can take advantage of the superior efficiency of disjunctive partitioning. However, if the sizes of the disjunctive partitions are larger than the corresponding conjunctive partitions it is not certain that we have gained anything. In this
section we examine the answer to this question. First let’s look at $or_{R_x} (\bar{v}, \bar{v}')$.

By definition, $or_{R_x} (\bar{v}, \bar{v}') = por_{R_x} (pc, x, \bar{y}, pc', x') \land (\bar{y} = \bar{y}')$. It is possible to build an example in which $|or_{R_x} (\bar{v}, \bar{v}')| = O(n \cdot |por_{R_x} (pc, x, \bar{y}, pc', x')|)$, where $n$ is the number of state variables. An example is the assignment $x \leftarrow y$, where $x$ is the first variable in the BDD order after $pc$ and $y$ is the last state variable in the BDD order.

In order to avoid this factor, we do not calculate $or_{R_x} (\bar{v}, \bar{v}')$. We calculate only $por_{R_x} (pc, x, \bar{y}, x', pc')$ and rewrite the procedures that calculate image and pre-image operations in such a way as to use $por_{R_x} (pc, x, \bar{y}, x', pc')$ instead of $or_{R_x} (\bar{v}, \bar{v}')$. In the next subsection, we present the new algorithm for image and pre-image computation and prove its correctness. After, that we will bound the size of $por_{R_x} (pc, x, \bar{y}, x', pc')$.

### 5.1 Image and Pre-Image Computations Using Partial Disjunctive Partitions

When computing image (pre-image) using disjunctive partitions, it is possible to calculate the image (pre-image) on each disjunctive partition independently and then union the results. In this subsection, we introduce how to compute image or pre-image when only $por_{R_x} (pc, x, \bar{y}, x', pc')$ is given for each variable $x$.

**Lemma 1.** $pre\_image(S(pc', x', \bar{y}'), or_{R_x}(pc, x, \bar{y}, pc', x', \bar{y}')) =$

$$pre\_image(S(pc', x', \bar{y}), por_{R_x}(pc, x, \bar{y}, pc', x'))$$

From this lemma, we get a simple algorithm that in the first step unprimes $\bar{y}'$ in $S(pc', x', \bar{y}')$ (linear in the size of the BDD), and then performs the ordinary pre-image algorithm on the result. The proof of this lemma is given in the full version of this paper.

**Lemma 2.** $image(S(pc, x, \bar{y}), or_{R_x}(pc, x, \bar{y}, pc', x', \bar{y}')) =$

$$image(S(pc, x, \bar{y}), por_{R_x}(pc, x, \bar{y}', pc', x'))$$

Here again, we have a simple algorithm. First prime $\bar{y}$ in $S(pc, x, \bar{y})$ and in $por_{R_x}(pc, x, \bar{y}, pc', x')$ and then calculate the image using the results. The proof is almost the same as of the previous lemma.

### 5.2 Bounding the Size of the Partial Disjunctive Partitions

In this subsection, we bound the size of partial disjunctive partitions. The proofs of these claims are long, technical, and tedious. Proof sketches are given in the full version of this paper. Despite the relatively large upper bound, in practice, these extreme examples are rare. See Section 7 for experimental results.

Since every variable is dependent on $pc$, it seems wise to place $pc$ as the first state variable in the BDD ordering. All of the following lemmas assume that the BDD ordering follows this idea.
We define $\text{por}_x(R_x(\bar{v}, x'))$ to be $\text{por}_x(R_x(\bar{v}, x', pc'))$ without the condition on the value of $pc'$:

$$\text{por}_x(R_x(\bar{v}, x')) = (\exists x(\text{dep\_states}(x, \bar{v}))) \land \text{and}_x(R_x(\bar{v}, x')).$$

We can now rewrite the definition of $\text{por}_x(R_x(\bar{v}, x', pc'))$ using $\text{por}_x(R_x(\bar{v}, x'))$:

$$\text{por}_x(R_x(\bar{v}, x', pc')) = \text{por}_x(R_x(\bar{v}, x')) \land \text{and}_{pc}(\bar{v}, pc').$$

The following lemmas will first bound the size of $\text{por}_x(R_x(\bar{v}, x'))$ and only then the size of $\text{por}_x(R_x(\bar{v}, x', pc'))$.

### 6 Scalability for Distributed Model Checking

We now turn to the scalability of disjunctive partitioning. We claim that symbolic model checking with disjunctive partitioning is not only more efficient than with conjunctive partitioning, it also scales better. This is a direct result of the fact that quantification distributes over disjunctive partitioning, but not over conjunctive partitioning. Since $\text{image}(S(\bar{v})) = \bigvee_x \exists \bar{v}(S(\bar{v}) \land \text{or}_x(\bar{v}, \bar{v'}))$, when using disjunctive partitions or partial disjunctive partitions we can calculate the image using one partition on each processor including quantification and then union the results of all processors. Because the image computation may be exponential in the number of BDD nodes and the union operation is linear in the number of BDD nodes, distributing the partitions between $n$ processors divides the “heavy” work by $n$. Note that when image computation is done distributively using conjunctive partitions it requires another step in which the partial results are “anded” together before quantification. Thus, the work done after all the processors have calculated their results may still be exponential in the number of BDD nodes. We now suggest two distributed algorithms for disjunctive partitions. The first algorithm is simple and uses a master and several slaves. The master will send $S(\bar{v})$ to all the slaves and start sending each idle slave a disjunctive partition. Each slave that gets a disjunctive partition will perform the image computation with this partition and union it with previous computations it made. When there are no more partitions and all slaves are idle, the master will gather all the slaves’ results and union them. Reachability computation is then performed by repeated image computations of the former algorithm. One drawback with this scheme is that while the server computes the union of all the slaves’ results, the slaves are idle.

The second algorithm avoids this problem. In this algorithm, each process $P_i$ is responsible for several partitions $TR_i$, and has its own reachability set $RS_i$. There is also a (shared) queue of sets of states and each process has two pointers to this queue: a shared pointer for entering sets to the queue and a private one for reading from the queue. As a result, all processors read all the sets that enter the queue. At the beginning the queue has the initial set of states. Each process $P_i$, at each iteration takes the next set $S$ from the queue (according to its pointer), removes from it the parts it already handled $S = S \setminus RS_i$ and adds the result to
<table>
<thead>
<tr>
<th>Example</th>
<th>Num of vars</th>
<th>Conjunctive partitions</th>
<th>Disjunctive partitions</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Reachability time</td>
<td>Maximal step time</td>
</tr>
<tr>
<td>simple</td>
<td>505</td>
<td>11024 s</td>
<td>95.7 s</td>
</tr>
<tr>
<td>factorial</td>
<td>159</td>
<td>31.8 s</td>
<td>0.9 s</td>
</tr>
<tr>
<td>insert sort</td>
<td>197</td>
<td>264.6 s</td>
<td>3.9 s</td>
</tr>
<tr>
<td>quick sort</td>
<td>282</td>
<td>10197 s</td>
<td>10 s</td>
</tr>
<tr>
<td>merge sort</td>
<td>654</td>
<td>952 s</td>
<td>7.77 s</td>
</tr>
<tr>
<td>pointer quick sort</td>
<td>693</td>
<td>1546 s</td>
<td>5.8 s</td>
</tr>
<tr>
<td>pointer merge sort</td>
<td>716</td>
<td>&gt; 8 h</td>
<td>&gt; 99 s</td>
</tr>
</tbody>
</table>

Fig. 4. Comparison of reachability computation using conjunctive partitions against using partial disjunctive partitions.

If $RS_i$, then calculates the image of $S$ using $TR_i$ getting $image_i = image(S, TR_i)$. In order to continue only with the new states, the reachable states are removed from $image_i$ getting $new_i = image_i \setminus RS_i$. In the case where $new_i \neq \emptyset$, it is put in the next entry of the queue. When all processors are trying to read from the queue and they are all pointing to an empty slot in the queue, the algorithm has ended. At the end, each process has the whole reachability set because it saw all the image computation results of all processes in the queue and no new set of states is entered to the queue.

7 Experimental Results

We implemented the translation from conjunctive partitioned transition relation to partial disjunctive partitioned transition relation in the IBM model checker RuleBase [1]. We compared reachability analysis using conjunctive partitions with reachability analysis using partial disjunctive partitions on models that were translated from software programs. These software programs were written in C and contain pointers and arrays. In both cases, we applied dynamic BDD reordering. In order to obtain a fair comparison between these algorithms, we ran each one twice. In the first run, the algorithm reordered the BDD with no time limit in order to find a good BDD order. The initial order of the second run was the BDD order found by the first run. The partial disjunctive partitioning outperforms the conjunctive partitioning with respect to execution time, as shown in Figure 4. We compared the sizes of partial disjunctive partitions with those of conjunctive partitions under the same BDD order. The table in Figure 5 shows the maximal and minimal ratios between a specific variable partial disjunctive partition size and its conjunctive partition size. We specifically note the ratio of the $pc$ variable and the size of its partial disjunctive partition. In addition, we show the maximal conjunctive partition and maximal partial disjunctive partition not including $pc$. We observed that the partial disjunctive partitions were in the same order of magnitude or even smaller than the conjunctive partitions. This was achieved by the use of partial disjunctive partitions instead of ordinary disjunctive partitions. In our experiments we found that the
Fig. 5. Comparison between size of conjunctive partitions and partial disjunctive partitions.

size of each ordinary disjunctive partition \((\text{or}_x R_x (\bar{v}, \bar{v}'))\) was up to 84 times the size of its corresponding partial disjunctive partition.

## 8 Conclusions and Future Work

Using partial disjunctive partitions seems to be a successful and natural scheme for software models. In this work, we show how to apply disjunctive partitioning to software models while keeping the partitions small. We also show how to enhance the image and pre-image computation to support our partial disjunctive partitions and make model checking algorithms more efficient. However, this is only the beginning and there are a number of directions for future work. As we note above, we handle variables with a large number of bits by creating a single partition for each variable containing the behaviors of all its bits. Future work will explore the possibility of implementing the DNF partitioned transition relation [4], where the disjunctive partition of a state variable is composed of conjunctive partitions of its bits.

As we claimed in Section 6, disjunctive partitioned transition relation is natural for distributed algorithms. It seems wise to implement and explore both algorithms presented in that section. Special attention should be given to finding a good distribution of the disjunctive partitions over the processes in order to achieve good load balancing.

### Acknowledgments

We thank Cindy Eisner, Yoad Lustig and Ziv Nevo for many helpful discussions.

### References

Instantiating Uninterpreted Functional Units and Memory System: Functional Verification of the VAMP

Sven Beyer¹, Chris Jacobi²*, Daniel Kröning³**, Dirk Leinenbach¹* ***, and Wolfgang J. Paul¹

¹ Saarland University, Computer Science Department, 66123 Saarbrücken, Germany
{s beyer,dirk l,wjp}@cs.uni-sb.de

² IBM Deutschland Entwicklung GmbH, Processor Dev. II, 71032 Böblingen, Germany
cjacobi@de.ibm.com

³ Carnegie Mellon University, Computer Science, Pittsburgh, PA
kroening@cs.cmu.edu

Abstract. In the VAMP (verified architecture microprocessor) project we have designed, functionally verified, and synthesized a processor with full DLX instruction set, delayed branch, Tomasulo scheduler, maskable nested precise interrupts, pipelined fully IEEE compatible dual precision floating point unit with variable latency, and separate instruction and data caches. The verification has been carried out in the theorem proving system PVS. The processor has been implemented on a Xilinx FPGA.

1 Introduction

Previous Work. Work on the formal verification of processors so far has concentrated mainly on the following aspects of architectures:

i) Processors with in-order scheduling, one or several pipelines including forwarding, stalling and interrupt mechanisms [3,13,28]. The verification of the very simple, non-pipelined FM9001 processor has been reported in [2]. Using the flushing method from [3] and uninterpreted functions for modeling execution units, superscalar processors with multicycle execution units, exceptions and branch prediction [28] have been verified by automatic BDD based methods. Also, one can transform specification machines into simple pipelines (with forwarding and stalling mechanism) by an automatic transformation, and automatically generate formal correctness proofs for this transformation [15].

ii) Tomasulo schedulers with reorder buffers for the support of precise interrupts [5,8,16,24]. Exploiting symmetries, McMillan [16] has shown the correctness of a powerful Tomasulo scheduler with a remarkable degree of automation. Using theorem proving, Sawada and Hunt [24] show the correctness of an entire out-of-order processor, precise interrupts, and a store buffer for the memory unit. They also consider self-modifying code (by means of a sync instruction).

* The work reported here was done while the author was with Saarland University.
** Research supported by the DFG graduate program ‘Effizienz und Komplexität von Algorithmen und Rechenanlagen’
*** Research supported by the DFG graduate program ‘Leistungsgarantien für Rechnersysteme’
iii) Floating point units (FPU). The correctness of an important collection of floating point algorithms is shown in [21,22] using the theorem prover ACL2. Correctness proofs using a combination of theorem proving and model checking techniques for the FPUs of Pentium processors are claimed in [4,19]. As the verified unit is part of an industrial product not all details have been published. Based on the constructions and on the paper and pencil proofs in [18] a fully IEEE compatible FPU has been verified [1,11] (using mostly but not exclusively theorem proving).

iv) Caches. Multiple cache coherence protocols have been formally verified, e.g., [6,17,25,26]. Paper and pencil proofs are extremely error prone, and hence the generation of proofs for interactive theorem proving systems is slow. The method of choice is model checking. The compositional techniques employed by McMillan [17] even allow for the verification of parameterized designs, i.e., cache coherence is shown for an arbitrary number of processors.

Simplifications, Abstractions, and Restrictions. Except for the work on floating point units, the cache coherence protocol in [6], and the FM9001 processor [2], none of the papers quoted above states that the verified design actually has been implemented. All results cited above except [1,2,6,11] use several simplifications and abstractions:

i) The realized instruction set is restricted: always included are the six instructions considered in [3]: load word, store word, jump, branch equal zero, three register ALU operations, ALU immediate operations. Five typical extra instructions are trap, return from exception, move to and from special registers, and sync [24]. The branch equal zero instruction is generalized in [28] by an uninterpreted test evaluation function. Most notably the verification of machines with load/store operations on half words and bytes has apparently not been reported. In [27] the authors report an attempt to handle these instructions by automatic methods which was unsuccessful due to memory overflow.

ii) Delayed branch is replaced by non-deterministic speculation (speculating branch taken/not taken).

iii) Sometimes, non-implementable constructs are used in the verification of the processors: e.g., Hosabettu et.al. [8] use tags from an infinite set. Obviously, this is not directly implementable in real hardware.

iv) The verification of the FPUs does neither cover the handling of denormal numbers nor of exception flags. The verification of a dual precision FPU has not been reported (though, obviously, Intel’s and AMD’s FPUs are capable of dual precision).

v) No verification of a memory unit with caches has been reported. Eiriksson [6] only reports the verification of a bit-level implementation of a cache coherence protocol without data consistency.

vi) The verification of pipelines or Tomasulo schedulers with instantiated floating point units and memory units with caches and main memory bus protocol has not been reported. Indeed, in [27] the authors state: “An area of future work will be to prove that the correctness of an abstract term-level model implies the correctness of the original bit-level design.”

Results and Overview. In the VAMP (verified architecture microprocessor) project we have designed, functionally verified, and synthesized a processor with full DLX in-
struction set, delayed branch, Tomasulo scheduler, maskable nested precise interrupts, pipelined fully IEEE 754 [9] compatible dual precision floating point unit with variable latency, as well as separate, coherent instruction and data caches. We use only finite tags in the hardware. Thus all abstractions, restrictions and simplifications mentioned above have been removed. Specification and verification was performed using the interactive theorem proving system PVS [20]. All formal specifications and proofs are on our web site. The hardware description was automatically extracted from PVS and translated into Verilog HDL by a tool sketched in section 7. Hardware with non verified rudimentary software is up and running on a Xilinx FPGA. The Verilog design can also be downloaded from our web site.

In section 2, we summarize the fixed point instruction set, its floating point extension, and the interrupt support realized. We give a micro-architectural overview with a focus on the memory system. Section 3 describes the correctness criterion, the main proof strategy, and the integration of the execution units into the Tomasulo core. Correctness criterion and proof strategy are based on scheduling functions [14,18] (similar to the stg-component of MAETTs [23]). The model of the execution unit is in a nontrivial way more general than previous models without complicating interactive proofs too much.

Section 4 presents a delayed branch mechanism, which is automatically constructed and proven correct by the methods for automatic pipeline construction from [15] and summarizes the specification of an interrupt mechanism for maskable nested precise interrupts and delayed PC from [18]. Section 5 deals with the integration of the floating point unit from [11] into our Tomasulo scheduler. Section 6 deals with loads and stores of double words, words, half words, and bytes at a 64 bit cache/memory interface. We also sketch correctness proofs of the implementation of a simple coherence protocol between data cache and instruction cache, as well as the implementation of a main memory bus protocol. Section 7 describes the implementation of the VAMP on a Xilinx FPGA. Section 8 gives an overview of the verification effort for various parts of the project, summarizes our work, and sketches directions of some future work.

2 Overview of the VAMP Processor

Instruction Set. The full DLX instruction set from [7] is realized. This includes loads and stores for double words, words, half words, and bytes, various shift operations, and two jump-and-link operations. Loads of bytes and half words can be unsigned or signed. In order to support the pipelining of instruction fetches, delayed branch with one delay slot is used. Note that delayed branch changes the sequential semantics of program execution.

The floating point extension of the DLX instruction set from [18] is supported. The user sees a floating point register file with 32 registers of single precision numbers as well as a single floating point condition code register FCC. Pairs of floating point registers can be accessed as registers for double precision numbers (with an even register address). Supported operations are: i) loads and stores for singles and doubles. ii) +, −, ×, ÷ both for single and double precision numbers. iii) test-and-set, the result is stored in FCC.

1 http://www-wjp.cs.uni-sb.de/forschung/projekte/VAMP/
iv) conditional branches as a function of FCC. v) conversions between singles, doubles and integers. vi) moves between the general purpose register file and the floating point register file. Operations are fully IEEE compatible [9]. In particular, all four rounding modes, denormal numbers, and exponent wrapping as a function of the interrupt masks are realized.

Interrupt Support. Presently, the interrupts from table 1 in section 4 are supported. Interrupts are maskable and precise. Floating point interrupts are accumulated in 5 bits of a special purpose register IEEE\textsubscript{f} (IEEE flag) as prescribed by the IEEE standard. All special purpose registers (details in section 4) are collected into a special purpose register file. Operations supporting the interrupt mechanism are: i) moves between general purpose registers and special purpose registers. ii) trap. iii) return-from-exception.

Microarchitecture Overview. Figure 1 gives a high level overview of the VAMP microarchitecture. Stages IF and ID are a pipelined implementation of delayed branch as explained in section 4. Stages EX, C and WB realize a Tomasulo scheduler with 5 execution units, a fair scheduling policy on the common data bus CDB, and a reorder buffer ROB (for precise interrupts). The execution units are i) MEM: memory unit with variable latency and internal pipelining. There is presently no store buffer. ii) XPU: the fixed point unit. iii) FPU1 to FPU3: specialized pipelined floating point units with variable latency. FPU1 performs additions and subtractions. FPU2 performs multiplications and divisions. FPU3 performs test-and-set as well as conversions. The data output of the reorder buffer is 64 bits wide. The floating point register file FPR is physically realized as 16 registers, each 64 bits wide. The general purpose registers file GPR and the special purpose register file SPR are both 32 bits wide, and have 32 and 9 entries, respectively. They are connected to the low-order bits of the ROB output.
Figure 2 depicts a simplified view of the memory unit. Internally, it has two pipeline stages. The first stage does address and control signal computations. The second stage performs the actual data cache access via signals $adr$, $din$, and $dout$. Instructions are fetched from the instruction cache via signals $pc$ and $inst$. The memory interface $Mif$ internally consists of a data cache, an instruction cache, and a main memory. The caches are kept coherent (this does not suffice to guarantee correct execution of self-modifying code). Details are explained in section 6.

3 Correctness Criterion and Tomasulo Algorithm

Notations. We consider a specification machine $S$ and an implementation machine $I$. Configurations of these machines are tuples, whose components $R_S$ and $R_I$, respectively, are registers or memories. Register contents are bit strings. Memory contents are modeled as mappings from addresses (bit strings) to bit strings. For example, $PC_S$ denotes the program counter of the specification machine, and $mem_I$ denotes the main memory of the implementation machine.

The specification machine processes a sequence of instructions $I_0, I_1, \ldots$ at the rate of one instruction per step. We denote by $R^i_S$ the content of component $R$ before execution of instruction $I_i$. One step of the implementation machine is a hardware cycle, and we denote by $R^T_I$ the content of component $R$ during cycle $T$. The fetch of the 4 bytes of an instruction into the instruction register $IR$ of the implementation machine during cycle $T$ can be specified by $IR^{T+1}_I := mem^T_I[PC^T_I + 3 : PC^T_I]$.

Although the instruction register is not a visible register, one can specify the desired content $IR^i_S$ of the instruction register for the specification machine for instruction $I_i$ as a function of the visible components by $IR^i_S = mem^i_S[PC^i_S + 3 : PC^i_S]$. Defining the
next configuration $c_{S}^{i+1}$ of the specification machine involves many such intermediate definitions, e.g., the immediate constant $imm_{S}^{i}$, the effective address $ea_{S}^{i}$, etc. Starting from the visible components $R_{S}$ we extend the configuration of the specification machine in this way by numerous (redundant) secondary components.

**Scheduling Functions.** For hardware cycles $T$ and pipeline stages $k$ of the implementation machine, we formally define an integer valued scheduling function $sI(k, T)$ [14], where $sI(k, T) = i$ has the intended meaning that an instruction $I_i$ is during cycle $T$ in stage $k$.

By treating instruction numbers like integer valued tags, the definition of these functions is straightforward. We initialize $sI(k, 0) := 0$ for all stages. We then “clock” these tags through the pipeline stages under the control of the update enable signals $ue_k$ for the output registers of stage $k$. If a stage is not clocked, the scheduling function is not changed, i.e., $sI(k, T) := sI(k, T - 1)$ if $ue_{k,T-1}$. Note that we introduce separate “stages” $k$ for each reservation station and ROB entry.

For the fetch stage, e.g., we define $sI(fetch, T) := sI(fetch, T - 1) + 1$ if $ue_{fetch,T-1}$, meaning that the content of the fetch stage progresses by one instruction in the instruction stream $I_0, I_1, \ldots$ If stage $k$ receives data from stage $k'$ in cycle $T$, we define $sI(k, T) := sI(k', T - 1)$. Note that this covers the case that a stage can receive data from two different stages and $k''$, since in a fixed cycle $T$, it receives data from only one of these stages. This occurs at the ROB, e.g., where we allow bypassing branch instructions from the instruction register directly into the ROB without going through an execution unit. Thus, the ROB can receive data from the CDB and from the instruction register.

As a form of bookkeeping for the memory unit, we introduce an additional “stage” $mem'$. The corresponding scheduling function $sI(mem', T)$ equals $sI(mem, T)$ if the memory unit is empty or the instruction in the unit has not accessed the main memory yet. Otherwise, we set $sI(mem', T) := sI(mem, T) + 1$. We need this bookkeeping function in order to model whether the memory is already updated by a store instruction.

**Correctness Criterion.** We are interested in the content of the main memory $mem$ and the register files $RF \in \{GPR, FPR, SPR\}$ after certain instructions $I_i$ respectively before instruction $I_{i+1}$. The main memory is an output “register” of stage $mem$ and the register files are output “registers” of stage $wb$. The functional correctness criterion requires an instruction $I_i$ in stage $mem'$ of the implementation machine $I$ to see the same memory content as the corresponding instruction of the specification machine $S$; formally $mem^T_I = mem^{sI(mem', T)}_S$. The corresponding condition for register files $RF$ is $RF^T_I = RF^{sI(wb, T)}_S$. In general, we prove by induction on $T$ for all stages $k$ and all output registers $R$ of stage $k$ that $R^T_I = R^{sI(k, T)}_S$, where $R^i_S$ can be a visible or

---

2 Having integer valued tags is only a proof trick. In hardware, we only use finite tags. During the proof of correctness for the Tomasulo scheduler, we prove that these finite tags properly match to the infinite instruction number.

3 Update enable signals are sometimes called ‘register activates’. They are used to (de-)activate updating of register contents.

4 We introduce symbolic names for some stages $k$, e.g., $fetch$ and $mem$. 
redundant component of the configuration of the specification machine. Note that for technical reasons, we claim for the instruction register that \( IR^T_I = IR^{S(I(fetch,T)} - 1 \).

The liveness criterion states that all instructions that are not interrupted reach the writeback stage. At the time of submission of this paper, we have separate formal liveness proofs for the scheduler and the execution units; we are currently working on combining them into a single formal liveness proof for the entire machine.

Paper and pencil proofs for the correctness of Tomasulo schedulers tend to follow a canonical pattern: i) For instructions \( I_i \) and register operand \( R \), one defines \( last(i,R) \) as the index of the last instruction before \( I_i \) which wrote register \( R \). ii) One shows by induction that the formal definitions of tags and valid bits have the intended meaning. In our setting, this means that the finite tags in hardware correspond to the integer valued tags provided by the scheduling function \( sI \). iii) Finally, one has to show that the reservation station of instruction \( I_i \) reconstructs \( R^{last(i,R)}_S \). The rest is easy.

It is important to observe that the structure of these paper and pencil proofs and their formal (theorem proving) counter parts do not depend much on the fixed or variable latency of execution units or whether these units are pipelined. The scheduler recognizes instructions completed by the execution units simply by examining the tags returned from the units. The situation is very different for model checking [28].

**Integration of Execution Units.** The proofs for the scheduler and the proofs for the execution units are separated by the following specifications for the execution units [11, 10]. Notations refer to figure 3.

i) \( \text{stall}^T_{in} \Rightarrow \neg \text{valid}^T_{out} \), i.e., if the scheduler asserts \( \text{stall}^T_{in} \), the execution unit does not return a valid instruction.

ii) \( \forall T \exists T' > T : \neg \text{stall}^{T'}_{out} \), i.e., the \( \text{stall}^{T}_{out} \) signal is never active indefinitely.

iii) Instructions dispatched with \( \text{tag}^{T}_{in} = \text{tg} \) at time \( T \) will eventually (at time \( T' \geq T \)) return a result with the same tag , i.e., \( \text{tag}^{T'}_{out} = \text{tg} \). Moreover, \( \text{data}^{T'}_{out} = f(\text{data}^{T}_{in}) \) where \( f \) is the (combinatorial) function the execution unit is supposed to compute.

iv) For each time \( T \) at which a result with tag \( \text{tg} \) is returned, there is an earlier time \( T' \leq T \) such that an instruction with tag \( \text{tg} \) was dispatched at time \( T' \), and \( \text{tg} \) was not returned between \( T' \) and \( T \). Hence, the execution units do not create spurious outputs.

Note that the instructions do not need to leave the execution units in the order they enter the units; all FPUs, e.g., exploit this by allowing instructions on some special operands to overtake other instructions. Moreover, multiplications may overtake divisions (cf. [10] for details).

The four conditions above must be shown for each of the execution units provided the scheduler guarantees the following three conditions: i) No instruction is dispatched
to an execution unit which sends a stall\textsubscript{out} signal to its reservation station. ii) The execution units are not stalled forever by the producers. iii) Tag-uniqueness: no tag which is dispatched into an execution unit is already in use.

4 Delayed Branch and Maskable Nested Precise Interrupts

In the delayed branch mechanism, taken branches yield a new PC of the form $PC + imm + 4$, taken branches are delayed, and $PC + 8$ is saved to the register file during jump-and-link. In the equivalent delayed PC mechanism [14,18], one uses an intermediate program counter $PC'$ with branch targets $PC' + imm$, all fetches use a delayed program counter $DPC$, and $PC' + 4$ is saved during jump-and-link.

Figure 4 depicts a pipelined implementation of the delayed PC mechanism in the VAMP processor. This construction and its formal correctness proof are automatically obtained by the method for automatic pipeline construction from [15]. Indeed, fetching instructions from the intermediate program counter $PC'$ is—not only intuitively but formally—forwarding of $DPC$. The role of the multiplexers above $PC'$ and $DPC$ are explained in the following paragraphs about interrupts.

The formal specification of the interrupt mechanism for delayed PC is based on the definitions of [18, Chap. 5, 9.1]. Table 1 shows the supported interrupts. The special purpose registers for the interrupt mechanism are: i) status register $SR$ for interrupt masks, ii) two registers $ECA$ for exception cause and $Edata$ for parameters passed to the interrupt service routine, iii) two registers $EPC$ and $EDPC$ for return addresses for $PC'$ and $DPC$ and iv) a register $IEEEf$ for the accumulation of masked floating point exceptions.

At issue time of an instruction $I_i$, it is unknow whether $I_i$ will be interrupted and whether the interrupt requires to repeat the interrupted instruction or not. Therefore, we have to save two pairs of potential return addresses in the reorder buffer: $(PC_{S,i}^r, DPC_{S,i}^r)$ for interrupts of type “repeat”, and the results of the uninterrupted next $PC'$ and next $DPC$ computations $(PC_{S,i+1}^r, DPC_{S,i+1}^r)$ for interrupts of type “continue”. The data paths of the PC environment are shown in figure 4.

Interrupt handling in the specification machine $S$ depends on the components $ECA$ and $Edata$. In the implementation, these two registers are treated as additional results of the execution units; thus, execution units have up to four 32-bit results. This affects the width of the ROB. The formal correctness of these components in the ROB at writeback time is asserted without additional verification effort by the consistency of the Tomasulo scheduler. Further lemmas are needed for the correctness of the PCs stored in the ROB. The return-from-exception instruction is treated like any other instruction; no special effort is needed here.

\footnote{Page fault signals are presently tied to zero.}
Table 1. Implemented interrupts

<table>
<thead>
<tr>
<th>index</th>
<th>name</th>
<th>maskable</th>
<th>type</th>
<th>index</th>
<th>name</th>
<th>maskable</th>
<th>type</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>reset</td>
<td>no</td>
<td>abort</td>
<td>7</td>
<td>FPU overflow</td>
<td>yes</td>
<td>continue</td>
</tr>
<tr>
<td>1</td>
<td>illegal instruction</td>
<td>no</td>
<td>repeat</td>
<td>8</td>
<td>FPU underflow</td>
<td>yes</td>
<td>continue</td>
</tr>
<tr>
<td>2</td>
<td>misalignment</td>
<td>no</td>
<td>repeat</td>
<td>9</td>
<td>FPU loss of accuracy</td>
<td>yes</td>
<td>continue</td>
</tr>
<tr>
<td>3</td>
<td>page fault on fetch</td>
<td>no</td>
<td>repeat</td>
<td>10</td>
<td>FPU division by zero</td>
<td>yes</td>
<td>continue</td>
</tr>
<tr>
<td>4</td>
<td>page fault load store</td>
<td>no</td>
<td>repeat</td>
<td>11</td>
<td>FPU invalid</td>
<td>yes</td>
<td>continue</td>
</tr>
<tr>
<td>5</td>
<td>trap</td>
<td>no</td>
<td>continue</td>
<td>12</td>
<td>FPU unimplemented</td>
<td>no</td>
<td>continue</td>
</tr>
<tr>
<td>6</td>
<td>arithmetic overflow</td>
<td>yes</td>
<td>continue</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Since the main memory is updated before writeback of an instruction, one has to guarantee that in case of an interrupt, all stores prior to the interrupted instruction are executed, but none of the instructions after it. Especially, one has to show that a store that has reached the writeback stage also has accessed the main memory, i.e., it did not enter the wrong execution unit.

5 Floating Point Unit

Execution Units. The FPUs and their verification are described in [11]. The construction and verification of the combinatorial circuits is based on the paper and pencil proofs from [18]. The internal control of the iterative unit for multiplication and division is complex: during cycles, when the division unit performs a subtraction step, the multiplier can be used by multiplication operations or by multiplication steps of other division operations. Moreover, operations with special operands are processed in a single cycle. Thus in general, the units do not process instructions in order, but that is not required by the specifications from section 4. We remark that we have formal proofs but no paper and pencil proofs for the correctness and liveness of the floating point control. The control was constructed and verified with the help of a model checker[10].

At first sight, floating point operations have two operands and one result. However, rounding mode (stored in a special purpose register $RM$) and interrupt masks (stored in $SR$) are two further operands of every floating point operation.

Moreover, there is aliasing in connection with the addressing of the floating point registers: each single precision floating point register can be accessed by single precision operations as well as by double precision operations. The ISA does not preclude the construction of a double precision operand by two writes with single precision to the upper and lower half of a double precision register. It can be necessary to forward these two results from separate places whether the double precision operand is read. This is easily realized by treating the upper half and the lower half of double precision operands as separate operands. Thus, reservation stations for dual precision floating point units have 6 operands.

IEEE Flags and Synchronization. The exception flags for interrupts 6 to 12 are part of the result of every floating point operation $I_i$. They are accumulated in special purpose register $IEEEf$ during writeback of $I_i$. We have already seen in section 4 that this affects
the width of the reorder buffer. A move operation \( I_j \) which reads from register \( IEEEf \) is issued only after the entire reorder buffer is empty. This simple modification of the issue logic makes it very easy to prove that the flags of all floating point operations preceding \( I_j \) are accumulated when \( IEEEf \) is read by \( I_j \). A move instruction from \( IEEEf \) to general purpose register 0, which is constantly 0, acts as a sync operation for self-modifying code as explained at the end of the following section.

6 Memory Interface

**Loads and Stores with Variable Operand Width.** The formal specification of the semantics of the memory instructions is based on the definitions in [18, Chap. 3]. Accesses are characterized by their effective address \( ea \) and their width in bytes \( d \in \{1, 2, 4, 8\} \). The access is aligned if \( ea \mod d = 0 \). Effective addresses \( ea \) define a double word address \( da(ea) = \lfloor ea/8 \rfloor \) and a byte address \( ba(ea) = ea \mod 8 \). A simple “alignment lemma” states that for aligned accesses, the memory operand \( mem[ea + d - 1 : ea] \) equals bytes \( [ba(ea) + d - 1 : ba(ea)] \) of the double word addressed by \( da(ea) \) at the memory interface.\(^6\) Details can be found in [18].

Circuits called \( \text{shift}4\text{load} \) and \( \text{shift}4\text{store} \) are used in order to ensure that data is loaded and stored correctly. These circuits are shown in figure 2. “Shift for store” denotes shifting the data, say the halfword which is to be stored, into the correct position of a double-word before it is sent to the 64-bit wide memory interface. Similarly, “shift for load” denotes extraction of the requested portion (say halfword) of the 64-bit delivered from the memory interface. Also, sign-extension is done during “shift for load” for signed byte- and halfword-loads. Shift for store and load are implemented by means of two simplified shifters with some control logic [18].

The proof of correctness of the VAMP memory interface is structured hierarchically. First, we verify the VAMP with an idealized memory interface \( m_{\text{spec}} \), a dual-ported memory without caches. Second, we show that a cache memory interface with split caches backed up by a unified main memory \( m_{\text{impl}} \) behaves exactly like the dual-ported memory \( m_{\text{spec}} \). Thus, \( m_{\text{spec}} \) serves as the specification for the cache memory interface. By putting these two independent proofs together, we obtain the correctness of the VAMP with split caches with respect to the memory \( mem_S \) of the specification machine.

**Cache Specification and Implementation.** The memory \( m_{\text{spec}} \) is defined recursively, i.e., it is updated on the double word address \( a \) iff a write access to address \( a \) terminates. Separate byte-enables \( mw_{b} \) allow for updating only some of the 8 bytes stored on address \( a \). Formally, we have for any byte \( b < 8 \) and any double word address \( a \):

\[
\begin{align*}
m_{\text{spec}}[8 \cdot a + b]^{T+1} := & \begin{cases} 
din[b]^{T} & \text{if } a = adr^{T} \wedge mw^{T} \wedge mw_{b}^{T} \wedge /dbusy^{T} \\
m_{\text{spec}}[8 \cdot a + b]^{T} & \text{else}
\end{cases}
\end{align*}
\]

The memory interface is implemented with split caches connected to a single main memory as depicted in figure 5. We use a write-back policy for the data cache, i.e., on a

\( ^6 \) Note that this specifies little endian memory organization.
write access of the CPU, the data cache is updated and the corresponding data is marked as dirty. Thus, a slow access to the main memory is avoided. If dirty data is to be evicted from the cache, it is written back to the main memory in order to ensure data consistency.

The protocol used to keep the caches coherent works as follows: If a cache signals a hit on a CPU access, the data is read directly from the cache or written to it, depending on the type of the CPU access. This allows for memory accesses that take only one cycle to complete. If, on the other hand, the cache signals a miss, the corresponding data has to be loaded into the cache. The control first examines the other cache in order to find out if it holds the required data. In this case, the data in the other cache is invalidated. If the data to be invalidated is dirty, this requires an additional write back to the main memory.

This consistency protocol guarantees exclusiveness, i.e., for any address, at most one of the two caches signals a hit. In this way, we ensure that on a hit of the instruction cache, the data cache does not contain newer data.

The instruction and data caches are implemented as \(k\)-way sectored set-associative caches using a LRU replacement policy. Cache sectors consist of 4 double words since the bus protocol supports bursts of length 4.

**Typical Lemmas.** The inductive invariant used to show consistency of split caches as described above consists of three parts. Two of these parts are obvious: if the data or instruction cache, respectively, signals a hit, then its output data equals the specified memory content. However, an invariant consisting only of these two claims is not inductive since caches are reloaded from the main memory. Therefore, we need a third part of our invariant stating the consistency of data in the main memory. Thus, we also claim that on a clean hit or a miss in cycle \(t\) on address \(Dadr^T\) in the data cache, the main memory \(m_{\text{impl}}\) on this address \(Dadr^T\) contains the specified memory content. Note that on a clean hit in the data cache, we thus claim data consistency in both the data cache and the main memory. Formally, we have the following claim:

\[
\begin{align*}
Ihit^T &\implies Idout[b]^T = m_{\text{spec}}[8 \cdot Iadr^T + b]^T \\
Dhit^T &\implies Ddout[b]^T = m_{\text{spec}}[8 \cdot Dadr^T + b]^T \\
/(Dhit^T \land \text{dirty}^T) &\implies m_{\text{impl}}[8 \cdot Dadr^T + b]^T = m_{\text{spec}}[8 \cdot Dadr^T + b]^T.
\end{align*}
\]

This invariant is strong enough to show transparency of the whole memory interface since the data word returned to the CPU on a read access is just the cache output in case of a hit, or the data written to the cache during reload in case of a miss. Note that the invariant relies on the exclusiveness property of the protocol, which has to be verified as part of the proof of the invariant.
Bus Protocol. The main memory is accessed via a bus protocol featuring bursts. The bus protocol signals ready data by raising \( brdy \) one cycle in advance. A sample timing of a 4-burst write is depicted in figure 6. Note that the data input \( din \) one cycle after \( brdy \) is written to the main memory and that the end of the access is signaled by \( \neg reqp \land brdy \).

As part of our correctness proof for the memory interface, we have formalized this bus protocol and proved that an automaton\(^7\) according to figure 7 implements this protocol correctly by means of theorem proving. The main invariant for this proof is the following: in the cycle of the \( i \)-th memory access of the burst, i.e., after the \( i \)-th \( brdy \), the automaton is in state \( \text{mem} \) for the \( i \)-th time. In the cycle of the last memory access, the automaton is in state \( \text{last mem} \).

Self-Modifying Code. We consider self-modifying code independent of the implementation of the memory interface. As an additional precondition for the correctness of code, we demand that in case an instruction is fetched from a memory location \( adr \), there is a special \( sync \)-instruction between the last write to \( adr \) and the fetch of \( adr \).\(^8\) In the VAMP architecture, this \( sync \) instruction is implemented without additional hardware by a special move from the \( IEEEf \) register to \( R0 \) as mentioned in section 5. We have formally verified that this use of the \( sync \) instruction suffices to show the correctness of the implementation in case of self-modifying code.

7 Synthesis

We have translated the PVS hardware description of the VAMP processor to Verilog HDL using an automated tool called \( \text{pvs2hd1} \). The tool unrolls recursive definitions and then performs fairly straightforward translation. The Verilog representation of the

---

\(^7\) Note that this bus control FSD is only a part of the FSD for the cache memory interface.

\(^8\) This implies the correspondency condition from [23].
processor (including caches and floating point unit) has been synthesized, implemented, and tested on a Xilinx FPGA hosted on a PCI board. Some additional unverified hardware for controlling the VAMP processor and for accessing its memory from the host PC is also present on this FPGA. The VAMP processor occupies about 18000 slices of a Xilinx Virtex FPGA. This accounts for a gate count of 1.5 million gates as reported by the Xilinx tools. The design contains 9100 bits of registers (not counting memory and caches) and runs at 10 MHz.

Note that we assume a fully synchronous design, i.e., all registers share the same clock and RAM blocks for register files or caches are also updated synchronous to this clock; thus, concerning timing, they can be treated like registers. In a fully synchronous design, valid data is needed only at the rising edge of the clock with certain setup- and hold-times. The synthesis software analyzes all paths between inputs and registers, registers and registers, and registers and outputs; thus, it can guarantee that our logical design can be implemented with a certain maximum clock speed preserving all our proved properties. In particular, we fully ignore any glitches, i.e., instabilities in signals during a clock period that are resolved until the next rising edge of the clock since these glitches do not influence fully synchronous designs. Thus, our approach does not cover designs where certain signals must be kept stable for several cycles, i.e., where glitches must not occur. This is the case for asynchronous EDO-RAM chips that need stable addresses for a fixed amount of time. Since we use synchronous RAM chips, our proofs guarantee the correctness of the design regardless of any occurring glitches.

We have ported the gcc and the GNU C library for the VAMP in order to execute test programs on the VAMP. As it was to be expected from our verified design, we found no errors in the VAMP processor. When testing some cases of denormal results of floating point operations, however, we found differences between the VAMP FPU and Intel’s Pentium II FPU. This is due to some discrepancies of Intel’s FPU to the IEEE standard. See [11] for further details.

8 Conclusion

Verification Effort. The formal verification of the VAMP microprocessor took about eight person-years; for the translation tool and synthesis on the FPGA, an additional person-year was required. Table 2 summarizes the verification effort for the different parts of the VAMP. Note especially that “Putting it all together” took a whole person-year for several reasons. First of all, the proof of the Tomasulo core from [12] was only generic and had to be applied to the VAMP architecture, especially the VAMP instruction set. Unfortunately, in spite of thorough planning on our part, the interfaces between the different parts did not match exactly. Thus, a lot of effort went into patching the interfaces. Additionally, self-modifying code and the special implementation of the IEEEf-register had to be considered. Also, interrupt support and a memory unit still had to be added to the formally verified Tomasulo core. Last but not least, PVS does not really scale too well for projects this large; typechecking of the VAMP alone takes already more than two hours on our fastest machine.

To the best of our knowledge, we have reported for the first time the formal verification of i) a processor with the full DLX instruction set including load and store
<table>
<thead>
<tr>
<th>Part</th>
<th>Effort in years</th>
<th>Lemmas</th>
<th>Proof steps</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tomasulo core &amp; ALU</td>
<td>2</td>
<td>521</td>
<td>14367</td>
</tr>
<tr>
<td>FPU</td>
<td>3</td>
<td>1046</td>
<td>25936</td>
</tr>
<tr>
<td>Cache Memory Interface</td>
<td>2</td>
<td>566</td>
<td>24432</td>
</tr>
<tr>
<td>Putting it all together</td>
<td>1</td>
<td>415</td>
<td>23887</td>
</tr>
<tr>
<td>Total</td>
<td>8</td>
<td>2548</td>
<td>88622</td>
</tr>
</tbody>
</table>

instructions for bytes, half words, words, and double words, ii) a processor with delayed branch, iii) a processor with maskable nested interrupts, iv) a processor with integrated floating point unit, v) a memory system with separate instruction and data cache. More importantly, the above mentioned constructions and proofs are integrated into a single design and a single correctness proof. Thus, we can be sure that no oversimplifications have been made in any part of the design. PVS ensures that there are no proof gaps left.

The design is synthesized\(^9\) and implemented on an FPGA. The complexity of the design is comparable to industrial controllers with FPUs. To the best of our knowledge, VAMP is by far the most complex processor formally verified so far.

We see several directions for further work in the near future. i) Adding a store buffer to the memory unit. ii) The treatment of a memory management unit with separate translation look aside buffers for data and instructions. iii) Proving formally that a machine with memory management unit and appropriate page fault handlers as part of the operating system gives a single user program the view of a uniform virtual memory. This requires to argue about hardware and software simultaneously. iv) Redoing as much as possible of the present correctness proof with automatic methods. For such methods any subset of our lemmas lends itself as a benchmark suite with a very nice property: we know that it can be completed to the correctness proof of a full bit-level design.

References


\(^9\) The trivial proof of synthesizability.


A Hazards-Based Correctness Statement for Pipelined Circuits

Mark D. Aagaard

Electrical and Computer Engr., University of Waterloo
maagaard@uwaterloo.ca

Abstract. The productivity and scalability of verifying pipelined circuits can be increased by exploiting the structural and behavioural characteristics that distinguish pipelines from other circuits. This paper presents a formal model of pipelines that augments a state machine with information to describe the transfer of parcels between stages, and reading and writing state variables. Using our model, we created a definition of correctness that is based on the well-established principles of structural, control, and data hazards. We have proved that any pipeline that satisfies our hazards-based definition of correctness is guaranteed to satisfy the conventional correctness statement of Burch-Dill style flushing.

1 Introduction

In early verifications of pipelined circuits, the manual effort to discover abstraction functions limited both the productivity and scalability of verification. Burch and Dill’s use of flushing a pipeline to derive an abstraction function automatically [5] improved verification productivity and scalability by sheltering the user from the complexities of the pipeline. Unfortunately, realistic circuits are beyond the scope of such push-button verification. To scale verification to larger pipelines, researchers invented a variety of decomposition strategies. Jones et al. used knowledge about pipeline behaviour to create incremental flushing [8]. Pnueli et al. [4] and Sawada and Hunt [12] used pipeline behaviour as a guide for defining intermediate models. Hosabettu et al. developed completion functions to decompose pipelines stage-by-stage [7]. McMillan used knowledge about the behaviour of pipelines to guide assume-guarantee decomposition [10].

We believe that a model of state machines that captures the distinguishing structure and behaviour of pipelined circuits will improve verification productivity and scalability. The structure of a pipeline is a network of stages through which parcels (instructions) flow. The behaviour of a pipeline can be described using the principles of structural, control, and data hazards. This paper presents a formal model and a correctness statement for pipelines based on stages, parcels, and hazards. Our goals were: remain true to the intuitive meaning of pipelines and hazards, separate orthogonal concerns into distinct correctness obligations, and support cutting-edge optimizations.

Our model of pipelines augments a state machine with pipeline-specific functions and predicates (Section 2): transferring a parcel between stages, writing to a variable,

* This work was supported in part by the National Sciences and Engineering Research Council of Canada and by the Semiconductor Research Corporation Contract RID 1030.001
and reading from a variable. The model supports superscalar and out-of-order execution, external kill signals, exceptions, external interrupts, bypass registers, and register renaming [2]. Our correctness statement, PipeOk, separates correctness obligations relating to different hazards, datapath functionality and flushing (Section 3). We have proved that any pipeline that satisfies PipeOk is guaranteed to satisfy the standard Burch-Dill flushing correctness statement (Section 4).

PipeOk contains thirteen correctness obligations that provide a natural decomposition strategy. Each obligation describes a single type of behaviour, for example, write-after-write hazards. Because hazards are well understood by both verification and design engineers, verification engineers will be able to more easily discuss test plans, verification strategies, and counter examples with designers. Because each obligation focuses on a single type of behaviour, verifying the obligations will be amenable to powerful abstraction mechanisms. For example, the ordering of reads and writes can be verified separately for each variable and need only reason about consecutive operations.

To prove that PipeOk implies Burch-Dill correctness, we prove that PipeOk implies Flushpoint Equality (flushed states are externally equivalent to specification states) and then use the previously proven result that Flushpoint Equality implies Burch-Dill correctness [3]. We prove that PipeOk implies Flushpoint Equality by showing: read and write operations happen in the correct order, the result of each write operation is correct, and finally that flushing works correctly.

2 Modelling Pipelines

This section describes our formal model of pipelines. We begin with an informal description of the “parcel view” of a pipeline, which motivates our approach. The remainder of the section presents the model, auxiliary functions to relate a pipeline to its specification, and conditions to ensure that the auxiliary functions are consistent.

2.1 The Parcel View of a Pipeline

A pipeline is a network of stages. Parcels, or instructions, flow through the stages and read-from and write-to variables, or signals, in the pipeline. Figure 1 shows the runs of a sample program on an instruction set architecture specification, a four-stage pipelined microprocessor, and a “parcel view” of the pipeline. Each run is annotated to show when each parcel moves between stages and when each variable is read or written. The value of a variable is denoted by the label of the instruction that writes to the variable.

Conventional verification strategies compare a snapshot of the pipeline state to a specification state. Because a pipeline state contains the effects of multiple partially executed parcels, it is difficult to relate the implementation to the specification. For example, step 4 of the pipeline contains parcels A, B, C, and D, which represents portions of steps 1, 2, 3, and 4 of the specification. A recent trend has been to examine the implementation only when it is in a flushed state, such as steps 0 and 9 of the pipeline, which are externally equivalent to steps 0 and 5 of the specification.

The parcel view shows slices of the pipeline state as perceived by each parcel. Different variables in the same slice come from different points in time. The slice to
Fig. 1. Specification, pipeline and parcel view of a sample program
the left (right) of each parcel shows the variables as read (written) by the parcel. Gray backgrounds denote values that are with the specification. For example, in step 2 of the parcel view, $R1$ is shown in gray, because $R1$ is $I$ in the pipeline and $A$ in the specification. The parcel for $B$ is able to execute correctly, because it reads its operand from the bypass register, which corresponds to $R1$ at that time.

The parcel view of pipelines was inspired by two observations: first, for each parcel, the only state variables that are relevant to its correctness are those that it reads or writes; second, if every parcel is executed correctly, then the pipeline is correct. Our proof that our correctness statement, $PipeOk$, implies Burch-Dill flushing relies on the parcel view of the pipeline. We have proved that if the order of read and write operations with respect to parcels in the pipeline is the same as the order with respect to states in the specification, then data dependencies are obeyed.

### 2.2 Formal Model of Pipelines

Our formal model of pipelines (Table 1) augments a standard model of non-deterministic state-machines with predicates to detect when parcels transfer between stages, read from state variables, and write to state variables. We use these predicates to compute the parcel view of a pipeline from the next-state relation.

The predicate $xfr$ detects the transfer of a parcel between two stages. We have defined instantiations of $xfr$ for wide variety of protocols for transferring parcels [1]. Transfers can often be detected using one or two signals, such as the valid bits for the stages. In the set of stages, $Top$ and $Bot$ are virtual stages: they do not exist in the pipeline. For input/output pipelines, such as systolic arrays or execution units in microprocessors, $Top$ represents the module in the environment from which parcels enter the pipeline and $Bot$ represents the module to which parcels exit. For closed systems, such as microprocessors

<table>
<thead>
<tr>
<th>Table 1. Definition of a pipeline</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Conventional state machine</strong></td>
</tr>
<tr>
<td>$state$ Set of states.</td>
</tr>
<tr>
<td>$Nsr$ : $state \rightarrow state \rightarrow bool$ Next-state relation.</td>
</tr>
<tr>
<td>$isInit$ : $state \rightarrow bool$ Initial-state predicate.</td>
</tr>
<tr>
<td><strong>Pipeline sets</strong></td>
</tr>
<tr>
<td>$stage$ Set of identifiers for stages in the pipeline, including $Top$ and $Bot$</td>
</tr>
<tr>
<td>$addr_i$ Set of identifiers for data storage variables in the pipeline.</td>
</tr>
<tr>
<td>$isExt$ : $(a : addr_i) \rightarrow (q : state) \rightarrow bool$ Variable is externally visible.</td>
</tr>
<tr>
<td>$isStore$ : $(a : addr_i) \rightarrow (q : state) \rightarrow bool$ Variable is for data storage.</td>
</tr>
<tr>
<td>$subPipes$ : $(s : stage) \rightarrow pipe$ One pipe record for each stage</td>
</tr>
<tr>
<td><strong>Probes</strong></td>
</tr>
<tr>
<td>$xfr$ : $(q : state) \rightarrow (s_1 : stage) \rightarrow (s_2 : stage) \rightarrow bool$</td>
</tr>
<tr>
<td>In state $q$, a parcel transfers from $s_1$ to $s_2$</td>
</tr>
<tr>
<td>$Wr$ : $(a : addr_i) \rightarrow (q : state) \rightarrow (s : stage) \rightarrow bool$</td>
</tr>
<tr>
<td>A parcel in $s$ writes to address $a$ in state $q$</td>
</tr>
<tr>
<td>$Rd$ : $(a : addr_i) \rightarrow (q : state) \rightarrow (s : stage) \rightarrow bool$</td>
</tr>
<tr>
<td>A parcel in $s$ reads from address $a$ in state $q$</td>
</tr>
</tbody>
</table>
Table 2. Functions for comparing a pipeline and specification

<table>
<thead>
<tr>
<th>Sets</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$addr_s$</td>
<td>Set of identifiers for data storage variables in the specification.</td>
</tr>
<tr>
<td>$data_s$</td>
<td>Set of data values in the specification.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Structural-hazard correctness</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$Match$</td>
<td>$(\sigma : \text{run}) \rightarrow (t_1 : \text{time}) \rightarrow (t_n : \text{time}) \rightarrow \text{bool}$</td>
</tr>
<tr>
<td>The parcel that enters at time $t_1$ exits at time $t_n$.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Control-hazard correctness</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$ShouldExit$</td>
<td>$(\sigma : \text{run}) \rightarrow (t : \text{time}) \rightarrow \text{bool}$</td>
</tr>
<tr>
<td>The parcel that enters should eventually exit</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Data-hazard and datapath correctness</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$addrmap$</td>
<td>$(a : addr_i) \rightarrow (q : \text{state}) \rightarrow addr_s$</td>
</tr>
<tr>
<td>Maps addresses of implementation to addresses in the specification</td>
<td></td>
</tr>
<tr>
<td>$datamap$</td>
<td>$(a : addr) \rightarrow (q : \text{state}) \rightarrow data_s$</td>
</tr>
<tr>
<td>Maps the data in $q.a$ to corresponding specification data value</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Flushing correctness</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$Flush$</td>
<td>$\text{state} \rightarrow \text{state}$</td>
</tr>
<tr>
<td>Flushes a state</td>
<td></td>
</tr>
<tr>
<td>$IsFlushed$</td>
<td>$\text{state} \rightarrow \text{bool}$</td>
</tr>
<tr>
<td>A state is flushed</td>
<td></td>
</tr>
</tbody>
</table>

with built-in memory, transferring from/to Top and Bot is defined in terms of operations in the pipeline, such as fetching an instruction. Pipelines may contain atomic stages, which hold at most one parcel, and hierarchical stages, which may themselves be pipelines. We support this with the subPipes field.

State machines commonly distinguish internal and external variables (isExt for “is external”). We refine this by dividing variables into data-storage and pipeline variables (isStore for “is storage”). Data-storage variables are used to represent variables in the specification, and can be either internal (e.g., bypass registers) or external (e.g., register files). Pipeline variables are the registers that hold parcels in stages. They are internal and have no corresponding variables in the specification. Read and write predicates need only monitor storage variables.

2.3 Relating Implementations and Specifications

To verify a pipeline against a specification, we need to compare the behaviours of the pipeline and specification. Typically, this is done with a function to say how many instructions are fetched and an external-equivalence relation. Table 2 shows the analogous objects for our model.

We use $Match$ to identify the entrance and exit time of each parcel. $Match$ supports superscalar pipelines by instantiating the type time with a pair of a clock cycle and a port [1]. When working with hierarchical pipelines, we want to treat the stages as black boxes. The $Match$ relation allows us to match parcels entering and exiting stages while hiding the internal structure of the stage. We have found five common instantiations for $Match$: degenerate, constant latency, in-order, unique tags, and tagged in-order [1].
Table 3. Consistency Conditions on Pipelines and Specifications

<table>
<thead>
<tr>
<th>Specification conditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 The specification is deterministic. This is required for flushpoint-equality correctness to imply Burch-Dill correctness. Implementations may be non-deterministic.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Traversal conditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 If ShouldExit is true, then a parcel entered the pipeline.</td>
</tr>
<tr>
<td>3 Parcels cannot transfer from the pipeline to the “Top” stage.</td>
</tr>
<tr>
<td>4 Parcels cannot transfer from the “Bot” stage to the pipeline.</td>
</tr>
<tr>
<td>5 Time increases monotonically as parcels traverse through the pipeline.</td>
</tr>
<tr>
<td>6 IsFlushed cannot be true while a parcel is traversing through the pipeline.</td>
</tr>
<tr>
<td>7 A storage operation can happen in a stage only if a parcel is in the stage.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Storage Conditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>8 If an address map changed, then a write must have happened in Impl.</td>
</tr>
<tr>
<td>9 If a data map changed, then a write must have happened in Impl.</td>
</tr>
<tr>
<td>10 If a Spec variable changed, then a write must have happened in Spec.</td>
</tr>
<tr>
<td>11 When a pipeline is flushed, external equality and storage equality are identical.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Flushing conditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>12 Flush is idempotent on flushed pipelines.</td>
</tr>
<tr>
<td>13 All reachable states are reachable from a flushed state.</td>
</tr>
<tr>
<td>14 From any state, a flushed state can be reached eventually.</td>
</tr>
</tbody>
</table>

The predicate ShouldExit says whether a parcel that enters the pipeline should be executed. We have identified instantiations for ShouldExit that include external kill signals, branch prediction, internal exceptions, and external interrupts [2].

We separate external equivalence into two functions: addrmap, which defines a mapping between variables in the pipeline and specification, and datamap, which maps data in the pipeline to the specification. Address maps may be dependent on the current state: the identity of the specification variable that a bypass register represents is dependent upon the contents the bypass register. When an implementation variable does not represent any specification variable (e.g., a bypass register when it contains a bubble), addrmap returns ⊥, as shown in steps 0–2 for the pipeline in Figure 1.

To relate PipeOk to flushpoint equality and Burch-Dill flushing, we require that each pipeline defines a function Flush and a predicate isFlushed.

2.4 Consistency Conditions

Table refconds summarizes the conditions required for the predicates and functions in the pipeline model to be consistent with the behaviour of the state machine in the model. The complete mathematical definitions appear in a technical report [2].

3 Correctness Obligations

We begin with a summary of our notation. We present our correctness obligations according to the different types of hazards, datapath functionality, and flushing
3.1 Notation

When working with theorems relating a run of a specification to a run of an implementation, we often find it useful to draw “box” or commuting diagrams (Figure 2a). In Figure 2a, \(x\) and \(y\) refer to the states shown as circles. Properties associated with states and edges are listed in Figure 2b. We denote the \(t\)th element of a run \(\sigma\) as: \(\sigma^t\). We use \(\text{run } m \sigma\) to mean that \(\sigma\) is a run of the state-machine \(m\), as defined by: \(\forall t. \ m \sigma^t \sigma^{t+1}\). As a syntactic shorthand, we write \(m q q'\) rather than \(m.Nsr q q'\), and we drop the name of the pipeline when referring to parameters other than \(Nsr\).

![Fig. 2a. Graphical notation](image)

**Fig. 2.** Notation and conventions

3.2 Top-Level Correctness Statements

Our top-level correctness statement, Definition 1, \(\text{PipeOk}\), is the conjunction of thirteen correctness obligations. Each correctness obligation guarantees that a particular type of behaviour is implemented correctly. Section 3.3 describes structural-hazard correctness; Section 3.4 describes data-hazard correctness; Section 3.5 describes datapath functionality correctness; Section 3.6 describes additional correctness obligations needed to ensure that flushed states are externally equivalent to specification states. There are no correctness obligations that address only control hazards. Instead, control hazards permeate both structural hazard correctness and data hazard correctness. For structural hazards, we make sure that correctly speculated parcels are executed and incorrectly speculated parcels do not exit the pipeline. For data hazards, we make sure that incorrectly speculated parcels do not leave behind data results that are read by correctly speculated parcels.
Definition 1 Correctness of pipelines

\[
\text{PipeOk Impl Spec} \equiv \\
\begin{align*}
\text{Struct-hazard correctness} & \\
& \land \text{1 EnterTotFun Impl} \\
& \land \text{2 ExitTotFun Impl} \\
& \land \text{3 MatchIfTrav Impl} \\
\text{Data-hazard correctness} & \\
& \land \text{5 WawHazOk Impl Spec} \\
& \land \text{4 RawHazOk Impl Spec} \\
& \land \text{6 WarHazOk Impl Spec} \\
& \land \text{7 SpecRdTotFun Impl Spec} \\
& \land \text{8 SpecWrTotFun Impl Spec} \\
& \land \text{9 ImplWrTotFun Impl Spec} \\
\end{align*}
\]

\[
\begin{align*}
\text{Datapath correctness} & \\
& \land \text{10 DatapathOk Impl Spec} \\
\text{Flushing correctness} & \\
& \land \text{11 ImplWrFlush Impl Spec} \\
& \land \text{12 SpecWrFlush Impl Spec} \\
& \land \text{13 ImplInvalidateFlush Impl Spec} \\
\end{align*}
\]

3.3 Structural-Hazard Correctness Obligations

Structural hazard correctness is concerned with contention between parcels for resources in the pipeline. Typical bugs associated with structural hazards are loss of parcels, duplication of parcels, generation of bogus parcels inside the pipeline, deadlock, and livelock. A pipeline handles its structural hazards correctly if there is a one-to-one mapping between parcels that enter the pipeline and should exit and those parcels that do exit, and if the parcels that exit do so in the correct order.

Definition 2 tracks a parcel as it traverses from stage to stage in a pipeline. The expression \((t_1, s_1) \trans {\sigma} (t_n, s_n)\) means that in the run \(\sigma\), a parcel enters the stage \(s_1\) at \(t_1\), traverses from \(s_1\) to \(s_n\), and exits the stage \(s_n\) at \(t_n\). In the base case \(s_1\) and \(s_n\) are the same stage. In the inductive case, there is an intermediate stage \(s_2\) such that the parcel transfers from \(s_1\) to \(s_2\) and then traverses from \(s_2\) to \(s_n\). To detect when the parcel exits \(s_1\), we use the matching relation provided by \(s_1\), according to our hierarchical model of pipelines. Definition 2 supports pipelines with loops, because Match separately identifies each iteration. We use \(\trans{\cdot}\) to define \(\text{Trav}\), which means a parcel traverses through the pipeline from Top to Bot.

Definition 2 Traversing between stages in a pipeline (\(\trans{\cdot}\))

\[
(t_1, s_1) \trans{\sigma} (t_n, s_n) \equiv \\
\left[ \begin{array}{c}
\land s_1 = s_n \\
\land s_1.\text{Match } \sigma t_1 t_x
\end{array} \right] \lor \left[ \begin{array}{c}
\exists t_2, s_2. \\
\land s_1.\text{Match } \sigma t_1 t_2 \\
\land xfr \sigma t_2 s_1 s_2 \\
\land (t_2, s_2) \trans{\sigma} (t_n, s_n)
\end{array} \right]
\]

Obligation 1, EnterTotFun, says that for each time \((t_1)\) that a parcel enters the pipeline and should exit, there exists exactly one time \((t_2)\) such that the parcel exits at \(t_2\) (total and functional). Obligation 2, ExitTotFun, says that each parcel that exits the pipeline \((XfrOut)\) comes from exactly one parcel that entered the pipeline and should have exited (surjective and injective). Together, Obligations 1 and 2 guarantee that the relationship between entering and exiting parcels is bijective.
**Obligation 1** Each entrance results in exactly one exit

\[ \text{EnterTotFun} \implies \forall \sigma_i, t_1. \]

\[ \left[ \land \text{run} \implies \sigma_i \land \text{isFlushed} \sigma_0^i \land \text{ShouldExit} \sigma_i t_1 \right] \implies \exists! t_2. \text{Trav} \implies \sigma_i t_1 t_2 \]

**Obligation 2** Each exit comes from exactly one entrance

\[ \text{ExitTotFun} \implies \forall \sigma_i, t_2. \]

\[ \left[ \land \text{run} \implies \sigma_i \land \text{IsFlushed} \sigma_0^i \land \text{XfrOut} \sigma_i t_1 t_2 \right] \implies \exists! t_1. \text{Trav} \implies \sigma_i t_1 t_2 \land \text{ShouldExit} \sigma_i t_1 \]

**Obligation 3** Match correctly identifies when a parcel traverses the pipeline

\[ \text{MatchIffTrav} \implies \forall \sigma, t_1, t_2. \]

\[ \left[ \text{Match} \implies \sigma t_1 t_2 \right] \iff \left[ \text{Trav} \implies \sigma t_1 t_2 \right] \]

### 3.4 Data-Hazard Correctness Obligations

A data-depenency exists between a producing (writing) instruction and a consuming (reading) instruction if the producing instruction writes to an address that the consuming instruction reads from and no instruction between the producer and the consumer writes to that address. A pipeline implements data dependencies correctly if every data dependency in the specification is obeyed in the implementation.

Data hazards are categorized as: read-after-write, write-after-read, and write-after-write. If a pipeline handles all three types of data hazards correctly, then it implements data dependencies correctly. In Figure 3, the gray lines represent orderings between specification and implementation operations that will violate the dependency between \( W_i \) and \( R_i \). Read-after-write (Raw) hazard correctness guarantees that \( R_i \) occurs after \( W_i \). Together, write-after-write and write-after-read hazard correctness guarantee that no write will occur to this address between \( W_i \) and \( R_i \). Write-after-write (Waw) correctness guarantees that no programmatically earlier write happens after \( W_i \). Write-after-read (War) correctness guarantees that no programmatically later write will occur before \( R_i \). Figure 3 has many simplifications that are violated by optimizations such as bypass registers, register renaming, and out-of-order execution. Our formalization supports these optimizations using dynamic address maps, multiple writes, and out-of-order writes [2].

The data hazard obligations ensure that reads and writes in the implementation occur in the correct order. We use the symbols \( \text{WR} \leftarrow \text{Rd} \), \( \text{RD} \leftarrow \text{WR} \), and \( \text{WR} \leftarrow \text{WR} \) to denote consecutive write and read operations in a run. Definition 3 describes a read following a write to the address \( a \) in the run \( \sigma \). To the right of the text is an illustration of the definition using the graphical notation presented in Figure 2a. The definitions for a write following a read and a write following a write are similar.
Definition 3 Consecutive read-after-write ordering

\[(t_w, s_w)_{\text{Spec}} \overset{\sigma}{\lambda} \text{Rd} (t_r, s_r) \equiv \]
\[\land \text{Wr } a \sigma t_w s_w \land \text{Rd } a \sigma t_r s_r \land t_w < t_r \land \forall t \in \{t_w + 1..t_r - 1\}. \forall s. \neg(\text{Wr } a \sigma t s)\]

Obligation 4, \text{RawHazOk}, says that if there is a data-dependency in the specification and a corresponding read in the implementation \((a_i, t_{ri}, s_r)\), then there must exist a corresponding write \((a_i, t_{wi}, s_w)\) that happens before the read.

Obligation 4 Correctness of read-after-write data hazards

\text{RawHazOk} \implies \text{Spec} \equiv \forall \sigma_s, \sigma_i, a_s, t_{ws}, t_{rs}, a_i, t_{ri}, s_r,

\[
\land \text{Spec } \sigma_s \overset{\text{PCL}}{\lambda} \text{Impl } t_{ws} \land (a_s, t_{rs}) \sigma_s \overset{\text{PCL}}{\lambda} (a_i, t_{ri}, s_r) \land \exists t_{wi}, s_w.
\]

\[
\land \text{Wr } a_i \sigma_t t_{wi} s_w \land (a_s, t_{ws}) \sigma_s \overset{\text{PCL}}{\lambda} (a_i, t_{wi}, s_w) \land t_{wi} < t_{ri} \land t_{wi} \in \{t_{ws} + 1..t_{ri} - 1\}.
\]

\text{RawHazOk} contains the first appearance of the relation \text{Spec} \sigma_s \overset{\text{PCL}}{\lambda} \text{Impl} (“run correspondence”) which says that: \(\sigma_s\) is a run of \text{Spec}, \(\sigma_i\) is a run of \text{Impl} from a flushed state, and the initial states of \(\sigma_s\) and \(\sigma_i\) are externally equivalent.

We formalize an operation in a run of an implementation corresponding to an operation in the specification by tracking a parcel as it traverses the pipeline. The \(n\)th parcel that enters the pipeline and should exit corresponds to the \(n\)th step of the specification. The expression \(t_s \overset{\text{PCL}}{\lambda} (t_{in}, s_n)\) means that at time \(t_{in}\), the \(t_s\)th parcel that entered the pipeline and should exit is either inside the stage \(s_n\) or is just exiting \(s_n\).

Read and write correspondences are defined in terms of parcel correspondence \(\overset{\text{PCL}}{\lambda}\). The expression \((a_s, t_{ws}) \overset{\text{PCL}}{\lambda} (a_i, t_{wi}, s)\) means: the specification instruction at time \(t_{ws}\) writes to address \(a_s\), the instruction corresponds to the parcel in stage \(s\) at time \(t_{wi}\), the parcel writes to \(a_i\), and the address map of \(a_i\) at time \(t_{wi}\) points to \(a_s\).

Write-after-write and write-after-read hazards are dealt with by Obligations 5 and 6, both of which have a case for in-order writes and a case for out-of-order writes.
The in-order cases are simpler, because they deal only with consecutive operations, as denoted by $W$. The out-of-order cases require looking beyond consecutive operations, because we do not know how far out-of-order the operations will be. We use "$W \preceq W$" and "$R \preceq W$" for the transitive ordering of write and read operations.

**Obligation 5** Correctness of write-after-write data hazards

$WawHazOk$ Impl Spec $\equiv$

\[
\forall \sigma_s, \sigma_i, a_s, t_{ws1}, a_i, t_{wi1}, s_{w1}, t_{wi2}, s_{w2}.
\]

\[
\begin{align*}
\text{Spec} \sigma_s \sigma_i \text{ Impl} \\
\wedge (a_s, t_{ws1}) \frac{\sigma_s \sigma_i}{W} (a_i, t_{wi1}, s_{w1})
\end{align*}
\]

\[
\Rightarrow \\
\forall t_{ws2},
\begin{align*}
&\left[ t_{ws1} \frac{\sigma_s}{W} t_{ws2} \\
&\wedge (a_s, t_{ws2}) \frac{\sigma_s \sigma_i}{W} (a_i, t_{wi1}, s_{w2})
\right]
\end{align*}
\]

\[
\Rightarrow t_{wi1} < t_{wi2}
\]

\[
\begin{align*}
&\left[ t_{ws1} \frac{\sigma_s}{W} t_{ws2} \\
&\wedge t_{ws1} \frac{\sigma_s}{W} t_{ws2} \\
&\wedge t_{ws2} \frac{\sigma_s}{Rd} t_{rs2} \\
&\wedge (a_s, t_{ws2}) \frac{\sigma_s \sigma_i}{W} (a_i, t_{wi1}, s_{w2}) \\
&\wedge (a_s, t_{rs2}) \frac{\sigma_i}{Rd} (a_i, t_{ri1}, s_{r1})
\right]
\end{align*}
\]

\[
\Rightarrow t_{rs2} \leq t_{wi1}
\]

**Obligation 6** Correctness of write-after-read data hazards

$WarHazOk$ Impl Spec $\equiv$

\[
\forall \sigma_s, \sigma_i, a_s, t_{rs1}, a_i, t_{ri1}, s_{r1}.
\]

\[
\begin{align*}
\text{Spec} \sigma_s \sigma_i \text{ Impl} \\
\wedge (a_s, t_{rs1}) \frac{\sigma_s \sigma_i}{Rd} (a_i, t_{ri1}, s_{r1})
\end{align*}
\]

\[
\Rightarrow \\
\forall t_{ws2},
\begin{align*}
&\left[ t_{rs1} \frac{\sigma_s}{Rd} t_{ws2} \\
&\wedge (a_s, t_{ws2}) \frac{\sigma_s \sigma_i}{W} (a_i, t_{wi1}, s_{w2})
\right]
\end{align*}
\]

\[
\Rightarrow t_{ri1} \leq t_{wi2}
\]

\[
\begin{align*}
&\left[ t_{rs1} \frac{\sigma_s}{Rd} t_{ws2} \\
&\wedge (t_{wi1}, s_{w1}) \frac{\sigma_i}{W} (t_{ri1}, s_{r1})
\right]
\end{align*}
\]

\[
\Rightarrow t_{wi2} < t_{wi1}
\]

The out-of-order case for Obligation 5 requires that $t_{wi1}$ does not corrupt data by occurring between another implementation write ($t_{wi2}$) and its dependent read ($t_{ri2}$).
The out-of-order case of Obligation 6, WarHazOk, is simpler than that of Obligation 5, WawHazOk, because we do not need to mention the specification write that corresponds to \( t_{wi1} \). The purpose of the out-of-order case is to allow \( t_{wi2} \) to happen before \( t_{ri1} \) while ensuring that \( t_{wi2} \) does not corrupt the data intended for \( t_{ri1} \). If \( t_{wi2} \) corrupts the data, then \( t_{wi2} \) will be the producer for \( t_{ri1} \), which causes the right-hand-side of the implication to be \( t_{wi2} < t_{wi2} \), which is clearly false.

Obligations 4–6 guarantee that, if read and write operations occur in the implementation, then they will occur in the correct order. These obligations do not guarantee that the operations actually do occur in the implementation. Obligations 7–9 ensure that reads and writes in the specification will also occur in the implementation and that writes that occur in the implementation correspond to writes in the specification. For brevity, we omit the mathematical definitions, which can be found elsewhere [2].

**Obligation 7** SpecRdTotFun Impl Spec \( \equiv \) Each read operation in Spec corresponds to exactly one read operation in Impl

We allow multiple writes in the implementation to correspond to a single write in the specification, so long as the writes are to different variables (Obligation 8, SpecWrTotFun). This feature is required to support simple optimizations, such as bypass registers, as well as complex optimizations, such as retirement register files.

**Obligation 8** SpecWrTotFun Impl Spec \( \equiv \) Each write in Spec has at least one corresponding write in Impl. If two writes in Impl correspond to the same write in Spec, then the Impl writes must be to different addresses in Impl.

We allow implementations to perform writes that do not correspond to writes in the specification, so long as these writes are not read (Obligation 9, ImplWrTotFun). This freedom provides a uniform mechanism for implementations to invalidate data, (remapping a register in register renaming) as well as modify the contents of variables that are not needed (bubbles changing the value of a bypass register as they propagate through it). A variable is invalid if its address map is changed so that it no longer points to an address in the specification. As shown in Figure 1, when a bypass register contains a bubble, we say that its address map returns \( \perp \). Obligations 11–13 in Section 3.6 ensure that these writes do not corrupt data before a flushed state.

**Obligation 9** ImplWrTotFun Impl Spec \( \equiv \) Each write in Impl that is the last write before a read from the same address must have a corresponding write in Spec.

### 3.5 Datapath Correctness Obligation

Definition 4 describes when two storage variables are equivalent: their address maps point to the same address and their data maps return the same data value.

**Definition 4** Equality of storage variables

\[
(a_1, q_1) =_{\text{STORE}} (a_2, q_2) \equiv [a_1 = \text{addrmap } a_2 q_2] \land [q_1.a_1 = \text{datamap } a_2 q_2]
\]

The datapath of a pipeline is correct if, assuming every read operation that a parcel performs will consume the correct data, then every write that parcel performs must produce the correct data (Obligation 10, DatapathOk). The clause dealing with reads is nested within the antecedent to provide a uniform way of dealing with both parcels that
performs reads and those whose results are independent of the contents of the pipeline storage variables.

**Obligation 10 Correctness of datapath (DatapathOk)**

\[
\text{DatapathOk Impl Spec } \equiv \\
\forall \sigma_s, \sigma_i, a_s, t_s, a_{wi}, t_{wi}, s_w.
\]

\[
\left( a_{rs}, t_s \right)^{\sigma_s} \overset{\text{Rd}}{\rightarrow} \left( a_{ri}, t_{ri}, s_r \right) \\
\left( a_{rs}, a_{ri}, t_{ri}, s_r \right) \\
\left( a_{rs}, t_s \right)^{\sigma_s} \overset{\text{STORE}}{\rightarrow} \left( a_{ri}, \sigma_i^{t_{ri}} \right) \\
\left( a_{ws}, t_s \right)^{\sigma_s} \overset{\text{Wr}}{\rightarrow} \left( a_{wi}, t_{wi}, s_w \right)
\]

\[
\Rightarrow \left( a_{ws}, \sigma_i^{t_{wi}+1} \right) \overset{\text{STORE}}{\rightarrow} \left( a_{wi}, \sigma_i^{t_{wi}+1} \right)
\]

3.6 **Flushing Correctness Obligations**

Using Obligations 1–10, we have proved that every parcel that enters the pipeline and should exit, will produce the correct result (WriteOk in Figure 4). It may seem that this is a sufficient definition of correctness, however it allows externally visible state variables that are written but never read to contain incorrect data. We solve this problem with Obligations 11–13 (mathematical definitions appear elsewhere [2]). Obligation 11, ImplWrFlush, is analogous to Obligation 9, ImplWrTotFun, except that it is concerned with writes before flushed states, rather than writes before reads. Obligation 12, SpecWrFlush, ensures that in a flushed implementation state, the last writes that happened in the specification have corresponding writes in the implementation. Finally, Obligation 13, ImplInvalidateFlush, ensures that for each specification variable, there is at least one corresponding implementation variable. This is done by preventing the invalidation of the last corresponding implementation variable.

**Obligation 11 ImplWrFlush Impl Spec \equiv Last visible writes in impl before flushed states correspond to writes in spec.**

**Obligation 12 SpecWrFlush Impl Spec \equiv Last visible writes in spec occur in impl**

**Obligation 13 ImplInvalidateFlush Impl Spec \equiv If the address map of a variable (a_i) changes, then in the next clock cycle there must be another implementation variable (a_2) such that the address map of a_2 points to the same specification address as a_1 used to point to.**

4 **Proof That Hazard-Correctness Implies Burch-Dill Correctness**

The proof that PipeOk implies Burch-Dill flushing (Theorem 1) contains four major steps that are linked by transitivity (Figure 4). In the first step, we used the correctness obligations for structural, control, and data hazards (Obligations 1–9) to prove that the read and write operations in the implementation obey data dependencies in the specification. That is, the operations exist and occur in the correct order (DataDepOk). In
the second step, we combined the ordering of data-storage operations with the correctness of the datapath (Obligation 10) to prove that every write operation writes the correct data (WriteOk). In the third step, we combined the correctness of write operations with the correctness obligations for flushing (Obligations 11–13) to prove that when a pipeline is in flushed state, it will correspond to the specification (FlushedEq). The definition of FlushedEq comes from the Microbox work of Aagaard et al [3], where it is identified by the acronym iFEND for “informed-flushpoint with equality between a non-deterministic implementation and a deterministic specification”.

**Definition 5 Burch-Dill correctness**

\[
\text{BurchDillOk Impl Spec} \equiv \\
\forall q_i, q_s, q'_i.
\left\{ \\
\text{Flush } q_i \equiv q_s \\
\text{Impl } q_i q'_i \\
\text{DoesFetch } q_i q'_i
\right\} 
\Rightarrow 
\exists q'_s,
\left\{ \\
\text{Flush } q'_i \equiv q'_s \\
\text{Spec } q_s q'_s
\right\}
\]

**Theorem 1** Pipeline correctness implies Burch-Dill correctness

\[
\text{PipeOkImpBurchDillOk} \equiv \\
\forall \text{Impl, Spec.}
\text{PipeOk Impl Spec} \implies \text{BurchDillOk Impl Spec}
\]

![Proof sketch that PipeOk implies Burch-Dill flushing](image)

5 Conclusions

Some related work has been on correctness for pipelined circuits. Tahar and Kumar defined correctness statements for the different types of hazards in a single-scalar, in-order microprocessor [13]. Manolios has used bisimulation and retiming to relate the run of a pipeline to a specification using state-based abstraction functions, such as flushing [9]. Mishra et al defined correctness for pipelined microprocessors with the restriction that instructions proceed from stage to stage in a lockstep order [11].

Some of the lemmas and decomposition strategies used by others are similar to correctness obligations in our work. McMillan’s inductive proof to show that each instruction that reads correct data will write correct results [10] is similar to our obligation for datapath correctness. Sawada’s MAETT annotates implementation states with history and prophecy variables to facilitate separating the effects of individual instructions [12].
This is similar in flavour to our use of read and write operations to identify the relevant state variables for each instruction. Ho’s *token networks* [6] are a verification strategy that might yield useful abstractions to verify our structural hazard obligations.

The goal of the work presented here was to establish a formal foundation for pipelined circuits that would increase verification capacity and productivity, be intuitive to both verification engineers and design engineers, and handle cutting-edge optimizations in pipelines. We have defined a formal model and correctness statement (*PipeOk*) based upon conventional notions of stages, parcels, and hazards. We have proved that the correctness statement guarantees Burch-Dill flushing correctness. *PipeOk* is comprised of thirteen correctness obligations: three for structural hazards, six for data hazards, one for the datapath, and three for flushing. Control hazards are integrated into structural and data hazard correctness. The correctness obligations each deal with a specific type of behaviour, which should make them amenable to powerful abstraction and problem reduction techniques. We have begun several case studies to evaluate the effectiveness of *PipeOk* using a combination of model checking and theorem proving. After the case studies indicate that our model and correctness statement are effective, we will mechanize the proof that *PipeOk* implies Flushpoint Equality.

**References**

Analyzing the Intel Itanium Memory Ordering Rules Using Logic Programming and SAT*

Yue Yang, Ganesh Gopalakrishnan, Gary Lindstrom, and Konrad Slind

School of Computing, University of Utah
{yyang, ganesh, gary, slind}@cs.utah.edu

Abstract. We present a non-operational approach to specifying and analyzing shared memory consistency models. The method uses higher order logic to capture a complete set of ordering constraints on execution traces, in an axiomatic style. A direct encoding of the semantics with a constraint logic programming language provides an interactive and incremental framework for exercising and verifying finite test programs. The framework has also been adapted to generate equivalent boolean satisfiability (SAT) problems. These techniques make a memory model specification executable, a powerful feature lacked in most non-operational methods. As an example, we provide a concise formalization of the Intel Itanium memory model and show how constraint solving and SAT solving can be effectively applied for computer aided analysis. Encouraging initial results demonstrate the scalability for complex industrial designs.

1 Introduction

Modern shared memory architectures rely on a rich set of memory access related instructions to provide the flexibility needed by software. For instance, the Intel Itanium\textsuperscript{TM} processor family [1] provides two varieties of loads and stores in addition to fence and semaphore instructions, each associated with different ordering restrictions. A memory model defines the underlying memory ordering semantics. Proper understanding of these ordering rules is essential for the correctness of shared memory consistency protocols that are aggressive in their ordering permissiveness, as well as for compiler transformations that rearrange multithreaded programs for higher performance. Due to the complexity of advanced computer architectures, however, practicing designers face a serious problem in reliably comprehending the memory model specification.

Consider, for example, the assembly code shown in Fig. 1 that is run concurrently on two Itanium processors (such code fragments are generally known as litmus tests): The first processor, P1, executes a store of datum 1 into ad-

* This work was supported by a grant from the Semiconductor Research Corporation for Task 1031.001, and Research Grants CCR-0081406 and CCR-0219805 of NSF.
Fig. 1. A litmus test showing the ordering properties of store-release and load-acquire. Initially, a = b = 0. Can it result in r1 = 1 and r2 = 0? The Itanium memory model does not permit this result.

dress a; it then performs a store-release\(^1\) of datum 1 into address b. Processor P2 performs a load-acquire from b, loading the result into register r1. It is followed by an ordinary load from location a into register r2. The question arises: if all locations initially contain 0, can the final register values be r1=1 and r2=0? To determine the answer, the Itanium memory model must be consulted. The formal specification of the Itanium memory model is given in an Intel application note [2]. It comprises a complex set of ordering rules, 24 of which are expressed explicitly based on a large amount of special terminology. One can follow a pencil-and-pen approach to reason that the execution shown in Fig. 1 is not permitted by the rules specified in [2]. Based on this, one can conclude that even though the instructions in P2 pertain to different addresses, the underlying hardware is not allowed to carry out the ordinary load at the beginning, and by the same token, a shared memory consistency protocol or an optimizing compiler cannot reorder the instructions in P2. A further investigation shows that the above result would be permitted if the st.rel in P1 is changed to a st, or the ld.acq in P2 is changed to a ld. Therefore, st.rel and ld.acq must both be used in pairs to achieve the “barrier” effect in this scenario.

A litmus test like this can reveal critical information to help system designers make right decisions in code selection and optimization. But as bigger tests are used and more intricate rules are involved, trace properties quickly become non-intuitive and hand-proving program compliance can be very difficult. How can one be assured that there does not exist an interacting rule that might introduce unexpected implications? Also, a large scale design is often composed of simpler components. To avoid being overwhelmed by the overall complexity, a useful technique is to isolate the rules related to specific architectural features so that the model can be analyzed piece by piece. For example, if one can selectively enable/disable certain rules, one may quickly find out that the “program order” rules in [2] are critical to the scenario in Fig. 1 while many others are irrelevant.

These issues suggest that a series of useful features is needed from the specification framework to help people better understand the underlying model. Unfortunately, most non-operational specification methods leave these issues unresolved because they use notations that do not support analysis through execu-

\(^1\) Briefly, a store-release instruction will, at its completion, ensure that all previous instructions are completed; a load-acquire instruction correspondingly ensures that all following instructions will complete only after it completes. These explanations are far from precise - what do “previous” and “completion” mean? A formal specification of a memory model is key to precisely capture these and all similar notions.
tion. Given that designers need lucid and reliable memory model specifications, and given that memory model specifications can live for decades, it is crucial that progress be made in this regard.

In this paper, we take a fresh look at the non-operational specification method and explore what verification techniques can be applied. We make the following contributions. First, we present a compositional method to axiomatically capture all aspects of the memory ordering requirements, resulting a comprehensive, constraint-based memory consistency model. Second, we propose a method to encode these specifications using FD-Prolog. This enables one to perform interactive and incremental analysis. Third, we have harnessed a boolean satisfiability checker to solve the constraints. To the best of our knowledge, this is the first application of SAT methods for analyzing memory model compliance. As a case study of this approach, we have formalized a core subset of the Itanium memory model and used constraint programming and boolean satisfiability for program analysis.

Related Work. The area of memory model specification has been pursued under different approaches. Some researchers have employed operational style specifications [3] [4] [5] [6], in which the update of a global state is defined step-by-step with the execution of each instruction. For example, an operational model [4] for Sparc V9 [7] was developed in Murphi. With the model checking capability supported by Murphi, this executable model was used to examine many code sequences for Sparc V9. While the operational descriptions often mirror the decision process of an implementer and can be exploited by a model checker, they are not declarative. Hence they tend to emphasize the how aspects through their usage of specific data structures, not the what aspects that formal specifications are supposed to emphasize.

Other researchers have used non-operational (also known as axiomatic) specifications, in which the desired properties are directly defined. Non-operational styles have been widely used to describe conceptual memory models [8] [9]. One noticeable limitation of these specifications is the lack of a means for automatic execution. An axiomatic specification of the Alpha memory model was written by Yu [10] in Lisp. Litmus tests were written in S-expression. Verification conditions were generated for the litmus tests and fed to the Simplify [11] verifier of Compaq/SRC. In contrast, our specification is much closer to the actual industrial specification, thanks to the declarative nature of FD-Prolog. The FD constraint solver offers a more interactive and incremental environment. We have also applied SAT and demonstrated its effectiveness.

Lamport and colleagues have specified the Alpha and Itanium memory models in TLA+ [12] [13]. These specifications build visibility order inductively and support the execution of litmus tests. While their approach also precisely specifies the ordering requirement, the manner in which such inductive definitions

---

2 FD-Prolog refers to Prolog with a finite domain (FD) constraint solver. For example, SICStus Prolog and GNU Prolog have this feature.
are constructed will vary from memory model to memory model, making comparisons among them harder. Our method instead relies on primitive relations and directly describes the components to make up a full memory model. This makes our specification easier to understand, and more importantly, to compare against other memory models. This also means we can enable or disable some sub-rules quite reliably without affecting the other primitive ordering rules - a danger in a style which merges all the ordering concerns in a monolithic manner.

**Roadmap.** In the next section, we introduce our methodology. Section 3 describes the Itanium memory ordering rules. Section 4 demonstrates the analysis of the Itanium memory model through execution. We conclude and propose future works in Section 5. The concise specification of the Itanium ordering constraints is provided in the Appendix, with additional details appearing at our web site [http://www.cs.utah.edu/formal_verification/itanium](http://www.cs.utah.edu/formal_verification/itanium).

2 Overview of the Framework

A pictorial representation of our methodology is shown in Fig. 2. We use a collection of primitive ordering rules, each serving a clear purpose, to specify even the most challenging commercial memory models. This approach mirrors the style adopted in modern declarative specifications written by the industry, such as [2]. Moreover, by using pure logic programs supported by certain modern flavors of Prolog that also include finite domain constraints, one can directly capture these higher order logic specifications and also interactively execute the specifications to obtain execution results for litmus tests. Alternatively, we can obtain SAT instances of the boolean constraints representing the memory model through symbolic execution, in which case boolean satisfiability tools can be employed to quickly answer whether the tests are legal or not.

2.1 Specification Method

To define a memory model, we use predicate calculus to specify all constraints imposed on an ordering relation *order*. The constraints are almost completely
first-order. However, since *order* is a parameter to the specification, the constraints are most easily captured with higher order predicate calculus (we use the HOL logic [14]). Previous non-operational specifications often *implicitly* require general ordering properties, such as totality, transitivity, and circuit-freeness. This is the main reason why such specifications cannot readily be executed. In contrast, we are fully explicit about such properties, and so our constraints completely characterize the memory model.

The flexibility of our notation allows us to specify different memory models under the same framework. We have assembled a large collection of constraints for many conventional memory models, such as Sequential Consistency [8], Coherence, Causal Consistency [9], PRAM [15], and Processor Consistency [16]. Due to space limitation, this paper concentrates on demonstrating how to specify and analyze the Itanium memory ordering rules.

### 2.2 Executing Axiomatic Specifications

A straightforward transcription of the formal predicate calculus specification into a Prolog-style logic program makes it possible for interactive and incremental execution of litmus tests. This encourages exploration and experiment in the development and validation of complex coherence protocols. To make a specification executable, we instantiate it over a finite execution and convert the verification problem to a satisfiability problem.

**The Algorithm.** Given a finite execution *ops* with *n* operations, there are *n*² ordering pairs, constituting an ordering matrix *M*, where the element *M*<sub>ij</sub> indicates whether operations *i* and *j* should be ordered. We go through each ordering rule in the specification and impose the corresponding constraint regarding the elements of *M*. Then we check the satisfiability of all the ordering requirements. If such a *M* exists, the trace *ops* is legal, and a valid interleaving can be derived from *M*. Otherwise, *ops* is not a legal trace.

**Applying Constraint Logic Programming.** Logic programming differs from conventional programming in that it describes the logical structure of the problems rather than prescribing the detailed steps of solving them. This naturally reflects the philosophy of the axiomatic specification style. As a result, our formal specification can be easily encoded using Prolog. Memory ordering constraints can be solved through a conjunction of two mechanisms that FD-Prolog readily provides. One applies backtracking search for all constraints expressed by logical variables, and the other uses non-backtracking constraint solving based on *arc consistency* [17] for FD variables, which is potentially more efficient and certainly more complete (especially under the presence of negation) than with logical variables. This works by adding constraints in a monotonically increasing manner to a constraint store, with the in-built constraint propagation rules of FD-Prolog helping refine the variable ranges (or concluding that the constraints are not satisfiable) when constraints are asserted to the constraint store.
Applying Boolean Satisfiability Techniques. The goal of a boolean satisfiability problem is to determine a satisfying variable assignment for a boolean formula or to conclude that no such assignment exists. A slight variant of the Prolog code can let us benefit from SAT solving techniques, which have advanced tremendously in recent years. Instead of solving constraints using a FD solver, we can let Prolog emit SAT instances through symbolic execution. The resultant formula is true if and only if the litmus test is legal under the memory model. It is then sent to a SAT solver to find out the result.

3 Specifying the Itanium Memory Consistency Model

The original Itanium memory ordering specification is informally given in various places in the Itanium architecture manual [1]. Intel later provided an application note [2] to guide system developers. This document uses a combination of English and informal mathematics to specify a core subset of memory operations in a non-operational style. We demonstrate how the specification of [2] can be adapted to our framework to enable computer aided analysis. Virtually the entire Intel application note has been captured.3 We assume proper address alignment and common address size for all memory accesses, which would be the common case encountered by programmers (even these restrictions could be easily lifted). The detailed definition of the Itanium memory model is presented in the Appendix. This section explains each of the rules. The following definitions are used throughout this paper:

Instructions – Instructions with memory access or memory ordering semantics. Five instruction types are defined in this paper: load-acquire (\texttt{ld.acq}), store-release (\texttt{st.rel}), unordered load (\texttt{ld}), unordered store (\texttt{st}), and memory fence (\texttt{mf}). An instruction \( i \) may have read semantics (\texttt{isRd} \( i = \text{true} \)) or write semantics (\texttt{isWr} \( i = \text{true} \)). \texttt{ld.acq} and \texttt{ld} have read semantics. \texttt{st.rel} and \texttt{st} have write semantics. \texttt{mf} has neither read nor write semantics. Instructions are decomposed into operations to allow a finer specification of the ordering properties.

Execution – Also known as an execution trace, contains all memory operations generated by a program. Stores are annotated with the write data and loads are annotated with the return data. An execution is legal if there exists an order among the operations that satisfies all memory model constraints.

Address Attributes – Every memory location is associated with an address attribute, which can be write-back (WB), uncachable (UC), or write-coalescing (WC). Memory ordering semantics may vary for different attributes. Predicate attribute is used to find the attribute of a location.

3 This paper formally captures 21 out of 24 rules from [2]. The remaining 3 rules deal with semaphore operations, which are straightforward to add using the same approach.
**Operation Tuple** – A tuple containing necessary attributes is used to mathematically describe memory operations. A memory operation \( i \) is represented by a tuple \( \langle P, Pc, Op, Var, Data, WrId, WrType, WrProc, Reg, UseReg, Id \rangle \), where

- \( p_i = P \): issuing processor
- \( pc_i = Pc \): program counter
- \( op_i = Op \): instruction type
- \( var_i = Var \): shared memory location
- \( data_i = Data \): data value
- \( wrID_i = WrId \): identifier of a write operation
- \( wrType_i = WrType \): type of a write operation
- \( wrProc_i = WrProc \): target processor observing a write operation
- \( reg_i = Reg \): register
- \( useReg_i = UseReg \): flag of a write indicating if it uses a register
- \( id_i = Id \): global identifier of the operation

A read instruction or a fence instruction is decomposed into a single operation. A write instruction is decomposed into multiple operations, comprising a local write operation (\( \text{wrType}_i = \text{Local} \)) and a set of remote write operations (\( \text{wrType}_i = \text{Remote} \)) for each target processor (\( \text{wrProc}_i \)), which also includes the issuing processor. Every write operation \( i \) that originates from a single write instruction shares the same program counter (\( pc_i \)) and write ID (\( \text{WrID}_i \)).

### 3.1 The Itanium Memory Ordering Rules

As shown below, predicate \( \text{legal} \) is a top-level constraint that defines the legality of a trace \( ops \) by checking the existence of an \( \text{order} \) among \( ops \) that satisfies all requirements. Each requirement is formally defined in the Appendix.

\[
\text{legal} \; ops \; \equiv \; \exists \; \text{order}.
\]

\[
\begin{align*}
\text{requireLinearOrder} \; & \; ops \; \text{order} \; \wedge \\
\text{requireWriteOperationOrder} \; & \; ops \; \text{order} \; \wedge \\
\text{requireProgramOrder} \; & \; ops \; \text{order} \; \wedge \\
\text{requireMemoryDataDependence} \; & \; ops \; \text{order} \; \wedge \\
\text{requireDataFlowDependence} \; & \; ops \; \text{order} \; \wedge \\
\text{requireCoherence} \; & \; ops \; \text{order} \; \wedge \\
\text{requireReadValue} \; & \; ops \; \text{order} \; \wedge \\
\text{requireAtomicWBRelease} \; & \; ops \; \text{order} \; \wedge \\
\text{requireSequentialUC} \; & \; ops \; \text{order} \; \wedge \\
\text{requireNoUCBypass} \; & \; ops \; \text{order}
\end{align*}
\]

Table 1 illustrates the hierarchy of the Itanium memory model definition. Most constraints strictly follow the rules from [2]. We also explicitly add a predicate \( \text{requireLinearOrder} \) to capture the general ordering requirement since [2] has only English to convey this important ordering property.
Table 1. The specification hierarchy of the Itanium memory ordering rules.

<table>
<thead>
<tr>
<th>requireLinearOrder</th>
<th>requireMemoryDataDependence</th>
<th>requireReadValue</th>
</tr>
</thead>
<tbody>
<tr>
<td>- requireWeakTotal</td>
<td>- MD:RAW</td>
<td>- validWr</td>
</tr>
<tr>
<td>- requireTransitive</td>
<td>- MD:WAR</td>
<td>- validLocalWr</td>
</tr>
<tr>
<td>- requireAsymmetric</td>
<td>- MD:WAW</td>
<td>- validRemoteWr</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- validDefaultWr</td>
</tr>
<tr>
<td>requireWriteOperationOrder</td>
<td>requireDataFlowDependence</td>
<td>requireNoUCBypasss</td>
</tr>
<tr>
<td>- local/remote case</td>
<td>- DF:RAR</td>
<td>- validRd</td>
</tr>
<tr>
<td>- remote/remote case</td>
<td>- DF:RAW</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- DF:WAR</td>
<td></td>
</tr>
<tr>
<td>requireProgramOrder</td>
<td>requireCoherence</td>
<td>requireSequentialUC</td>
</tr>
<tr>
<td>- acquire case</td>
<td>- local/local case</td>
<td>- RAR case</td>
</tr>
<tr>
<td>- release case</td>
<td>- remote/remote case</td>
<td>- RAW case</td>
</tr>
<tr>
<td>- fence case</td>
<td></td>
<td>- WAR case</td>
</tr>
<tr>
<td></td>
<td>requireAtomicWBRelease</td>
<td>- WAW case</td>
</tr>
</tbody>
</table>

General Ordering Requirement (Appendix A.1). This requires order to be a weak total order which is also circuit-free.

Write Operation Order (Appendix A.2). This specifies the ordering among write operations that originate from a single write instruction. It guarantees that no write can become visible remotely before it becomes visible locally.

Program Order (Appendix A.3). This restricts reordering among instructions of the same processor with respect to the program order.

Memory-Data Dependence (Appendix A.4). This restricts reordering among instructions from the same processor when they access common locations.

Data-Flow Dependence (Appendix A.5). This is intended to specify how local data dependency and control dependency should be treated. However, this is an area that is not fully specified in [2]. Instead of pointing to an informal document as done in [2], we provide a formal specification covering most cases of data dependency, namely establishing data dependency between two memory operations by checking the conflict usages of local registers.4

Coherence (Appendix A.6). This constrains the order of writes to a common location. If two writes to the same location with the attribute of WB or UC become visible to a processor in some order, they must become visible to all processors in that order.

4 We do not cover branch instructions or indirect-mode instructions that also induce data dependency. We provide enough data dependency specification to let designers experiment with straight-line code that uses registers - this is an important requirement to support execution.
P1
(1) st_local(a,1); (7) ld.acq(1,b);
(2) st_remote1(a,1); (8) ld(0,a);
(3) st_remote2(a,1);
(4) st.rel_local(b,1);
(5) st.rel_remote1(b,1);
(6) st.rel_remote2(b,1);

P2

Fig. 3. An execution resulted from the program in Fig. 1. Stores are decomposed into local stores and remote stores. Loads are associated with return values.

Read Value (Appendix A.7). This defines what data can be observed by a read operation. There are three scenarios: a read can get the data from a local write (validLocalWr), a remote write (validRemoteWr), or the default value (validDefaultWr). Similar to shared memory read value rules, predicate validRd guarantees consistent assignments of registers - the value of a register is obtained from the most recent previous assignment of the same register.

Total Ordering of WB Releases (Appendix A.8). This specifies that store-releases to write-back (WB) memory must obey remote write atomicity, i.e., they become remotely visible atomically.

Sequentiaility of UC Operations (Appendix A.9). This specifies that operations to uncacheable(UC) memory locations must have the property of sequentiaility, i.e., they must become visible in program order.

No UC Bypassing (Appendix A.10). This specifies that uncacheable(UC) memory does not allow local bypassing from UC writes.

4 Making the Itanium Memory Model Executable

We have developed two methods to analyze the Itanium memory model. The first, as mentioned earlier, uses Prolog backtracking search, augmented with finite-domain constraint solving. The second approach targets the powerful SAT engines that have recently emerged.

The Constraint Logic Programming Approach
Our formal Itanium specification is implemented in SICStus Prolog [18]. Litmus tests are contained in a separate test file. When a test number is selected, the FD constraint solver examines all constraints automatically and answers whether the selected execution is legal. By running the litmus tests we can learn the degree to which executions are constrained, i.e., we can obtain a general view of the global ordering relation between pairs of instructions.
Consider, for example, the program discussed earlier in Fig. 1. Its instructions are decomposed into operations as shown in Fig. 3. After taking this trace as input, the Prolog tool attempts all possible orders until it can find an instantiation that satisfies all constraints. For this particular example, it returns "illegal trace" as the result. If one comments out the requireProgramOrder rule and examines the trace again, the tool quickly finds a legal ordering matrix and a corresponding interleaving as shown in Fig. 4. Many other experiments can be conveniently performed in a similar way. Therefore, not only does this approach give people the notation to write rigorous as well as readable specifications, it also allows users to play with the model, asking "what if" queries after selectively enabling/disabling the ordering rules that are crucial to their work. We can also use the built-in predicate setof provided by Prolog to collect all legal return values for read operations. This is achieved by repeatedly backtracking and gradually building up a list of the solutions.

Although translating the formal specification to Prolog is fairly straightforward, there does exist some "logic gap" between predicate calculus and Prolog. Most Prolog systems do not directly support quantifiers. Therefore, we need to implement the effect of a universal quantifier by enumerating the related finite domain. The existential quantifier is realized by the backtracking mechanism of Prolog when proper predicate conditions are set.

### The SAT Approach

As an alternative method, we use our Prolog program as a driver to emit propositional formulae asserting the solvability of the constraints. After being converted to the DIMACS format, the final formula is sent to a SAT solver, such as ZChaff [19] or berkmin [20]. Although the clause generation phase can be detached from the logic programming approach, the ability to have it coexist with FD-Prolog might be advantageous since it allows the two methods to share the same specification base.

### Performance Results

Performance statistics from some litmus tests is shown below. These tests are chosen from [2] and represented by their original table numbers. Results are

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

**Fig. 4.** A legal ordering matrix for the execution shown in Fig. 3 when requireProgramOrder is disabled. A value 1 indicates that the two operations are ordered. A possible interleaving 8 4 5 6 7 1 2 3 is also automatically derived from this matrix.
measured on a Pentium III 900 MHz machine with 256 MB of RAM running Windows 2000. SICStus Prolog is run under compiled mode. The SAT solver used is ZChaff.

<table>
<thead>
<tr>
<th>Test</th>
<th>Result</th>
<th>FD Solver(sec)</th>
<th>Vars</th>
<th>Clauses</th>
<th>SAT(sec)</th>
<th>CNF Gen (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>[2, Table 5]</td>
<td>illegal</td>
<td>0.49</td>
<td>64</td>
<td>679</td>
<td>0.01</td>
<td>3.67</td>
</tr>
<tr>
<td>[2, Table 10]</td>
<td>legal</td>
<td>3.00</td>
<td>100</td>
<td>1280</td>
<td>0.01</td>
<td>8.23</td>
</tr>
<tr>
<td>[2, Table 15]</td>
<td>illegal</td>
<td>22.29</td>
<td>576</td>
<td>15706</td>
<td>0.01</td>
<td>211.76</td>
</tr>
<tr>
<td>[2, Table 18]</td>
<td>illegal</td>
<td>2.40</td>
<td>144</td>
<td>2125</td>
<td>0.01</td>
<td>15.75</td>
</tr>
<tr>
<td>[2, Table 19]</td>
<td>legal</td>
<td>4.81</td>
<td>144</td>
<td>2044</td>
<td>0.01</td>
<td>15.68</td>
</tr>
</tbody>
</table>

Although satisfiability problems are NP-complete, the performance in practice has been acceptable. For the method using SAT solvers, the clause generation time is noticeably larger than the actual SAT solving time, since the entire formula is encoded at once through symbolic execution and is recursively simplified afterwards. Alternative boolean formula encoding techniques, such as the one discussed in [21], may help speed up this process.

5 Conclusions

The setting in which contemporary memory models are expressed and analyzed needs to be improved. Towards this, we present a framework based on axiomatic specifications (expressed in higher order logic) of memory ordering requirements. It is straightforward to encode these requirements as constraint logic programs or, by an extra level of translation, as boolean satisfiability problems. Our techniques are demonstrated through the adaptation and analysis of the Itanium memory model. Being able to tackle such a complex design also attests to the scalability of our framework for cutting-edge commercial architectures.

Our methodology provides several benefits. First, the ability to execute the underlying model is a powerful feature that promotes understanding. Second, the compositional specification style provides modularity, reusability, and scalability. It also allows one to change constraints incrementally for investigation purposes. Third, the expressive power of the underlying logic allows one to define a wide range of requirements using the same notation, providing a rich taxonomy for memory consistency models. Finally, the method of converting axiomatic rules to a propositional formula allows one to perform property checking through boolean reasoning, thus opening up a new means to conduct memory model verification.

Future Work. One possible enhancement is to develop the capability of exercising symbolic (non-ground) litmus tests. Such a tool may be used to automatically synthesize critical instructions of concurrent code fragments comprising compiler idioms or other synchronization primitives. For example, one could imagine using a symbolic store instruction in a program and asking a tool to solve whether it should be an “ordinary” or a “release” store to help generate aggressive code. Another area of improvement is in reducing the logic gap between the formal specification and the tools that execute the specification. One possibility is to
apply a quantified boolean formulae (QBF) solver that directly accepts quantifiers. The research of QBF solvers is still at a preliminary stage compared to propositional SAT. We hope our work can help accelerate its development by providing industrially motivated benchmarks.

References

5. Prosenjit Chatterjee, Ganesh Gopalakrishnan: Towards a Formal Model of Shared Memory Consistency for Intel Itanium. ICCD 2001, Austin, TX (Sept 2001)
10. Yuan Yu, through personal communication
15. R. J. Lipton, J. S. Sandberg: PRAM: A Scalable Shared Memory. CS-TR-180-88
18. SICStus Prolog, http://www.sics.se/sicstus
Appendix: Formal Itanium Memory Ordering Specification

legal \( ops \equiv \exists \) order.
- requireLinearOrder \( ops \) order \( \wedge \) requireWriteOperationOrder \( ops \) order \( \wedge \)
- requireProgramOrder \( ops \) order \( \wedge \)
- requireMemoryDataDependence \( ops \) order \( \wedge \)
- requireDataFlowDependence \( ops \) order \( \wedge \) requireCoherence \( ops \) order \( \wedge \)
- requireReadValue \( ops \) order \( \wedge \) requireAtomicWBRRelease \( ops \) order \( \wedge \)
- requireSequentialUC \( ops \) order \( \wedge \) requireNoUCBypass \( ops \) order

A.1 General Ordering Requirement

requireLinearOrder \( ops \) order \( \equiv \)
- requireWeakTotal \( ops \) order \( \wedge \) requireTransitive \( ops \) order \( \wedge \)
- requireAsymmetric \( ops \) order

requireWeakTotal \( ops \) order \( \equiv \forall i, j \in \text{ops}. \text{id} i \neq j \Rightarrow (\text{order} i j \vee \text{order} j i) \)

requireTransitive \( ops \) order \( \equiv \forall i, j, k \in \text{ops}. (\text{order} i j \wedge \text{order} j k) \Rightarrow \text{order} i k \)

requireAsymmetric \( ops \) order \( \equiv \forall i, j \in \text{ops}. \text{order} i j \Rightarrow \neg (\text{order} j i) \)

A.2 Write Operation Order

requireWriteOperationOrder \( ops \) order \( \equiv \forall i, j \in \text{ops}. \)
- orderedByWriteOperation \( i j \Rightarrow \text{order} i j \)

orderedByWriteOperation \( i j \equiv \text{isWr} i \wedge \text{isWr} j \wedge \text{wrID} i = \text{wrID} j \wedge \)
- (\text{wrType} i = \text{Local} \wedge \text{wrType} j = \text{Remote} \wedge \text{wrProc} j = p i \vee \)
- \text{wrType} i = \text{Remote} \wedge \text{wrType} j = \text{Remote} \wedge \)
- \text{wrProc} i = p i \wedge \text{wrProc} j \neq p i)

A.3 Program Order

requireProgramOrder \( ops \) order \( \equiv \forall i, j \in \text{ops}. \)
- (\text{orderedByAcquire} i j \vee \text{orderedByRelease} i j \vee \text{orderedByFence} i j) \Rightarrow \)
- \text{order} i j

orderedByProgram \( i j \equiv p i = p j \wedge pc i < pc j \)

orderedByAcquire \( i j \equiv \text{orderedByProgram} i j \wedge \text{op} i = \text{ld.acq} \)

orderedByRelease \( i j \equiv \text{orderedByProgram} i j \wedge \text{op} j = \text{st.rel} \wedge \)
- (\text{isWr} i \Rightarrow (\text{wrType} i = \text{Local} \wedge \text{wrType} j = \text{Local} \vee \)
- \text{wrType} i = \text{Remote} \wedge \text{wrType} j = \text{Remote} \wedge \text{wrProc} i = \text{wrProc} j)) \)

orderedByFence \( i j \equiv \text{orderedByProgram} i j \wedge (\text{op} i = \text{mf} \vee \text{op} j = \text{mf}) \)
A.4 Memory-Data Dependence

\text{requireMemoryDataDependence \ ops \ order} \equiv \forall \ i, j \in \ ops.
\text{(orderedByRAW \ i \ j \lor \ orderedByWAR \ i \ j \lor \ orderedByWAW \ i \ j)} \Rightarrow
\text{order \ i \ j}

\text{orderedByMemoryData \ i \ j} \equiv \text{orderedByProgram \ i \ j} \land \ \text{var \ i} = \text{var \ j}

\text{orderedByRAW \ i \ j} \equiv
\text{orderedByMemoryData \ i \ j} \land \ \text{isWr \ i} \land \ \text{wrType \ i} = \text{Local} \land \ \text{isRd \ j}

\text{orderedByWAR \ i \ j} \equiv
\text{orderedByMemoryData \ i \ j} \land \ \text{isRd \ i} \land \ \text{isWr \ j} \land \ \text{wrType \ j} = \text{Local}

\text{orderedByWAW \ i \ j} \equiv \text{orderedByMemoryData \ i \ j} \land \ \text{isWr \ i} \land \ \text{isWr \ j}
\text{(wrType \ i} = \text{Local} \land \ \text{wrType \ j} = \text{Local} \lor
\text{wrType \ i} = \text{Remote} \land \ \text{wrType \ j} = \text{Remote} \land
\text{wrProc \ i} = \text{p \ i} \land \ \text{wrProc \ j} = \text{p \ i})

A.5 Data-Flow Dependence

\text{requireDataFlowDependence \ ops \ order} \equiv \forall \ i, j \in \ ops.
\text{orderedByLocalDepencence \ i \ j} \Rightarrow \text{order \ i \ j}

\text{orderedByLocalDepencence \ i \ j} \equiv \text{orderedByProgram \ i \ j} \land \ \text{reg \ i} = \text{reg \ j} \land
\text{(isRd \ i} \land \ \text{isRd \ j} \lor
\text{isWr \ i} \land \ \text{wrType \ i} = \text{Local} \land \ \text{useReg \ i} \land \ \text{isRd \ j} \lor
\text{isRd \ i} \land \ \text{isWr \ j} \land \ \text{wrType \ j} = \text{Local} \land \ \text{useReg \ j})

A.6 Coherence

\text{requireCoherence \ ops \ order} \equiv \forall \ i, j \in \ ops.
\text{(isWr \ i} \land \ \text{isWr \ j} \land \ \text{var \ i} = \text{var \ j} \land
\text{(attribute \ (var \ i)} = \text{WB} \lor \text{attribute \ (var \ i)} = \text{UC}) \land
\text{(wrType \ i} = \text{Local} \land \ \text{wrType \ j} = \text{Local} \land \ \text{p \ i} = \text{p \ j} \lor
\text{wrType \ i} = \text{Remote} \land \ \text{wrType \ j} = \text{Remote} \land \ \text{wrProc \ i} = \text{wrProc \ j}) \land
\text{order \ i \ j})
\Rightarrow
\forall \ p, q \in \ ops.
\text{(isWr \ p \land \ isWr \ q \land \ wrID \ p} = \text{wrID \ i} \land \ \text{wrID \ q} = \text{wrID \ j} \land
\text{wrType \ p} = \text{Remote} \land \ \text{wrType \ q} = \text{Remote} \land \ \text{wrProc \ p} = \text{wrProc \ q}) \Rightarrow
\text{order \ p \ q}

A.7 Read Value

\text{requireReadValue \ ops \ order} \equiv \forall \ j \in \ ops.
\text{(isRd \ j} \Rightarrow \text{(validLocalWr \ ops \ order \ j} \lor \text{validRemoteWr \ ops \ order \ j} \lor
\text{validDefaultWr \ ops \ order \ j)}) \land \text{((isWr \ j \land \ useReg \ j) \Rightarrow \text{validRd \ ops \ order \ j})}
validLocalWr $\text{ops order } j \equiv \exists i \in \text{ops.}$

$$\text{(isWr } i \land \text{wrType } i = \text{Local } \land \text{var } i = \text{var } j \land p \text{ } i = p \text{ } j \land$$
$$\text{data } i = \text{data } j \land \text{order } i \text{ } j \land$$
$$\neg \exists k \in \text{ops. isWr } k \land \text{wrType } k = \text{Local } \land \text{var } k = \text{var } j \land p \text{ } k = p \text{ } j \land$$
$$\text{order } i \text{ } k \land \text{order } k \text{ } j)$$

validRemoteWr $\text{ops order } j \equiv \exists i \in \text{ops.}$

$$\text{(isWr } i \land \text{wrType } i = \text{Remote } \land \text{wrProc } i = p \text{ } j \land \text{var } i = \text{var } j \land$$
$$\text{data } j = \text{data } i \land \neg (\text{order } j \text{ } i) \land$$
$$\neg \exists k \in \text{ops. isWr } k \land \text{wrType } k = \text{Remote } \land \text{var } k = \text{var } j \land \text{wrProc } k = p \text{ } j \land$$
$$\text{order } i \text{ } k \land \text{order } k \text{ } j)$$

validDefaultWr $\text{ops order } j \equiv$

$$\neg \exists i \in \text{ops. isWr } i \land \text{var } i = \text{var } j \land \text{order } i \text{ } j \land$$
$$\text{(wrType } i = \text{Local } \land p \text{ } i = p \text{ } j \lor \text{wrType } i = \text{Remote } \land \text{wrProc } i = p \text{ } j)) \Rightarrow$$
$$\text{data } j = \text{default } (\text{var } j)$$

validRd $\text{ops order } j \equiv \exists i \in \text{ops.}$

$$\text{(isRd } i \land \text{reg } i = \text{reg } j \land \text{orderedByProgram } i \text{ } j \land \text{data } j = \text{data } i) \land$$
$$\neg \exists k \in \text{ops. isRd } k \land \text{reg } k = \text{reg } j \land$$
$$\text{orderedByProgram } i \text{ } k \land \text{orderedByProgram } k \text{ } j)$$

A.8 Total Ordering of WB Releases

requireAtomicWBRelease $\text{ops order } \equiv \forall i, j, k \in \text{ops.}$

$$\text{(op } i = \text{st.rel } \land \text{wrType } i = \text{Remote } \land \text{op } k = \text{st.rel } \land \text{wrType } k = \text{Remote } \land$$
$$\text{wrID } i = \text{wrID } k \land \text{attribute } (\text{var } i) = \text{WB } \land \text{order } i \text{ } j \land \text{order } j \text{ } k) \Rightarrow$$
$$\text{(op } j = \text{st.rel } \land \text{wrType } j = \text{Remote } \land \text{wrID } j = \text{wrID } i)$$

A.9 Sequentiality of UC Operations

requireSequentialUC $\text{ops order } \equiv \forall i, j \in \text{ops. orderedByUC } i \text{ } j \Rightarrow \text{order } i \text{ } j$

orderedByUC $i \text{ } j \equiv$

$$\text{orderedByProgram } i \text{ } j \land \text{attribute } (\text{var } i) = \text{UC } \land \text{attribute } (\text{var } j) = \text{UC } \land$$
$$\text{isRd } i \land \text{isRd } j \lor$$
$$\text{isRd } i \land \text{isWr } j \land \text{wrType } j = \text{Local } \lor$$
$$\text{isWr } i \land \text{wrType } i = \text{Local } \land \text{isRd } j \lor$$
$$\text{isWr } i \land \text{wrType } i = \text{Local } \land \text{isWr } j \land \text{wrType } j = \text{Local})$$

A.10 No UC Bypassing

requireNoUCBypass $\text{ops order } \equiv \forall i, j, k \in \text{ops.}$

$$\text{(isWr } i \land \text{wrType } i = \text{Local } \land \text{attribute } (\text{var } i) = \text{UC } \land \text{isRd } j \land$$
$$\text{isWr } k \land \text{wrType } k = \text{Remote } \land \text{wrProc } k = p \text{ } k \land \text{wrID } k = \text{wrID } i \land$$
$$\text{order } i \text{ } j \land \text{order } j \text{ } k) \Rightarrow$$
$$\text{(wrProc } k \neq p \text{ } j \lor \text{var } i \neq \text{var } j)$$
On Complementing Nondeterministic Büchi Automata

Sankar Gurumurthy\textsuperscript{1}, Orna Kupferman\textsuperscript{2\star}, Fabio Somenzi\textsuperscript{1\star\star}, and Moshe Y. Vardi\textsuperscript{3\star\star\star}

\textsuperscript{1} University of Colorado at Boulder  
\textsuperscript{2} Hebrew University  
\textsuperscript{3} Rice University

\textbf{Abstract.} Several optimal algorithms have been proposed for the complementation of nondeterministic Büchi word automata. Due to the intricacy of the problem and the exponential blow-up that complementation involves, these algorithms have never been used in practice, even though an effective complementation construction would be of significant practical value. Recently, Kupferman and Vardi described a complementation algorithm that goes through weak alternating automata and that seems simpler than previous algorithms. We combine their algorithm with known and new minimization techniques. Our approach is based on optimizations of both the intermediate weak alternating automaton and the final nondeterministic automaton, and involves techniques of rank and height reductions, as well as direct and fair simulation.

1 Introduction

Efforts for developing simple complementation algorithms for nondeterministic Büchi automata started early in the 60s, motivated by decision problems of second order logics. In [5], Büchi suggested a complementation construction that involved a complicated combinatorial argument and a doubly-exponential blow-up in the state space. Thus, complementing an automaton with \( n \) states resulted in an automaton with \( 2^{O(n)} \) states. In [22], Sistla, Vardi, and Wolper suggested an improved construction, with \( 2^{O(n^2)} \) states. Only in [20], however, Safra introduced an optimal determinization construction, which also enabled a \( 2^{O(n \log n)} \) complementation construction, matching the known lower bound [18]. Another \( 2^{O(n \log n)} \) construction was suggested by Klarlund in [10], which circumvented the need for determinization. While being the heart of many complexity results in verification, the constructions in [20,10] are complicated and difficult to program. We know of no implementation of Klarlund’s algorithm, and the implementation of Safra’s algorithm [24] has to cope with the involved structure of the states in the complementary automaton.

The lack of a simple implementation is not due to a lack of need. In the automata-theoretic approach to verification, we check correctness of a system with respect to a specification by checking containment of the language of the system in the language of

\textsuperscript{\star} Supported in part by NSF grant CCR-9988172.  
\textsuperscript{\star\star} Supported in part by SRC contract 2002-TJ-920 and NSF grant CCR-99-71195.  
\textsuperscript{\star\star\star} Supported in part by NSF grants CCR-9988322, CCR-0124077, CCR-0311326, IIS-9908435, IIS-9978135, and EIA-0086264, and by a grant from the Intel Corporation.
an automaton that accepts exactly all computations that satisfy the specification. In order to check the latter, we check that the intersection of the system with an automaton that accepts exactly all the computations that violate the specification is empty. For instance, LTL model checking [15,25] usually proceeds by translating the negation of an LTL formula into a Büchi automaton. When properties are specified by \( \omega \)-regular automata, one needs to complement the property automaton. Due to the lack of a simple complementation construction, the user is typically required to specify the property by deterministic Büchi automata [14] (it is easy to complement a deterministic automaton), or to supply the automaton for the negation of the property [9]. Similarly, specification formalisms like ETL [26], which have automata within the logic, involve complementation of automata, and the difficulty of complementing Büchi automata is an obstacle to practical use [3]. In fact, even when the properties are specified in LTL, complementation is useful: the translators from LTL into automata have reached a remarkable level of sophistication (cf. [23,8]). Even though complementation of the automata is not explicitly required, the translations are so involved that it is useful to checks their correctness, which involves complementation\(^1\). Complementation is interesting in practice also because it enables refinement and optimization techniques that are based on language containment rather than simulation. Thus, an effective algorithm for the complementation of Büchi automata would be of significant practical value.

In [12], Kupferman and Vardi describe a complementation procedure that is simpler than those in [20,10]. The key idea of [12] is to go via weak alternating automata. In an alternating automaton [6], both existential and universal branching modes are allowed, and the transitions are given as Boolean formulas over the set of states. For example, a transition \( \delta(q, \sigma) = q_1 \lor (q_2 \land q_3) \) means that when the automaton is in state \( q \) and it reads a letter \( \sigma \), it should accept the suffix of the word either from state \( q_1 \) or from both states \( q_2 \) and \( q_3 \). Let \( \alpha \) be the set of the automaton’s accepting states. In a weak automaton, each strongly connected component of the graph induced by the transition function is either accepting (trivial, or contained in \( \alpha \)) or rejecting (its intersection with \( \alpha \) is empty). Since the strongly connected components are partially ordered, each path in the run eventually gets trapped in one of them. The run is accepting if all paths get trapped in accepting components. The height of a weak automaton is the maximal number of alternations between accepting and rejecting components in a path in the graph of the automaton, plus one.

The rich structure of alternating automata makes their complementation trivial—one only has to dualize the transition function and the acceptance condition. Removing alternation from Büchi automata involves a simple extension of the subset construction [19]. Unfortunately, by dualizing the given nondeterministic Büchi automaton, one gets a universal co-Büchi automaton, creating a gap in the construction. This gap is closed in [12], whose complementation construction consists of the following steps.

1. Dualize the given nondeterministic Büchi automaton \( B \), and obtain a universal co-Büchi automaton \( C \) for the complement language. This step is trivial and involves no blow up.

\(^1\) For an LTL formula \( \psi \), one typically checks that both the intersection of \( A_\psi \) with \( A_{\neg \psi} \) and the intersection of their complementary automata are empty.
(2) Translate $C$ to an alternating weak automaton $W$ accepting the same language. If $C$ has $n$ states, then $W$ has $O(n^2)$ states.

(3) Translate $W$ to a nondeterministic Büchi automaton $M$. This step follows the exponential subset construction of [19]. The state space of $M$ can be restricted to consistent subsets\(^2\), making the overall blow up $2^{O(n \log n)}$ rather than $2^{O(n^2)}$.

In this paper we study and describe an arsenal of optimization techniques that can be applied in the last two steps. The techniques can be partitioned into the following classes.

**Rank Reduction.** The translation in Step (2) is based on an analysis of the accepting runs of $C$. Each vertex of the run is associated with a *rank* in the range $\{0, \ldots, 2n\}$. Like the progress measure of [10], the rank of a vertex indicates how easy it is to prove that all the paths that start at the vertex visit $\alpha$ only finitely often. The rank of a universal co-Büchi automaton $C$ is the maximal rank of a vertex in an accepting run of $C$.

If the state space of $C$ is $Q$ and its rank is $k$, the state space of $W$ can be restricted to $Q \times \{0, \ldots, k\}$. Hence, finding and/or reducing the rank of $C$ is desirable. We study ranks of languages, namely the minimal rank of a universal co-Büchi automaton that recognizes their complement. We show that, surprisingly, the rank of all $\omega$-regular languages is 3 (a nice corollary, also proved in [16], is that all $\omega$-regular languages can be recognized by an alternating weak automaton of height 3). Reducing the rank to 3, however, has a flavor of determinization, and involves an exponential blow-up in the state space. Accordingly, we prefer the approach of finding the rank $k$ of $C$. We show that the rank of $C$ is bounded by $2(n - |\alpha|)$, and that there are automata for which this bound is tight. As suggested in [12], the rank is often smaller. We find the rank by checking for language equivalence between $W$ and its restrictions to $Q \times \{0, \ldots, j\}$, for $j < 2(n - |\alpha|)$.

**Minimization of $W$.** Once we found the rank $k$ of $C$ and restrict the state space of $W$ accordingly, we minimize $W$ further. The transition function of $W$ as described in [12] is of size $|\delta||k|^2$, where $\delta$ is the transition function of $C$. It is suggested in [12] to simplify it and obtain a function of size $3|\delta||k|$. We simplify it further to $2|\delta||k|$. The simplification is based on simulation minimization we apply to $W$, and which often reduces the state space and the transitions even more. Our simulation relation is similar to the alternating simulation of [2], extended to automata with acceptance conditions on the states (direct simulation) as well as an extension of it in which acceptance conditions are moved to the arcs. Finally, we reduce the height of $W$ by repeatedly removing its minimal strongly connected component, as long as such a removal does not change its language.

**Minimization of $M$.** Once $M$ is produced by the subset construction, we apply further simplification techniques to it. The first is the fair simulation minimization of [8], and the second is similar to the height reduction described for $W$, performed on the strongly connected components of $M$. We note that the same reductions are applied also to the nondeterministic Büchi automaton $B$ with which we start.

As shown in [18], complementation of a nondeterministic Büchi automaton with $n$ states may involve a $2^{O(n \log n)}$ blow up. Accordingly, we measure the efficiency of

\(^2\) We describe the consistency condition in Section 3.
our optimizations by the following two criteria: (1) we would like the result of complementing a nondeterministic Büchi automaton derived from an LTL formula to be comparable with what we get by negating the formula and then translating to a nondeterministic Büchi automaton. (2) we would like the result of complementing a nondeterministic Büchi automaton twice to be comparable with the original automaton. We demonstrate the effectiveness of our construction by examining several examples for which our construction produces the minimal nondeterministic Büchi automaton. We have implemented our procedure as an extension of the Wring translator from LTL to Büchi automata [23, 8], and our experimental results are reported in Section 7.

2 Preliminaries

Let $B^+(Q)$ denote the set of positive Boolean formulas over $Q$. An alternating automaton on infinite words $A = \langle \Sigma, Q, q_{in}, \delta, \alpha \rangle$ consists of a finite alphabet $\Sigma$, a finite set of states $Q$, an initial state $q_{in} \in Q$, a transition function $\delta : Q \times \Sigma \rightarrow B^+(Q)$, and an acceptance condition $\alpha \subseteq Q$. For $A = \langle \Sigma, Q, q_{in}, \delta, \alpha \rangle$ and $q \in Q$, let $A^q = \langle \Sigma, Q, q, \delta, \alpha \rangle$. That is, $A^q$ is obtained from $A$ by making $q$ the initial state.

A run of an alternating automaton $A$ on a word $\sigma \in \Sigma^\omega$ is a $Q$-labeled tree $\langle T_r, r \rangle$, where $T_r$ is a prefix-closed subset of $\mathbb{N}^*$, and $r : T_r \rightarrow Q$ is a labeling function. A run of $A$ on $\sigma = \sigma_0, \sigma_1, \ldots$ satisfies the following conditions: (1) $r(\varepsilon) = q_{in}$. (2) For a tree node $t \in T_r$ such that $r(t) = q$ and $\delta(q, \sigma_i) = \beta$, there is a subset $Q_t \subseteq Q$ that satisfies $\beta$, and such that the successors of $t$ are labeled by the elements of $Q_t$.

A run is accepting if all its infinite paths satisfy the acceptance condition. In a Büchi automaton, a path satisfies $\alpha$ if it intersects $\alpha$ infinitely often. In a co-Büchi automaton, a path satisfies $\alpha$ if it intersects it finitely many times. A word $w \in \Sigma^\omega$ is accepted by $A$ if $A$ has an accepting run on $w$. The words accepted by $A$ form the language of $A$, denoted by $L(A)$.

Complementation of an alternating automaton is accomplished by dualizing its transition function, and changing the acceptance condition from Büchi to co-Büchi or vice versa. Dualization consists of exchanging $\land$ with $\lor$, and true with false in $\delta$.

A positive Boolean formula has a unique minimal DNF. Therefore $\delta(q, \sigma_i) \in B^+(Q)$ identifies a set of sets of states $\Delta(q, \sigma_i) \subseteq 2^Q$. For instance, if $\delta(q_0, \sigma_0) = (q_1 \land (q_2 \lor q_3)) \lor (q_1 \land q_2 \land q_4)$, then $\Delta(q_0, \sigma_0) = \{\{q_1, q_2\}, \{q_1, q_3\}\}$. The Boolean formulas true and false translate into $\{\emptyset\}$ and $\emptyset$, respectively. The choice of $Q_t \subseteq Q$ required by the definition of run can always be restricted so that $Q_t \in \Delta(q, \sigma_i)$.

Nondeterministic automata are alternating automata in which each $C \in \Delta(q, l)$ is a singleton for every $q \in Q$ and $l \in \Sigma$. Universal automata are alternating automata in which $\Delta(q, l)$ is a singleton for every $q \in Q$ and $l \in \Sigma$. Deterministic automata are at the same time nondeterministic and universal.

A maximal strongly connected component (MSCC) of a directed graph is a maximal subgraph such that each node in the subgraph has a path to every node in the subgraph. A trivial MSCC contains one node and no arcs. We assume that all the trivial MSCCs of an automaton are contained in $\alpha$. A weak alternating automaton is such that each MSCC of its transition graph is either disjoint from $\alpha$ or contained in it. From a weak alternating
automaton with co-Büchi acceptance $A$ one obtains a weak alternating automaton with Büchi acceptance $A'$ such that $L(A') = L(A)$ simply by taking $\alpha' = Q \setminus \alpha$.

We use three-letter abbreviations to designate types of automata: The first letter characterizes the transition structure and is one of “D” (deterministic), “N” (nondeterministic), “U” (universal), and “A” (alternating). The second letter identifies the acceptance condition and is one of “B” (Büchi), “C” (co-Büchi), and “W” (weak). Finally, the third letter designates the objects accepted by the automata; in this paper we are only concerned with “W” (infinite words). Hence, NBW designates a nondeterministic Büchi automaton, UCW designates a universal co-Büchi automaton, and AWW designates a weak alternating automaton, all on infinite words.

## 3 Ranks and Complementation

In this section we review the relevant technical details of [12]. Consider a UCW $A = \langle \Sigma, Q, q_{in}, \delta, \alpha \rangle$ obtained by dualizing NBW $B$, and a word $w$. Let $|Q| = n$. The run of $A$ on $w$ can be represented by a directed acyclic graph (DAG) $G_r = \langle V, E \rangle$, where

- $V \subseteq Q \times \mathbb{N}$ is such that $\langle q, l \rangle \in V$ iff there exists $x \in T_r$ with $|x| = l$ and $r(x) = q$.
- $E \subseteq \bigcup_{l \geq 0} (Q \times \{l\}) \times (Q \times \{l + 1\})$ is such that $E(\langle q, l \rangle, \langle q', l' \rangle)$ iff there exists $x \in T_r$ with $|x| = l$, $r(x) = q$, and $r(x \cdot c) = q'$ for some $c \in \mathbb{N}$.

We say that a vertex $\langle q', l' \rangle$ is a successor of a vertex $\langle q, l \rangle$ iff $E(\langle q, l \rangle, \langle q', l' \rangle)$. We say that $\langle q', l' \rangle$ is reachable from $\langle q, l \rangle$ iff there exists a sequence $\langle q_0, l_0 \rangle, \ldots, \langle q_i, l_i \rangle$ of successive vertices such that $\langle q, l \rangle = \langle q_0, l_0 \rangle$, and $\langle q', l' \rangle = \langle q_i, l_i \rangle$. Finally, we say that a vertex $\langle q, l \rangle$ is an $\alpha$-vertex iff $q \in \alpha$. It is easy to see that $\langle T_r, r \rangle$ is accepting iff all paths in $G_r$ have only finitely many $\alpha$-vertices.

Consider a (possibly finite) DAG $G \subseteq G_r$. We say that a vertex $\langle q, l \rangle$ is finite in $G$ iff only finitely many vertices in $G$ are reachable from $\langle q, l \rangle$. We say that a vertex $\langle q, i \rangle$ is $\alpha$-free in $G$ iff all the vertices in $G$ that are reachable from $\langle q, l \rangle$ are not $\alpha$-vertices. Finally, we say that the width of $G$ is $k$ if $k$ is the maximal number for which there are infinitely many levels $l$ such that there are $k$ vertices of the form $\langle q, l \rangle$ in $G$. Note that the width of $G_r$ is at most $n$. Given an accepting run DAG $G_r$, we define an infinite sequence $G_0 \supseteq G_1 \supseteq G_2 \supseteq \ldots$ of DAGs inductively as follows.

- $G_0 = G_r$.
- $G_{2i+1} = G_{2i} \setminus \{\langle q, l \rangle \mid \langle q, l \rangle$ is finite in $G_{2i}\}$.
- $G_{2i+2} = G_{2i+1} \setminus \{\langle q, l \rangle \mid \langle q, l \rangle$ is $\alpha$-free in $G_{2i+1}\}$.

It is shown in [12] that for every $i \geq 0$, the transition from $G_{2i+1}$ to $G_{2i+2}$ involves the removal of an infinite path from $G_{2i+1}$. Since the width of $G_0$ is bounded by $n$, it follows that the width of $G_{2i}$ is at most $n - i$. Hence, $G_{2n}$ is finite, and $G_{2n+1}$ is empty.

Each vertex $\langle q, l \rangle$ in $G_r$ has a unique index $i \geq 1$ such that $\langle q, l \rangle$ is either finite in $G_{2i}$ or $\alpha$-free in $G_{2i+1}$. Given a vertex $\langle q, l \rangle$, we define the rank of $\langle q, l \rangle$, denoted $\text{rank}(q, l)$, as follows.

$$
\text{rank}(q, l) = \begin{cases} 
2i & \text{if } \langle q, l \rangle \text{ is finite in } G_{2i}.
\end{cases}
$$

$$
2i + 1 & \text{if } \langle q, l \rangle \text{ is } \alpha\text{-free in } G_{2i+1}.
\end{cases}
$$
For $k \in \mathbb{N}$, let $[k]$ denote the set $\{0, 1, \ldots, k\}$, and let $[k]^{\text{odd}}$ denote the set of odd members of $[k]$. By the above, the rank of every vertex in $G_r$ is in $[2n]$. Recall that when the run is accepting, all the paths in $G_r$ visit only finitely many $\alpha$-vertices. Intuitively, $\text{rank}(q, l)$ hints at how difficult it is to prove that all the paths of $G_r$ that visit the vertex $\langle q, l \rangle$ visit only finitely many $\alpha$-vertices. Easiest to prove are vertices that are finite in $G_0$. Accordingly, they get the minimal rank 0. Then come vertices that are $\alpha$-free in the graph $G_1$. These vertices get the rank 1. The process repeats until all vertices get some rank.

We say that an integer $j$ is a required rank for a UCW $A$ if there exists a word $w \in L(A)$ such that some vertex in the run of $A$ on $w$ gets rank $j$. Then, the rank of $A$ is the maximal rank required for $A$. The annotation of runs with ranks is used in order to translate UCW into AWW:

**Theorem 1.** Let $A$ be a UCW with $n$ states and rank $k$. There is an AWW $A'$ with $n(k+1)$ states such that $L(A') = L(A)$.

**Proof.** Let $A = \langle \Sigma, Q, q_{in}, \delta, \alpha \rangle$. We define $A' = \langle \Sigma, Q', q'_{in}, \delta', \alpha' \rangle$, where

- $Q' = Q \times [k]$. Intuitively, $A'$ is in state $\langle q, i \rangle$, if it guesses that in the accepting run of $A$ on $w$, the rank of $\langle q, l \rangle$ is $i$. An exception is the initial state $q'_{in}$ explained below.
- $q'_{in} = \langle q_{in}, k \rangle$. That is, $q_{in}$ is paired with $k$, which is an upper bound on the rank of $\langle q_{in}, 0 \rangle$.
- We define $\delta'$ by means of a function $\text{release} : B^+(Q) \times [k] \to B^+(Q')$. Given a formula $\theta \in B^+(Q)$, and a rank $i \in [k]$, the formula $\text{release}(\theta, i)$ is obtained from $\theta$ by replacing an atom $q$ by the disjunction $\bigvee_{i \leq i} \langle q, i \rangle$. For example, $\text{release}(q_3 \land q_5, 2) = (\langle q_3, 2 \rangle \lor \langle q_3, 1 \rangle \lor \langle q_3, 0 \rangle) \land (\langle q_5, 2 \rangle \lor \langle q_5, 1 \rangle \lor \langle q_5, 0 \rangle)$.

Now, $\delta' : Q' \times \Sigma \to B^+(Q')$ is defined, for a state $\langle q, i \rangle \in Q'$ and $\sigma \in \Sigma$, as follows.

$$
\delta'((q, i), \sigma) = \begin{cases} 
\text{release}(\delta(q, \sigma), i) & \text{if } q \notin \alpha \text{ or } i \text{ is even.} \\
\text{false} & \text{if } q \in \alpha \text{ and } i \text{ is odd.}
\end{cases}
$$

That is, if the current guessed rank is $i$ then, by employing $\text{release}$, the run can move to its successors at any rank that is smaller than $i$. If, however, $q \in \alpha$ and the current guessed rank is odd, then, by the definition of ranks, the current guessed rank is wrong, and the run is rejecting.

- $\alpha' = Q \times [k]^{\text{odd}}$. That is, infinitely many guessed ranks along a path should be odd.

To see that the automaton $A'$ is weak, note that each set $Q \times \{i\}$ is a collection of strongly connected components that agree on their classification as accepting or rejecting. Indeed, membership in $\alpha'$ depends on the parity of $i$, and the transitions in $\delta'$ are such that from a state in $Q \times \{i\}$ the automaton $A'$ proceeds only to states in $Q \times \{j\}$, for $j \leq i$. 

Once we know how to translate UCW to AWW, complementation is reduced to removal of alternation from ABW (recall that AWW are a special case of ABW). In [19], Miyano and Hayashi describe such a translation. We present (a simplified version of) their translation in Theorem 2 below.
Theorem 2. [19] Let $\mathcal{A}$ be an alternating Büchi automaton. There is a nondeterministic Büchi automaton $\mathcal{A}'$, with exponentially many states, such that $L(\mathcal{A}') = L(\mathcal{A})$.

Proof. The automaton $\mathcal{A}'$ guesses a run of $\mathcal{A}$. At a given point of a run of $\mathcal{A}'$, it keeps in its memory a whole level of the run tree of $\mathcal{A}$. As it reads the next input letter, it guesses the next level of the run tree of $\mathcal{A}$. In order to make sure that every infinite path visits states in $\alpha$ infinitely often, $\mathcal{A}'$ keeps track of states that “owe” a visit to $\alpha$. Let $\mathcal{A} = \langle \Sigma, Q, q_{in}, \delta, \alpha \rangle$. Then $\mathcal{A}' = \langle \Sigma, 2^Q \times 2^Q, \langle \{q_{in}\}, \emptyset \rangle, \delta', 2^Q \times \emptyset \rangle$, where $\delta'$ is defined, for all $\langle S, O \rangle \in 2^Q \times 2^Q$ and $\sigma \in \Sigma$, as follows.

- If $O \neq \emptyset$, then $\delta'(\langle S, O \rangle, \sigma) = \{\langle S', O' \setminus \alpha \rangle \mid S'$ satisfies $\bigwedge_{q \in S} \delta(q, \sigma), O' \subseteq S'$, and $O'$ satisfies $\bigwedge_{q \in O} \delta(q, \sigma)\}.$

- If $O = \emptyset$, then $\delta'(\langle S, O \rangle, \sigma) = \{\langle S', S' \setminus \alpha \rangle \mid S'$ satisfies $\bigwedge_{q \in S} \delta(q, \sigma)\}.$

$\square$

For an NBW $B$, the rank of $B$ is the rank of its dual UCW. Complementing an NBW $B$ with $n$ states and rank $k$, its dual UCW has $n$ states and rank $k$ as well, the AWW $W$ constructed in Theorem 1 has $O(nk)$ states, and the final NBW $M$ constructed in Theorem 2 has $2^O(nk)$ states. By [18,20], however, an optimal complementation construction for nondeterministic Büchi automata results in an automaton with $2^O(n \log n)$ states, which may be smaller. Let $B = \langle \Sigma, Q, q_{in}, \delta, \alpha \rangle$. Consider a state $\langle S, O \rangle$ of $M$. Each of the sets $S$ and $O$ is a subset of $Q \times [k]$. We say that $P \subseteq Q \times [k]$ is consistent iff for every two states $\langle q, i \rangle$ and $\langle q', i' \rangle$ in $P$, if $q = q'$ then $i = i'$. It is shown in [12] that restricting the state space of $M$ to pairs $\langle S, O \rangle$ for which $S$ is a consistent subset of $Q \times [k]$ is allowable; that is, the resulting $M$ still complements $B$. Since there are $2^O(n \log k)$ consistent subsets of $Q \times [k]$, we have the following.

Theorem 3. Let $A$ be an NBW with $n$ states and rank $k$. There is an NBW $\mathcal{A}'$ with $2^O(n \log k)$ states such that $L(\mathcal{A}') = \text{comp}(L(\mathcal{A}))$.

4 Ranks of Automata and Languages

Consider a UCW $A$ with $n$ states and a word $w \in \Sigma^\omega$. Let $G_0, G_1, \ldots, G_{2n+1}$ be the sequence of DAGS constructed in [12] for the run of $A$ on $w$. Recall that the transition from $G_{2i+1}$ to $G_{2i+2}$ involves a removal of an infinite path from $G_{2i+1}$, which is why the width of $G_{2i}$ is at most $n - i$. As noted to us by Doron Bustan, all the vertices in the removed path are not $\alpha$-vertices. Hence, one could argue that the $n - i$ bound on the width of $G_{2i}$ holds also for a tighter definition of width: let the $\alpha$-less width of $G_i$ be the maximal number $k$ for which there are infinitely many levels $l$ such that there are $k$ vertices not in $\alpha$ of the form $\langle q, l \rangle$. With this tighter definition, the $\alpha$-less width of $G_0$ is bounded by $n - |\alpha|$, implying that the $\alpha$-less width of $G_{2i}$ is at most $n - (|\alpha| + i)$. In particular, the $\alpha$-less width of $G_{2(n-|\alpha|)}$ is at most 0. Hence $G_{2(n-|\alpha|)}$ has only finitely many vertices that are not $\alpha$-vertices. Since $G_0$ is accepting, then, by König’s Lemma, $G_{2(n-|\alpha|)}$ also has only finitely many $\alpha$ vertices. It follows that $G_{2(n-|\alpha|)}$ is finite, implying that all vertices get ranks in $0, \ldots, 2(n - |\alpha|)$.

In practice, the transition from $G_{2i}$ to $G_{2i+2}$ often reduces the width by more than one. One may wonder whether it is possible to tighten the analysis above even more in
order to show that a rank of $2(n - |\alpha|)$ is never required. Recall that an integer $j$ is a required rank for $A$ if there exists a word $w \in \mathcal{L}(A)$ such that some vertex in the run of $A$ on $w$ gets rank $j$. Equivalently, the $\alpha$-less width of $G_j$ (with $G_0$ being the run dag of $A$ on $w$) is strictly larger than 0. As follows from Theorems 1 and 3, the rank of $A$ plays an important role in the sizes of equivalent AWW and NBW for it. It is shown in [12] that the problem of finding the rank of a UCW $A$ is PSPACE-complete. By the above, the rank of $A$ is at most $2(n - |\alpha|)$. By the following theorem, there are cases in which this bound is tight.

**Theorem 4.** There is a family $A_1, A_2, \ldots$ of UCW such that $A_n$ has $n + 1$ states, acceptance set of size 1, and rank 2n.

We now turn to study ranks of $\omega$-regular languages. For an $\omega$-regular language $L$, we say that the rank of $L$ is $k$ iff there is a UCW of rank at most $k$ for $\text{comp}(L)$. It is tempting to think that ranks induce an infinite hierarchy $R_0 \subset R_1 \subset \cdots$ of languages, with $R_i$ containing all languages of rank $i$. We show that the hierarchy collapses at $R_3$ (that is, all $\omega$-regular languages have rank at most 3) and characterize its four levels. For a definition of safety and co-safety languages, see [1,21].

**Theorem 5.** $R_3 = \omega$-regular languages, $R_2 = DBW$, $R_1 = co-safety$ languages, and $R_0 = safety$ languages.

The hierarchy induced by ranks is closely related to a hierarchy induced by heights of AWW. Intuitively, the height of an AWW is the number of accepting and rejecting layers it has. Formally, the height of an AWW $A$ is the number of alternations between accepting and rejecting components in the graph of $A$, plus one, where the constants true and false are counted as accepting and rejecting components, respectively. For an integer $k$, let $\text{AWW}[k]$ denote the set of AWW of height at most $k$, or the $\omega$-regular languages accepted by such automata. Theorem 5 implies Theorem 6 below, which was proved first in [16]. Note that Theorem 5 is stronger than Theorem 6 and does not follow from it.

**Theorem 6.** $\text{AWW}[3] = \omega$-regular languages, $\text{AWW}[2] = DBW \cup DCW$, and $\text{AWW}[1] = safety$ or co-safety languages.

The results in this section imply that procedures for rank reduction that modify the given UCW are much stronger than those that calculate its rank. On the other hand, the reduction of the rank to 3 involves determinization, which we are trying to avoid, and which may cause an exponential blow-up. In view of this trade-off between the size of UCW and their ranks, our efforts focus on calculating the rank of the given UCW, rather than on modifying it.

5 Simplifying Alternating Büchi Automata

The construction of Theorem 2 may cause an exponential blow-up. Hence, before applying it, we try to simplify the AWW $\mathcal{W}$ in three ways: by simulation minimization, by computing the rank of the UCW $\mathcal{C}$, and by removing redundant MSCCs.
5.1 Simulation Minimization

We recall that for an ABW $\Delta(q, l)$ is a set of sets. Each member of $\Delta(q, l)$ is a conjunction of states. We define simulation between alternating automata in terms of a game as in [2]. Let $A = (\Sigma, Q_A, q_A, \delta, \alpha)$ and $P = (\Sigma, Q_P, q_P, \delta, \alpha)$ be two ABWs; automaton $P$ simulates automaton $A$ if, given players $P$ and $A$, $P$ has a winning strategy for the following game. The positions of the game are the elements of $Q_A \times Q_P$; the initial position is $(q_A, q_P)$, and the possible successors of a position $(s_A, s_P)$ are all pairs $(t_A, t_P)$ obtained by application of the following rule:

- $A$ chooses a letter $l \in \Sigma$ and a set of states $C_A \in \Delta_A(s_A, l);
- P$ chooses a set of states $C_P \in \Delta_P(s_P, l);
- $A$ chooses $t_P \in C_P$;
- $P$ chooses $t_A \in C_A$.

A player who has to choose from an empty set loses. If this never happens, the play is infinite. The winner of an infinite play depends on whether one considers direct simulation or fair simulation. For direct simulation, $A$ wins iff for some position $(s_A, s_P)$ encountered, $s_A \in \alpha_A$ and $s_P \not\in \alpha_P$. For fair simulation, $A$ wins iff there are infinitely many positions such that $s_A \in \alpha_A$, but only finitely many positions such that $s_P \in \alpha_P$. $P$ wins if $A$ does not. As in the case of NBWs, direct simulation implies fair simulation, and fair simulation implies language containment; the converse is not true [2].

**Theorem 7.** Let $A = (\Sigma, Q, q_in, \delta, \alpha)$ and $A' = (\Sigma, Q', q'_in, \delta', \alpha')$ be two ABWs. If $q_in$ directly simulates $q'_in$, then $q_in$ fair simulates $q'_in$. If $q_in$ fair simulates $q'_in$, then $L(A) \supseteq L(A')$.

If two states $q_1$ and $q_2$ are such that each simulates the other, we say that $q_1$ and $q_2$ are simulation equivalent. Two ABWs are simulation equivalent if their initial states are. Of particular interest to us is the case in which the two automata are $A^{q_1}$ and $A^{q_2}$ for $q_1, q_2 \in Q$; that is, we are interested in the simulation relation on the states of ABW $A$. The “layered” structure of the AWW $W$ implies the existence of a nontrivial simulation relation.

**Theorem 8.** Let $A = (\Sigma, Q, q_in, \delta, \alpha)$ be a UCW with rank $k$; let $A'$ be the equivalent AWW of Theorem 1. Then, for every $(q, j) \in Q \times \{0, \ldots, k\}$ and $i \in \{0, \ldots, j\}$, if $j$ is even or $q \not\in \alpha$, then $(q, j)$ fair simulates $(q, i)$ in $A'$. If in addition $j$ is odd or $i$ is even, then $(q, j)$ direct simulates $(q, i)$.

The simulation of Theorem 8 allows us to improve on [12, Remark 4.2] and reduce the size of the transition relation of $W$ from $3|\delta|k$ to $2|\delta|k$, where $\delta$ is the transition function and $k$ is the rank of the UCW $C$.

**Theorem 9.** If in Theorem 1, release $(\theta, i)$ is redefined so that an atom $q$ is replaced by $(q, i) \lor (q, i - 1)$ if $i > 0$, and by $(q, 0)$ for $i = 0$, then $L(A') = L(A)$.

In general, simulations between states of an ABW can be used to merge states (in case of simulation equivalence), remove transitions, or simplify transitions.\(^3\) The last

\(^3\)This is in contrast to [7], which only considers simulation equivalence quotients. Besides, its model of alternating automata with existential and universal states makes even direct simulation unsafe for minimization.
Theorem 10. Let $A = (\Sigma, Q, q_{in}, \delta, \alpha)$ be an ABW. Let $q_1$ and $q_2$ be two states in $Q$ such that $q_2$ direct simulates $q_1$. Suppose $\{q_1, q_2\} \subseteq C \in \Delta(q, l)$, for some $q \in Q$ and $l \in \Sigma$. Then the automaton $A'$ obtained from $A$ by replacing $C$ in $\Delta(q, l)$ with $C' = C \setminus \{q_2\}$ is direct simulation equivalent to $A$.

Theorem 11. Let $A = (\Sigma, Q, q_{in}, \delta, \alpha)$ be an ABW. Let $C_1, C_2 \in \Delta(q, l)$, for some $q \in Q$, $l \in \Sigma$. Suppose that $C_1 \neq C_2$, and that $\forall q_1 \in C_1$, $\exists q_2 \in C_2$ such that $q_1$ direct simulates $q_2$. Then the automaton $A'$ obtained from $A$ by replacing $\Delta(q, l)$ with $\Delta'(q, l) = \Delta(q, l) \setminus \{C_2\}$ is direct simulation equivalent to $A$.

Two simulation equivalent states $q_1$ and $q_2$ are merged by the following steps: (1) for every letter $l$, $\delta(q_1, l)$ is replaced by $\delta(q_1, l) \lor \delta(q_2, l)$; (2) $q_2$ is replaced by $q_1$ throughout $\delta$; (3) $q_1$ is made initial if $q_2$ is; (4) $q_2$ is dropped.

Corollary 1. Let $A = (\Sigma, Q, q_{in}, \delta, \alpha)$ be an ABW. If two states $q_1, q_2 \in Q$ are direct simulation equivalent, the automaton obtained by merging $q_1$ and $q_2$ is simulation equivalent to $A$.

The computation of the direct simulation relation is based on the following observation [2]. Let $S$ be a simulation relation on the states of an ABW over alphabet $\Sigma$. Then $(u, v) \in S$ implies

$$\forall l \in \Sigma. \forall C \in \Delta(u, l). \exists C' \in \Delta(v, l). \forall v' \in C'. \exists u' \in C. (u', v') \in S.$$ 

We can therefore compute the direct simulation relation as a greatest fixpoint by starting with all the pairs of states $(u, v)$ such that acceptance of $u$ implies acceptance of $v$, and removing pairs that violate the condition above.

5.2 Simulation with Accepting Arcs

The definition of direct simulation given in Section 5.1 assumes that $u \in \alpha$ implies $v \in \alpha$. However, we may compute a larger relation by considering the acceptance conditions to be on the arcs. Let every set of states $C \in \Delta(q, l)$ be a transition out of $q \in Q$ enabled by $l \in \Sigma$. An arc of transition $C$ is the pair $(q, q')$, for some state $q' \in C$. An arc $(q, q')$ is accepting if $q' \in \alpha$. We can modify the definition of direct simulation as follows. Player $A$ wins an infinite play if for some position $(s_A, s_P)$, the arc $(s_A, t_A)$ of $C_A$ is accepting, but the arc $(s_P, t_P)$ is not. Player $P$ wins if $A$ does not.

This approach may lead to simplifications not allowed by the original definition of direct simulation. However, Theorems 10 and 11 do not hold when acceptance conditions are moved to the arcs. Consider an AWW with $\Sigma = \{0\}$, $Q = \{a, b\}$, $q_{in} = a$, $\delta(a, 0) = a \land b$, $\delta(b, 0) = b$, and $\alpha = \{a\}$. Here $b$ direct simulates $a$ when acceptance is on the arcs. In this case the only accepting arc is the self-loop on $a$. However, $\delta(a, 0)$ cannot be simplified to $a$ lest the language changes from empty to $\Sigma^\omega$. To obviate this problem, while computing the direct simulation relation with accepting arcs, we mark all the arcs that are used to justify the relation itself. We then allow simplification of a transition according to Theorem 10 only if the arcs to be removed are not marked.
5.3 Simplification Based on Language Containment

Theorem 8 gives conditions under which \(\langle q, j \rangle\) simulates \(\langle q, i \rangle\) for \(j > i\). However, no such general result can be proved for \(j < i\). To determine the rank of the UCW \(C\) obtained by dualization of the given NBW \(B\), and hence the required height of the AWW \(W\), we resort to a language containment check. Specifically, since the rank is bounded by \(2(n - |\alpha|)\), we apply the construction of Theorem 1 with \(k = 2(n - |\alpha|)\) to build AWW \(W'\) such that \(L(W') = L(C)\). The construction of Theorem 2 applied to \(W'\) yields \(M'\).

To check whether \(k \in \{0, 2, \ldots, 2(n - |\alpha| - 1)\}\) is the rank of \(C\), we restrict \(W'\) to \(Q \times \{0, \ldots, k\}\), make \(\langle q_{in}, k \rangle\) initial, and call the result \(W''\). We then obtain an AWW \(D\) for \(\text{comp}(L(W''))\) by dualization of \(W''\), and apply Theorem 2 to it to produce \(M''\). Since we know that \(L(W'') \subseteq L(W')\), if the intersection of \(M'\) and \(M''\) is empty, then \(k\) is an upper bound on the rank of \(C\). If one tries the possible values of \(k\) in increasing order, the first time the intersection is empty, \(k\) is the rank of \(C\), and \(W = W''\). It is important to note that the restriction to consistent subsets is allowed when converting \(W\) to NBW, but is not allowed when converting \(D\). This makes the determination of the rank a particularly expensive operation. To partially offset this cost, simulation minimization is always applied to \(D\) before the subset construction.

The language-containment approach can be used to further simplify \(W\). Specifically, we try to remove an MSCC from \(W\), and all the transitions with at least one destination state in the chosen MSCC. This guarantees that the language of the resulting automaton is contained in the language of the original one. A single language containment check then suffices to check whether the language remains the same. The MSCCCs are examined in topological order from terminal to initial. If the language does not change, the removal of the MSCC is greedily accepted. We refer to this process as pruning the AWW.

5.4 Simplification Procedure

If the NBW \(B\) is weak, so is the UCW \(C\). Hence, the construction of Theorem 1 is not required, because a UWW is a special case of AWW. Since \(B\) has been minimized, no further simplification of \(W = C\) is attempted. Testing this special case avoids the potentially expensive simplification of \(W\) and makes complementation of NWB efficient. This is practically relevant because many natural specifications induce weak automata [11,4]. (In [17] it is shown that the intersection of ACTL and LTL is UCW[1], which is included in UWW.)

If \(C\) is not weak, first its rank is determined, and \(W\) is built accordingly, simplifying transitions as discussed in Section 5.2, and applying Corollary 1, and Theorems 10–11. The states with index 0 are included only if \(C\) has at least one transition equal to true. (Otherwise, no accepting path can visit them.) Pruning based on language containment (see Section 5.3) is then performed as the last optimization of the AWW before computing the NBW equivalent to \(W\).

If \(B\) is a DBW that is not weak, the resulting AWW is an NWW, and the subset construction does not change it. In such a case, our algorithm behaves like the one of [13]. In some cases, simplification of an AWW also produces an NWW, making the subset construction redundant.
6 Simplification of Nondeterministic Büchi Automata

The complementation algorithm starts and ends with two NBWs, \( B \) and \( M \). It is important to minimize both. For \( B \), every simplification is likely to alleviate the burden for the successive stages of the computation. For \( M \), minimization recovers inefficiencies due, in particular, to the subset construction. In this section we describe how this minimization is carried out. Two procedures are applied to the NBWs \( B \) and \( M \). One is fair simulation minimization [8]. The other is a pruning technique akin to the one described in Section 5.3, but based on checking direct simulation, rather than language containment. Its objective is to reduce the height of the NBW, and it works as follows.

1. Mark all states simulated by an initial state as initial.
2. Process MSCCs that intersect \( \alpha \) in topological order from sources to sinks.
3. Remove arcs out of MSCC and compute simulation relation for result.
4. If initial states with path to MSCC are simulated by initial states without a path to the MSCC, make all the states in the MSCC non-accepting.
5. Minimize automaton if some MSCCs were made non-accepting; otherwise, make non-initial all states that were made initial in the first step.

We rely on the fact that direct simulation minimization removes from the initial states a state that is simulated by another initial state. Hence, we end up with only one initial state if we started with one.

7 Experimental Results

We have implemented the complementation algorithm presented in this paper as an extension of the Wring translator from LTL to Büchi automata [23,8], which is written in Perl. All experiments were run on an IBM IntelliStation running Linux with a 1.7 GHz Pentium 4 CPU and 1 GB of RAM. Complementation experiments were allotted 1 minute if the input NBW was weak, and 2 minutes if it was not.

We use a set of 1000 LTL formulas distributed with Wring to evaluate the complementation algorithm. Two types of comparisons were conducted. In the first, each formula is converted by Wring into a Büchi automaton whose complement is then computed if it has exactly one fairness constraint. (Wring produces generalized Büchi automata, which may have 0, 1, or more sets of accepting states. Our implementation of the complementation algorithm only deals with one set of accepting states.) The complement is compared to the automaton obtained by translating the negation of the LTL formula. In the second comparison, the automaton obtained from an LTL formula is compared to the complement of its complement. Table 1 summarizes our results with regard to the quality of the automata produced by the complementation algorithm. For the two experiments, the table reports the ratios of total numbers of states and transitions produced by the complementation procedure and those in the reference automata.

Several steps in the translation from LTL to automaton are order dependent. Since Wring’s data structures heavily rely on hash tables, even minimal differences in two runs like the addition of a diagnostic print command may cause some differences in the
Table 1. Our complementation procedure produces small automata

<table>
<thead>
<tr>
<th>experiment</th>
<th>states</th>
<th>trans.</th>
</tr>
</thead>
<tbody>
<tr>
<td>negation</td>
<td>1.09</td>
<td>1.26</td>
</tr>
<tr>
<td>double complementation</td>
<td>1.13</td>
<td>2.23</td>
</tr>
</tbody>
</table>

Table 2. Experimental results

<table>
<thead>
<tr>
<th>method</th>
<th>weak timeouts</th>
<th>strong timeouts</th>
<th>time</th>
<th>states</th>
<th>trans.</th>
<th>$M$ opt.</th>
<th>$W$ states</th>
</tr>
</thead>
<tbody>
<tr>
<td>base</td>
<td>406</td>
<td>215</td>
<td>67</td>
<td>56</td>
<td>47303</td>
<td>4.08</td>
<td>7.05</td>
</tr>
<tr>
<td>+w</td>
<td>404</td>
<td>4</td>
<td>70</td>
<td>60</td>
<td>9556</td>
<td>5.96</td>
<td>14.03</td>
</tr>
<tr>
<td>+t9</td>
<td>405</td>
<td>4</td>
<td>69</td>
<td>49</td>
<td>7672</td>
<td>6.07</td>
<td>13.67</td>
</tr>
<tr>
<td>+ds</td>
<td>405</td>
<td>4</td>
<td>68</td>
<td>53</td>
<td>10233</td>
<td>5.96</td>
<td>13.36</td>
</tr>
<tr>
<td>+lc</td>
<td>405</td>
<td>3</td>
<td>69</td>
<td>59</td>
<td>9240</td>
<td>6.02</td>
<td>13.52</td>
</tr>
<tr>
<td>–lc</td>
<td>405</td>
<td>4</td>
<td>68</td>
<td>38</td>
<td>6263</td>
<td>6.48</td>
<td>14.93</td>
</tr>
<tr>
<td>–hr</td>
<td>405</td>
<td>3</td>
<td>68</td>
<td>39</td>
<td>6129</td>
<td>6.38</td>
<td>14.71</td>
</tr>
<tr>
<td>–arc</td>
<td>404</td>
<td>4</td>
<td>69</td>
<td>53</td>
<td>6267</td>
<td>5.95</td>
<td>13.36</td>
</tr>
<tr>
<td>all</td>
<td>406</td>
<td>3</td>
<td>68</td>
<td>39</td>
<td>6568</td>
<td>6.02</td>
<td>13.83</td>
</tr>
</tbody>
</table>

Table 3. Definition of methods compared in Table 2

<table>
<thead>
<tr>
<th>method</th>
<th>$B$ sim</th>
<th>weak test</th>
<th>Thm. 9</th>
<th>$C$ bound</th>
<th>$C$ rank</th>
<th>$W$ arc</th>
<th>$W$ sim</th>
<th>$W$ lc</th>
<th>$M$ hr</th>
<th>$M$ sim</th>
</tr>
</thead>
<tbody>
<tr>
<td>base</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>+w</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>+t9</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>+ds</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>+lc</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>–lc</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>–hr</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>–arc</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>all</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

Table 4. Feature description

<table>
<thead>
<tr>
<th>feature</th>
<th>description</th>
<th>section</th>
</tr>
</thead>
<tbody>
<tr>
<td>$B$ sim</td>
<td>fair simulation minimization of $B$</td>
<td>6</td>
</tr>
<tr>
<td>weak test</td>
<td>simplified treatment for weak $B$</td>
<td>5.4</td>
</tr>
<tr>
<td>Thm. 9</td>
<td>reduce the number of transitions of $W$</td>
<td>5.1</td>
</tr>
<tr>
<td>$C$ bound</td>
<td>use of $2(n -</td>
<td>\alpha</td>
</tr>
<tr>
<td>$C$ rank</td>
<td>exact computation of the rank of $C$</td>
<td>5.3</td>
</tr>
<tr>
<td>$W$ arc</td>
<td>simulation minimization of $W$ with accepting arcs</td>
<td>5.2</td>
</tr>
<tr>
<td>$W$ sim</td>
<td>direct simulation minimization of $W$</td>
<td>5.1</td>
</tr>
<tr>
<td>$W$ lc</td>
<td>removal of MSCCs by language containment</td>
<td>5.3</td>
</tr>
<tr>
<td>$M$ hr</td>
<td>height reduction of $M$</td>
<td>6</td>
</tr>
<tr>
<td>$M$ sim</td>
<td>fair simulation minimization of $M$</td>
<td>6</td>
</tr>
</tbody>
</table>
results. Hence, the number of automata with one set of accepting states presents small fluctuations in the various experiments. The same applies to most quantities we report.

Table 2 compares variants of the complementation algorithm ranging from the basic procedure presented in [12] (base) to the procedure that implements all the improvements described in this paper (all). Table 3 defines all variants in terms for their features, and Table 4 summarizes each feature used to define the methods and refers to the section of this paper that discusses it.

The first column of Table 2 designates the algorithm variant. Columns weak and timeout weak report the number of automata from those with one accepting set that were found to be weak and how many of those timed out. Columns strong and timeout strong do the same for the automata that were not weak. The next column gives the total CPU time in seconds. Columns 7 and 8 give the average number of states and transitions in $M$ for the cases that completed. For comparison, the average numbers of states and transitions of the input automaton $B$ are 6.04 and 12.23, respectively. The last two columns report the average ratio between the size of $M$ before and after optimization ($M$ opt. ratio), and the total number of states of the AWWs.

A few observations can be made about the data in Table 2. First, checking the input automaton $B$ for weakness is a simple way to dramatically improve performance. However, method w+, that adds this simple check to the base approach, can only complete 10 automata that are not weak: Though there seems to remain considerable room for improvement in the complementation of automata that are not weak, the optimizations presented in this paper triple the number of successes.

Comparing the average sizes of the automata obtained with the several variants is hindered by the fact that the largest automata tend to cause the most timeouts. Comparing variants that produce about the same number of timeouts, however, shows that more optimization tends to produce smaller automata. It is also instructive to examine the effects of optimization of the NBW $M$ produced by the subset construction of Theorem 2. The variants that skip direct simulation minimization of the AWW $W$ have higher $M$ opt. ratios because the final optimization has to make up for the “sloppiness” of the preceding stage. While fair simulation minimization of $M$ discharges its duties well, minimization of $W$ leads to a more robust solution.

Acknowledgment. Doron Bustan called our attention to the improved bound on the rank of a UCW.

References


Abstract. In formal verification, we verify that a system is correct with respect to a specification. Even when the system is proven to be correct, there is still a question of how complete the specification is, and whether it really covers all the behaviors of the system. The challenge of making the verification process as exhaustive as possible is even more crucial in simulation-based verification, where the infeasible task of checking all input sequences is replaced by checking a test suite consisting of a finite subset of them. It is very important to measure the exhaustiveness of the test suite, and indeed, there has been an extensive research in the simulation-based verification community on coverage metrics, which provide such a measure. It turns out that no single measure can be absolute, leading to the development of numerous coverage metrics whose usage is determined by industrial verification methodologies. On the other hand, prior research of coverage in formal verification has focused solely on state-based coverage. In this paper we adapt the work done on coverage in simulation-based verification to the formal-verification setting in order to obtain new coverage metrics. Thus, for each of the metrics used in simulation-based verification, we present a corresponding metric that is suitable for the setting of formal verification, and describe an algorithmic way to check it.

1 Introduction

Today’s rapid development of complex hardware designs requires reliable verification methods. In formal verification, we verify the correctness of a design with respect to a desired behavior by checking whether a labeled state-transition graph that models the design satisfies a specification of this behavior, expressed in terms of a temporal logic formula or a finite automaton [CGP99]. Beyond being fully-automatic and reliable, an additional attraction of formal-verification tools is their ability to accompany a negative answer to the correctness query by a counterexample to the satisfaction of the specification in the design [CGMZ95]. On the other hand, when the answer to the correctness query is positive, most formal-verification tools terminate with no further information.
to the user. Since a positive answer means that the design is correct with respect to the specification, this seems like a reasonable policy. In the last few years, however, there has been growing awareness of the importance of suspecting the design of containing an error also in the case verification succeeds. The main justification for such suspicion are possible errors in the modeling of the design or of the behavior, and possible incompleteness in the specification.

Several *sanity checks* have been suggested for further assessment of the modeling of the design and the specification [Kur97]. One direction is to detect *vacuous satisfaction* of the specification [BBER01,KV03,PS02], where cases like antecedent failure [BB94] make parts of the specification irrelevant to its satisfaction. For example, the specification “every request is eventually granted” is vacuously satisfied in a design in which no requests are sent. A similar direction is to check the *validity* of the specification (a specification is valid if it holds for all designs). Clearly, vacuity or validity of the specification suggests some problem. It is less clear how to check completeness of the specification. Indeed, specifications are written manually, and their completeness depends entirely on the competence of the person who writes them. The motivation for a completeness check is clear: an erroneous behavior of the design can escape the verification efforts if this behavior is not captured by the specification. In fact, it is likely that a behavior not captured by the specification also escapes the attention of the designer, who is often the one to provide the specification.

The challenge of making the verification process as exhaustive as possible is even more crucial in *simulation-based* verification. Each input vector for the design induces a different execution of it, and a design is correct if it behaves as required for all possible input vectors. Checking all the executions of a design is an infeasible task. Simulation-based verification is traditionally used in order to check the design with respect to some input vectors [BF00]. The vectors are chosen so that the verification would be as exhaustive as possible, but still, design errors may escape the verification process. Since simulation-based verification is a heuristic that replaces the infeasible task of checking all input vectors, it is very important to measure the exhaustiveness of the input sequences that are checked. Indeed, there has been an extensive research in the simulation-based verification community on *coverage metrics*, which provide such a measure [TK01]. Coverage metrics are used in order to monitor progress of the verification process, estimate whether more input sequences are needed, and direct simulation towards unexplored areas of the design. Essentially, the metrics measure the part of the design that has been activated by the input sequences. For example, in *code-based* coverage metrics, the design is given as a program in some *hardware description language* (HDL), and one measures the number of code lines executed during the simulation. In Section 3, we survey the variety of metrics that are used in simulation-based verification (see also [ZHM97,Dil98,Pel01,TK01]). Coverage metrics today play an important role in the design validation effort [Ver03].

Measuring the exhaustiveness of a specification in formal verification (“do more properties need to be checked?”) has a similar flavor as measuring the exhaustiveness of the input sequences in simulation-based verification (“are more sequences need to be checked?”). Nevertheless, while for simulation-based verification it is clear that coverage corresponds to activation during the execution on the input sequence, it is less
Coverage Metrics for Formal Verification

clear what coverage should correspond to in formal verification, as in model checking all reachable parts of the design are visited. Early work on coverage metrics in formal verification [HKHZ99,KGG99] suggested two directions. Both directions reason about a finite-state machine (FSM) that models the design. The metric in [HKHZ99], later followed by [CKV01,CKKV01,CK02], is based on mutations applied to the FSM. Essentially, a state $s$ in the FSM is covered by the specification if modifying the value of a variable in the state renders the specification untrue. The metric in [KGG99] is based on a comparison between the FSM and a reduced tableau for the specification. See [CKV01] for a discussion of pros and cons of this metric.

Coming up with an exhaustive specification is of great importance and challenge in formal verification. Sanity checks have been helpful in detecting design errors that escape the verification process [BBER01,HKHZ99,PS02]. The main lesson to be learned from several years of research in coverage in simulation-based verification [Pel01,TK01] is that coverage is a heuristic that measures the exhaustiveness of the verification effort, but no single measure can be absolute. Consequently, research in simulation-based coverage has identified numerous coverage metrics; their usage is determined by practical verification methodologies. Prior research of coverage in formal verification [HKHZ99,KGG99,CKV01,CKKV01,CK02] has focused solely on state-based coverage. In contrast, in simulation-based coverage one finds many other coverage metrics, including several metrics of code coverage, which measure that all syntactic aspects of the design have been covered [Pel01,TK01]. Our goal in this paper is to adapt the work done on coverage in simulation-based verification to the formal-verification setting in order to obtain new coverage metrics. Thus, for each of the metrics used in simulation-based verification, we present a corresponding metric that is suitable for the setting of formal verification. In addition, we describe symbolic algorithms for computing each of the new metrics.

The adoption of metrics from simulation-based verification is not straightforward. To see this, consider for example code-based coverage and a check whether both branches of an if statement have been executed during the simulation. A straightforward adoption would check the satisfaction of the specification in a mutant design, one for each branch, in which the branch is disabled. Such a mutant design, however, has less behaviors than the original design, and would clearly satisfy all universal specifications (i.e., specifications that apply to all behaviors, as in linear temporal logic) that are satisfied by the original design. In general, the problem we are facing is the need to assess the role a behavior has played in the satisfaction of a universal specification – one that is clearly satisfied in the design obtained by removing this behavior. The way we suggest to do so is to check whether the specification is vacuously satisfied in a mutant design in which this behavior is disabled: a vacuous satisfaction of the specification in such a design (we assume that the specification is not vacuously satisfied in the original design) indicates that the specification does refer to this behavior; on the other hand, a non-vacuous satisfaction of the specification in the mutant design indicates that the specification does not refer to the missing behavior. Accordingly, some of the new metrics we suggest reduce coverage to queries about vacuous satisfaction. On the other hand, a code-based metric that checks whether a particular assignment in the code has been executed may also be reduced to a metric that checks the satisfaction of the specification in a mutant de-
sign in which the assignment is changed. Accordingly, some of the metrics we suggest follow the approach in [HKHZ99] and reduce coverage to queries about satisfaction of the specification in mutant designs. Unlike previous work, however, the mutant designs we consider are not arbitrary, and capture the different metrics of coverage used in simulation-based verification.

Due to lack of space, this version misses many technical details. A fuller version can be found at the authors’ URLs.

2 Preliminaries

2.1 Simulation-Based Verification

In simulation-based verification, the implementation of a hardware design is executed in parallel with a reference model described at a different level of abstraction or with monitors and assertions that check for certain behavior of the implementation [KN96]. The execution is done with respect to a selected set of finite input sequences, referred to as tests. Thus, assuming the implementation has a set $I$ of input signals, a test is a finite sequence $t = i_0, i_1, \ldots, i_n \in (2^I)^*$ of input assignments. Implementations of hardware designs can be described by different formalisms. We consider two formalisms with respect to which coverage metrics are naturally defined.

The first formalism is that of hardware description languages (HDL). A typical HDL program specifies the input and output variables of the various modules of the design, and, using control and assignment statements, the interaction of the modules among themselves and with an environment that provides the input signals. Reasoning about rich HDL such as Verilog involves difficult technical details.\footnote{For a description of a formal model of a real-life HDL see, for example, [FLLO95].} We consider here the simplified model of control flow graph (CFG). Each HDL statement corresponds to a control state and induces a node in the CFG. We refer to CFG nodes as locations. Assignment statements have a single successor, and control statements, such as if or while, have several successors, corresponding to the possible locations to which the control can jump. Transitions from a control statement to its successors are labeled by an expression that guards the transition. Recall that the design interacts with an environment that supplies its input signals. When the design is described as a CFG, the interaction induces a traversal of the CFG. Formally, given a CFG $G$ with a set $V$ of locations, and a test $t = i_0, i_1, \ldots, i_n \in (2^I)^{n+1}$ of input assignments, the execution of $G$ on $t$ is a sequence $(i_0, l_0^{i_0}, o_0), (i_1, l_0^{i_1}, \ldots, l_1^{i_1}, o_1), \ldots, (i_n, l_0^{i_n}, \ldots, l_m^{i_m}, o_n) \in (2^I \times V^+ \times 2^O)^{n+1}$ such that $l_0^{i_0}$ is the initial location of $G$, for all $0 \leq i \leq n$, the location $l_m^{i_m}$ corresponds to a read and write assertion, $o_i$ is the new assignment to the output variables, and $l_m^{i_m+1}$ matches the control flow of the CFG from location $l_m^{i_m}$ upon reading $i_{m+1}$. The locations $l_1^{i_1}, \ldots, l_{m+1}^{i_{m+1}}$ then correspond to the control flow of the CFG from $l_1^{i_1}$ until the next input assignment is read. We often ignore the input and output variables and refer to the interaction as a word in $V^*$ obtained by projecting the execution above on $V$.

The second formalism is that of sequential circuits. We refer to a circuit as a tuple $S = \langle I, O, L, \mathcal{F}, \mathcal{G}, c_0 \rangle$, where $I$ and $O$ are the sets of input and output signals, respectively, $L$ is a set of latches, $c_0 \in 2^L$ describes the initial values of the latches, and $\mathcal{F}$ and $\mathcal{G}$ are...
families of the next-state and output functions. Thus, each latch $l \in \mathcal{L}$ has a function $f_l : 2^I \times 2^\mathcal{L} \to \{0, 1\}$ in $\mathcal{F}$, and each output signal $o \in O$ has a function $g_o : 2^I \times 2^\mathcal{L} \to \{0, 1\}$ in $\mathcal{G}$. A configuration $c \in 2^\mathcal{L}$ of the circuit describes the value of each latch. The circuit starts its interaction with the environment in configuration $c_0$. When the circuit is in configuration $c$ and it reads a set $i \in 2^I$ of input signals, it moves to configuration $c'$ in which the value of each latch $l$ is $f_l(i, c)$, and in which it sends to the environment the set of output signals $o$ with $g_o(i, c) = 1$. Accordingly, the execution of a circuit $\mathcal{S}$ on a test $t = i_0, i_1, \ldots, i_n \in (2^I)^{n+1}$, is a sequence $\langle i_0, o_0 \rangle, \langle i_1, c_1, o_1 \rangle, \ldots, \langle i_n, c_n, o_n \rangle \in (2^I \times 2^\mathcal{L} \times 2^O)^{n+1}$ that satisfies the conditions above.

Both HDL and circuits enable a description of the design at different levels of abstraction [Hos95], yet abstraction is most naturally supported when the design is modeled as a symbolic finite state machine (FSM). We assume that the design is defined with respect to a set $X$ of state variables, and it is specified by predicates on $X$ and $X'$ — a primed version of the variables in $X$. Formally, an FSM is a tuple $F = \langle I, O, X, \theta_{in}, \theta_{next}, \mathcal{G} \rangle$, where $I$ and $O$ are the input and output variables, $X$ is the set of state variables, inducing the state space $2^X$, $\theta_{in}$ is a predicate on $X$ describing the set of initial states, $\theta_{next}$ is a predicate on $X \cup X'$ describing the transition relation (there is a transition from state $u$ to state $v$ iff $\theta_{next}(u, v')$), and $\mathcal{G}$ is a family of predicates that associates with each input or output variable $s$ a predicate $\tau_s$ on $X$ describing the set of states in which $s$ holds. Likewise, predicates on $X$ are used to describe other sets of interest, for example, the set of fair states when the design comes with an unconditional fairness constraint. Formally, a fair FSM $F$ is a tuple $F = \langle I, O, X, \theta_{in}, \theta_{next}, \mathcal{G}, \alpha \rangle$, where $\alpha$ is a predicate on $X$ describing the accepting condition. A behavior $\pi$ is accepted by $F$ if it satisfies $\alpha$. The simplest accepting condition is Büchi condition [Büc62] (called impartiality in [MP92]), where $\alpha$ is a predicate on $X$ and a behavior $\pi$ satisfies $\alpha$ if it visit a state satisfying $\alpha$ infinitely often.

### 2.2 Model Checking, Vacuity, and Coverage

In linear-time model checking, we check whether a design has a desired behavior by checking whether a Büchi automaton for the negation of the specification has accepting runs on an FSM describing the design [VW86]. The specification can be expressed as an LTL formula [Hol97], as a ForSpec formula [AFG+02], or as a Büchi automaton [HHK96,Kur98]. A specification $\varphi$ in linear temporal logic can be translated to a non-deterministic Büchi automaton $\mathcal{A}_{\neg \varphi}$ that accepts all words that do not satisfy $\varphi$ [VW94]. Given an automaton $\mathcal{A}_{\neg \varphi}$, we check that the product of $F$ with $\mathcal{A}_{\neg \varphi}$, which is a fair FSM $F \times \mathcal{A}_{\neg \varphi}$, does not contain accepting paths.

Sanity checks for model checking address the problem of errors in the modeling of the design and the desired behavior, which are not discovered by model checking. These problems may cause “false positive” results of model checking and conceal errors in the design. Two such checks are vacuity and coverage, which we briefly review below (for the full details, see [BBER01,KV03,HKHZ99,CKV01]).

Intuitively, an FSM $F$ satisfies a formula $\varphi$ vacuously if $F$ satisfies $\varphi$ yet it does so in a non-interesting way, which is likely to point on some trouble with either $F$ or $\varphi$. In order to formalize this intuition, we first say that a subformula $\psi$ of $\varphi$ does not affect $\varphi$ in $F$ if for every formula $\xi$, the FSM $F$ satisfies $\varphi[\psi \leftarrow \xi]$ iff $F$ satisfies $\varphi$. |
As shown in [KV03], when $\psi$ of $\phi$ in $F$ vacuously $\phi$ true, the formulas $F$ a naive algorithm that model checks $\psi$ the replacement of on the polarity of $\psi$ sophisticated algorithms are suggested in [PS02, KV03, Cho03, AFF].

In this paper we introduce new types of mutations and new types of coverage metrics [CKKV01]. Coverage in model checking was introduced in [HKHZ99, KGG99]. The metric in [HKHZ99] is based on FSM mutations. For an FSM $F = (I, O, X, \theta_{in}, \theta_{next}, G)$, a state $w \in 2^X$ and an output variable $q \in O$, a mutant FSM $\tilde{F}_{w,q}$ is obtained from $F$ by dualizing the value of $q$ in the state $w$. Thus, if $\tau_q$ is the predicate describing the set of states satisfying $q$ in $F$, then the predicate $\tilde{\tau}_{w,q}$, which describes the set of states satisfying $q$ in $\tilde{F}_{w,q}$, is satisfied by $w$ iff $\tau_q$ is not satisfied by $w$. For all states $v \neq w$, the predicate $\tilde{\tau}_{w,q}$ is satisfied by $v$ iff $\tau_q$ is satisfied by $v$. For an FSM $F$, a specification $\phi$ that is satisfied in $F$, and an output variable $q$, we say that $\phi$ $q$-covers $w$ iff $\tilde{F}_{w,q}$ no longer satisfies $\phi$. By [HKHZ99], a state is covered if it is $q$-covered for some output variable $q$. It is easy to see that the set of states $q$-covered by $\phi$ can be computed by a naive algorithm that performs model checking of $\phi$ in $\tilde{F}_{w,q}$ for each state $w$ of $F$. More sophisticated algorithms are suggested in [HKHZ99, CKV01, CKKV01].

Chockler et al. also suggest the following refinement of coverage metrics [CKKV01]. Instead of performing local mutations in $F$, we can perform local mutations in the infinite tree $T_F$ obtained by unwinding $F$. A state $w$ of $F$ can appear many (possibly an infinite number of) times in $T_F$. Flipping the value of $q$ in one occurrence of $w$ in $T_F$ can have a different effect from flipping the value of $q$ in all or some of the occurrences of $w$ in $T_F$. These differences are captured by the notions of node, structure, and tree coverage. Node coverage of a state $w$ corresponds to flipping the value of $q$ in one occurrence of $w$ in the infinite tree. Structure coverage corresponds to flipping the value of $q$ in all the occurrences of $w$ in the tree. Chockler et al. describe a framework in which node, structure, and tree coverage can be computed by a symbolic algorithm; minor changes are required to capture the different types of coverage [CKKV01]. We describe their algorithm in more detail in Section 5.

In this paper we introduce new types of mutations and new types of coverage metrics in model checking in order to capture better the different notions of coverage used in simulation-based verification. Coverage in model checking is performed by applying mutations to a given FSM and then examining the resulting mutant FSMs with respect to a given specification. Each mutation is generated in order to check whether a specific element of the design is essential for the satisfaction of the specification. As we explain in more detail in Section 4, mutations correspond to omissions and replacements of small elements of the design, which can be given as an HDL program, an FSM, or a sequential circuit. Once we have a mutant FSM, there are two coverage checks we can perform on it.
1. **Falsity coverage**: does the mutant FSM still satisfy the specification?
2. **Vacuity coverage**: if the mutant FSM still satisfies the specification, does it satisfy it vacuously?

Falsity coverage is the metric introduced in [HKHZ99], and we extend it here to handle mutations richer than those studied in the literature so far. Vacuity coverage is new. As we demonstrate in Example 1, it often provides information that falsity coverage fails to detect. In particular, in mutations that are based on omission of elements from the original design (as we are going to see in Section 4, such mutations are popular in metrics adopted from simulation-based verification), falsity coverage is useless for universal specifications. Indeed, having less behaviors, the mutant design is guaranteed to satisfy all the specifications satisfied by the original design.

**Example 1.** Consider the FSM $F$ described below, which abstracts a design with respect to the output signals $grant_1$ and $grant_2$. Let $\varphi = G(grant_1 \rightarrow Fgrant_2)$. Thus, $\varphi$ requires that (in all execution paths) each grant to the first user is followed by a grant to the second user. It is easy to see that $\varphi$ is satisfied in $F$. Recall that the goal of coverage metrics is to check whether all the elements of the design play some role in the satisfaction of $\varphi$. Let us see which parts of $F$ are covered by $\varphi$. We refer only to structure coverage in this example.

- The positive value of $grant_2$ in $w_4$ is essential to the satisfaction of $\varphi$: the state $w_4$ is falsity covered by $\varphi$ with respect to mutations that flip the value of $grant_2$.
- The value of $grant_1$ in $w_1$ is not essential to the satisfaction of $\varphi$. On the other hand, the designer had a reason to set it to $true$ in $w_1$, as it is essential to the non-vacuous satisfaction of $\varphi$: the state $w_1$ is vacuity covered by $\varphi$ with respect to mutations in which $w_1$ is omitted and with respect to mutations that flip the value of $grant_1$.
- One may also question negative values of variables. For example, while the negative value of $grant_2$ in $w_0$ is not essential to the satisfaction of $\varphi$, it is essential to its non-vacuous satisfaction: the state $w_0$ is vacuity covered by $\varphi$ with respect to mutations that flip the value of $grant_2$.
- Consider now the value of $grant_2$ in the state $w_2$. All the paths of $F$ that pass through $w_2$ describe a behavior in which two grants – in both $w_2$ and in $w_4$, are given to the second user, after at most one grant was given to the first user. The specification does not require such a behavior, nor does it require a correspondence between the number of grants that each user gets. The labeling of $w_2$ indeed does not play a role in the satisfaction of $\varphi$: the state $w_2$ is neither falsity nor vacuity covered by $\varphi$ with respect to mutations that omit $w_2$ or flip the value of $grant_2$. This information may hint on a possible impreciseness or incompleteness in the definition of $\varphi$.

### 3 Coverage Metrics in Simulation-Based Verification

In this section we survey coverage metrics in simulation-based verification – metrics we are going to adopt for the setting of formal verification in the next section. Each of the
metrics is “tailored” for a specific representation of the design or a specific verification goal. The reader is referred to [TK01] for a detailed survey. All metrics refer to a set of input sequences (or tests) $t \in (2^I)^*$ with respect to which the design is simulated.

3.1 Syntactic Coverage Metrics

Syntactic coverage metrics assume a specific formalism for the description of the design and measure the syntactic part of the design visited in the process of execution of a given input sequence. Commonly [Mar99,TK01], high coverage according to syntactic-based metrics is considered a precondition to moving to other more sophisticated (and time consuming) coverage metrics.

**Code Coverage.** Code-based coverage metrics refer to the HDL program that describes the design or to its CFG. Measuring code coverage requires little overhead and it is easy to interpret the coverage information. This makes code coverage the most popular metric [UZ98,TK01]. The most widely used code-coverage metrics are statement and branch coverage. Essentially, an object is covered if it is visited during the execution of the input sequence. Again, the fully-formal definition depends on the particular HDL used, but a semi-formal definition is given in terms of the computation of the CFG as follows. Let $G$ be a CFG. For an input sequence $t \in (2^I)^*$ such that the execution of $G$ on $t$, projected on the sequence of locations, is $l_0, \ldots, l_m$, we say that a statement $\tau$ is covered by $t$ if there is $0 \leq j \leq m$ such that the control location $l_j$ corresponds to $\tau$. We say that a branch $(l, l')$ between two control locations is covered by $t$ if there is $0 \leq j \leq m - 1$ such that $l_j = l$ and $l_{j+1} = l'$. More sophisticated metrics measure the way expressions in the guards labeling the CFG’s transitions are satisfied. For example, expression coverage checks whether a Boolean expression has been satisfied by all its satisfying assignments (e.g., whether $a_1 == a_2$ has been satisfied by both an $a_1 = a_2 = 0$ and an $a_1 = a_2 = 1$ assignment).

**Circuit Coverage.** Circuit-structure based coverage metrics refer to the circuit that describes the design. Thus they identify the physical parts of the circuit that are covered. Measuring circuit coverage is usually easy and it is easy to interpret the coverage information. Unlike code coverage, however, it is not easy to use the coverage information in order to generate new tests that direct simulation towards the unexplored areas of the design. The most widely used circuit-coverage metrics are latch and toggle coverage [HH96,KN96]. Essentially, a latch is covered if it changes its value at least once during the execution of the input sequence. Similarly, an output variable is covered if its value has been toggled. Formally, for a circuit $S$ and an input sequence $t \in (2^I)^{n+1}$ such that the execution of $S$ on $t$ is $\langle i_0, c_0, o_0 \rangle, \langle i_1, c_1, o_1 \rangle, \ldots, \langle i_n, c_n, o_n \rangle$, we say that a latch $l \in L$ is covered by $t$ if there is $j \geq 0$ such that $l \in c_0$ iff $l \notin c_j$. Similarly, an output variable $o \in O$ is covered by $t$ if there are $0 < j_1 < j_2$ such that $o \in o_0$ iff $o \notin o_{j_1}$ iff $o \in o_{j_2}$. Note that toggle coverage requires that the value of an output variable should be changed at least twice during the execution of $t$. 
3.2 Semantic Coverage Metrics

Semantic coverage metrics measure the part of the functionality of the design exercised by the set of input sequences. Semantic coverage metrics require user help and are more sophisticated than syntactic coverage metrics. We consider the following metrics.

**FSM Coverage.** Due to the large size of FSMs for complete systems, FSM-based coverage metrics refer to more abstract FSMs constructed manually by the designer, or automatically extracted from the design by projecting its symbolic description on a subset of the state variables as explained in Section 2.1 [TK01]. Similarly to code coverage, a state or a transition of the abstract FSM is covered if it is visited during the execution of the input sequence. The fact that coverage is checked with respect to an abstract FSM makes the interpretation of the coverage information harder (linking the uncovered parts of the FSM to uncovered parts of the HDL program is not trivial) and have led to the use of more sophisticated metrics. In particular, limited-path coverage metrics check that important sequences of behavior are exercised [SA99]. Transition coverage can be viewed as a special case of path coverage, for paths of length 1.

**Assertion Coverage.** In assertion coverage (“functional coverage”, in [TK01, Cad03]), the user provides a list of assertions referring to the variables of the design. The assertions describe some conditions that may be satisfied during the execution or a state of the design during the execution. They may be propositional (“snapshot tasks”) or temporal (describing a behavior along several clock cycles). A test \( t \) covers an assertion \( a \) if the execution of the design on \( t \) satisfies \( a \). The assertion-coverage metric measures what assertions are covered by a given set of input sequences.

**Mutation Coverage.** In mutation coverage, the user introduces a small change (aka “mutation”) to the design, and checks whether the change leads to an erroneous behavior [DLS78, Bud81, ZHM97]. The coverage of a test \( t \) is measured as the percentage of the mutant designs that fail on \( t \), that is, the percentage of the mutations that \( t \) “catches”. The list of interesting mutations can be written manually or automatically following some mutation criteria. For example, a local mutation can be flipping a value of one output variable in a circuit. In mutation coverage the goal is to find a set of input sequences such that for each mutant design there exists at least one test that fails on it. As discussed in Section 2.2, mutation coverage is the metric that inspired most of the work on coverage in model checking.

4 Coverage Metrics in Model Checking

In this section we discuss how the coverage metrics from simulation-based verification can be adopted in model checking. Thus, for each of the metrics described in Section 3, we define a metric that can be used in the context of model checking.
4.1 Syntactic Coverage

In syntactic coverage, we assume that we are given the syntactic representation of the design (an HDL code or a CFG) with respect to which we measure the coverage. Since in the process of model checking we visit the whole reachable part of the design, metrics that measure the part of the design exercised during the simulation cannot be applied directly to model checking. Essentially, we adopt these metrics by replacing the question whether a part of the design has been visited during the simulation by the question whether the part plays a role in the success of the verification process, where playing a role means that the part is essential for the satisfaction or the non-vacuous satisfaction of the specification. The latter is checked by reasoning about the behavior of a mutant design in which the part is modified or omitted.

**Code Coverage.** Let $G$ be a CFG and $\varphi$ a specification that is satisfied in $G$. We say that a statement $\tau$ of $G$ is covered by $\varphi$ if omitting $\tau$ from $G$ causes vacuous satisfaction of $\varphi$ in the mutant CFG. Similarly, a branch $\langle l, l' \rangle$ of $G$ is covered if omitting it causes vacuous satisfaction of $\varphi$. Note that falsity coverage would be meaningless here, since omitting a statement or a branch of CFG results in a design with fewer behaviors, which is guaranteed to satisfy the universal specification. In expression coverage, we check whether omitting the behaviors in which the variables have a particular satisfying assignment for a particular expression leads to vacuous satisfaction of $\varphi$.

**Circuit Coverage.** Recall that latch and toggle coverage metrics check whether the value of a specific latch or variable in the circuit changes during the execution of an input sequence. We replace this question by the question whether disabling the change causes the specification to be satisfied vacuously. Thus, a latch $l \in L$ is covered if the specification is vacuously satisfied in the circuit obtained by fixing the value of $l$ to its initial value. Similarly, an output variable $o \in O$ is covered if the specification is vacuously satisfied in the circuit obtained by allowing $o$ to change its value only once. Thus, if the initial value of $o$ is 0, the circuit is obtained by fixing $o$ to 1 as soon as it changes its value to 1, and if the initial value of $o$ is 1, the circuit is obtained by fixing $o$ to 0 as soon as it changes its value to 0.

4.2 Semantic Coverage

Among the semantic coverage metrics, mutation coverage has already been adopted to the setting of model checking. As discussed in Section 2.2, we suggest a strengthening of the adopted metrics by checking the effect of the mutation not only on the satisfaction of the specification, but also on its vacuous satisfaction. Below we describe the adoption of the other semantic coverage metrics.

**FSM Coverage.** In FSM coverage we are given an abstract FSM $F$ and we check the influence of mutations and omissions in this FSM on the result of model checking of the specification $\varphi$ in the design. In state coverage, for a state $w$ of $F$ we check the influence of omission of $w$ or changing the values of output variables in $w$ on the (non-vacuous) satisfaction of the specification in the design. Clearly, a mutan FSM $\tilde{F}_w$ obtained from
Coverage Metrics for Formal Verification

F by omitting w has fewer behaviors than F, thus for omissions of a state we only check vacuity coverage. On the other hand, a mutant FSM $\tilde{F}_{w,o}$ obtained from F by flipping the value of the output variable $o \in O$ in w can also falsify the specification, thus we check falsity and vacuity coverage.

In path coverage, we check the influence of omitting or mutating a finite path on the (non-vacuous) satisfaction of the specification in the design. A path $\pi$ of length $c$ in F is a sequence of states $w_1, \ldots, w_c$ of F such that for all $1 \leq i \leq c - 1$ we have $\theta_{next}(w_i, w_{i+1})$. Let us first define coverage for omissions of a path. A path $\pi$ is covered by $\varphi$ if the mutant FSM $\tilde{F}_{\pi}$ obtained from F by omitting all behaviors that contain $\pi$ satisfies $\varphi$ vacuously. On the other hand, we can also introduce mutations that replace $\pi$ with a mutant path $\tilde{\pi}$ in the FSM. Then, the mutant FSM $\tilde{F}_{\pi,\tilde{\pi}}$ is obtained from F by replacing $\pi$ with $\tilde{\pi}$. The mutant FSM $\tilde{F}_{\pi,\tilde{\pi}}$ can falsify $\varphi$ or can satisfy $\varphi$ vacuously, thus for mutations that replace a path with another, mutant path we check both falsity and vacuity coverage. We note that all possible mutations in the FSM can be introduced consistently on each occurrence of the mutated element, on exactly one occurrence, or on a subset of occurrences, thus resulting in structure, node, or tree coverage, respectively.

Assertion Coverage. An input to assertion-coverage check is an FSM F, a specification $\varphi$ that is satisfied non-vacuously in F, and a list of LTL assertions $a_1, \ldots, a_k$. An assertion $a_i$ is covered by $\varphi$ in F if the mutant FSM $\tilde{F}_{a_i}$ obtained from F by omitting all behaviors that do not satisfy $a$ satisfies $\varphi$ vacuously. We note that this definition is similar to the definition of FSM path coverage. The only difference is in the description of the mutation: in FSM path coverage we omit behaviors that contain a given finite path $\pi$, whereas in assertion coverage we omit behaviors that do not satisfy a given assertion.

5 Coverage Computation

In Section 4 we described new coverage metrics for model checking. In this section we discuss how to compute these metrics. We first show that both vacuity and falsity coverage can be reduced to model checking (possibly of mutant specifications and/or mutant designs). Let $F$ be an FSM, $\varphi$ a specification that is satisfied in $F$ non-vacuously, and $\tilde{F}$ a mutant FSM. If $\tilde{F}$ does not satisfy $\varphi$, we say that $\tilde{F}$ is falsity covered by $\varphi$. If $\tilde{F}$ satisfies $\varphi$, it still may be vacuity covered by $\varphi$ if it satisfies $\varphi$ vacuously. Formally, $\tilde{F}$ satisfies $\varphi$ vacuously if $\tilde{F} \models \varphi$ and there exists $\psi \in cl(\varphi)$ such that $\tilde{F}$ satisfies $\varphi[\psi \leftarrow \bot]$. Thus, like falsity coverage, we check whether a mutant design $\tilde{F}$ satisfies a specification, only that here the specification is also mutated.

Mutation Coverage. The algorithm we present for falsity-coverage computation is based on the coverage algorithm described in [CKKV01]. That algorithm computes symbolically falsity coverage for mutations that flip the value of a variable $q \in O$ in one state $w$ of the FSM. The idea is to look for a fair path in the product of the mutant FSM $\tilde{F}$ and an automaton $A_{\neg \varphi}$ for the negation of $\varphi$. The state space of the product is $2^X \times S$, where $X$ is the set of state variables of $F$, $S$ is the state space of $A_{\neg \psi}$, and the transitions of the product are induced by the transition relations of $F$ and $A_{\neg \psi}$. In order to compute the set of covered states, it is suggested in [CKKV01] to add $|X|$ new variables that encode
the state $w$ in which the value of $q$ is flipped. It is now possible to define symbolically an augmented product, with state space $2^X\times2^X\times S$, where the first component of a state $\langle w, u, s \rangle$ is the state $w$ that is being considered, and the two other components are as in the usual product automaton. The value of the first component is chosen nondeterministically at initialization and is kept unchanged. The copy of the augmented product with first component $w$ checks whether the mutation of $F$ in which $q$ is flipped in $w$ contains a fair path (in which case flipping $q$ in $w$ violates the specification). Thus, when the augmented product is in a state $\langle w, w, s \rangle$, the set of successor states contains all triples $\langle w, u, t \rangle$ such that $u$ is a successor of $w$ and $t \in \delta(s, \tilde{\sigma})$, where $\tilde{\sigma}$ is the label of $w$ in $\tilde{F}_{w,q}$. The above describes structure coverage, where the value of $w$ is flipped in all visits. Likewise, we can define an augmented product in which the value of $q$ in $w$ is flipped only one time (node coverage) or some of the times (tree coverage). We can now use a symbolic algorithm in order to find the set $P$ of all triples $\langle w, u, s \rangle$ from which there exists a fair path in the augmented product automaton. The covered states are those $w$ such that $\langle w, u_0, s_0 \rangle \in P$, for some initial states $u_0$ of $F$ and $s_0$ of $A_{\neg \psi}$.

Vacuity Coverage. Recall that checking whether a system satisfies a specification vacuously involves model checking of a mutant specification. We adjust the symbolic algorithm in [CKKV01] to this setting by adding a new variable $x$ that encodes the subformula $\psi \in \text{cl}(\varphi)$ that is being replaced with $\bot$. The variable $x$ is an integer in the range $0, \ldots, |\text{cl}(\varphi)|$, thus it can be encoded with $O(\log |\varphi|)$ Boolean variables. The value $0$ of $x$ stands for “no replacement”, thus it checks the satisfaction of $\varphi$ in the system. As with mutations, the values of these variables are chosen nondeterministically at initialization and are kept unchanged. In the automaton $A_{\neg \varphi}$, each state variable corresponds to a subformula (cf. [BCM92]), thus the nondeterministic choice of the subformula leads to a mutant automaton $A_{\neg \varphi[\psi \leftarrow \bot]}$. The state space of the augmented product now consists of triples $\langle x, u, s \rangle$, where $x$ encodes the subformula replaced with $\bot$, and $u$ and $s$ are the components of the product automaton. The successors of $\langle x, u, s \rangle$ are the triples $\langle x, u', s' \rangle$ such that $\langle u', s' \rangle$ is a possible successor of $\langle u, s \rangle$ in a product between the system with the automaton $A_{\neg \varphi[\psi \leftarrow \bot]}$, where $\psi$ is the subformula encoded by $x$. The subformulas that affect the value of $\varphi$ in the systems are these encoded by a value $x$ for which there are initial states $u_0$ and $s_0$ of the system and the automaton, respectively, such that there is a fair path from $\langle x, u_0, s_0 \rangle$. Let $P$ be the set of triples from which a fair path exists in the augmented product (as above, $P$ can be found symbolically), and let $P'$ be the intersection of $P$ with the initial states of the system and the automaton, projected on the first element. Note that $x \in P'$ iff the subformula associated with $x$ affects the value of $\varphi$ in the system. Thus, $\psi$ is satisfied vacuously in the system if $\neg P'(0)$ and $P' \neq \{1, \ldots, \text{cl}(\psi)\}$.

In order to get a symbolic algorithm for vacuity coverage, we combine the above algorithm with the one of [CKKV01]. For example, if we want to find the set of states $w$ such that flipping the value of $q$ in $w$ causes the specification to be satisfied vacuously, we augment the state space of the product of $F$ and $A_{\neg \varphi}$ by variables that encode both the state in which we do the mutation and the subformula that is being replaced with $\bot$. As we specify below, if we want to check vacuity coverage for other types of mutations, we use the variables in order to encode the other types of mutations.
Code Coverage. Recall that in code coverage we need to check whether the omission of parts of the code causes the specification to be satisfied vacuously. Accordingly, for code coverage, it is simpler to define the mutations with respect to the HDL code. Let $k$ be the number of elements in the code we want to check (e.g., the number of lines). We introduce a new variable $mut$, which is an integer in the interval $[1, \ldots, k]$. The value $i$ of $mut$ indicates that the mutation is in element $l_i$, which we want to omit, and we need $O(\log k)$ Boolean variables to encode it. The HDL code is instrumented using source-to-source translation in (see [BKM02] as an example of such instrumentation) so that $l_i$ in the code is replaced by the statement “if ($mut \neq i$) then $l_i$ else skip”. The instrumented code represents all the mutant designs. The product of the FSM induced by the instrumented code and $A_{\neg \varphi}$ subsumes all the mutations of the code. It is now possible to apply the symbolic algorithm described above (instead of the variables that encode $w$, we now have the variables that encode $mut$) for detecting the mutations that lead to vacuous satisfaction.

In expression coverage, we do something similar. Let $e_1, \ldots, e_m$ be the expressions we want to check, and let $V_i = \{v^1_{i}, \ldots, v^n_{i}\}$ be the Boolean variables over which $e_i$ is defined. Assume that $n$ bounds the number of variables in every expression. Let “if $e_i$ then $B_i$” be the statement that contains $e_i$ as a guard (handling of “while” or “until” statements is similar). Recall that we want to check, for each $e_i$ and for each satisfying assignment $f \in 2^{V_i}$, whether skipping $B_i$ when the variables have value $f$ causes the specification to be satisfied vacuously. Accordingly, we add a variable $mut$ (encoded by $O(\log m)$ Boolean variables) that indicates the expression to be checked, and $n$ variables $u_1, \ldots, u_n$ that encode assignments to $n$ variables. As usual, the variables get their value nondeterministically at initialization. The HDL code is now instrumented so that “if $e_i$ then $B_i$” in the code is replaced by “if ($mut \neq i$) or ($e_i \land \bigvee_{1 \leq j \leq n} v^j_{i} \neq u_j$) then $B_i$ else skip”. It is now possible to apply the symbolic algorithm described above for detecting the expressions and assignments that lead to vacuous satisfaction.

Circuit Coverage. In latch coverage, we restrict the product of $F$ and $A_{\neg \varphi}$ to paths in which the value of a latch is not allowed to change, and check whether this causes vacuous satisfaction. Thus, we augment the product with variables that encode the examined latch and (for the vacuity check) the subformula of $\varphi$ that we replace with $\bot$.

FSM Coverage. State and transition coverage can be computed using the techniques of mutation-based metrics. We now describe the computation of path coverage. We start with mutations that omit all behaviors that contain a given finite path $\pi = w_1, \ldots, w_c$. Let $M_\pi$ be a monitor that filters away paths that contain $\pi$ as a sub-path. That is, $M_\pi$ is a fair FSM that accepts paths $\rho$ such that $\pi$ is not a sub-path of $\rho$. Since $M_\pi$ only cares for the values of control variables that encode the states (and not, for example, for the values of output variables in these states), the set of input variables of $M_\pi$ is the set of control variables $X$ of $F$, and $M_\pi$ does not have output variables. For a given path $\pi$, the mutant FSM $\tilde{F}_\pi$ is the product FSM $F \times M_\pi$, which contains only the computations of $F$ that do not have $\pi$ as a sub-path. Then, $\pi$ is vacuity covered by $\varphi$ if $\tilde{F}_\pi$ satisfies

---

2 The user may wish to include 0 (no mutation) in the range of $mut$, in which case the instrumented code represents also the original design.
ϕ vacuously. For a set of paths \{π₁, ..., πₖ\}, we can compute the set of covered paths symbolically using the techniques as described above for vacuity coverage.

In a similar way we can define mutations of paths that replace a finite path π with a path \(\tilde{π}\) of the same length, redirecting the system to another execution. If a mutated path is of length 1, the mutation redirects one transition. For a path π replaced with a mutant path \(\tilde{π}\), we use a monitor \(M_{π,\tilde{π}}\). In the product \(\tilde{F}_π\) of \(F\) with \(M_{π,\tilde{π}}\), all the occurrences of π are replaced by \(\tilde{π}\). Note that for mutated (rather than omitted) paths we can compute both falsity and vacuity coverage.

**Assertion Coverage.** For an LTL assertion \(a\), a monitor for \(a\) is the automaton \(A_{\neg a}\). Given assertions \(a₁, ..., aₖ\), the mutant FSM is the product \(F \times A_{\neg a₁} \times ... \times A_{\neg aₖ}\). Falsity and vacuity coverage of a set of assertions is computed similarly to FSM path coverage, where the variable \(\text{mut}\) encodes the assertion \(a_{\text{mut}}\) for \(1 \leq \text{mut} \leq k\).

**References**


Coverage Metrics for Formal Verification


More Deterministic” vs. “Smaller” Büchi Automata for Efficient LTL Model Checking

Roberto Sebastiani and Stefano Tonetta

DIT, Università di Trento, via Sommarive 14, 38050 Povo, Trento, Italy
{rseba, stonetta}@dit.unitn.it

Abstract. The standard technique for LTL model checking (M |= ¬ϕ) consists on translating the negation of the LTL specification, ϕ, into a Büchi automaton Aϕ, and then on checking if the product M × Aϕ has an empty language. The efforts to maximize the efficiency of this process have so far concentrated on developing translation algorithms producing Büchi automata which are “as small as possible”, under the implicit conjecture that this fact should make the final product smaller. In this paper we build on a different conjecture and present an alternative approach in which we generate instead Büchi automata which are “as deterministic as possible”, in the sense that we try to reduce as much as we are able to the presence of non-deterministic decision states in Aϕ. We motivate our choice and present some empirical tests to support this approach.

1 Introduction

Model checking is a formal verification technique which allows for checking if the model of a system verifies some desired property. In LTL model checking, the system is modeled as a Kripke structure M, and (the negation of) the property is encoded as an LTL formula ϕ. The standard technique for LTL model checking consists on translating ϕ into a Büchi automaton Aϕ, and then on checking if the product M × Aϕ has an empty language. To this extent, the quality of the translation technique plays a key role in the efficiency of the overall process.

Since the seminal work in [6], the efforts to maximize the efficiency of this process have so far concentrated on developing translation algorithms which produce from each LTL formula a Büchi automaton (BA henceforth) which is “as small as possible” (see, e.g., [1,12,3,5,4,9,7]). This is motivated by the implicit heuristic conjecture that, as the size of the product M × Aϕ of the Kripke structure M and the BA Aϕ is in worst-case the product of the sizes of M and Aϕ, reducing the size of Aϕ is likely to reduce the size of the final product also in the average case. This conjecture is implicitly assumed in most of papers (e.g., [1,12,5,7]), which use the size of the BA’s as the only measurement of efficiency in empirical tests.

Remarkably, Etessami and Holtzmann [3] tested their translation procedures by measuring both the size of resulting BA’s and the actual efficiency of the LTL model checking.

* This work has been sponsored by the CALCULEMUS! IHP-RTN EC project, contract code HPRN-CT-2000-00102, and has thus benefited of the financial contribution of the Commission through the IHP programme. The authors are also sponsored by a MIUR COFIN02 project, code 2002097822_003.
process, and noticed that “... a smaller number of states in the automaton does not necessarily improve the running time and can actually hurt it in ways that are difficult to predict” [3].

In this paper we propose and explore a new research direction. Instead of wondering what makes the BA $A_\varphi$ smaller, we wonder directly what may make the product automaton $M \times A_\varphi$ smaller, independently on the size of the BA $A_\varphi$. We start from noticing the following fact: if a state $s$ in $M \times A_\varphi$ is given by the combination of the states $s'$ in $M$ and $s''$ in $A_\varphi$, and if $s''$ is a deterministic decision state —that is, each label may match with at most only one successor of $s''$— then $s$ has at most the same amount of successor states as $s'$, no matter the number of successors of $s''$. From this fact, we conjecture that reducing the presence of non-deterministic decision states in the BA is likely to reduce the size of the final product in the average case, no matter if this produces bigger BA’s. (Notice that it is not always possible to reduce completely the presence of non-deterministic decision states, as not every LTL formula $\varphi$ can be translated into a deterministic BA, and even deciding whether the translation is possible belongs to EXPSPACE and is PSPACE-Hard [11].)

In order to explore the effectiveness of the above conjecture, we thus present a new approach in which we generate from each LTL formula a BA which is “as deterministic as possible”, in the sense that we try to reduce as much as we are able to the presence of non-deterministic decision states in the generated automaton. This is done by exploiting the idea of semantic branching, which has proved very effective in the domain of modal theorem proving [8].

The rest of the paper is structured as follows. In Section 2 we present some preliminary notions. In Section 3 we describe the main ideas of our approach. In Section 4 we describe the LTL to BA algorithm we have implemented. In Section 5 we present the results of an extensive empirical test. In Section 6 we conclude, describing also some future work. For lack of space, the correctness and completeness of the algorithm is proved in an extended technical report, which is available at [link]

2 Preliminaries

We use Linear Temporal Logic (LTL) with its standard syntax and semantics [2] to specify properties. Let $\Sigma$ be a set of elementary propositions. A propositional literal (i.e., a proposition $p$ in $\Sigma$ or its negation $\neg p$) is a LTL formula; if $\varphi_1$ and $\varphi_2$ are LTL formulae, then $\neg \varphi_1, \varphi_1 \land \varphi_2, \varphi_1 \lor \varphi_2, X \varphi_1, \varphi_1 U \varphi_2, \varphi_1 R \varphi_2$ are LTL formulae, where $X$, $U$ and $R$ are the standard “next”, “until” and “releases” temporal operators respectively. We see the familiar $\top$ (true), $\bot$ (false), $F \varphi_1$ (eventually $\varphi_1$) and $G \varphi_1$ (globally $\varphi_1$) as standard abbreviations of $p \lor \neg p$, $p \land \neg p$, $\top U \varphi_1$ and $\bot R \varphi_1$ respectively.

For every operator $op$ in $\{\land, \lor, X, F, G, U, R\}$, we say that $\varphi$ is an $op$-formula if $op$ is the root operator of $\varphi$ (e.g., $X(p U q)$ is an X-formula). We say that the occurrence of a subformula $\varphi_1$ in an LTL formula $\varphi$ is a top level occurrence if it occurs in the scope of only boolean operators $\neg, \land, \lor$ (e.g., $F p$ occurs at top level in $F p \lor X F q$, while $F q$ does not).
A Kripke Structure $M$ is a tuple $\langle S, S_0, T, \mathcal{L} \rangle$ with a finite set of states $S$, a set of initial states $S_0 \subseteq S$, a transition relation $T \subseteq S \times S$ and a labeling function $\mathcal{L} : S \rightarrow 2^\Sigma$, where $\Sigma$ is the set of atomic propositions.

A labeled generalized BA (LGBA) [6] is a tuple $A := \langle Q, Q_0, T, \mathcal{L}, D, \mathcal{F} \rangle$, where $Q$ is a finite set of states, $Q_0 \subseteq Q$ is the set of initial states, $T \subseteq Q \times Q$ is the transition relation, $D := 2^\Sigma$ is the finite domain (alphabet), $\mathcal{L} : Q \rightarrow 2^D$ is the labeling function, and $\mathcal{F} \subseteq 2^Q$ is the set of accepting conditions (fair sets). A run of $A$ is an infinite sequence $\sigma := \sigma(0), \sigma(1), \ldots$ of states in $Q$, such that $\sigma(0) \in Q_0$ and $T(\sigma(i), \sigma(i+1))$ holds for every $i \geq 0$. A run $\sigma$ is an accepting run if, for every $F_i \in \mathcal{F}$, there exists $\sigma(j) \in F_i$ that appears infinitely often in $\sigma$. An LGBA $A$ accepts an infinite word $\xi := \xi(0), \xi(1), \ldots \in D^\omega$ if there exists an accepting run $\sigma := \sigma(0), \sigma(1), \ldots$ so that $\xi(i) \in L(\sigma(i))$, for every $i \geq 0$. Henceforth, if not otherwise specified, we will refer to an LGBA simply as a Büchi automaton (BA).

Notice that each state in a Kripke structure is labeled by one total truth assignment to the propositions in $\Sigma$, whilst the label of a state in a BA represents a set of such assignments. A partial assignment represents the set of all total assignments/labels which entail it. We represent truth assignments indifferently as sets of literals $\{l_i\}_i$ or as conjunctions of literals $\bigwedge_i l_i$, with the intended meaning that a literal $p$ (resp. $\neg p$) in the set/conjunction assigns $p$ to true (resp. false).

Notationally, we use $\xi$ for representing an infinite word over $2^\Sigma$; $\xi(i)$ is the $i$-th element and $\xi_s$ is the suffix starting from $\xi(i)$. We use $\sigma$ for an infinite sequence of states (runs); $\sigma(i)$ is the $i$-th element and $\sigma_s$ is the suffix starting from $\sigma(i)$. We use $\mu$ for truth assignments. We use $\varphi, \psi, \vartheta$ for general formulae. We denote by $\text{succ}(s, A_{\varphi})$ ($\text{succ}(s, M)$) the set of successor states of the state $s$ in a BA $A_{\varphi}$ (Kripke structure $M$).

If $\mu$ is a truth assignment and $\varphi$ is an LTL formula, we denote by $\varphi[\mu]$ the formula obtained by substituting every top level literal $l \in \mu$ in $\varphi$ with $\top$ (resp. $\bot$ with $\bot$) and by propagating the $\top$ and $\bot$ values in the obvious ways. (E.g., $(p \lor X\varphi_1) \land (q \lor X\varphi_2)[\{p, \neg q\}] = X\varphi_2$.)

An elementary formula is an LTL formula which is either a constant in $\{\top, \bot\}$, a propositional literal or a $X$-formula. A cover for a set of LTL formulae $\{\varphi_k\}_k$ is a set of sets of elementary formulae $\{\{\vartheta_{ij}\}_j\}_i$ s.t. $\bigwedge_i \varphi_k \leftrightarrow \bigvee_i \bigwedge_j \vartheta_{ij}$. (Henceforth, we indifferently represent covers either as sets of sets or as disjunctions of conjunctions of elementary formulae.) A cover for $\{\varphi_k\}_k$ is typically obtained by computing the disjunctive normal form (DNF) of $\bigwedge_i \varphi_k$, considering $X$-subformulae as boolean propositions.

The general translation schema of an LTL formula $\varphi$ into a BA $A_{\varphi}$ works as follows [6]. First, $\varphi$ is written in negative normal form (NNF), that is, all negations are pushed down to literal level. Second, $\varphi$ is expanded by applying the tableau rewriting rules:

$$\varphi_1 U \varphi_2 \Rightarrow \varphi_2 \lor (\varphi_1 \land X(\varphi_1 U \varphi_2)), \quad \varphi_1 R \varphi_2 \Rightarrow \varphi_2 \land (\varphi_1 \lor X(\varphi_1 R \varphi_2))$$ (1)

until no $U$-formula or $R$-formula occurs at top level. Then the resulting formula is rewritten into a cover by computing its DNF. Each disjunct of the cover represents a state of the automaton: all propositional literals represent the label of the state—that is, the condition the input word must satisfy in that state—and the remaining $X$-formulae
Fig. 1. Product of a generic Kripke structure with a non-deterministic (up) and a deterministic (down) cover expansion of $\varphi := (p \lor X\varphi_1) \land (q \lor X\varphi_2)$.

represent the next part of the state —that is, the obligations that must be fulfilled to get an accepting run— and determine the transitions outcome from the state.

The process above is applied recursively to the next part of each state, until no new obligation is produced. This results into a closed set of covers, so that, for each cover $C$ in the set, the next part of each disjunct in $C$ has a cover in the set. Then $A_\varphi = \langle Q, Q_0, T, \mathcal{L}, D, F \rangle$ is built as follows. The initial states are given by the cover of $\varphi$. The transition relation is given by connecting each state to those in the cover of its next part. An acceptance condition $F_i$ is added for every elementary subformula in the form $\psi \mathcal{U} \vartheta$, so that $F_i$ contains every state $s \in Q$ such that $s \not\models (\psi \mathcal{U} \vartheta)$ or $s \models \vartheta$.

3 A New Approach

3.1 Deterministic and Non-deterministic Decision States

We say that two states are mutually consistent if their respective labels are mutually consistent, mutually inconsistent otherwise. We say that a state $s$ in a BA is a deterministic decision state if the labels of all successor states of $s$ are pairwise mutually inconsistent, a non-deterministic decision state otherwise. Intuitively, if $s$ is a deterministic decision state, then every label in the alphabet is consistent with (the label of) at most one successor of $s$. A BA is deterministic if its states are all deterministic decision states and if its initial states are pairwise mutually inconsistent.

We consider an LTL model checking problem $M \models \neg \varphi$, where $M$ is a Kripke structure and $\varphi$ is an LTL formula. $A_\varphi$ is the BA into which $\varphi$ is converted, and $M \times A_\varphi$ is the product of $M$ and $A_\varphi$. Each state $s$ in $M \times A_\varphi$ is given by the (consistent) pairwise combination $s' s''$ of some states $s'$ in $M$ and $s''$ in $A_\varphi$, and the successor states of $s$ are given by all the consistent combinations of one successor of $s'$ and one of $s''$: 
\[ succ(s, M \times A_\varphi) = \{ s'_i s''_i | s'_i \in succ(s', M), s''_i \in succ(s'', A_\varphi), s'_i s''_i \not\equiv \bot \}, \]
\[ |succ(s, M \times A_\varphi)| \leq |succ(s', M)| \cdot | succ(s'', A_\varphi)|, \]
where \( s's'' \) denotes the combination of the states \( s' \) and \( s'' \) and \( "s'_i s''_i \not\equiv \bot" \) denotes the fact that the combination of \( s' \) and \( s'' \) is consistent.

We make the following key observation: if \( s'' \) is a deterministic decision state, then each successor state of \( s' \) can combine consistently with at most one successor of \( s'' \), so that \( s \) has at most as many successor states as \( s' \). Thus (3) reduces to
\[ | succ(s, M \times A_\varphi) | \leq | succ(s', M) |. \]  

The above observation suggests to us the following heuristic consideration: in order to minimize the size of the product \( M \times A_\varphi \), we should try to make \( A_\varphi \) “as deterministic as we can” —that is, to reduce as much as we can the presence of non-deterministic decision states in \( A_\varphi \)— no matter if the resulting BA is greater than other equivalent but “less deterministic” BA’s.

**Example 1.** Consider the state \( s' \) of a Kripke structure \( M \) in Figure 1 (left) and its successor states \( s'_1, s'_2, s'_3 \) and \( s'_4 \) with labels \( \{ p, q, \ldots \}, \{ p, \neg q, \ldots \}, \{ \neg p, q, \ldots \} \) and \( \{ \neg p, \neg q, \ldots \} \) respectively. Consider the LTL formula \( \varphi := (p \lor X \varphi_1) \land (q \lor X \varphi_2) \) for some LTL subformulae \( \varphi_1 \) and \( \varphi_2 \). Consider the two covers of \( \varphi \):

\[ C_1 := \{ \{ p, q \}, \{ p, X \varphi_2 \}, \{ q, X \varphi_1 \}, \{ X \varphi_1, X \varphi_2 \} \}, \]
\[ C_2 := \{ \{ p, q \}, \{ p, \neg q, X \varphi_2 \}, \{ \neg p, q, X \varphi_1 \}, \{ \neg p, \neg q, X \varphi_1, X \varphi_2 \} \}, \]

which generate the two BA’s \( A \) in Figure 1 (center) respectively. In the first BA the state \( s'' \) is a non-deterministic decision state. Thus the successors of \( s's'' \) in \( M \times A \) are the consistent states belonging to the cartesian product of the successor sets of \( s' \) and \( s'' \). In particular, \( s'_1 \) matches with all successor states of \( s'' \), \( s'_2 \) matches with \( s''_2 \) and \( s''_4 \), \( s'_3 \) matches with \( s''_3 \) and \( s''_4 \), and \( s'_4 \) matches with \( s''_4 \). In the second BA \( s'' \) is a deterministic decision state. Thus, each successor of \( s' \) matches with only one successor of \( s'' \).

**Remark 1.** It is well-known (see, e.g., [11]) that converting a non-deterministic BA \( A \) into a deterministic one \( A' \) (when possible) may make the size of the latter blow up exponentially wrt. the size of the former in the worst case. This is due to the fact that each state \( s' \) of \( A' \) represents a subset of states \( \{ s_i \} \) of \( A \), so that \( |A'| \leq 2^{|A|} \), and hence \( |M \times A'| \leq |M| \cdot 2^{|A|} \), whilst \( |M \times A| \leq |M| \cdot |A| \). Thus, despite the local effect described above (4), one may suppose that globally our approach worsens the global performance.

We notice instead that \( L(s') \models \bigwedge_i L(s_i) \), so that the set of states in \( M \) matching with \( s' \) is a subset of the intersection of the set of states in \( M \) matching with each \( s_i \):

\[ \{ s^* \in M | s^* s' \not\equiv \bot \} \subseteq \bigcap_i \{ s^* \in M | s^* s_i \not\equiv \bot \}. \]

Thus, the process of determinization may increase the number of states in the BA, but reduces as well the number of states in \( M \) with which each state in the BA matches.
Example 2. Consider the LTL formula and the covers of Example 1. (Notationally, we denote by $C_{ij}$ the $j$th element of $C_i$.) Then $C_{21}, C_{22}, C_{23}$ and $C_{24}$ match with $1/4$ of the possible labels, whilst $C_{11}, C_{12}, C_{13}$ and $C_{14}$ match with $1/4, 1/2, 1/2$ and $1/1$ of the possible labels respectively.

3.2 Deterministic and Non-deterministic Covers

Let $\{\varphi_k\}_k$ be a set of LTL formulae in NNF, let $\varphi := \bigwedge_k \varphi_k$, and let $C := \{\{\vartheta_{ij}\}_j\}_i$ be a cover for $\varphi$. $C$ can be written as $\{\mu_i \cup \chi_i\}_i$ where $\mu_i := \{\vartheta_{ij} \in \{\vartheta_{ij}\}_j | \vartheta_{ij}$ prop. literal$\}$ and $\chi_i := \{\vartheta_{ij} \in \{\vartheta_{ij}\}_j | \vartheta_{ij}$ X-formula$\}$ are the set of propositional literals and X-formulae in $\{\vartheta_{ij}\}_j$ respectively. Thus

$$\varphi \leftrightarrow \bigvee_i (\mu_i \land \chi_i).$$

(8)

We say that a cover $C = \{\mu_i \cup \chi_i\}_i$ as in (8) is a deterministic cover if and only if all $\mu_i$’s are pairwise mutually inconsistent, non-deterministic otherwise.

Example 3. Consider the LTL formula and the covers of Example 1. $C_1$ is non-deterministic because, e.g., $\{p,q\}$ and $\{p\}$ are mutually consistent. $C_2$ is deterministic because $\{p,q\}, \{p, \neg q\}, \{\neg p, q\}$ and $\{\neg p, \neg q\}$ are pairwise mutually inconsistent.

In the construction of a BA, each element $\mu_i \land \chi_i$ in a cover $C$ represents a state $s_i$, where $\mu_i$ is the label of the state and $\chi_i$ is its next part (by abuse of notation, we henceforth call such a formula “state”). Thus, a deterministic cover $C$ represents a set of states whose labels are pairwise mutually inconsistent. Consequently, deterministic covers (when admissible) give rise to deterministic decision states.

3.3 Computing Deterministic Covers

As said in the previous sections, the standard approach for computing covers is based on the recursive application of the tableau rules (1) and on the subsequent computation of the DNF of the resulting formula. The latter step is achieved by applying recursively to the top level formulae the rewriting rule

$$\varphi' \land (\varphi_1 \lor \varphi_2) \Rightarrow (\varphi' \land \varphi_1) \lor (\varphi' \land \varphi_2)$$

(9)

and then by removing every disjunct which propositionally implies another one. As in [8], we call step (9) syntactic branching because it splits “syntactically” on the disjuncts of the top level $\lor$-subformulae. As noticed in [8], a major weakness of syntactic branching is that it generates subbranches which are not mutually inconsistent, so that, even after the removal of implicant disjuncts, the distinct disjuncts of the final DNF may share models. As a consequence, if the boolean parts of two disjuncts in a cover are mutually consistent, non-deterministic decision states are generated.

To avoid this fact we compute a cover in a new way. After applying the tableau rules, we apply recursively to the top level boolean propositions the Shannon expansion

$$\varphi \Rightarrow (p \land (\varphi[\{p\}]) \lor (\neg p \land (\varphi[\{\neg p\}]).$$

(10)
As in [8], we call step (10) semantic branching because it splits “semantically” on the truth values of top level propositions. The key issue of semantic branching is that it generates subbranches which are all mutually inconsistent [8]. Thus, after applying (10) to all top level literals in \( \varphi \), we obtain an expression in the form
\[
\bigvee_i (\mu_i \land \varphi[\mu_i]), 
\]
such that all \( \mu_i \)'s are all pairwise mutually inconsistent and \( \varphi[\mu_i] \) is a boolean combination of \( X \)-formulae. If all \( \varphi[\mu_i] \)'s are conjunctions of \( X \)-formulae, then (11) is in the form (8), so that we have obtained a deterministic cover. If not, every disjunct \( (\mu_i \land \varphi[\mu_i]) \) in (11) represents a set of states \( S_i \) such that all states belonging to the same set \( S_i \) have the same label \( \mu_i \) but different next-part, whilst any two states belonging to different sets \( S_i \)'s are mutually inconsistent.

As a consequence, the presence of non-unary sets \( S_i \) is a potential source of non-determinism. Thus, if this does not affect the correctness of the encoding (see below), we rewrite each formula \( \varphi[\mu_i] \) into a single \( X \)-formula by applying the rewriting rules:
\[
\begin{align*}
X\varphi_1 \land X\varphi_2 & \implies X(\varphi_1 \land \varphi_2), & (12) \\
X\varphi_1 \lor X\varphi_2 & \implies X(\varphi_1 \lor \varphi_2). & (13)
\end{align*}
\]
The result is clearly a deterministic cover. We call this step branching postponement because (13) allows for postponing the or-branching to the expansion of the next part.

**Example 4.** Consider the LTL formula and the covers of Example 1. The cover \( C_1 \) is obtained by applying syntactic branching to \( \varphi \) from left to right, whilst \( C_2 \) is obtained by applying semantic branching to \( \varphi \), splitting on \( p \) and \( q \). (As all \( \varphi[\mu_i] \)'s are conjunctions of \( X \)-formulae, no further step is necessary.)

Unfortunately, branching postponement is not always safely applicable. In fact, while rule (12) can always be applied without affecting the correctness of the encoding, this is not the case of rule (13). For example, it may be the case that \( X\varphi_1 \) and \( X\varphi_2 \) in (13) represent two states \( s_1 \) and \( s_2 \) respectively so that \( s_1 \) is in a fair set \( F_1 \) and \( s_2 \) is not, and that the state corresponding to \( X(\varphi_1 \lor \varphi_2) \) is not in \( F_1 \); if so, we may lose the fairness condition \( F_1 \) if we apply (13). This fact should not be a surprise: if branching postponement were always applicable, then we could always generate a deterministic BA from an LTL formula, which is not the case [11]. Our idea is thus to apply branching postponement only to those formulae \( \varphi[\mu_i] \) for which we are guaranteed it does not cause incorrectness, and to apply standard DNF otherwise. This will be described in detail in the next section.

To sum up, semantic branching allows for partitioning the next states into mutually inconsistent sets of states \( S_i \), whilst branching postponement, when applied, collapses each \( S_i \) into only one state. Notice that

- unlike syntactic branching, semantic branching guarantees that the only possible sources of non-determinism (if any) are due to the next-part components \( \varphi[\mu_i] \)'s.

No source of non-determinism is introduced by the boolean components \( \mu_i \)'s;
cover compute_cover(\(\varphi\))
  1. apply_tableau_rules(\(\varphi\));
  2. for each \(p\) occurring at top level in \(\varphi\) {
      \(\varphi := (p \land \varphi[\{p\}]) \lor (\neg p \land \varphi[\{-p\}]);\) // semantic branching on labels
      simplify(\(\varphi\)); // boolean simplification
  }
  3. \(\varphi := \bigvee_{i \in I}(\mu_i \land DNF(\varphi[\mu_i]));\)
  4. \(\varphi := \bigvee_{i \in I}(\mu_i \land \bigvee_{j \in J_i} X \land_{k \in K_{ij}} \psi_{ijk});\) // factoring out the \(X\) operators
  5. \(C^*(\varphi) := \bigvee_{i \in I,j \in J_i}(\mu_i \land X \psi_{ij});\) // \(\psi_{ij}\) being \(\land_{k \in K_{ij}} \psi_{ijk}\)
  6. \(C(\varphi) := \bot;\) // initialization of \(C(\varphi)\)
  7. for each \(i \in I\) {
      8. \(s_i := (\mu_i \land X \bigvee_{j \in J_i} \psi_{ij});\)
      9. \(Subs(s_i) := \bigvee_{j \in J_i}(\mu_i \land X \psi_{ij});\)
      10. if (Postponement_is_Safe(s_i)) then \(C(\varphi) := C(\varphi) \lor s_i;\) // postponement applied
          else \(C(\varphi) := C(\varphi) \lor Subs(s_i);\) // postponement not applied
  }
  11. return \(C(\varphi);\)

Fig. 2. The schema of the cover computation algorithm

- branching postponement reduces the number of states sharing the same labels even if it is applied only to a strict subset of the subformulae \(\varphi[\mu_i]\) in (11). Thus, also partial applications of branching postponement make the BA “more deterministic”.

4 The MoDELLA Algorithm

In the current state-of-the-art algorithms the translation from an LTL formula \(\varphi\) into a BA \(A_\varphi\) can be divided into three main phases:

1. Formula rewriting: apply a finite set of rewriting rules to \(\varphi\) in order to remove redundancies and make it more suitable for an efficient translation.
2. BA construction from \(\varphi\): build a BA with the same language of the input formula \(\varphi\).
3. BA reduction: reduce redundancies in the BA (e.g., by exploiting simulations).

In our work, we focus on phase 2. According to the new approach proposed in the previous section, we have conceived and implemented a new translation algorithm, called MoDELLA (More Deterministic LTL to Automata) which builds a BA from an LTL formula trying to apply branching postponement as often as it is able to.

4.1 The Basic Algorithm

The general schema of the BA construction in MoDELLA, in its basic form, is the standard one proposed in [6] and briefly recalled in Section 2. MoDELLA differs from previous conversion algorithms in two steps: the computation of the covers and the computation of the fair sets.
Computation of the cover. The function which computes the cover of the formula $\varphi$ is described in Figure 2. First, we apply, as usual, the tableau rewriting rules (1) (line 1). The formula obtained is a boolean combination of literals and $X$-formulae. After applying the semantic branching rules on labels (10), we get a disjunction of formulae in the form (11) (lines 2-5).

If now we applied branching postponement (12) and (13), denoting $\bigwedge_{k \in K_{ij}} \psi_{ijk}$ by $\psi_{ij}$, we would obtain the deterministic cover:

$$C^D(\varphi) := \{ \mu_i \land X \bigvee_{j \in J_i} \psi_{ij} \}_{i \in I}. \quad (14)$$

Unfortunately, as pointed out in section 3.3, branching postponement may affect the correctness of the BA. Thus, we apply it only in “safe” cases. First, for every disjunct $\mu_i \land \varphi[\mu_i]$ we temporarily compute $\text{DNF}(\varphi[\mu_i])$ and then we factor $X$ out of every conjunction in $\text{DNF}(\varphi[\mu_i])$ (lines 6-7). We obtain a temporary non-deterministic cover

$$C^*(\varphi) := \{ \mu_i \land X \psi_{ij} \}_{i \in I, j \in J_i}. \quad (15)$$

Notice that every state $s_i$ in $C^D(\varphi)$ is equivalent to the disjunction of $|J_i|$ states in $C^*(\varphi)$:

$$s_i = \mu_i \land X \bigvee_{j \in J_i} \psi_{ij} = \bigvee_{j \in J_i} (\mu_i \land X \psi_{ij}). \quad (16)$$

For every $i \in I$, we define the set of substates of $s_i$ as:

$$\text{Subs}(s_i) := \{ \mu_i \land X \psi_{ij} \}_{j \in J_i}. \quad (17)$$

($\text{Subs}(s_i)$ is the set $S_i$ in Section 3.3.) We extend the definition to every state $s^*$ of $C^*(\varphi)$ by saying that $\text{Subs}(s^*) := \{ s^* \}$.

Then, the cover $C(\varphi)$ is built in the following way (lines 10-16): for every $i \in I$, we add to $C(\varphi)$ $s_i$ if postponement is safe for $s_i$, $\text{Subs}(s_i)$ otherwise. $\text{Postpone\_is\_Safe}(s)$ decides if branching postponement is safe for a state $s$ according to a sufficient condition described in the following paragraphs.

Computation of fair sets. If $\mathcal{U}_\varphi$ is the set of $U$-formulae which are subformulae of $\varphi$, the usual set of accepting conditions is:

$$F^* := \{ F^*_{\psi U \vartheta} \mid \psi U \vartheta \in \mathcal{U}_\varphi \}, \quad (18)$$

$$F^*_{\psi U \vartheta} := \{ s \in Q \mid s \not\models \psi U \vartheta \text{ or } s \models \vartheta \}. \quad (19)$$

We extend these definitions as follows:

$$\mathcal{F} := \{ F_H \mid H \in 2^{\mathcal{U}_\varphi} \}, \quad (20)$$

$$F_H := \{ s \in Q \mid \text{there exists } \psi U \vartheta \in H \text{ s.t.} \} \quad (21)$$

for each $s^* \in \text{Subs}(s)$, $s^* \not\models \psi U \vartheta$ or for each $s^* \in \text{Subs}(s), s^* \models \vartheta_h$.

Notice that, if $|H| = 1$ and, for every $s \in Q$, $|\text{Subs}(s)| = 1$ (i.e. we have never applied branching postponement), this is the usual notion (i.e. $F_{\psi U \vartheta} = F^*_{\psi U \vartheta}$).
We say that the branching postponement is not safe for a state $s$ if there exists $F_H \in \mathcal{F}$ such that $s \not\in F_H$ and there exist $\psi U \vartheta \in \mathcal{H}$, $s^* \in \text{Subs}(s)$ such that $s^* \in F_{\psi U \vartheta}$.

With this condition we are guaranteed that if the BA $A_\varphi^*$ built without branching postponement has an accepting run $\sigma^*$ over a word $\xi$, then the correspondent run $\sigma$ of the BA $A_\varphi$ built with safe branching postponement is also accepting.

Example 5. Consider the LTL formula $\varphi := \mathcal{F}\mathcal{G}p$. After having applied the tableau rules and semantic branching on labels, we obtain $\varphi = (p \land (\mathcal{X}\mathcal{F}\mathcal{G}p \lor \mathcal{X}\mathcal{G}p)) \lor (\neg p \land \mathcal{F}\mathcal{G}p)$. If $s = (p \land \mathcal{X}(\mathcal{F}\mathcal{G}p \lor \mathcal{G}p))$, the branching postponement is not safe for $s$. Indeed, $\text{Subs} = \{(p \land \mathcal{X}\mathcal{F}\mathcal{G}p), (p \land \mathcal{X}\mathcal{G}p)\}$ and $(p \land \mathcal{X}\mathcal{G}p) \in F_{\mathcal{F}\mathcal{G}p}$ but $s \not\in F_{\mathcal{F}\mathcal{G}p}$. Thus, compute_cover produces the cover:

$$\{(p \land \mathcal{X}\mathcal{F}\mathcal{G}p), (p \land \mathcal{X}\mathcal{G}p), (\neg p \land \mathcal{X}\mathcal{F}\mathcal{G}p)\}.$$ (22)

4.2 Improvements

We describe some improvements to the basic schema of MoDeLLA described in the previous section. Most of them are adapted from known optimizations.

Pruning the fair sets. In the previous section, we have noticed that the basic version of MoDeLLA computes $2^{2|\mathcal{F}|}$ fair sets. Thus, in order to reduce this number, in the final computation of the fair conditions, $\mathcal{F}$, we apply the following simplification rules, which are a simple version of an optimization introduced in [12]:

- for all $F \in \mathcal{F}$, if $F = Q$ then $\mathcal{F} := \mathcal{F}\setminus\{F\}$,
- for all $F, F' \in \mathcal{F}$, if $F \subseteq F'$ then $\mathcal{F} := \mathcal{F}\setminus\{F'\}$.

Remark 2. Due to the existential quantifier in the definition (18) of $F_H$, for every formula $\psi U \vartheta \in \mathcal{H}$, we have that $F_{\{\psi U \vartheta\}} \subseteq F_H$. For this reason, after the above fair sets pruning, MoDeLLA will keep only those accepting condition $F_H$ for which $\mathcal{H}$ is a singleton. Thus, we obtain that $|\mathcal{F}| \leq |\mathcal{U}_\varphi|$, as in the usual construction.

Merging states. After computing a cover, if two states $s_1 = (\mu_1, \chi), s_2 = (\mu_2, \chi)$ have the same next part $\chi$ and satisfy the following property:

for all $\psi U \vartheta \in \mathcal{U}_\varphi$,

(1) for all $s_1^* \in \text{Subs}(s_1), s_1^* \models \psi U \vartheta$ ⇔ (for all $s_2^* \in \text{Subs}(s_2), s_2^* \models \psi U \vartheta$) and

(2) for all $s_1^* \in \text{Subs}(s_1), s_1^* \models \vartheta$ ⇔ (for all $s_2^* \in \text{Subs}(s_2), s_2^* \models \vartheta$),

then we substitute them with $s = (\mu_1 \lor \mu_2, \chi)$ where $\text{Subs}(s) := \text{Subs}(s_1) \cup \text{Subs}(s_2)$. Notice that for every $F \in \mathcal{F}$, we have $s_1 \in F \iff s_2 \in F \iff s \in F$. This technique is a simpler version of the one introduced in [7], which however applies the merging only after moving labels from the states to the transitions.

Example 6. Consider the formula of Example 5 and the cover produced by the basic version of MoDeLLA. After merging the states with the above technique, the cover (22) becomes $\{(\top \land \mathcal{X}\mathcal{F}\mathcal{G}p), (p \land \mathcal{X}\mathcal{G}p)\}$. Notice that the labels $\top$ and $p$ of the two states are mutually consistent so that the BA is still non-deterministic. However, we have reduced the number of states without increasing the non-determinism. ☐
5 Empirical Results

MoDELLA is an implementation in C of the algorithm described in Section 4. It implements only phase 2, so that it can be used as kernel of optimized algorithms including also formula rewriting (phase 1) and BA reduction (phase 3). (Indeed, we believe our technique is orthogonal to the rewriting rules of phase 1 and to BA reductions.) We extensively tested MoDELLA in comparison with the state-of-the-art algorithms. Unlike, e.g., [1,12,5,7], we did not consider as parameters for the comparison the size of the BA produced, but rather the number of states and transitions of the product $M \times A_\varphi$ between the BA and a randomly-generated Kripke structure. To accomplish this, we used LBTT 1.0.1 [13], a randomized testbench which takes as input a set of translation algorithms for testing their correctness. In particular, LBTT gives the same formula (either randomly-generated or provided by the user) to the distinct algorithms, it gets their output BA’s and it builds the product of these automata with a randomly-generated Kripke structure $M$ of given size $|M|$ and (approximated) average branching factor $b$. LBTT provides also a random generator producing formulae of given size $|\varphi|$ and maximum number of propositions $P$.

To compare MoDELLA with state-of-the-art algorithms, we provided interfaces between LBTT and WRING 1.1.0 [12,9] and between LBTT and TMP 2.0 [3,4]. Since LBTT computes the direct product between the BA and the state space, the size of the product is not affected by the number of fair sets of the BA. Thus, to get more reliable results, we have dealt only with degeneralized BA, and we have applied a simple procedure described in [6] to convert a BA into a Büchi automata with a single fair set.

We have run LBTT on three PCs Dual Processor with 2GB RAM on Linux RedHat. All the tools and the files used in our experiments can be downloaded at http://www.science.unitn.it/~stonetta/modella.html.

5.1 Comparing Pure Translators

In a first session of tests, we wanted to verify the effectiveness of MoDELLA as a pure “phase 2” translator. Thus, we compared MoDELLA with “pure” translators (no formula rewriting, no BA reduction), i.e. with GPVW [6], LTL2AUT [1] and WRING [12] with rewriting rules and simulation-based reduction disabled (WRING(2) henceforth). Notice that TMP uses LTL2AUT as phase 2 algorithm [3]. For reasons which will be described in the next section, we run also a version of MoDELLA without the merging of states (MS) optimization of Section 4.2 (which we call MoDELLA–MS henceforth).

We fixed $|M|$ to 5000 states and we made $b$ grow exponentially in $\{2, 4, 8, 16, 32, 64\}$. We did four series of tests: 1) tests with 200 random formulae with $|\varphi| = 15$ and $P = 4$; 2) tests with 200 random formulae with $|\varphi| = 15$ and $P = 8$; 3) tests on the 27 formulae proposed in [12]; 4) tests on the 12 formulae proposed in [3]. For every formula $\varphi$, we tested both $M \models \varphi$ and $M \models \neg \varphi$. The results are reported in Figure 3. (In the fourth series, the run of GPVW and LTL2AUT were stopped for $b \geq 16$ because they caused a memory blowup.)

\[2\] For GPVW and LTL2AUT, we have used the reimplementation provided by WRING.
Fig. 3. Performances of the pure “phase 2” algorithms. X axis: approximate average branching factor of \( M \). Y axis: mean number of states (left column) and of transitions (right column) of the product \( M \times A_\varphi \). 1st row: 400 random formulae, 4 propositions; 2nd row: 400 random formulae, 8 propositions; 3rd row: 24 formulae from [12]; 4th row: 54 formulae from [3].
Fig. 4. Same experiments as in Figure 3, adding phases 1 and 3 to the pure “phase 2” algorithms.
Comparing the plots in the first column (number of states of $M \times A_\varphi$) we notice that (i) GPVW and LTL2AUT are significantly less performing than the other algorithms; (ii) MoDELLA performs better than WRING(2) in all the test series; (iii) even with MS optimization disabled, MoDELLA performs mostly better than WRING(2).

Comparing the plots in the second column (number of transitions of $M \times A_\varphi$) we notice that WRING(2) performs much better than LTL2AUT and GPVW, and that both MoDELLA and MoDELLA–MS perform always better than WRING(2). In particular, the performance gaps are very relevant in the fourth test series.

5.2 Comparing Translators with Rewriting Rules and Simulation-Based Reduction

In a second section of tests, we investigated the behaviour of MoDELLA as the kernel of a more general algorithm, embedding also the rewriting rules (phase 1) and the simulation-based reduction (phase 3) of WRING and TMP. This allows us for investigating the effective “orthogonality” of our new algorithm wrt. the introduction of rewriting rules and of simulation-based reduction.

First, we applied to our algorithm the rewriting rules described in [12] and interfaced MoDELLA–MS with the simulation-based reduction engine of WRING. Unfortunately, since WRING accepts only states labeled with conjunctions of literals, we could interface WRING only with MoDELLA–MS and not with the full version of MoDELLA. (We denote the former as MoDELLA–MS+WRING(13) henceforth.) Second, we applied to MoDELLA the rewriting rules described in [3] and the simulation-based reduction described in [4] which are respectively the phase 1 and the phase 3 of TMP. (We call this enhanced version of our algorithm MoDELLA+TMP(13) henceforth.) Finally, we implemented the optimization technique described in [7]. When we enable this technique, together with the rewriting rules and the TMP’s automata reduction, we refer to it as MoDELLA+ALL.

We run the tests with the same parameters of the first session of tests, obtaining the results of Figure 4. By looking at the plots, one can observe the following facts for both the columns (number of states and number of transitions of $M \times A_\varphi$): (i) if compared with the correspondent phase 2, MoDELLA–MS+WRING(13) and MoDELLA+TMP(13) benefit a lot respectively from WRING’s and TMP’s rewriting rules and simulation-based reduction, although slightly less than WRING and TMP theirselves do; (ii) MoDELLA–MS+WRING(13) and MoDELLA+TMP(13) perform mostly better respectively than WRING(123) and than TMP, although the gap we had with “pure” algorithms is reduced; (iii) MoDELLA+ALL performs better than all the others, except with the third test series where MoDELLA–MS+WRING(13) is the best performer.

6 Conclusions and Future Work

In this paper we have presented a new approach to build BA from LTL formulae, which is based on the idea of reducing as much as possible the presence of nondeterministic decision states in the automata; we have motivated this choice and presented a new conversion algorithm, MoDELLA, which implements these ideas; we have presented an
extensive empirical test, which suggests that MoDELLA is a valuable alternative as a core engine for state-of-the-art algorithms.

We plan to extend our work on various directions. From the implementation viewpoint, we want to implement in MoDELLA the simulation-based reduction techniques presented in [12] in order to have a tool which exploits the power of all state-of-the-art automata reductions. From an algorithmic viewpoint, we want to investigate new optimizations steps ad hoc for our approach. From a theoretical viewpoint, we want to investigate more general sufficient conditions for branching postponement.

Another interesting research direction, though much less straightforward, might be to investigate the feasibility and effectiveness of introducing semantic branching in the alternating-automata based approach of [5].

Finally, we would like to test the performance (wrt. time and memory consuming) of state-of-the-art LTL model checkers, e.g. SPIN [10], on real-world benchmarks by using the automata built by MoDELLA.

References

An Optimized Symbolic Bounded Model Checking Engine

Rachel Tzoref, Mark Matusevich, Eli Berger, and Ilan Beer
IBM Haifa Research Lab, Haifa, Israel
rachelt@il.ibm.com

Abstract. It has been shown that bounded model checking using a SAT solver can solve many verification problems that would cause BDD based symbolic model checking engines to explode. However, no single algorithmic solution has proven to be totally superior in resolving all types of model checking problems. We present an optimized bounded model checker based on BDDs and describe the advantages and drawbacks of this model checker as compared to BDD-based symbolic model checking and SAT-based model checking. We show that, in some cases, this engine solves verification problems that could not be solved by other methods.

1 Introduction

As the use of formal verification in industrial settings continues to grow [3,5], contemporary research seeks diverse ways to solve the “state explosion” problem inherent in model checking. In recent years, the traditional methods of BDD-based symbolic model checking [10] have been augmented by methods which are based on Boolean Satisfiability (SAT) [13,11] that can solve the Bounded Model Checking (BMC) [7] problem. Unlike the model checking problem that, given a model $M$ and a property $\phi$, tries to determine if $M \models \phi$, the BMC problem restricts itself to determining whether $M \models \phi$ on the first $k$ iterations of $M$. The class of properties that can be checked this way is smaller than the one handled by model checking, as described in Section 2.

The BMC problem is usually solved by reducing the model and the bug detection circuit, unfolded $k$ cycles, to a propositional formula, and then solving this formula using a SAT solver. However, other approaches are also applicable. Bertacco and Olukotun [6] suggest a BDD-based algorithm that unfolds the sequential circuit $k$ times in order to calculate the values of signals on the first $k$ cycles. This algorithm is based on symbolic simulation methods [8], and has some advantages over the SAT approach described in [7]. The main advantage is that the unfolded structure uses BDD variables only for inputs to the model. Therefore, when the number of inputs is small compared to the number of state variables, as in the case of datapath, this approach is advantageous. In this paper, we describe an optimized BDD-based BMC engine, based on this unfolded structure.

2 Basic Concepts

We consider bounded model checking to be the following problem: given a nondeterministic Finite State Machine (FSM) $M$, $n$ RCTL [4] properties $(\phi_1, \ldots, \phi_n)$ and a
An FSM can be defined by the following 6-tuple $(CC_0, I_0, CC, I, S, P)$:

- $CC_0$ is combinatorial logic that generates the initial states of the flip-flops.
- $I_0 = (i_{(1,0)}, \ldots, i_{(t,0)})$ is an ordered set of Boolean inputs to $CC_0$.
- $CC$ is combinatorial logic that generates the next state function of the flip-flops.
- $I = (i_1, \ldots, i_q)$ is an ordered set of Boolean inputs to $CC$.
- $S = (s_1, \ldots, s_r)$ is a set of symbols representing the outputs of the flip-flops.
- $P = (p_1, \ldots, p_n)$ is an ordered set of Boolean outputs representing the properties $(\phi_1, \ldots, \phi_n)$.

$(CC_0, I_0, CC, I, S, P)$ is illustrated in Figure 1.

### 3 BDD-Based BMC

This section describes how an FSM is transformed into a combinatorial circuit that represents the first $k$ cycles of the FSM, as well as the computation process applied to the combinatorial circuit in order to evaluate the properties in the first $k$ cycles.

#### 3.1 Circuit Unfolding

The unfolding process transforms an FSM, which is a sequential circuit, into an iterative logic array, as depicted in Figure 2. The combinatorial logic, inputs, and properties of the FSM are duplicated $k$ times, and the flip-flops are replaced by wires connecting the copies of the different iterations. Therefore, the $S$ parts do not actually exist; they are depicted only to indicate where the flip-flops existed previously. Assuming there are no combinatorial loops in $CC_0$ and $CC$ of the original FSM, there are no combinatorial loops in the combinatorial circuit resulting from the unfolding process.
Definition 1 (Closed machine). The circuit that results from the unfolding process is called a closed machine.

We use the netlist representation of the unfolded FSM as our basic data structure. This data structure is referred to as the circuit.

3.2 Verification Using the BDD-Based BMC

We use the following terms in the description of the computation process:

- **Cycle** is the pair \((S_{j-1}, (CC_j \cup I_j \cup P_j))\) (corresponds to cycles in calculations of FSM). This cycle is denoted as cycle number \(j\).
- \(p_{m,j}\) is the gate that represents property \(p_m\) in cycle \(j\).
- \(g_j\) represents the replication of a certain gate \(g\) of the FSM in cycle \(j\).
- The **cone** of a gate \(g_j\) is the set of all gates on which \(g_j\) topologically depends.
- A **fanin** of a gate \(g_j\) is a gate \(f_j\) whose output is a direct input to \(g_j\).
- A **fanout** of a gate \(g_j\) is a gate \(h_j\) that has a direct input, which is the output of \(g_j\).

Definition 2 (Gate function). The function of a gate \(g_j\) (denoted \(f[g_j]\)) is the parametric representation of the gate \(g_j\) depending on \((I_0, \ldots, I_k)\). \(f[g_j]\) operates on all of the FSM inputs \((I_0 \times \ldots \times I_k)\) and goes to \(\{0, 1\}\), \(f : B^{t+k} \rightarrow B\).

Definition 3 (Frontier). The frontier \(F\) is a set of gates where for each gate \(g \in F\), two conditions hold: all of the fanins of \(g\) have a calculated BDD and the BDD of \(g\) is not yet calculated.

The initial frontier is built by going backwards from the properties, until we reach primary inputs or gates for which there is a calculated BDD. (These gates were in the cone of influence of properties in previous cycles.) The fanouts of these inputs and gates compose the initial frontier. The frontier may change whenever we calculate a BDD of a gate.

For each gate \(p_{m,j}\) of \(p_{(1,1)}, \ldots, p_{(n,k)}\), we build the BDD that represents the function of the gate \(p_{m,j}\). If the BDD of \(p_{m,j}\) equals the function \(true\), then \(p_m\) holds in cycle \(j\). Otherwise, we extract out of the BDD a non-satisfying assignment as a counter example. In order to calculate the BDD of \(p_{m,j}\), we must first calculate the BDDs in the cone of \(p_{m,j}\). When building the BDD of \(g_j\), we use the BDDs of all of the fanins of \(g_j\). Therefore, the structure of the closed machine dictates a partial order of calculation on the gates. Note that different copies of the same gate \(g\) in different cycles may have different BDDs.

3.3 Advantages and Drawbacks of BDD-Based BMC

The BDD-based BMC approach uses a parametric representation of the state of the flip-flops, depending only on the inputs of the model. That is, the set of reachable states in cycle \(j\) is represented by a collection of BDDs representing \(f[g_j]\), for all gates \(g_j\) that represent outputs of the flip-flops in cycle \(j\). As a result, the BDD-based BMC is only sensitive to the amount of nondeterminism in the model. In contrast, symbolic
model checking and SAT solvers represent the states by state variables. Therefore, they are sensitive both to the amount of nondeterminism and to the number of state variables. In addition, the functions computed by the BDD-based BMC describe the natural functionality of the original model. Symbolic model checking computes a characteristic representation of the reachable states, which is randomly shaped, and its BDD tends to be bigger than those of the natural functions. Another advantage versus SAT is that multiple properties are computed in the same run, without repeating calculations of overlapping cones of influence of these properties. SAT solvers need to backtrack after a counter example is found and thus repeat parts of the calculations. The main drawback of our approach is its sensitivity to the number of calculated cycles. In each cycle, \( q \) variables are added and therefore the complexity of calculation increases as the cycles advance. As a result of these advantages and drawbacks, the BDD-based BMC approach performs better than the other methods in wide and shallow circuits (i.e., circuits that have many state variables, but their state space can be covered by a few cycles) and in circuits with many state variables, but with a low amount of nondeterminism.

Due to the static unfolding, the circuit is amenable to static BDD variable ordering, based on its topology. In many cases, this order is sufficient for calculation without a need for dynamic BDD reordering. We can also simplify the evaluation of the properties by performing easy calculations before the difficult ones. Our measure of difficulty is the expected BDD size of the gate, which we estimate according to the sizes of the input BDDs. We traverse first the easier calculations paths, and in many cases, as a result of constant propagation during the computation process, some more difficult calculations that were not yet performed become redundant.

## 4 Open Machine

We will now introduce a variation of the unfolding algorithm, which enables powerful optimizations to the BDD-based BMC engine, as will be described later. Additionally, this variation enables us to prove properties in some cases, despite the fact that we are calculating only a bounded number of cycles.

### Definition 4 (Open machine). An open machine is a closed machine whose logic \( CC_0 \) is replaced by free inputs, as depicted in Figure 3.

These free inputs are denoted with \( I_0' \). Note that the number of inputs in \( I_0' \) may be different from the number of inputs in \( I_0 \).

### 4.1 The Difference between the Open Machine and the Closed Machine

Let \( f^{op}[g_j] \) denote a gate function in the open machine, and \( f^{cl}[g_j] \) denote a gate function in the closed machine.

### Definition 5 (Equivalence between gate functions). Two gate functions \( f[g_x] \) and \( f[g'_y] \) are equal, if and only if the BDD of \( g_x \) equals the BDD of \( g'_y \). This equivalence is denoted by \( f[g_x] \equiv f[g'_y] \).

Note that \( f^{op}[g_j] \) is not necessarily equal to \( f^{cl}[g_j] \).
Theorem 6. If \( f^{\text{op}}[g_x] \equiv f^{\text{op}}[g'_y] \), then \( \forall j \geq 0 \, f^{\text{op}}[g_{x+j}] \equiv f^{\text{op}}[g'_{y+j}] \) and \( f^{\text{cl}}[g_{x+j}] \equiv f^{\text{cl}}[g'_{y+j}] \).

For proof see [14]. Note that a closed machine version of Theorem 6 does not hold, i.e., if \( f^{\text{cl}}[g_x] \equiv f^{\text{cl}}[g'_y] \), we cannot conclude anything about other gates in the closed machine or in the open machine.

Corollary 7 It stems from Theorem 6 that if \( f^{\text{op}}[g_x] \equiv b, b \in \{0, 1\} \), then \( \forall j \geq 0 \, f^{\text{op}}[g_{x+j}] \equiv b \) and \( f^{\text{cl}}[g_{x+j}] \equiv b \).

4.2 Uses of the Open Machine

Proving Properties

In some cases, Theorem 6 gives us the ability to prove properties, despite the fact that we are calculating a bounded number of cycles. We prove \( \phi_m \) by calculating the BDD of \( p_{m,j} \) for all \( j = 1, \ldots, k \) in the open machine. Calculation is performed in the same manner described for the closed machine. If we find that the BDD of \( p_{m,j} \) equals true for some \( 1 \leq j \leq k \), we can conclude that \( \phi_m \) holds both in the open machine and in the closed machine for all cycles \( \geq j \). As described in [9], we can prove a property in a bounded circuit in this way only if the circuit is \( k \)-definite in respect to the property (i.e., the property in each cycle depends only on inputs of at most the last \( k \) cycles).

While the method in [9] is performed only in order to try and prove properties, we use a more general characteristic of the open machine (introduced in Theorem 6) mainly for optimizations, as described in the next subsection.

An induction-based algorithm, based on a SAT solver, is suggested in [12] for proving safety properties. We chose a different approach in order to accommodate large, real-world, circuits. Our method is suitable only for a subset of the circuits for which the method in [12] is suitable. However, our method can be efficiently implemented using the BDD-based BMC.

Optimizations Based on the Open Machine

Before applying the computation process to the closed machine, we perform two powerful optimizations that simplify further calculations, based on the open machine:
1. **Constant propagation.** There are constant signals in the FSM that originate in restrictions of the environment on the design’s inputs. When we find that \( g_i \) is the constant \( b \) in the open machine, we automatically propagate \( b \) to all \( g_j \), for \( j \geq i \), both in the open machine and in the closed machine, according to Corollary 7. Due to the special data structure, described later, the time complexity of the propagation is independent of \( k \).

2. **Logical equivalence.** If a gate \( g \) is \( k \)-definite, the copy of \( g \) in cycle \( j \) has the same BDD as the copy of \( g \) in cycle \( j + k \), for all \( j \geq 1 \). Another case in which different gates have equal BDDs occurs as a result of logic duplication in the original model. We find in the open machine sets of gates with equal BDDs and gather them in equivalence sets. Each equivalence set actually represents up to an infinite number of equivalence sets, since the next cycle replications of the gates in each equivalence set are also an equivalence set. When the computation process runs in the closed machine, we only calculate one BDD for each equivalence set.

**Data Structure for the BDD-Based BMC**

Our data structure represents both the closed machine and the open machine. While our implementation of the data structure conceptually allows us to perform operations on each of the \( 2 \times k \) replications of each gate \( g \) at any time, initially there is only one object (whose size is independent of \( k \)) in the data structure for every gate \( g \) of the original FSM. This representation may change as various operations are performed on the circuit. As a result, the common size of the objects representing the replications of \( g \) may grow and, in the worst case, depend on \( k \). In practice, most of the data structure remains folded during the entire run. When an operation is performed on a gate \( g_j \) in the open machine, it also applies to all of the relevant gates of the subsequent cycles, according to Theorem 6. In most cases, the time complexity is independent of \( k \), since all of the relevant gates are a single object in the data structure.

5 **Under-Approximation**

Despite the simplification methods and despite applying reordering algorithms, the BDDs can still grow as the cycles advance and may eventually outgrow the memory resources. One solution is to perform under-approximations, although this compromises on coverage. Each under-approximation is performed by choosing an input \( i_l \in I_j : 0 \leq j \leq k \) (denoted \( i_{l,j} \)) and setting it to a constant value \( b \in \{0, 1\} \) for the rest of the run. Next, we simplify the already calculated BDDs accordingly. The heuristics we use to choose \( i_{l,j} \) and \( b \) try to find the best variable assignment that will balance between causing a significant reduction in the BDDs sizes and leaving many behaviors in the scope of the calculation. The heuristics also take into account that if \( i_{l,j} \) was set to \( b \) and we are performing a new under-approximation, then we prefer not to choose any of the inputs \( i_{l,(j+t)} \) for \( t \neq 0 \), or if we choose one of them, then set it to \( \neg b \). In this way, we degenerate the behavior of an input only in a specific cycle, rather than for the entire run. Examples of heuristics for choosing \( i_{l,j} \) and \( b \) appear in [14]. Running the computation process with under-approximations is especially useful for finding bugs that, on one hand occur after many cycles, and therefore an exhaustive search would
be difficult, and on the other hand are quite common (occur for many possible sets of inputs) and therefore can be found even when the search is partial.

We also implemented a mode that combines under-approximations with backtracking, to perform exact evaluation of the properties. In this mode, whenever reaching the cycle bound, we backtrack and compute parts of the search space which were neglected as a result of previous under-approximations.

### 6 Experimental Results

We implemented the optimized BDD-based BMC in the framework of IBM’s model checker RuleBase [2], and used the CUDD package [1] for BDD calculations. The table in Figure 4 presents the results of our engine versus an IBM zChaff-based SAT solver. The engines ran on real-life examples taken from various projects. Both engines operated using default configurations. We set a timeout of 36000 seconds, memory limit of 1G, and a bound of 100 cycles.

The number of inputs, flip-flops, and properties is shown for each circuit. The total run-time is in seconds and the memory is in MB. The cycles column is the number of cycles the engine calculated until reaching either the cycle bound, timeout, or memory limit, or until all properties failed. The res column displays whether the engine managed to disprove the properties. The # app column displays the number of under-approximations performed during the computation process. We also ran several symbolic model checkers on these examples, all of these outgrew memory resources on design1 to design5, while computing the set of initial states. When under-approximations were used, we report, in parentheses, the time and memory consumption of the run without under-approximations. These results demonstrate the significant decrease in time and memory demands our under-approximations achieve.

(*) The SAT solver reached timeout after 70 cycles in each of the 15 runs.

(**) The SAT solver reached timeout while constructing the CNF formula. Using a SAT expert advice, we ran the SAT solver without the bounded cone of influence reduction. With this configuration, it found a counter example for the first property after 189 seconds and for the second property after 139 seconds — about 10 times slower than the unfolding engine (combining the run-time of the two properties).

The table in Figure 5 reports the run-time in seconds of the constant propagation performed on the FSM unfolded 100 cycles. The open and closed machine column
presents the run-time of constant propagation, as it is performed in our optimized engine — first on the open machine (according to Corollary 7) and then on the closed machine. Note that constant propagation on the open machine changes both the topology of the open machine and of the closed machine. The only closed machine column presents constant propagation as it would have been performed in a standard implementation (i.e., only on the closed machine). We conclude that there is a significant decrease in run-time when performing constant propagation on the open machine. Note, that in many cases, constant propagation on the closed machine alone dominates the running time and may even cause timeout.

The table in Figure 6 reports the run-time for each circuit in seconds of unfolding the FSM $k$ cycles, out of the netlist representation of the original FSM, for $k = 100$ and for $k = 300$. This table demonstrates the fact that, due to our data structure, the circuit unfolding time does not have a linear dependency on the cycle-bound $k$.

Acknowledgments. The authors would like to thank Danny Geist and Cindy Eisner for their valuable contribution to this paper.

References

Constrained Symbolic Simulation with Mathematica and ACL2

Ghiath Al Sammane, Diana Toma, Julien Schmaltz, Pierre Ostier, and Dominique Borrione

TIMA Laboratory, VDS Group, Grenoble, France
http://tima.imag.fr

Abstract. We use symbolic simulation for the verification of high level circuit specifications. We combine Mathematica for algebraic computation and ACL2 for branching decision to increase the efficiency of the method.

1 Introduction

Symbolic simulation, proposed as early as 79 by J.Darringer, is intermediate between conventional simulation and mathematical reasoning, to verify abstract, pre-RTL design specifications. Instead of simulating a design with numerical values, symbolic inputs are given to the symbolic simulator, which produces an algebraic expression for the memory and output variables, as a function of the initial state and of the inputs. These difficulties arise: (1) the symbolic expressions may become exponentially large in the number of simulation cycles; (2) in the presence of conditional statements, when the condition is a symbolic term, all alternative paths must be explored. The simulator generates a simulation tree, which may also grow exponentially; (3) the automatic simplification and reduction of the computed symbolic expressions is needed, else the outputs of symbolic simulation are unreadable.

Previous works have tackled one or more of the above difficulties: e.g. GSTE [9] at switch and gate-level, PVS [7] and ACL2 [4] at the initial abstract design levels. To simplify symbolic simulation by reducing algebraic expressions and controlling the expansion of the simulation tree, most proposed solutions use an automated reasoning tool.

A systematic approach for using ACL2 as a symbolic simulation engine was proposed by J. Moore [6]. On this base, the semantics of a subset of VHDL [3] were defined in ACL2 in order to simulate a VHDL design symbolically [2]. In this paper we propose a different approach based on the separation of algebraic computation and branching decision. We combine Mathematica [8] a computer algebra system and ACL2 [4] an automatic theorem prover to perform what we call constrained symbolic simulation. This association increases the efficiency of the symbolic simulation by using two tools, each one being powerful in its domain.
2 Overview of the Method

Figure 1 shows the overall combined verification system taking VHDL inputs. The front-end compiler performs syntactic and static semantics checks, and serves as common starting point to all EDA tools. NIF is an intermediate format developed by our group. The elaboration of the Mathematica model, called M-Code, is performed on the NIF file. During this step, data type restrictions are extracted as constraints. Before starting the simulation, the user, who is not necessary a proof expert, can add constraints on the inputs. Those are inequalities or equalities between expressions composed of design variables or input signals and arithmetic operators \((+, -, /, \times)\). M-code and constraints are submitted to Mathematica for \(n\) simulation cycles, \(n\) is user defined. During simulation, symbolic expressions are simplified using rewrite rules. Standard Mathematica simplification rules are algebraic axioms like \((x - x \rightarrow 0)\) and arithmetic simplifications like \((n + n \rightarrow 2n)\), for terms defined on real or integer types. VHDL simplification rules were defined by us for the hardware types unknown to Mathematica (e.g. Bit). To reduce the simulation tree, whenever path conditions are encountered, ACL2 is called as a reasoning engine. ACL2 evaluates a given condition under simulation constraints using pre-proved theorems. Depending on the ACL2 answer, Mathematica chooses a path. After each simulation cycle, the values of all variables and signals are stored in a file. This is the result of the constrained symbolic simulation of the VHDL description.
Table 1. Example of stabilizing concurrent assignments

<table>
<thead>
<tr>
<th>cycle</th>
<th>VHDL expressions</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a &lt;=(d and not(c)) or (b and c);</td>
</tr>
<tr>
<td></td>
<td>b &lt;=(a and not(c)) or (d and c);</td>
</tr>
<tr>
<td>2</td>
<td>a &lt;=(d and not(c)) or ((a and not(c)) or (d and c) and c);</td>
</tr>
<tr>
<td></td>
<td>b &lt;=((d and not(c)) or (b and c) and not(c)) or (d and c);</td>
</tr>
<tr>
<td>3</td>
<td>a &lt;= d;</td>
</tr>
<tr>
<td></td>
<td>b &lt;= d;</td>
</tr>
</tbody>
</table>

3 Modeling VHDL in Mathematica

The VHDL supported by our tool is based on the standard subset for Register Transfer Level (RTL) synthesis [3], enlarged with full arithmetic types. Combinational logic and clock-edge synchronized sequential logic may be described using a behavioral, structural or dataflow style, or any combination thereof. A model is a component, i.e. an entity coupled with its associated architecture.

Due to the absence of explicit time [3], the simulation algorithm is simplified, as described in [2]: the driver of a signal only holds one current and one next value, since right hand side waveforms are a single zero delay expression (the after clause is not recognized in the subset). Concurrent signal assignments and combinational processes are stabilized by performing delta computation cycles between each two clock simulation cycles. In this context, the model is observable only at the clock cycle level.

In the M-code, a VHDL component built from a (entity Ent, architecture A) pair is modeled by a Mathematica function named: EntA. Its arguments are all the objects declared in the corresponding entity-architecture: input, output and local signals, and local variables. All are named Mathematica blank patterns, i.e. no data type is defined. However, the information about data types is not lost: it will serve as simulation constraints.

Two Mathematica variables are necessary to model each local or output signal: one for the current value, passed to EntA as argument; one for the next value, declared as temporary variable inside the body of EntA. Input signals, that cannot be modified in the architecture, only have a current value.

The body of EntA is the Mathematica model for the VHDL statements inside the architecture. All processes are flattened inside the body. To eliminate simulation delta cycles, we perform a symbolic fixed point computation during M-code generation. We repeat the execution, symbolically and sequentially, of all concurrent signal assignments, and simplify the expressions, until they stabilize. The next values of all signals can then be computed in one step.

Table 1 displays the three cycles needed to stabilize the symbolic value for the concurrent assignments shown at cycle 1. In the M-code symbolic delta cycles are no more needed. The corresponding M-code for this example is:

```mathematica
NextSig[a,d];
NextSig[b,d];
```
Table 2. Examples of M-code assignment functions

<table>
<thead>
<tr>
<th>VHDL</th>
<th>M-code</th>
</tr>
</thead>
<tbody>
<tr>
<td>A &lt;= d + g;</td>
<td>NextSig[A, Plus[d, g]];</td>
</tr>
<tr>
<td>V := 2 + j;</td>
<td>ChangeVar[V, Plus[2, j]];</td>
</tr>
<tr>
<td>Q := V + 1;</td>
<td>ChangeVar[Q, Plus[V, 1]];</td>
</tr>
</tbody>
</table>

Table 3. Syntax of VHDL branching statements in M-code

<table>
<thead>
<tr>
<th>VHDL</th>
<th>M-code</th>
</tr>
</thead>
<tbody>
<tr>
<td>If B then state-bloc-1 else state-bloc-2 end if</td>
<td>If [B, state-bloc1, state-bloc-2, decideACL2]</td>
</tr>
<tr>
<td>For I in start to end loop Statements End loop;</td>
<td>For [Set[I, start], Equal[I, end], Incr[I], decideACL2, Statements] (*Comment: B = Equal[I, end] *)</td>
</tr>
</tbody>
</table>

At this stage, the body of \( EntA \) contains only sequential statements: assignments, conditionals or instantiations of components. Each one of them is represented by a function in Mathematica syntax.

An assignment is modeled by \( NextSig \) for signals and \( ChangeVar \) for variables (Table 2). \( NextSig \) assigns the next value of the signal while \( ChangeVar \) assigns the variable directly. \( NextSig[Sig, \text{terms}] \) or \( ChangeVar[Var, \text{terms}] \) also create rewrite rules [5] that transform \( Sig \) or \( Var \) to \( \text{terms} \). These rules are not applied during M-code generation, but during simulation.

Branching statements are modeled by functions in which their semantics consider a three state logic (Table 3). When \( B \) is a symbolic formula that cannot be evaluated to true or false by Mathematica, ACL2 is called to decide \( B \) under constraints. Details about the decision procedure are discussed in the next section.

4 Simulation Algorithm

First, all objects are initialized with their values according to their VHDL declaration. The consistency of simulation constraints is verified by ACL2. After that, the M-code function is executed \( \text{NbCYCLE} \) times (\( \text{NbCYCLE} \) is user defined).

At each simulation cycle, the function \( \text{Test-vectors} \) can be customized to generate specific inputs; for instance, reset signals can be active in the first simulation cycle, inactive otherwise. Then, the \( \text{EntA} \) function is interpreted in Mathematica, where two operations are performed: simplification of terms and branch decision. At the end of each cycle an execution tree is generated, which contains all symbolic values for each signal and variable in the design.

4.1 Computation of Terms

When assignment functions \( NextSig[Sig, \text{terms}] \) or \( ChangeVar[Var, \text{terms}] \) are encountered, right hand side \( \text{terms} \) are simplified into \( \text{termst} \), using standard
Initialize($\text{Sin, Sout, Slocal, Vlocal}$)
Verify-by-$\text{ACL2(Constraints)}$
For cycle:=1 to $\text{NbCYCLE}$ do
    Test-vectors($\text{Reset, cycle, Sin}$) 
    EntA($\text{Reset, Sin, Sout, Slocal, Vlocal}$) 
    Print-Tree($\text{Sin, Sout, Slocal, Vlocal}$)
End for;

**Fig. 2.** Simulation algorithm

<table>
<thead>
<tr>
<th>MATHEMATICA</th>
<th>ACL2</th>
</tr>
</thead>
</table>
|Call of $\text{ACL2}$ to check consistency of constraints| $\text{check\_consistency(Lh)}$
|If $\text{Lh}$ is not empty, show $\text{Lh}$ to the user, else $\text{Lh}$ implies branch condition $\text{B}$?| $\text{Lh}\Rightarrow\text{B}$?|
|If answer is Q.E.D simulate “true” branch else $\text{Lh}$ implies not $\text{B}$?| $\text{Lh}\Rightarrow\text{not B}$?|
|If answer is Q.E.D simulate “false” branch else ask the user to add constraints or fork| $\text{answer}$|

**Fig. 3.** Branch decision scheme

Mathematica and static VHDL rules. Then, the left hand side $\text{Sig}$ or $\text{Var}$ is assigned with $\text{termst}$ and the rewriting rule $\text{Sig} \rightarrow \text{termst}$ or $\text{Var} \rightarrow \text{termst}$ is added to a library called dynamic VHDL simplification rules. Those rules are now available to simplify all successive assignments. This on the fly simplification of terms is essential for time and memory efficiency.

In Table 2, $\text{ChangeVar}[V, \text{Plus}[2, j]]$ assigns $V$ with $\text{Plus}[2, j]$ and creates the rewrite rule $V \rightarrow 2 + j$. In the next assignment ($V + 1$) is simplified using ($V \rightarrow 2 + j$). Then, $Q$ is assigned with $3 + j$. Finally, the rewrite rule ($Q \rightarrow 3 + j$) is created.

### 4.2 Branch Decision

During simulation, Mathematica, whenever it cannot decide a branch condition, calls $\text{ACL2}$. Figure 3 shows the principle of their interaction.

First, Mathematica asks $\text{ACL2}$ to check the consistency of the set of simulation constraints $L_h$. Function $\text{check\_consistency}$ takes $L_h$ as input and returns a
minimal set of contradictory hypothesis $I_h$, or the empty set. If $I_h$ is not empty, the simulation is stopped and the contradiction is shown to the user.

If $I_h$ is empty, Mathematica sends $L_h \Rightarrow B$ to ACL2. If ACL2 finds a proof, it returns $Q.E.D$; the ”true” branch is considered for simulation. If ACL2 fails or is not able to find a proof in a given time, it returns Failed. In this case, Mathematica sends $L_h \Rightarrow \neg B$. If it succeeds, the ”false” branch is considered for simulation. Otherwise, the simulation stops and the user is asked for more constraints. If more constraints are given, simulation is reinitialized. Otherwise, the symbolic simulation forks into two branches, one assuming the branch condition is true and the other its negation.

Branch decision is generally not decidable. However, most cases are limited to equalities and inequalities formulae, and resolved by using some pre-proved theorems on them (written as ACL2 books). At each cycle the proved theorems are added to the ACL2 database and they are available for the future proofs.

Example Euclid’s GCD algorithm (Table 4):

<table>
<thead>
<tr>
<th>VHDL</th>
<th>M-code</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1: process begin</td>
<td>GCDmath[CLK_,RST_,a_,b_,OK_, \text{res}<em>a0</em>,b0_,c0_] :=</td>
</tr>
<tr>
<td>wait until clk='1';</td>
<td>Module[</td>
</tr>
<tr>
<td>if RST='1' then</td>
<td>If[RST=1,</td>
</tr>
<tr>
<td>a0:=a;</td>
<td>ChangeVar[a0,a];</td>
</tr>
<tr>
<td>b0:=b;</td>
<td>ChangeVar[b0,b];</td>
</tr>
<tr>
<td>ok&lt;=False;</td>
<td>NextSig[OK,False]</td>
</tr>
<tr>
<td>elsif a0=b0 then</td>
<td>,If[Equal[a0,b0]</td>
</tr>
<tr>
<td>ok&lt;=True;</td>
<td>,NextSig[OK,True];</td>
</tr>
<tr>
<td>res&lt;==a0;</td>
<td>NextSig[res,a0]</td>
</tr>
<tr>
<td>elsif a0&gt;b0 then</td>
<td>,If[a0&gt;b0]</td>
</tr>
<tr>
<td>a0:=a0-b0;</td>
<td>,ChangeVar[a0,a0-b0]</td>
</tr>
<tr>
<td>else b0:=b0-a0;</td>
<td>,ChangeVar[b0,b0-a0]</td>
</tr>
<tr>
<td>end if;</td>
<td>decideACL2]</td>
</tr>
<tr>
<td>end process P1;</td>
<td>,decideACL2]</td>
</tr>
</tbody>
</table>

Before beginning the simulation, the function Test_vectors has been customized to generate an active reset at the first simulation cycle and inactive hereafter. The initial values are $a = 3n$ and $b = n$ and the constraints are $L_h = \{n \in \mathbb{N}^*\}$. The simulation of four cycles runs as follows.

At Cycle1, $RST$ has the numeric value 1 and $a_0$ and $b_0$ are assigned with initial values $3n$ and $n$. In all subsequent cycles, $RST$ is set to 0 and Mathematica will always decide to simulate the ”false” branch of the first if–then–else statement. We do not mention it anymore. At Cycle2, Mathematica cannot decide if $a_0$ is equal to $b_0$, i.e. if $3n$ is equal to $n$. So, it calls decideACL2, which works as shown on Figure 3. The constraint $\{n \in \mathbb{N}^*\}$ is transformed into the ACL2
list \(((\text{integerp } n) \wedge (0 < n))\) and its consistency is checked. As ACL2 returns an empty list of contradictions \(I_h\), Mathematica sends the following "defthm event" to ACL2:

```lisp
(defthm branch-1
  (implies (and (integerp n) (< 0 n))
    (equal (* 3 n) n)))
```

Because the ACL2 answer is "Failed", Mathematica sends the event:

```lisp
(defthm branch-1-negation
  (implies (and (integerp n) (< 0 n))
    (not (equal (* 3 n) n))))
```

ACL2 answers "Q.E.D", Mathematica considers the "false" branch for simulation and simplifies \(a_0 - b_0\) to \(2n\). The reader may be surprised by the simplicity of the theorems, but without ACL2 Mathematica is not able to prove them. At Cycle3, \(a_0\) is simplified to \(n\) and at Cycle4 ACL2 answers "Q.E.D" to the event:

```lisp
(defthm branch-4
  (implies (and (integerp n) (< 0 n))
    (equal n n)))
```

As four cycles have been simulated, the simulation is stopped. Figure 4 shows the execution tree without any constraints. With constraints, only the bold path is simulated (reset has been omitted).

---

**Fig. 4.** Execution tree of the GCD example
5 Discussion and Conclusions

Our prototype system for Constrained Symbolic Simulation takes advantage of the best qualities of two powerful automatic systems: Mathematica to simplify algebraic expressions, and ACL2 to decide the truth value of expressions under a set of hypotheses. Clock synchronized sequential circuits and delay-free combinational circuits, written in a synthesizable VHDL subset, are automatically translated into a M-code file, its simulation model.

The automatic generation of proof obligations for ACL2, under the form of “deftlm events” is implemented. Mathematica and ACL2 are executed as concurrent processes, and communicate via a pipeline. Our technique efficiently prunes the execution tree, and proves VHDL assert statements [1] on small circuit blocks; we are working on bigger systems, like the AMBA architecture. We intend to extend our method to more abstract specifications, as describable in the next version of the VHDL subset for system-level synthesis, or SystemC.

References

Semi-formal Verification of Memory Systems by
Symbolic Simulation*

Husam Abu-Haimed, Sergey Berezin, and David L. Dill
Stanford University
{husam,berezin,dill}@stanford.edu

Abstract. We propose a debugging method for data-path intensive sys-
tems, in particular, memory systems. The approach is based on strength-
ening invariants by deriving constraints on data in the design using sym-
bolic simulation with constrained inputs. A new heuristic is introduced
for finding the appropriate input constraints for the symbolic simulation.
We give up soundness in order to gain more automation and efficiency,
minimizing or even eliminating the required manual effort. While it is
no longer possible to prove the correctness of the design, experimental
results demonstrate that the technique is quite effective in finding design
errors.

1 Introduction

Most hardware systems of interest today are much larger than what can be
reliably tested by conventional methods, and some form of formal verification
becomes a necessity. In order for non-expert users to be able to apply formal
methods, the tools must be mostly automatic. Some of the most successful ap-
proaches to date are model checking [5], theorem proving [11], and validity check-
ing [12]. However, these approaches are often applicable only to relatively small
systems, or require significant manual guidance.

In this paper, we are interested in verifying memory systems and similar data-
intensive designs. Due to the large sizes of data structures used in memories, we
model them as infinite systems. Proving the correctness of such designs usually
boils down to proving an invariant. The approach we propose can be used in
verifying arbitrary safety properties which can be expressed as invariants.

The standard way to prove invariants for infinite systems is by induction over
time. Most of the time, however, the invariant we want to prove is not inductive
and has to be strengthened. Often, invariants are strengthened manually in a very
tedious iterative process that requires experience and familiarity with the design.
This is the most difficult and time consuming part of the verification process.

* This research was supported by GSRC contract DABT63-96-C-0097-P00005, by Na-
tional Science Foundation CCR-0121403, and by King Fahd University of Petroleum
and Minerals, Saudi Arabia. The content of this paper does not necessarily reflect
the position or the policy of GSRC, NSF, or the Government, and no official endor-
sement should be inferred.

© Springer-Verlag Berlin Heidelberg 2003
Many techniques have been proposed in the literature to partially automate the process of strengthening invariants \[7,8,6,13,14,3,9,10,2,15,4\].

In a previous work \[1\] we introduced a method for strengthening and proving invariants by the technique called consistency testing which uses symbolic simulation. In that method, the user may have to supply the consistency test manually, and the tool then constructs the remaining part of the inductive invariant, proves it, and verifies that the supplied test satisfies certain properties to guarantee soundness.

In this work, we propose a similar method, but without the soundness check and with a simplified induction scheme. The consistency test is replaced by input constraints constructed automatically using a special heuristic. This results in a potentially unsound method, but it becomes completely automatic and serves as a very efficient debugging tool. Besides skipping the soundness check, the efficiency is also gained by reducing the number of cycles in symbolic simulation compared to the previous method. We use CVC \[12\] as a symbolic simulator and a validity checker in our experiments.

Our approach is based on the empirical observation from several examples that most of the invariants in data path intensive systems can be obtained by symbolically simulating the system for a few cycles with specific inputs. The inductive step is then proven only for the states that can be reached by such symbolic simulation, instead of for all reachable states. In order to complete the proof, we need to show that all the reachable state are included in this set of states. However we do not discuss this problem in the paper.

Instead, we give up this soundness check and propose our approach as a debugging tool. We tested the effectiveness of this approach by applying it to several examples of memory systems. In all the examples we considered, it was able to find all design errors in addition to several errors we inserted to test the effectiveness of our approach. This gives us confidence in the effectiveness and reliability of our approach as a debugging technique.

The paper is organized as follows. Sections 2 and 3 formally introduce induction on time and functional equivalence, followed by a detailed description of our verification technique in section 4. An automatic technique for finding input constraints is given in section 5. Section 6 concludes the paper.

## 2 Induction on Time

We model a hardware design as a transition system \(T = (S, s_0, N, R, D_{\text{in}}, D_{\text{out}})\), where \(S\) is a non-empty (and possibly infinite) set of states, \(s_0 \in S\) is the initial state, \(D_{\text{in}}\) and \(D_{\text{out}}\) are the domains of inputs and outputs, \(N : S \times D_{\text{in}} \rightarrow S\) is the transition function, and \(R : S \times D_{\text{in}} \rightarrow D_{\text{out}}\) is the output function. We write \(N(s, \alpha^\ell)\) to denote the final state of running \(T\) on the input sequence \(\alpha^\ell\) of length \(\ell\) starting from the state \(s\):

\[
N(s, \alpha^\ell) = N(N(\ldots N(s, \alpha_0), \alpha_1), \ldots, \alpha_{\ell-1}).
\]
It is important to note that a single transition in \( T \) can actually represent a complex transaction in the real hardware implementation requiring multiple cycles of execution.

A state \( s \) is called reachable in a transition system \( T \), if there is an input sequence \( \alpha^\ell \) such that \( s = N(s_0, \alpha^\ell) \), where \( s_0 \) is the initial state of \( T \). In this paper, we only consider safety properties, or invariants over the set of reachable states. We say that a transition system \( T \) satisfies a safety property \( Q(s) \), if \( Q(s) \) holds for every reachable state \( s \) of \( T \). This can be stated as follows:

\[
\forall \ell, \alpha^\ell. \ Q(N(s_0, \alpha^\ell)) \quad (1)
\]

The conventional way of proving (1) is by induction on time, when \( Q \) is first shown to hold in the initial state \( s_0 \), and then the transition function \( N \) is shown to preserve \( Q \):

\[
Q(s_0), \quad \forall s, \sigma. \ Q(s) \Rightarrow Q(N(s, \sigma)). \quad (2)
\]

In practice, this induction scheme requires finding an inductive the invariant, which is often the hardest and most tedious part of verification process.

### 3 Functional Equivalence

We prove correctness of systems using the idea of functional equivalence. The problem is stated as follows. Given two systems, the concrete system \( T^c \) (the system we want to verify) and the abstract system \( T^a \) (which defines the required functionality of \( T^c \)), prove that \( T^c \) is functionally equivalent to \( T^a \). Two systems are said to be functionally equivalent if they produce the same sequence of outputs for the same sequence of inputs. Formally, this is expressed as follows:

\[
\forall \ell, \alpha^\ell, \lambda. \ R^c(N^c(s_0^c, \alpha^\ell), \lambda) = R^a(N^a(s_0^a, \alpha^\ell), \lambda). \quad (3)
\]

If we define \( Q(s) \) to be \( \forall \lambda. \ R^a(s_0^a, \lambda) = R^c(s_0^c, \lambda), \) (3) becomes \( \forall \ell, \alpha^\ell. \ Q(N(s, \alpha^\ell)) \), which is the same as formula (1). So, we can use the same induction principle given by (2) to prove the functional equivalence (3) of the two modules.

### 4 The Verification Method

In this section we introduce our approach through a simple example. We show how the direct use of (2) to prove the correctness of a memory system fails. Then we show how our method can be used to deal with the problem.

Consider a small example of a read-only memory with a single-line cache given in figure 1(a). To verify the correctness of this design, we show that it is functionally equivalent to a simple (uncached) array of data in figure 1(b). Since the memories are read-only, the input to both modules is the address \( (D_{in} = \text{Addr}) \), and the output is the data read from that address \( (D_{out} = \text{Data}) \).
The transition systems $T^c$ and $T^a$ are defined as follows. The abstract state $s^a$ of $T^a$ is just an array $M$ indexed by $\text{Addr}$ and holding the $\text{Data}$ elements. The next state function $N^a$ is the identity function, and $R^a(s^a, \lambda) = M[\lambda]$. The concrete state $s^c$ of $T^c$ contains the state of the cache in addition to the same array $M$. Initially, in $s^c_0$, some arbitrary address is cached such that the cache is coherent with the main memory $M$. The next state function $N^c(s^c, \lambda)$ adds the address $\lambda$ and the data stored under that address $M[\lambda]$ to the cache, yielding the new state. The output function $R^c(s^c, \lambda)$ is similar to $N^c$, except that it returns the data associated with the address $\lambda$.

Unfortunately, proving the functional equivalence of the two memories by simple induction fails. Consider the state in figure 1 (a) and (b) where $a \neq b$ and $a = e$. In this case, $s^c$ and $s^a$ are functionally equivalent and hence the induction hypothesis $Q(s)$ is satisfied. However, transitioning to the next state by reading some address $\sigma \neq \pi$ brings $T^c$ to a new state $s^{c'}$, shown in figure 1 (c), where the address $\pi$ is no longer cached. Therefore, reading $\pi$ again yields $b \neq a$, which no longer agrees with $T^a$. The induction fails in this case because it starts out from an incoherent state, which is not reachable. The natural way to strengthen the invariant is to require the state to be coherent. In this example it means that the cached value must be the same as in the main memory. So, in general, we can strengthen invariants for such systems by asserting their coherence.

Now suppose we simulate the incoherent state $s^c$ for one step with the input constraint $C(s^c, \alpha) \equiv \alpha \neq \pi$. The resulting state $s^{c'}$ is shown in figure 1(d). Clearly, state $s^{c'}$ is coherent, and the induction (2) for such a state is valid. Formally, (2) is restricted to the set of states $\Sigma'$ defined as follows:

$$\Sigma' = \{(s^a, s^{c'}) \mid \exists s^{c''}, \alpha. s^{c''} = N^c(s^{c''}, \alpha) \land \alpha \neq \pi\}.$$ 

The induction (2) with $\Sigma'$ becomes:

$$Q(s_0), \forall s \in \Sigma', \sigma. Q(s) \Rightarrow Q(N(s, \sigma)). \quad (4)$$

Proving (4) does not complete the proof of correctness for the memory system; it simply says that the concrete system behaves according to the specifications when started from any state in $\Sigma'$. To complete the proof of correctness,
we need to prove that all reachable states $\Sigma$ are included in $\Sigma'$. That can be done by proving the following induction:

$$\Sigma'(s_0), \quad \forall s, \sigma. \Sigma'(s) \Rightarrow \Sigma'(N(s, \sigma)),$$

(5)

In general, (5) is undecidable. For some memory systems, however, proving it can be a matter of a simple intuition of the designer. For cases where we fail to prove (5), our approach can still be used as an effective debugging tool. For the cache example, it is easy to show that (5) is valid and that completes the proof of correctness for this example.

The general idea in our approach is to find an input constraint $C(s, \alpha^k)$ on an input vector $\alpha^k$ that when executed on an arbitrary state $s$ will remove the incoherences in it. For instance, in the example above, the read from $\alpha \neq \pi$ removes the incoherence by causing $c$ to be copied from the main memory to the cache.

5 Finding Input Constraints

Data path intensive systems consist mainly of registers interconnected by buses (or links). Each link has a condition or predicate associated with it. When the condition is true, the data is transferred along the link. In any system transition many data transfers may happen. These data transfers imply some constraints on the state of the system. In the cache example, the data transfer from the main memory to the cache implies the constraint that the cache and the main memory are always coherent. In general, we can control which data transfers happen in each system transition by constraining the inputs. In the cache example, we constrained the input by $\alpha \neq \pi$.

Our heuristic looks for the right input constraints that will exercises the right links and get the data synchronized. The idea is to look at counterexamples of failed proofs. Suppose we try to prove

$$\forall s, \sigma.[Q(s) \Rightarrow Q(N(s, \sigma))].$$

(6)

If the proof fails, we get back a counterexample $C$. Intuitively, $C$ defines the data transfers that contributed to the failure of the proof. Based on our assumption, the proof failed due to incoherences between the data involved in these transfers. If we simulate $s$ for one transition and exercise the same links as in $C$, we are likely to get rid of these incoherences. Let $c_i$ be the condition associated with a link $l_i$. If $l_i$ is activated in $C$, its condition $c_i$ becomes true in $C$. Let $\Sigma'$ be the set of states where every $c_i$ holds for each link $l_i$ activated in $C$. That is, the input constraint becomes $C(s, \alpha) = \bigwedge_i c_i$. Then we try to prove:

$$\forall s' \in \Sigma', \sigma.[Q(s') \Rightarrow Q(N(s', \sigma))].$$

(7)

By simulating $s$ with the constraint $C(s, \alpha)$, it is likely that we will get rid of the incoherences. If (7) is not valid, we get a new counterexample and repeat
the process. If at any point we get a counterexample with the same set of activated links as in any previous counterexample, we report it as a potentially true counterexample. The user can also put a limit on the number of iterations to guarantee termination.

6 Conclusion

In this paper, we presented an automatic technique for finding design errors in memories and data path systems. The method is based on a semi-formal version of invariant checking using symbolic simulation with automatically generated input constraints. We tested the method on various types of memory systems (one and two-level direct-mapped cache, set-associative cache, and a memory system with SDRAM controller), and the method found all the bugs in these designs without any manual effort, which demonstrates its effectiveness. The longest runtime was for the two-level direct-mapped cache, and it took 10 minutes on a machine with a 800MHz Pentium processor.

References

7. Satyaki Das and David L. Dill. Counter-example based predicate discovery in predicate abstraction. In FMCAD’02.
CTL May Be Ambiguous When Model Checking Moore Machines

Cédric Roux and Emmanuelle Encrenaz

UPMC – LIP6 – ASIM
12, rue Cuvier, 75252 Paris CEDEX 5 – France

Abstract. The model checking problem is defined over Kripke structures. However, hardware designers often handle other models, such as Moore machines. When model checking their designs using CTL as a logic, they must translate them into Kripke structures. A given CTL property may be believed to be true (conversely false) over the Moore machine and in fact be false (conversely true) on the derived Kripke structure. This may lead to ambiguities if the designer does not fully understand the translation scheme he uses, which may be the case if he uses automatic tools. We present iCTL, a logic specifically designed to work with Moore machines, which extends CTL to help the designer removing possible ambiguities when model checking Moore machines. We show that it is strictly more expressive than CTL.

1 Introduction

While developing a symbolic model checker to verify hardware systems described as a composition of synchronous Moore machines, we came across an interesting problem. We use CTL [2] as logic and the formulae we want to verify may include values of input signals of the Moore machines. These input signals do label the transitions of the Moore machine. Since CTL is defined over Kripke structures and not Moore machines, and because the transitions of Kripke structures are not labelled, when translating a Moore machine into a Kripke structure, one has to integrate the input signals in the states of the Kripke structure. Several choices are possible. Depending on the translation chosen, the truth value of a given property may either be true or false over the derived Kripke structure. This introduces an ambiguity that the designer must be aware of when verifying his designs. He has to know how his model is translated into the one used by the model checker, and has to write properties with this in mind, so not to get confused by the answer of the tool. Not doing so could even lead to a counter-intuitive situation, where the designer might view his model as being buggy where in fact he simply wrote wrong formulae, thinking them over Moore machines and not over the derived Kripke structures.

In [3] the authors translate a Moore machine into a Kripke structure by incorporating the input configurations in the source state of the transitions. And they define the truth value of a CTL property over a Moore machine as
being the truth value of this property over the Kripke structure. We think that
such an approach leads to ambiguities.

In SMV [5], one directly writes Kripke structures and CTL formulae over
these structures. It is possible to create free variables (that may represent input
signals of Moore machines incorporated into the current state of the Kripke
structure). This leads to exactly the same situation as [3].

The VIS model checker [1] accepts, among others, systems described in a
Verilog subset, in which collections of Moore machines can be represented. It
supports modularity and the concept of input and output signals is present.
However, an input signal can appear in a CTL formula only if it is declared
of type reg, which forces its assignment in guarded blocks. As a consequence,
depending on the way this assignment is done, input signals of a Moore machine
will be included into the source or target state of the transitions in the Kripke
structure, which influences the results of the verification of a given formula.

The purpose of this article is to suggest to add two new operators to CTL
to bring together the intuitive idea one can have regarding the truth value of a
formula over the Moore machine and the one obtained by the verification algo-

2 Translating a Moore Machine into a Kripke Structure

Several translation schemes from a Moore machine into a Kripke structure are
possible. The simplest one is to remove the inputs labelling transitions. Since we
want to express properties including input signals, we abandoned such a scheme.
Another way is to put the input signals into the target state of a transition. Since
we plan to compose Moore machines, this solution can’t be retained because the
outputs of one machine which are inputs of one other have to have the same
temporal behavior as the other inputs of the second machine. So, we have to put
the inputs into the source state of a transition.

Here follows the formal definitions of Kripke structures, Moore machines,
and the translation scheme we adopted.

Definition 1. A Kripke structure is a five-tuple \( \langle S, S_0, P, L, R \rangle \) where

1. \( S \) is a finite set of states,
2. \( S_0 \subseteq S \) is the set of initial states,
3. \( P \) is a finite set of atomic propositions (we define \( n_P = |P| \)),
4. \( L = \{ l_0, \ldots, l_{n_P-1} \} \) is a vector of \( n_P \) functions, each function defining the
   value of exactly one atomic proposition; for all \( 0 \leq i \leq n_P - 1 \) we have
   \( l_i : S \rightarrow \mathbb{B} \); for all \( s \in S \), we have that \( l_i(s) \) is true iff the atomic proposition
   associated to \( l_i \) is true in \( s \),
5. \( R \subseteq S \times S \) is the transition relation.
Definition 2. A Moore machine is a structure $\langle S, S_0, I, O, L, R \rangle$ where

1. $S$ is a finite set of states,
2. $S_0 \subseteq S$ is the set of initial states,
3. $I$ is the finite set of input symbols,
4. $O$ is the finite set of output symbols (we define $n_O = |O|$),
5. $L = \{l_0, \ldots, l_{n_O-1}\}$ is a vector of $n_O$ functions, each function defining the value of exactly one output symbol; for all $0 \leq i \leq n_O-1$ we have $l_i : S \rightarrow \mathbb{B}$;
   for all $s \in S$, we have that $l_i(s)$ will be true iff the output symbol associated to $l_i$ is true in the state $s$,
6. $R \subseteq S \times 2^I \times S$ is the transition relation.

The Moore machines we handle are complete and deterministic. Complete means that each state has one successor for any input configuration. Deterministic means that for a given input configuration, a state $s$ will always lead to the same state $s'$.

Definition 3. Translating a Moore Machine by Putting the Inputs in the Source State Given a Moore machine $\langle S_M, S_{M0}, I_M, O_M, L_M, R_M \rangle$, we deduce the Kripke structure $\langle S_K, S_{K0}, P_K, L_K, R_K \rangle$ where:

- $S_K = S_M \times 2^{I_M}$,
- $S_{K0} = S_{M0} \times 2^{I_M}$,
- $P_K = I_M \cup O_M$ (we define $n_{I_M} = |I_M|$ and $n_{O_M} = |O_M|$),
- $L_K = \{l_{O_0}, \ldots, l_{O_{n_{O_M}-1}}\} \cdot \{l_{I_0}, \ldots, l_{I_{n_{I_M}-1}}\}$; for all $0 \leq i \leq n_{O_M} - 1$, we have $l_{O_i} : S_K \rightarrow \mathbb{B}$; for all $i$, for all $s = (s_1, c_1) \in S_K$, we have that $l_{O_i}(s)$ is true iff $l_{I_i}(s_1)$ is true; for all $0 \leq i \leq n_{I_M} - 1$, we have $l_{I_i} : S_K \rightarrow \mathbb{B}$ (each $l_{I_i}$ is associated to one and only one input signal); for all $i$, for all $s = (s_1, c_1) \in S_K$, we have that $l_{I_i}(s)$ is true iff the component of $c_1$ corresponding to the input signal associated to $l_{I_i}$ is true,
- $R_K \subseteq S_K \times S_K$ and $\forall (s, c_i) \in S_K; \forall (s', c_i') \in S_K$, we have $((s, c_i), (s', c_i')) \in R_K$ iff $(s, c_i, s') \in R_M$.

An example of a trivial Moore machine and the derived Kripke structure is shown in figure 1.

3 A Disturbing Example

We could simply state that a CTL formula is true in a Moore machine if and only if it is true in the corresponding Kripke structure as done in [3] but the verification results obtained may disturb the designer.

As an illustration, we propose to check the CTL property $(\text{EX } p) \land (\text{EX } \neg p)$ over the Moore machine depicted on figure 1.

This formula would be true on a Kripke structure obtained from the Moore machine by removing the inputs, but it is false on the Kripke structure shown on figure 1 (which is the one obtained with the translation of definition 3), because neither $A_0$ nor $A_1$ has a successor verifying $\neg p$ and a successor verifying $p$.

In fact, the formula $(\text{EX } p) \land (\text{EX } \neg p)$ is ambiguous over the Moore machine: do we mean that both successors are selected by the same input configuration or by different input configurations?
4  iCTL – CTL Model Checking with Input Configurations

We introduce two new operators to CTL. These two operators are ∀I and ∃I. This defines a new logic, that we call iCTL. Given φ, an iCTL formula (that may contain ∀I and ∃I operators), ∀Iφ stands for “for all input configuration, φ holds” and ∃Iφ stands for “there is an input configuration for which φ holds”.

Here follows the formal definition of iCTL.

4.1 Syntax and Semantics of iCTL

The syntax is the same as the one of CTL, with the following added rule for state formulae.

– if f is a state formula, then ∀If and ∃If are state formulae.

The semantics remains the same, with the following added rules.

As the two new operators deal with input configurations, the Kripke structure they apply on are the ones given by our translation from Moore machines. The symbols are thus the same than those from definition 3.

\[ M, s \models ∀f \iff s = (s_M, c_M) \text{ and } \forall c'_M \in 2^{I_M}, s' = (s_M, c'_M) \text{ and we have that } s' \models f, \]

\[ M, s \models ∃I f \iff s = (s_M, c_M) \text{ and for one } c'_M \in 2^{I_M}, s' = (s_M, c'_M) \text{ and we have that } s' \models f. \]

Since our Moore machines are complete, for all input configurations, the state s’ exists in the Kripke structure, thus s’ \models f is sound.

Using iCTL, we now can define when a Moore machine validates a logical formula.

**Definition 4.** A Moore machine M validates a formula f of iCTL if and only if the formula is true in the corresponding Kripke structure, as given by the transformation of definition 3.
This definition is the same as in [3], but we expect the designer to remove the ambiguities of CTL by using $\exists_I$ and $\forall_I$ in the places where they are needed.

### 4.2 Examples

The Moore machine of figure 2 will be used as example.

On the Kripke structure derived from it by the translation of definition 3, we’ve got that the formula $\text{AX EX } p$ is false in $s_1.\bar{i}$ and $s_1.i$. Looking at the Moore machine, one might think that this formula is true in $s_1$, since all its successors have a successor where $p$ is true (states $s_4$ and $s_6$). The formula $\text{AX } (\exists_I (\text{EX } p))$ is true in $s_1.\bar{i}$ and $s_1.i$ on the derived Kripke structure. This corresponds to the intuition one might have about the truth value of $\text{AX EX } p$ over the Moore machine. We see here that to capture this intuition, $\exists_I$ is necessary.

Similarly, the formula $\text{EX AX } p$ is true in $s_1.\bar{i}$ and $s_1.i$ in the derived Kripke structure while $\text{EX } (\forall_I (\text{AX } p))$ is false in $s_1.\bar{i}$ and $s_1.i$. This latest interpretation seems to be consistent with the intuition that one might have for the truth value of $\text{EX AX } p$ in the state $s_1$ of the Moore machine.

### 4.3 iCTL Is More Expressive than CTL

Given a formula $f \in \text{iCTL}$ and a formula $g \in \text{CTL}$, we say that $f$ is equivalent to $g$ if and only if for all Kripke structure $K$ derived from a Moore machine $M$ using the translation of definition 3, for all state $s$ of $K$, we have that $K, s \models f$ iff $K, s \models g$. (This is the global equivalence of [4].)

On the Kripke structure of figure 3, we can prove (by induction over its size) that any CTL formula won’t see its truth value changed in $s_1.\bar{i}$, $s_2.\bar{i}$ and $s_2.i$ if we change the labelling of $s_3$. But the iCTL formula “$\exists_I \text{EX } p$” distinguishes both cases. Since all CTL formulae are in iCTL, we have that iCTL is more expressive than CTL (for Kripke structures coming from definition 3).

### 4.4 iCTL and Other Logics

Modal $\mu$-calculus is a logic dealing with labelled transition systems (thus, able to handle Moore machines), which contains the ($\ast$) and $[\ast]$ operators. ($\ast$) $p$ is
true in a state $s$ if $p$ is true in at least one of its successor, reachable by any transition. $[\ast]p$ is true in a state $s$ if $p$ is true in all the successors of $s$, reachable by any transition. We think that $\langle \ast \rangle$ in the $\mu$-calculus is equivalent to $\exists_I \text{EX}$ in $i$CTL and that $[\ast]$ is equivalent to $\forall_I \text{AX}$. Formulae $\exists_I \text{AX} p$ or $\forall_I \text{EX} p$ are in $i$CTL and have a meaning over Kripke structures obtained from Mealy machines. We didn’t find equivalent formulae to those in the $\mu$-calculus.

$LTL$ does not present the same ambiguities than $CTL$ since it only captures a set of infinite sequences and the sets of sequences of the Moore machine and of the derived Kripke structure are equivalent. So, something like “$iLTL$” would be useless.

5 Conclusion

The paper discusses the consequences of placing input configurations labelling transitions in Moore machines into the source states in the derived Kripke structure built to perform $CTL$ model checking. This translation has an impact on the verification since a given $CTL$ formula believed to be true or false on the Moore machine can have a different truth value on the obtained Kripke structure. This is due to the lack of expressiveness of $CTL$ that does not take into account labelled transitions, as we find in Moore machines. To overcome this ambiguity, we introduce two operators, $\exists_I$ and $\forall_I$. We show that the obtained logic, named $iCTL$, is more expressive than $CTL$. We have implemented these operators in our model checker and it is our intention to verify complex systems with this logic.

References


Reasoning about GSTE Assertion Graphs

Alan J. Hu\textsuperscript{1}, Jeremy Casas\textsuperscript{2}, and Jin Yang\textsuperscript{2}

\textsuperscript{1} Department of Computer Science, University of British Columbia,
2366 Main Mall, Vancouver, BC V6T 1Z4, Canada,
+1-604-822-6667, FAX +1-604-822-5485
ajh@cs.ubc.ca

\textsuperscript{2} Strategic CAD Labs, Intel Corporation

\begin{abstract}
Generalized symbolic trajectory evaluation (GSTE) is a new model-checking approach that combines the industrially-proven scalability and capacity of classical symbolic trajectory evaluation with the expressive power of temporal-logic model checking. GSTE was originally developed at Intel and has been used successfully on Intel’s next-generation microprocessors. However, the supporting theory and algorithms for GSTE are still immature. In particular, GSTE specifications are given as assertion graphs, a variety of $\forall$-automata, and although an efficient model-checking algorithm exists to verify whether a circuit model obeys a specification assertion graph, there is no work on reasoning about assertion graphs themselves. This paper presents new algorithms to leverage GSTE model checking to efficiently decide whether one assertion graph implies another, and to model check one assertion graph under the assumption that another is true (under regular GSTE acceptance conditions). These two operations — deciding whether one specification implies another and verifying under an assumption — are the fundamental building blocks of compositional verification and any higher-level reasoning about model-checking results, so the algorithms presented here are key steps to using GSTE in a broader verification framework. Preliminary experimental results applying our algorithms to real, industrial circuits and specifications show that our algorithms are useful in practice.
\end{abstract}

\section{Introduction}

Generalized symbolic trajectory evaluation (GSTE) is a powerful, new model-checking approach [20]. GSTE is based on classical symbolic trajectory evaluation [16], which has proven itself able to handle large, industrial designs and has been in active use at Compaq (now HP), IBM, Intel, and Motorola (e.g., [12,10,1,4]). Classical symbolic trajectory evaluation, although efficient, is very limited in the types of properties that it can specify and verify. GSTE extends classical symbolic trajectory evaluation to handle $\omega$-regular properties, giving it comparable expressive power to more established model-checking approaches [5,13,18,8,6], while still maintaining the efficiency and capacity of classical symbolic trajectory evaluation. GSTE was originally developed at Intel and has been used successfully on Intel’s next-generation microprocessors (e.g., [3]).

Key to the efficiency and usability of GSTE is the manner in which properties are specified, in a variety of automata called an assertion graph. Existing GSTE theory provides an efficient procedure for model checking that a circuit obeys an assertion graph,
Reasoning about GSTE Assertion Graphs

as well as techniques based on abstract interpretation to combat state explosion [21]. What is missing, however, is all the supporting theory and algorithms that have developed around more established formalisms like CTL [5] or LTL [18]. In particular, there has been no published research on how to reason about assertion graphs.

This paper presents the foundational pieces for reasoning about specifications given as assertion graphs. Specifically, we give new algorithms to decide whether one assertion graph implies another, and to model check one assertion graph under the assumption that another is true. These two operations — deciding whether one specification implies another and verifying under an assumption — are the fundamental building blocks for decomposing a verification task, composing verification results, and any other higher-level reasoning about specifications. Our current verification system is a mixed deductive-algorithmic system, with an efficient GSTE model-checking procedure built into a lightweight theorem prover. Our new algorithms exploit the existing GSTE model-checking procedure, creating an efficient, algorithmic means to discharge basic deductive reasoning steps about assertion graphs. Preliminary experimental results on real, industrial circuits and specifications show that the algorithms are efficient in practice.

2 Background

2.1 GSTE and Assertion Graphs

GSTE is explained in several sources (e.g., [20,21,19], etc.). Here, we concentrate on the specification style used by GSTE and highlight its characteristics.

GSTE is basically a linear-time model-checking method, i.e., the possible behaviors of the system being verified is considered to be the set of all possible execution traces, and verification consists of checking that all of these traces obey the specification. The specification in GSTE is called an assertion graph, and is basically a variety of automaton. One can think of the assertion graph as defining the set of execution traces that it accepts, so the verification problem is basically language containment. Figure 1 gives a simple example and intuitive explanation of an assertion graph.

In general, an assertion graph is a directed graph with distinguished initial vertex \(v_0\), and the restriction that all vertices must have non-zero out-degree. Each edge \(e\) is labeled with an antecedent \(\text{ant}(e)\) and a consequent \(\text{cons}(e)\). The antecedents and consequents are simply propositional formulas over some set of atomic propositions \(AP\). Traditionally, the atomic propositions correspond exactly to the state variables of the system being verified, so the antecedents and consequents are formulas over the state of the system at some point in time. The assertion graph also has acceptance conditions, described below.

A path in the assertion graph is a directed path (defined in the usual manner for directed graphs) starting from the initial vertex \(v_0\). Every path in the assertion graph specifies a temporal if-then assertion: if the antecedents hold, then the consequents must hold as well. More precisely, a path of length \(n\) (i.e., with \(n\) edges) is an assertion about the system’s behavior over a period of \(n\) clock cycles. If all of the antecedents along the path hold at the corresponding points in the system’s behavior, then all of the consequents
**Fig. 1. GSTE Assertion Graph Example.** This assertion graph, adapted from [20], was used in the verification of an industrial memory design, which reads and writes data with a large variety of selection and alignment options. The property being verified is that, if data value $D$ is written to address $A$, followed by an arbitrary number of clock cycles that don’t overwrite the same address, followed by a read of the address, then the value returned is the value that was written, appropriately aligned and masked. The edge labels are of the form “antecedent / consequent”, where the antecedents and consequents are simply propositional formulas over the state of the system at a given clock cycle. For example, the antecedent WRITE specifies that the value of the write-enable input $we$ is high, that the address input $addr$ is equal to some value $A$, etc. The capital letters denoting values, like $A$, $D$, etc., are symbolic constants, which are essentially skolem constants that can be equal to any value, making the verification result hold for all possible values of the symbolic constants. A path is a sequence of edges that start from the initial vertex $v_0$. A terminal path is a path that ends with a terminal edge (shown in the figure by a tic-mark on the edge, e.g., the edge from $v_2$ to $v_3$). A path accepts an execution trace if at least one antecedent on that path fails (is false on the state of the system at that clock cycle) or if all antecedents and all consequents on the path succeed (are true on that clock cycle). Intuitively, a path is an if-then assertion: the antecedents say when the assertion is relevant; the consequents say what must hold whenever the assertion is relevant. If any antecedent fails, the assertion is vacuously true; if all antecedents are satisfied, then all consequents must be satisfied as well. The assertion graph as a whole accepts an execution trace if every terminal path in the assertion graph accepts that trace. Intuitively, the assertion graph takes a potentially infinite set of assertions about the system and rolls them up into a graph; therefore, every trace must satisfy every assertion (vacuously or otherwise).

**Fig. 2. Monitor Circuit.** Our algorithms rely on a linear-space, linear-time construction for a monitor circuit from an assertion graph $G$. The generated circuit has inputs corresponding to the atomic propositions in $G$ and an output that is true iff the sequence of states presented at the input would have been accepted by $G$. The init input initializes the internal state of the circuit.
must also hold at the corresponding points, in order for the assertion to be satisfied. If any antecedent doesn’t hold, then the assertion is vacuously true. Formally, if \( \rho \) is a path of length \( n \), with \( \rho[i] \) denoting the \( i \)th edge in \( \rho \), and if \( \sigma \) is a trace consisting of \( n \) system states, with \( \sigma[i] \) denoting the \( i \)th state, then \( \sigma \) satisfies or is accepted by \( \rho \) iff

\[
(\forall i_1 \leq i \leq n. \sigma_i \models \text{ant}(\rho[i])) \Rightarrow (\forall i_1 \leq i \leq n. \sigma_i \models \text{cons}(\rho[i])).
\]

For convenience, we will say that “\( \sigma \) satisfies the antecedents of \( \rho \)” if \( \forall i_1 \leq i \leq n. \sigma_i \models \text{ant}(\rho[i]) \), and that “\( \sigma \) fails at least one of the consequents of \( \rho \)” if \( \exists i_1 \leq i \leq n. \sigma_i \not\models \text{cons}(\rho[i]) \).

An assertion graph as a whole accepts a given trace iff all “appropriate” paths in the assertion graph are satisfied. Appropriate is defined by the four different kinds of acceptance in GSTE:

- In strong satisfiability, a finite-length trace is accepted iff it satisfies all paths of the same length in the assertion graph.
- In terminal satisfiability, some edges are marked as terminal edges, and a terminal path is a path that starts from \( v_0 \) and ends with a terminal edge. A finite-length trace is accepted iff it satisfies all terminal paths of the same length.
- In normal satisfiability, an infinite trace is accepted iff it satisfies all infinite paths.
- In fair satisfiability, there is a finite set of fair edge sets. A path is fair iff it visits each fair edge set infinitely often (generalized Büchi fairness). An infinite trace is accepted iff it satisfies all fair paths.

The different kinds of acceptance are listed in (roughly) increasing order of model-checking complexity.

An assertion graph \( G \) defines the set of traces that it accepts. Call that set the language of \( G \), denoted \( L(G) \). Similarly, a system \( M \) defines the set of traces that it can produce, denoted \( L(M) \). Verification consists of proving that \( L(M) \subseteq L(G) \). In subsequent sections of this paper, unless otherwise stated, we will restrict ourselves to terminal satisfiability, which includes strong satisfiability as a special case, because the finite-trace satisfiabilities are currently the most commonly used in practice.

At first glance, assertion graphs may appear somewhat bizarre: the antecedent/consequent edge labels are unusual, as is acceptance based on all paths accepting. However, assertion graphs are actually the natural combination of symbolic trajectory evaluation and automata-theoretic specification. The antecedent/consequent style comes from classical symbolic trajectory evaluation [16] and is a natural way to specify temporal properties. For example, timing diagrams, one of the most widely used hardware specifications in practice, are typically interpreted this way (e.g., if some sequence of events happens, then some other events must happen) [2]. In addition, the explicit identification of antecedents and consequents provides an efficiency benefit, because the model-checking algorithm can limit its search on-the-fly to paths that satisfy the antecedents. The “for all paths” acceptance criteria makes assertion graphs a variety of \( \forall \)-automata [9], which are less familiar than the usual existential acceptance of non-deterministic automata (where a trace is accepted if there exists a corresponding path through the automata), but the \( \forall \) semantics also provides both usability and efficiency benefits. The usability arises because an assertion graph defines a set of assertions, and
one typically wants all assertions to be true; in contrast, usually with automata as specifications, the automata directly defines a set of possible behaviors, so verification consists of determining if the system’s behavior exists in the set provided by the specification. The efficiency advantage of the ∀ semantics — as in other works that use ∀-automata as specifications [9,8,2] — is that a ∀-automaton is essentially pre-complemented, so checking language containment can bypass the expensive step of complementing a non-deterministic automaton. Indeed, GSTE model-checking is very efficient in practice, and the correctness of the algorithm relies on the ∀ semantics.

We emphasize that assertion graphs take their present form as the direct result of practical considerations. The natural theoretical question is what relationship they have to more established formalisms. Assertion graphs with fairness can express all ω-regular properties: an easy construction is to start with a non-deterministic, generalized Büchi automata and then to note that the almost-isomorphic assertion graph (with the same structure, the same fairness constraints, the Büchi automaton’s edge labels moved to the antecedents, and all consequents labeled with False) accepts the complement language. ω-regular expressiveness follows because ω-regular languages are closed under complementation. The same construction also shows that non-deterministic Büchi automata can be simulated with a single-exponential blow-up (to pre-complement the Büchi automaton), and that LTL model checking can be translated to GSTE with at worst the same complexity as the translation to generalized Büchi automata, for which efficient tools exist (e.g., [17]). In the other direction, assertion graphs can be simulated by more conventional automata.¹ Analogous results hold for assertion graphs with terminal satisfiability and ordinary regular automata. In theory, therefore, assertion graphs are no more expressive.

In our case, we have an existing user community with practical experience using GSTE assertion graphs as well as an industrially-proven, efficient GSTE model-checking tool. The short-term need was for algorithms for rudimentary reasoning with assertion graphs — implication and model-checking under assumptions — so we sought to develop efficient algorithms to perform these operations directly on assertion graphs (with terminal satisfiability), exploiting the existing GSTE model-checking engine as much as possible.

2.2 Monitor Circuits fromAssertion Graphs

Our algorithms for reasoning about assertion graphs rely on an efficient (linear space and time) algorithm for constructing circuits from assertion graphs, which was inspired by efficient methods for generating circuits from regular expressions [15,14,11]. The construction is rather intricate and is described elsewhere [7]. Here, we give a brief overview.

Given an assertion graph $G$, we construct a monitor circuit for $G$. A monitor circuit is simply a small circuit that watches, without interfering, the system being verified and

¹ Simulation by a conventionally labeled ∀-automata can be done with twice as many states; simulation by a normal ∃-automata requires an exponential blow-up. We would like to thank the anonymous reviewers for suggesting the construction for simulation via conventional ∀-automata, and for pointing out that there cannot be a general sub-exponential construction to simulate assertion graphs via normal ∃-automata or vice-versa.
flags whether or not the system is obeying some user-specified correctness property. In this case, the monitor circuit has inputs corresponding to the atomic propositions $AP$ that are used in $G$. The monitor circuit has a single output $\text{accept}$, which is true iff the trace that has been observed on the inputs would be accepted by $G$. The circuit is a Mealy machine, so the value at the inputs is immediately reflected at the $\text{accept}$ output. The circuit also has an $\text{init}$ input, which initializes the internal state of the circuit; $\text{init}$ is asserted at the same time that the first state of the execution trace is presented at the inputs, and then de-asserted from then on. See Figure 2.

Intuitively, the monitor circuit has an internal copy of the assertion graph and keeps track of paths by placing tokens on the edges in its copy. In theory, each token represents a path that ends on that edge at that clock cycle, and the token remembers the history of which antecedents and consequents were true during preceding clock cycles. At each clock cycle, tokens can update their histories and advance to the next edge, possibly splitting into multiple tokens if there are multiple out-going edges. The circuit accepts a trace iff all tokens represent accepting paths. The key insight to making this construction efficient is that the tokens can actually be almost memoryless. The only history necessary is to distinguish between three different kinds of pasts: (1) if an antecedent has failed already, this path and its continuations will always accept, so they need not be tracked any further, (2) if all antecedents and all consequents so far have succeeded, then this path currently accepts, but its continuations might not, and (3) if all antecedents have succeeded, but at least one consequent has failed, then this path currently rejects, but its continuations might eventually accept if an antecedent fails in the future. All paths with the same history that arrive at the same edge at the same time will share the same future, so their tokens can be merged. Hence, the constructed monitor circuit has a structure that exactly corresponds to the assertion graph, with two state bits per edge to track the two kinds of tokens, and a constant amount of circuitry per edge and per vertex to update the tokens appropriately. The constructed circuit is clearly linear-size compared to $G$.

3 Assertion Graph Implication

We now consider determining whether one assertion graph $G_1$ implies another assertion graph $G_2$, or, equivalently, whether $L(G_1) \subseteq L(G_2)$.

3.1 Implication via Product Construction

The monitor circuit construction immediately yields an obvious way to determine whether $L(G_1) \subseteq L(G_2)$:

1. Build circuits $C_1$ and $C_2$ for the assertion graphs $G_1$ and $G_2$.
2. Tie the inputs together.
3. Verify on the combined machine, using GSTE or any other model checking method, whether $\text{accept}_1 \Rightarrow \text{accept}_2$ in all reachable states.

The disadvantage of this approach is that we are building circuits for both $G_1$ and $G_2$, rather than using $G_2$ as a specification, potentially increasing the possibility of state explosion. Instead, we would like to harness the efficiency of GSTE and avoid adding $G_2$ to the state space.
3.2 Implication via GSTE

Given a circuit $M$ and an assertion graph $G$, GSTE model checking provides an efficient way to determine whether $L(M) \subseteq L(G)$, or equivalently, whether $M$ is a model of $G$, notated $M \models_T G$. (The “$T$” is for terminal satisfiability.) With our construction of a circuit from an assertion graph, one might consider generating a circuit $C$ doubling. We prove that $C$ is not accepted by $M$.

Given a circuit $C$.

1. Without loss of generality, we assume that the initial vertex $v_0$ of $G_2$ has in-degree of 0. (If this is not the case, we can modify $G_2$ by creating a duplicate initial vertex $v_0'$, which has the same incoming and outgoing edges as $v_0$, and then we delete the incoming edges to the true initial vertex $v_0$.)

2. Apply the monitor circuit construction to $G_1$, resulting in circuit $C_1$.

3. Modify $G_2$ to work with $C_1$, creating a new assertion graph $G_2'$:
   
   a) The new graph $G_2'$ has all of the same vertices as $G_2$.

   b) For every edge $e$ in $G_2$ from vertex $v_i$ to vertex $v_j$, create two edges $e'$ and $e''$, both from vertex $v_i$ to vertex $v_j$. Set

   $$\text{ant}(e') = \begin{cases} 
   \text{ant}(e) \land \text{accept} \land \text{init} & \text{if } v_i = v_0 \\
   \text{ant}(e) \land \text{accept} \land \neg \text{init} & \text{otherwise}
   \end{cases}$$

   $$\text{ant}(e'') = \begin{cases} 
   \text{ant}(e) \land \neg \text{accept} \land \text{init} & \text{if } v_i = v_0 \\
   \text{ant}(e) \land \neg \text{accept} \land \neg \text{init} & \text{otherwise}
   \end{cases}$$

   The consequents do not change: $\text{cons}(e) = \text{cons}(e') = \text{cons}(e'')$. Edge $e'$ is a terminal edge in $G_2'$ iff edge $e$ is a terminal edge in $G_2$. Edge $e''$ is not a terminal edge.

   c) Add init and accept to the atomic proposition set.

   Figure 3 shows this construction applied to the assertion graph from Figure 1.

4. Use GSTE to model check whether $C_1 \models_T G_2'$. The result is true iff $G_1 \Rightarrow G_2$.

**Proof** that $G_1 \Rightarrow G_2$ implies $C_1 \models_T G_2'$:

Suppose $C_1 \not\models_T G_2'$. Then, there exists a trace $\sigma'$ of $C_1$ and a terminal path $\rho'$ of $G_2'$, of the same length, where $\sigma'$ satisfies all the antecedents in $\rho'$, but fails at least one consequent. Define the trace $\sigma$ by projecting out the accept and init signals from each state of $\sigma'$. Define path $\rho$ in $G_2$ formed from $\rho'$ by mapping back through the edge doubling. We prove that $\sigma$ is a witness that $G_1 \not\Rightarrow G_2$ by showing that:

1. $\sigma \models_T G_1$.
2. $\rho$ is a terminal path in $G_2$.
3. $\sigma$ satisfies the antecedents along $\rho$.
4. $\sigma$ fails at least one consequent along $\rho$. 
We are given that

\begin{align*}
\text{Claim 4:} & \quad \text{Recall that} \quad \text{Claim 3:} \\
& \quad \text{Because} \quad C \text{ of the edge in} \quad \rho : = \text{RI} \\
& \quad \text{Edge labels are defined as in Figure 1, with AI} \quad : = \text{RI} \\
& \quad \text{result of modifying the assertion graph in Figure 1 using the construction from Section 3.2.}
\end{align*}

We know that

\begin{align*}
\text{Claim 1:} & \quad \text{Suppose} \quad G \sigma \quad \text{propositions) such that} \\
& \quad \text{For the initial state of} \quad G \sigma \quad \text{atomic proposition values that are the same as} \\
& \quad \text{Since} \quad \sigma \quad \text{is a Mealy machine, we can always compute the value of} \quad 1 \\
& \quad \text{Thus,} \quad \sigma' \quad \text{is a trace of} \quad \sigma' \quad \text{by construction. The resulting trace} \quad \sigma' \quad \text{has atomic proposition values that are the same as} \\
& \quad \text{Unpacking the circuit} \quad \sigma' \quad \text{with values for} \quad \text{init} \quad \text{and} \quad \text{accept} \\
& \quad \text{Initial state of} \quad \sigma' \quad \text{set} \quad \text{init} \quad \text{to be} \quad 1 \quad \text{in all other states of} \quad \sigma' \quad \text{set} \quad \text{init} \quad \text{to be} \quad 0 \\
& \quad \text{Because} \quad C_1 \quad \text{is a Mealy machine, we can always compute the value of} \quad \text{accept} \\
& \quad \text{Thus,} \quad \sigma' \quad \text{is a trace of} \quad C_1 \quad \text{by construction. The resulting trace} \quad \sigma' \quad \text{has atomic proposition values that are the same as} \\
& \quad \text{Since} \quad \sigma' \quad \text{ends with} \quad \text{accept} \quad \text{true, the constructed path} \quad \rho' \quad \text{ends at a terminal edge in} \quad G'_2.
\end{align*}

\textbf{Fig. 3. Assertion Graph Modified to Consider Only Accepting Paths.} This figure shows the result of modifying the assertion graph in Figure 1 using the construction from Section 3.2. Edge labels are defined as in Figure 1, with AI := accept ∧ init, AN := accept ∧ ¬init, RI := ¬accept ∧ init, and RN := ¬accept ∧ ¬init. The implication construction modifies an assertion graph so that it considers only the accepting paths of the other assertion graph. The basic idea is to double all edges, with one edge guessing that the path is accepting and the other edge guessing that the path is rejecting. Because these guesses are in the antecedents, paths that guess wrong are disregarded. The modification also ensures that the monitor circuit is initialized properly, via the init signal.

\textbf{Claim 1:} We know that \(\sigma'\) satisfies the antecedents of \(\rho'\). Therefore, the circuit \(C_1\) is initialized properly, because the antecedent constrain the init signal. Also, the accept signal is true in the last state of \(\sigma'\), because \(\rho'\) ends on a terminal edge, so \(\sigma\) is an input sequence that would end up with \(C_1\) accepting. Therefore, \(\sigma \models_T C_1\), by the construction of \(C_1\).

\textbf{Claim 2:} \(G'_2\) is created by doubling the edges of \(G_2\). Undoing the doubling maps the path back to a path on \(G_2\). Since \(\rho'\) ended on a terminal edge in \(G'_2\), the corresponding edge in \(G_2\) must also be a terminal edge, so \(\rho\) is a terminal path.

\textbf{Claim 3:} Recall that \(\sigma'\) satisfies the antecedents of \(\rho'\). The path \(\rho\) has antecedents that are strictly weaker than the corresponding antecedents in \(\rho'\), because they are missing the conjuncts about accept and init. Therefore, \(\sigma\) satisfies the antecedents of \(\rho\).

\textbf{Claim 4:} We are given that \(\sigma'\) fails at least one consequent along \(\rho'\). The consequents are the same in \(\rho\) and \(\rho'\), so \(\sigma\) must fail the corresponding consequent along \(\rho\).

\textbf{Proof} that \(G_1 \Rightarrow G_2\) is implied by \(C_1 \models_T C'_2\):

Suppose \(G_1 \not\models G_2\). Then, there exists a trace \(\sigma\) (in the state space defined by the atomic propositions) such that \(\sigma \models_T G_1\), but \(\sigma \not\models_T G_2\). We will construct a trace \(\sigma'\) of \(C_1\) that is not accepted by \(G'_2\), witnessing that \(C_1 \not\models_T G'_2\).

We construct \(\sigma'\) by augmenting the state space of \(\sigma\) with values for init and accept. For the initial state of \(\sigma'\), set init to be 1. In all other states of \(\sigma'\), set init to be 0. Because \(C_1\) is a Mealy machine, we can always compute the value of accept by feeding \(\sigma\) as input to \(C_1\). Thus, \(\sigma'\) is a trace of \(C_1\) by construction. The resulting trace \(\sigma'\) has atomic proposition values that are the same as \(\sigma\) and has accept true in the last state (because \(\sigma\) is accepted by \(G_1\)).

Since \(\sigma \not\models_T G_2\), we know there exists a terminal path \(\rho\) in \(G_2\), of the same length as \(\sigma\), such that \(\sigma\) satisfies all the antecedents in \(\rho\) but fails at least one consequent. Construct path \(\rho'\) in \(G'_2\) as follows: Match \(\rho\) edge-for-edge, picking the accept or \(\neg\) accept version of the edge in \(G'_2\) depending on the value of the accept signal in \(\sigma'\). Since \(\sigma'\) ends with accept true, the constructed path \(\rho'\) ends at a terminal edge in \(G'_2\).
Now, we see that $\sigma'$ satisfies the antecedents in $\rho'$ because the states/antecedents are the same as in $\sigma$ and $\rho$ (with the accept' or $\neg$accept' edge chosen correctly by the construction of $\rho'$). On the other hand, $\sigma$ fails at least one consequent of $\rho$, so $\sigma'$ must fail the corresponding consequent of $\rho'$, since the consequents are the same in both paths. Therefore, $\sigma'$ witnesses that $C'_1 \not\models_T G'_2$.

4 Model Checking under an Assumption

Besides assertion graph implication, the other main reasoning tool we wanted was how to perform GSTE model checking under an assumption. We notate this problem $C_0 \models_T (G_1 \Rightarrow G_2)$, meaning that all behaviors of a circuit $C_0$ that satisfy an assertion graph $G_1$ (the assumptions) also satisfy the assertion graph $G_2$. This construction is closely related to the preceding one.

The basic idea is that we build a monitor circuit $C_1$ for $G_1$ and augment $C_0$ with this monitor, in a non-interfering manner. Then, we modify $G_2$ so that it ignores traces that are not accepted by the monitor, resulting in verifying only the behaviors of $C_0$ that satisfy the assumptions of $G_1$. An alternative intuition is to consider the implication construction in Section 3.2 as the special case of model checking a completely unconstrained machine under the assumption of $G_1$; here, we constrain the inputs of $C_1$ to be the behaviors of $C_0$.

1. Without loss of generality, we assume that the initial vertex $v0$ of $G_2$ has in-degree of 0.
2. Build the monitor circuit $C_1$ from $G_1$.
3. Connect the inputs of $C_1$ to the state variables of $C_0$. In this way, $C_1$ will watch $C_0$ and indicate accept/reject depending on whether or not $C_0$’s behavior obeys the assertion graph $G_1$. Call this combined circuit $C_{01}$.
4. Build $G'_2$ from $G_2$ by edge-doubling and modifying the antecedents, exactly as in the implication construction.
5. $C_{01} \models_T G'_2$ iff $C_0 \models_T (G_1 \Rightarrow G_2)$.

Proof:
The constraints on init in the antecedents of $G'_2$ guarantee that we only consider traces in which $C_1$ is properly initialized.

The monitor circuit $C_1$ has no effect on $C_0$. Therefore, $C_{01}$ has the same traces as $C_0$, except for some additional state bits that determine whether or not $G_1$ would have accepted the trace.

Any path in $G'_2$ that guesses accept/reject incorrectly on any edge will have its antecedent fail and will be ignored. For any path in $G_2$, there will always exist a corresponding path in $G'_2$ that guesses accept/reject correctly for every edge. The only paths that are checked are the ones that are terminal in $G'_2$, which means that they were terminal in $G_2$ as well, and also that the accept signal is true, which means that $G_1$ would have accepted the path. Thus, we check only the traces of $C_0$ that satisfy $G_1$. 


5 Experimental Results

We have implemented the above algorithms into Intel’s Forte verification system\(^2\) and report their effectiveness on two verification tasks taken from real, industrial problems.

5.1 Decomposing a Verification Property: Verifying a Memory Unit

The first example is the verification of an industrial memory unit, using the assertion graph from Figure 1. Verifying this assertion graph on the memory unit by directly applying GSTE model checking required 56 seconds.

Alternatively, we manually decomposed the assertion graph into two smaller assertion graphs \(G_1\) and \(G_2\), which separates the memory behavior from the selection and alignment specifications. See Figure 4. GSTE model checking these two specifications on the memory unit took 28 seconds and 7 seconds, respectively. Note that, because of the \(\forall\) semantics, we can produce the assertion graph for \(G_1 \land G_2\) simply by having the two graphs share a single initial vertex. Accordingly, we verified that \(G_1 \land G_2\) implies the original assertion graph, using the implication construction from Section 3.2. This step took 0.3 seconds, and the generated monitor circuit for \(G_1 \land G_2\) had 5338 gates and 44 latches — far smaller than the memory unit. The total verification runtime was, therefore, less than 36 seconds, compared to the original 56 seconds.

Obviously, for such a small property, the time savings are not enough to repay the effort of decomposing the property. Nevertheless, we see that the decomposition does reduce the overall model-checking complexity, and our new algorithm does enable verifying automatically that a combination of sub-properties implies a more complex one. For larger, more challenging verification tasks, being able to decompose a difficult

\(^2\) Forte is available for download at http://www.intel.com/software/products/opensource/tools1/verification/ but our new algorithms are not yet part of the the standard distribution.
Fig. 5. Content-Addressable Memory (CAM). A CAM allows finding data by matching the value of a tag. In this CAM, a 64-bit data value is written at the same time as an 8-bit tag. Values can be read by supplying the correct tag. The \( \text{match}[i] \) signals indicate which of the 16 tags matches a supplied tag. The “outputs” on the right are for verification only: \( \text{hit} = \bigvee_i \text{match}[i] \), and \( \text{matchout}[i] = \text{datamem}[i] \) if \( \text{match}[i] \) is true, otherwise \( \text{matchout}[i] = 0 \). The overall CAM has 1152 latches. Our verification will cut the circuit at the dotted line. We first verify the tag portion of the circuit, then use that assertion graph as an assumption to verify the data portion of the circuit.

property into smaller ones, verify the smaller properties, and then conclude that the original property holds, is extremely useful.

5.2 GSTE with an Assumption: Content-Addressable Memory

The second example is from the verification of a content-addressable memory (CAM). This example illustrates GSTE model checking under an assumption.

A CAM allows finding data in its memory by matching a given tag value in an array of stored tags, i.e., by matching a value to the content of storage locations, rather than by address. CAMs are ubiquitous in modern microprocessors, where they are used to cache small amounts of frequently accessed data (e.g., in caches, TLBs, and assorted other buffers). Figure 5 shows the CAM for this example.

We wish to verify that the CAM as a whole satisfies the assertion graph \( G_2 \) in Figure 6. Verifying this assertion graph on the CAM by directly applying GSTE model checking required 15 seconds. Alternatively, to evaluate our algorithm for model checking under an assumption, we first verified the correct operation of the tag portion, in isolation, against the tag-correctness assertion graph \( G_1 \) in Figure 7. This verification took 0.8 seconds. Then, we abstracted away the tag portion of the CAM and used our algorithm for verification under an assumption to verify that \( G_2 \) holds, assuming that \( G_1 \) does:

\[
(\text{data portion of CAM}) \models_T (G_1 \Rightarrow G_2).
\]

This verification took 7 seconds. Altogether, the decomposed verification was roughly twice as fast as the direct approach, and the monitor circuit for \( G_1 \) had only 12 latches, an order of magnitude less than the tag memory that was abstracted away.
Fig. 6. CAM Correctness Specification. This assertion graph specifies that if a tag and data values are written, followed by an arbitrary number of cycles in which they are not overwritten, followed by a read by the same tag, then the CAM must indicate a hit, and the \texttt{matchout} signal must give the correct data value at any matching locations.

\begin{align*}
\text{TAG\_WRITE} & := (\texttt{twrite} = 1) \land (\texttt{taddr} = A) \land (\texttt{tagin} = T) \\
\text{DATA\_WRITE} & := (\texttt{dwrite} = 1) \land (\texttt{daddr} = A) \land (\texttt{din} = D) \\
\text{TAG\_RETAIN} & := (\texttt{twrite} = 0) \lor (\texttt{taddr} \neq A) \\
\text{DATA\_RETAIN} & := (\texttt{dwrite} = 0) \lor (\texttt{daddr} \neq A) \\
\text{TAG\_READ} & := (\texttt{aread} = 1) \land (\texttt{tagin} = T) \\
\text{TAG\_RESULT} & := (\texttt{hit} = 1) \land \forall i[(i = A) \Rightarrow (\texttt{match}[i] = 1)] \\
\text{DATA\_RESULT} & := \forall i[(i = A) \Rightarrow (\texttt{matchout}[i] = D)]
\end{align*}

Fig. 7. Tag Correctness Specification. This assertion graph specifies that if a tag is written, not overwritten for an arbitrary number of cycles, and then the same tag is presented, the \texttt{hit} signal and the correct \texttt{match} signal must be asserted. We first verify this property on the tag portion of the circuit. Then, we use this assertion graph as an assumption to abstract away the tag portion of the circuit when verifying the whole CAM.

As in the previous example, the time savings on a small verification task are not enough to repay the time to manually decompose the problem. Nevertheless, this example does demonstrate how our new algorithm runs efficiently and enables decomposing a harder verification problem into smaller, easier ones. In general, we envision using this style of proof for simplifying complex verification tasks, and also for verification with IP cores (portions of a circuit supplied by third-parties, for which functionality is specified, but internal details are not visible) as well as the verification of partial or incomplete circuits.

6 Conclusion and Future Work

We have presented new algorithms for reasoning about GSTE assertion graphs. These algorithms appear efficient in theory, and preliminary experiments indicate that they are efficient in practice as well. Given the increasing practical importance of GSTE model
checking, the need for (practically efficient) supporting theory and algorithms is great. This work is a first step.

The practical success of GSTE is the justification for studying assertion graphs. In theory, assertion graphs are simply a new variety of automata, with equivalent expressive power to established varieties of automata, so an obvious, fundamental question is to elucidate whether and how GSTE is gaining efficiency advantages over older techniques. Do assertion graphs facilitate writing specifications in a manner that enables more efficient model checking? Are other aspects of GSTE, completely separable from assertion graphs, more important for efficiency? Can we leverage these ideas with other verification methods? On the other hand, perhaps the practical successes have been primarily the result of the overall verification methodology, the types of verification tasks undertaken, or the skill of the verification engineers. Assertion graphs and GSTE give symbolic-trajectory-evaluation-based approaches comparable expressive power to other model-checking approaches, so it is now possible to make direct comparisons.

Focusing on assertion graphs, research is needed on composing and decomposing assertion graphs. For example, given the $\forall$ semantics, it should be possible to decompose a large assertion graph into the conjunction of smaller ones, as is possible in formalizations of timing graphs [2]. Such a decomposition could reduce the complexity of model checking.

A related, and perhaps more immediately applicable, direction for research is to look for transformations and inference rules for assertion graphs. For example, it is easy to see that adding edges, weakening antecedents, or strengthening consequents are all operations that cannot enlarge the set of traces accepted by an assertion graph. Perhaps it is possible to develop a powerful set of inference rules to reason about assertion graphs, without having to perform model checking.

The work presented here are fundamental building blocks for reasoning about assertion graphs. An important next step is to develop compositional verification theorems, so that we can automate the process of stitching together partial verification results.

Finally, although assertion graphs are interesting to consider in isolation as a variety of automata, in practice their use is intimately tied to GSTE model checking. This connection suggests that it may be interesting to consider weaker notions of implication (and equivalence). For example, rather than defining $G_1 \Rightarrow G_2$ to mean $L(G_1) \subseteq L(G_2)$, we could use the weaker definition: $\forall$ circuits $M. (M \models G_1) \Rightarrow (M \models G_2)$. Under all the different acceptance conditions, we have constructed small assertion graphs $G_1$ and $G_2$ such that $L(G_1) \neq L(G_2)$, but that are equivalent under the weaker definition because no circuit satisfies either one. (The intuition is that real circuits cannot generate arbitrary sets of strings, e.g., a circuit can always be run for one more clock cycle, generating a longer string.) We do not know whether the difference between these definitions is theoretically interesting or practically important.

In general, increasing evidence demonstrates the practical value of GSTE and assertion graphs, but the supporting infrastructure is underdeveloped. Much work remains to be done.
References


Towards Diagrammability and Efficiency in Event Sequence Languages

Kahti Fisler
Department of Computer Science
WPI (Worcester, MA, USA)
kfisler@cs.wpi.edu

Abstract. Many industrial verification teams are developing suitable event sequence languages for hardware verification. Such languages must be expressive, designer friendly, and hardware specific, as well as efficient to verify. While the formal verification community has formal models for assessing the efficiency of an event sequence language, none of these models also accounts for designer friendliness. We propose an intermediate language for event sequences that addresses both concerns. The language achieves usability through a correlation to timing diagrams; its efficiency arises from its mapping into deterministic weak automata. We present the language, relate it to existing event sequence languages, and prove its relationship to deterministic weak automata. These results indicate that timing diagrams can become more expressive while remaining more efficient for symbolic model checking than LTL.

1 Introduction

The increasing adoption of formal verification has led to a flurry of research into property specification languages for hardware verification. Large-scale efforts include Accellera’s standardization of Sugar [1], Synopsys’ OVA [13], and Intel’s FTL [4]. Generally speaking, these are event sequence languages: they allow designers to express sequences of events to monitor and check during verification. The proliferation of work from industry on event sequence languages emphasizes that they must be designer friendly, expressive, and specific to the hardware domain in addition to efficient to verify. Although practical experience and theoretical results give insights into how to achieve these goals individually, few formal models attempt to address usability and efficiency simultaneously.

In the space of event sequence languages, timing diagrams provide an appealing combination of usability and efficiency. Designers have established their utility by regularly employing them as an informal design tool. Mappings from formalized timing diagrams to deterministic weak automata [8] provide effectively linear symbolic verification algorithms [5]. That timing diagrams are not more widely used as event sequence languages suggests that they lack the expressiveness needed in industrial verification [3]. Their combination of utility and

* This research is supported through NSF grant CCR-0132659.
efficiency, however, raises an interesting question: how expressive can we make an event sequence language while retaining both diagrammability and efficiency?

This paper explores this question by proposing a (textual) intermediate language for capturing event sequence languages. To target diagrammability, we design the core of the language around timing diagrams. To target expressiveness, we extend the core language to capture constructs from other event sequence languages. To target efficiency, we syntactically characterize which expressions in this language map to deterministic weak automata. The results of this work are twofold: first, our language provides a framework in which to assess both usability and efficiency of other event sequence languages; second, our characterization proves that timing diagrams can be extended with several new features—such as partial orders between events, interleaved environmental assumptions, escaping conditions, and event clocks—without losing their mapping to deterministic weak automata. Our long-term goal is to develop formal models that simultaneously characterize both usability and efficiency in event sequence languages. This paper focuses on the efficiency of verifying our proposed language; future papers will treat formal models of diagrammability as a measure of usability.

2 Preliminaries

2.1 Event Sequences and Timing Diagrams

Event sequences, as their name implies, capture sequences of events on signals in a design; they express properties for verification or simulation. Regular expressions and linear temporal logic have similar goals, but also some subtle differences. Event sequences often monitor transitions on signals in the design, rather than just boolean values of propositions. In addition, event sequences generally capture timing constraints between events. While both regular expressions and linear temporal logic can capture these features, the resulting expressions can be rather cumbersome, especially in contrast to event sequences and timing diagrams. Figure 1 shows a simple example of the same event sequence expressed as a timing diagram, in linear temporal logic (LTL), and in Sugar.

Although timing diagrams present event sequences somewhat intuitively, they are not as expressive as some other event sequence languages. For example, textual event sequence languages easily express disjunctions, while diagrams in general capture disjunctive information poorly. The mapping from timing diagrams to weak automata, which does not hold for full LTL, demonstrates benefits to
this limited expressive power. The question, then, is how far we can push timing diagrams while retaining this mapping. The timing diagram shown in Figure 2, for example, expresses some disjunction as the order of events is left unspecified (a partial order rather than a total one). This extension adds expressive power without sacrificing diagrammability or weakness. We are interested in similar extensions based on constructs from modern event sequence languages.

2.2 Weak Automata

A Büchi automaton \( \langle Q, \Sigma, q_0, R, L, \mathcal{F} \rangle \) is weak if it has only one fair set and each of its strongly connected components has either all states fair or no states fair [10]. Weak automata are attractive in verification because symbolic cycle detection is effectively linear for weak automata, as opposed to quadratic for full LTL [5]. Deterministic weak automata are particularly interesting for their properties under complementation. Automata-based verification approaches complement automata that capture properties. In the general case, complementing a Büchi automaton can blowup the number of states exponentially. Complementing a deterministic weak automaton, however, requires only complementing the fair set; the structure of an automaton and its complement are otherwise identical. This represents a substantial savings in construction time, and more importantly, in the size of automata used to represent complemented properties.

3 An Intermediate Language for Event Sequences

This section presents a regular-expression-like notation for event sequences. We motivate the development of the language using the example timing diagram shown in Figure 2. We explain the semantics of the diagram informally; the formal details appear elsewhere [7].

To capture the diagram, the language must express transitions on signals and constraints (timing and ordering) between these transitions. Let propositional literals \((p, \neg q)\) denote boolean values and propositional variables annotated with arrows \(p \downarrow, p \uparrow\) denote falling and rising transitions, respectively. Let semicolons denote concatenation (temporal sequencing) of events. Using these notations and reading off the timing diagram from left to right suggests the expression \(\langle a \uparrow; b \uparrow; c \uparrow; a \downarrow; b \downarrow\rangle\). If we interpret semicolons as implying order between events (a common interpretation of concatenation), this expression is inconsistent with the semantics of the timing diagram. The rising transitions on \(a\) and \(b\) may

\[
C = \{a \uparrow, b \uparrow, c \uparrow, a \downarrow; b \downarrow\}
\]

\[
T = \{(a \uparrow, c \uparrow; 2, 5, \text{true}),
\langle c \uparrow, a \downarrow, 1, \infty, \text{true}\rangle,
\langle a \downarrow, b \downarrow, 3, 9, \text{true}\}\}.
\]

**Fig. 2.** A timing diagram with partial orders and its mapping into an event sequence.
occur in any order since no constraint orders them (the falling events on $a$ and $b$, in contrast, must occur in order). The event sequence language must therefore support partial, rather than only total, orders between events.

Timing diagrams consist of totally-ordered regions within which individual events are partially ordered. For sake of generality, our event sequence language supports hierarchical combinations of ordered, unordered and iterated groups of events. In the formal syntax and semantics that follows, we refer to these groups of events as clusters. We capture partial orders within unordered clusters using a separate annotation for transition (timing) constraints between events; a timing constraint specifies the events covered, lower and upper bounds on the time between the events, and the clock against which the bounds are measured (true specifies the system clock). This approach treats constraints between events uniformly, whether they occur in ordered or unordered clusters. Figure 2 shows the resulting event sequence for our example timing diagram.

### 3.1 Syntax

The timing diagram example suggests the following syntax for event sequences:

**Definition 1** Clusters are defined hierarchically as follows:

- An event is a conjunction of values of and transitions on variables that contains at least one transition. Propositional literals ($p, \neg q$) denote boolean values; propositions with arrows ($p \downarrow, p \uparrow$) denote transitions.
- A cluster is either:
  - a single event, or
  - an unordered cluster $\{C_1, \ldots, C_k\}$ where each $C_i$ is a cluster, or
  - an ordered cluster $(C_1; \ldots; C_k)$ where each $C_i$ is a cluster, or
  - a repeating cluster $C^M$ where $C$ is a non-repeating cluster and $M$ is a positive number, $\ast$, or $+$ (called a repetition marker; markers $\ast$ and $+$ are called unbounded).

An event sequence consists of a (top level) cluster and three kinds of modifiers. Temporal constraints, already motivated, may be relative to a designer-specified event clock, as captured by a boolean expression (this is a common feature in event sequence languages). To indicate that certain variables hold value during regions (between events) in a diagram, holding patterns constrain variable values within clusters. To allow portions of diagrams to serve as assumptions rather than requirements, escape conditions capture circumstances under which the sequence should be immediately rejected or accepted.

**Definition 2** An event sequence is a tuple $\langle C, H, T, S \rangle$ where $C$ is a cluster, $H$ (the holding patterns) is a partial function from $C$ to propositional formulas, $T$ is a set of temporal constraints and $S$ is a set of escape conditions.
\[ C = \langle a \uparrow^+; \{b \uparrow, c \uparrow\}; d \uparrow\rangle \]
\[ H = \{b \uparrow, c \uparrow\} \rightarrow a \]
\[ T = \{c \uparrow, d \uparrow, 2,5, \text{true}\} \]
\[ S = \{\text{accept-if-don’t-complete}(a \uparrow^+)\} \]

![Figure 3. A sample event sequence and an example of its semantics.](image)

- A **temporal constraint** is a tuple \( \langle e_1, e_2, l, u, \text{clk} \rangle \) where \( e_1 \) and \( e_2 \) are (uniquely identified\(^1\)) events in \( C \), \( l \) is a positive integer, \( u \) is either an integer at least as large as \( l \) or the symbol \( \infty \), and \( \text{clk} \) is a boolean expression (the clock for the constraint; \text{true} indicates the system clock). Events \( e_1 \) and \( e_2 \) may lie in different clusters, but then they must lie in the same repeated clusters.
- An **escape condition** has one of three types, where \( X \) is a boolean expression over events (the events need not be in \( C \)) and \( C' \) is a cluster within \( C \):
  - “accept if don’t complete \( C' \)”
  - “reject if see \( X \) in \( C' \)”
  - “accept if see \( X \) in \( C' \)”

Figure 3 illustrates an event sequence of some number of rising transitions on \( a \), followed by rising transitions on \( b \) and \( c \) (in either order), followed by a rising transition on \( d \). The transition on \( d \) must occur between 2 and 5 ticks (inclusive) after the transition on \( c \) (the timing constraint), signal \( a \) must remain true until the transition on \( d \) occurs (the holding pattern), and the rest of the sequence is only checked if the transition on \( a \) occurs (the escape condition).

The language contains some redundancy for sake of clarity: ordered clusters, for example, can be viewed as unordered clusters plus timing constraints. To simplify the semantics and proofs, we assume that all sequences are in reduced form, in which all clusters \( C^+ \) are replaced with \( \langle C; C^* \rangle \), all \( C^M \) for a concrete number \( M \) are replaced with an ordered cluster of \( M \) copies of \( C \), and all ordered clusters \( \langle C_1; \ldots; C_k \rangle \) are replaced with unordered clusters and timing constraints that require an event from each \( C_i \) to occur before an event from each \( C_{i+1} \).

### 3.2 Semantics

The semantics of event sequences is defined in terms of languages over infinite words, where each character in a word is an assignment of boolean values to variables. An infinite word models an event sequence if there exists a mapping from the clusters in the sequence to ranges of indices into the word (herein called *windows*) such that the windows assigned to each cluster preserve the cluster’s constraints; these mappings are called *index assignments*.

As an example, consider the event sequence and word shown in Figure 3. The word is divided into windows per cluster (demarced by solid lines), and subwindows as necessary for nested clusters (demarced by dashed lines). We first formalize the mappings from clusters to windows.

\(^1\) A numbering scheme could distinguish syntactically similar events.
Definition 3 Given a word $W$, a window of $W$ is a subword of $W$; a pair of indices into $W$, denoted $[i, j]$ where $i \leq j$, defines a window. Furthermore,

- An individual index $i$ defines a trivial window $[i, i]$.
- Window $[i_1, i_2]$ contains window $[i_3, i_4]$ iff $i_1 \leq i_3$ and $i_4 \leq i_2$.
- Window $[i_1, i_2]$ is earlier than window $[i_3, i_4]$ iff $i_1 < i_3$ or $i_1 = i_3$ and $i_2 < i_4$.
- Given a window $w = [\text{start}, \text{end}]$, a sequence $[s_1, e_1], \ldots, [s_k, e_k]$ forms a non-overlapping covering sequence of windows for $w$ if $s_1 = \text{start}$, $e_k = \text{end}$, and for all $1 \leq j < k$, $e_j < s_{j+1}$.

Definition 4 A (partial) index assignment for event sequence $V$ and word $W$ is a (partial) function from the clusters in (including nested within) $V$ to non-empty sets of windows of $W$.

A window must meet certain requirements in order to capture the constraints of a cluster. The following definitions formalize those requirements.

Definition 5 Let $E = v_1 \land \ldots \land v_k$ where each $v_i$ is a proposition, its negation, or a rising or falling transition on a proposition. Let $W$ be a word and $i$ an index into $W$. Let $W_i(q)$ denote the value of proposition $q$ at index $i$ into $W$. Index $i$ satisfies $E$ if for every $v_i$, $W_i(p) = 0$ if $v_i = \neg p$, $W_i(p) = 1$ if $v_i = p$, $W_i(p) = 0$ and $W_{i+1}(p) = 1$ if $v_i = p \uparrow$, and $W_i(p) = 1$ and $W_{i+1}(p) = 0$ if $v_i = p \downarrow$.

Definition 6 Given an unordered cluster $C = \{C_1, \ldots, C_k\}$, a schedule of $C$ is a sequence $CO_1, \ldots, CO_j$ of non-empty subsets of $C$ such that

- $CO_1, \ldots, CO_j$ partition $C$,
- In every $CO_i$ that contains multiple elements of $C$, all elements of $CO_i$ are single events (rather than other complex clusters), and
- For each timing constraint $\langle e_1, e_2, l, u, clk \rangle$ such that $e_1 \in CO_i$ and $e_2 \in CO_j$, $i < j$.

Definition 7 Let $V$ be an event sequence, $W$ a word, and $I$ a partial index assignment for $V$ and $W$. $I$ is structurally valid iff for every cluster $C$ in $V$:

- If $C$ is an event, then for every $[i, i] \in I(c)$, $i$ satisfies $C$ (Defn 5).
- If $C$ is a repeating cluster $C'\ast$, then for every $wp$ in $I(C'\ast)$ there exists a natural number $m$ and some sequence $wp_1, \ldots, wp_m$ of non-overlapping covering windows for $wp$ such that each $wp_i \in I(C')$.
- If $C$ is an unordered cluster $\{C_1, \ldots, C_k\}$, then for every window $w \in I(C)$ there exists a schedule $CO_1, \ldots, CO_j$ for $C$ and a sequence $w_1, \ldots, w_j$ of non-overlapping covering windows for $w$ such that for all $i \leq h \leq j$ and all $e \in CP_h$, $w_h \in I(e)$.
**Definition 8** Let $V = \langle C, T, H, S \rangle$ be an event sequence, let $W$ be a word, and let $I$ be an index assignment for $V$ and $W$. $I$ is constraint valid for $V$ and $W$ iff

1. $I$ satisfies the holding patterns, in that for all clusters $C'$, every $x \in H(C')$ and every window $[w_1, w_2] \in I(C')$, every index $w_1 \leq i \leq w_2$ satisfies $x$, and

2. $I$ satisfies the timing constraints, in that for every $(e_1, e_2, l, u, clk) \in T$ and every $t_1 \in I(e_1)$ and $t_2 \in I(e_2)$ such that $t_1$ and $t_2$ fall in a common window for the smallest cluster containing both $e_1$ and $e_2$, the number of indices satisfying $clk$ between $t_1$ and $t_2$ (inclusive) is within the range $[l, u]$.

Constraint validity handles timing constraints and holding patterns, but not escape conditions. The next two definitions handle escape conditions. **Definition 12** relates words and event sequences based on the existence of index assignments that may or may not invoke escape conditions. Given index assignment $I$, let $\overline{I}$ be the inverse of $I$ (mapping windows to sets of clusters).

**Definition 9** Let $V$ be an event sequence, $W$ a word, and $I$ a structurally valid index assignment for $V$ and $W$. Let $E$ be an escape condition of type “accept/reject if see $X$ in $C$.” Index $i$ into $W$ invokes $E$ under $I$ if $i \in I(C)$, $i$ satisfies $X$, and $I$ is defined for all clusters in the images of $\overline{I}$ for windows occurring before $i$. We also say that $I$ invokes an escape condition of $V$.

**Definition 10** Let $V$ be an event sequence, $W$ a word, and $I$ a structurally valid index assignment for $V$ and $W$. $I$ loops under escape condition $E$ if $E$ is of the form “accept if don’t complete $C$” and $I$ is defined for all clusters in the images of $\overline{I}$ for windows occurring before $i$, but not for a window containing $i$.

For the semantics to yield a deterministic procedure for checking whether a word satisfies an event sequence, index assignments must assign the fewest and earliest possible windows to clusters (in particular, this renders both * and scheduling deterministic). We formally define this notion of minimality as follows: 

**Definition 11** Let $V$ be an event sequence and let $W$ be a word. Let $I$ and $I'$ be non-equivalent index assignments for $V$ and $W$. Let $Rg$ denote the range of a function. $I \prec I'$ iff

1. the earliest window in one but not both of $Rg(I)$ and $Rg(I')$ is in $Rg(I)$, or

2. $Rg(I) = Rg(I')$ but for $w$, the earliest window such that $\overline{I}(w) \neq \overline{T}(w)$, $\overline{T}'(w) \subset \overline{I}(w)$.

Given a set $\Sigma$ of index assignments, $I \in \Sigma$ is minimal in $\Sigma$ iff for all $I' \in \Sigma$, $I \prec I'$. ($\prec$ does not order all pairs, but is sufficient for our theorems [9].)

We now define when a word models an event sequence:

**Definition 12** Let $V$ be an event sequence and let $W$ be a word. $W \models V$ if there exists a minimal and structurally valid index assignment $I$ for $V$ and $W$ such that $I$ is a total function and constraint valid, or $I$ loops under some escape condition in $V$, or $I$ invokes some escape condition in $V$. 
The semantics captures one occurrence of an event sequence, rather than the multiple occurrences needed to treat an event sequence as an invariant. The one-occurrence semantics offers two benefits: it provides a foundation for defining different multiple occurrence semantics [7], and it enables the mapping to weak automata. This restriction is not as limiting as it might seem: in prior work [8], we showed that relabeling fair sets and adding a few transitions constructs the automaton for a negated invariant event sequence (the machine most commonly needed for verification) from the machine that accepts one occurrence.

4 Relationship to Existing Event Sequence Languages

To motivate the intersection between our simultaneous goals of diagrammability and efficiency, this section shows how several features of existing event sequence languages do or do not map into the proposed intermediate language.

4.1 Timing Diagrams

Section 3 illustrated the connection between timing diagrams and our proposed event sequence language. The language presented here extends our previous results on the relationship between timing diagrams and weak automata [8] in two ways. The previous result held for timing diagrams with a total order on their transitions and a prefix of the diagram as an environmental assumption (as in, “if the rising transition on a occurs, then match the whole diagram”). As a corollary to the results in this paper, timing diagrams with partial event orders and multiple non-contiguous assumptions on the environment also map to deterministic weak automata. We view environment assumptions as events that are only constrained if they occur [6]; unlike other events, their failure to occur does not violate the diagram’s requirements. For the diagram in Figure 2, we could treat the two transitions on a as environment assumptions by rewriting the event chain using nested clusters (as \( \langle \{a \uparrow, b \uparrow, c \uparrow, a \downarrow\}; b \downarrow \rangle \) and adding “accept-if-don’t-complete” escape conditions on the two clusters for a.

The proposed language is more expressive than our current timing diagram formalization. Consider the cluster \( \langle a \uparrow^*; b \uparrow \rangle \). The timing diagram semantics requires all depicted transitions to occur unless an escape condition matches, so this expression (without escape conditions) is currently not expressible as a timing diagram (since \( a \uparrow \) might not occur). Similar examples involving repetitions also exist. Enriching the timing diagram notation could resolve some of these issues; this remains an issue for future work.

4.2 LTL, Sugar, and FTL

Sugar and FTL are similar in that each extends conventional LTL. Since there exist LTL formulas that cannot be captured by weak automata, certain FTL and Sugar formulas will not map into our intermediate language. Weakness primarily characterizes the location of fair sets in automata. In LTL, fairness
constraints arise from combinations of eventualities and cycles (the operators $\mathbf{U}$ and $\mathbf{G}$). Figure 4 shows automata that capture two formulas: $(p \lor q) \lor r$ and $p \lor (\mathbf{G}(q \lor r))$. The first example yields a weak automaton and corresponds to cluster $\langle (p^*; q^*; r^+) \rangle$. The second corresponds to cluster $\langle p^*; (q^*; r^+)^+ \rangle$ with escape condition “accept if don’t complete $r^+$”; this expression violates our syntactic restrictions for weakness presented in Theorem 3 (Section 5.3).

One key difference between these two formulas is that the second contains a repetition within its last cluster, while the first does not. This same difference characterizes the automata for the regular expressions $(aa)^*$ and $(aa)^*b$, the first of which cannot be captured by a deterministic weak automaton while the second one can. An automaton can recognize a nonrepeating final pattern without creating a fair set. This motivates our characterization of weakness: the final cluster cannot end with an unbounded repetition marker.

Certain other features of Sugar and FTL do not adversely impact weakness. FTL’s `change_on` and `reject_on` constructs indicate when a sequence should be immediately accepted or rejected; escape conditions capture such scenarios in the proposed intermediate language. For example, augmenting $(p \lor q) \lor r$ with escape condition “accept if see `reset` in $q$” would introduce a new state labeled `reset` with an incoming edge from the state for $q$; this automaton is also weak.

### 4.3 OVA

Of the recent event sequence languages discussed in this paper, OVA most closely matches the proposed language. Unlike Sugar and FTL, OVA does not explicitly support LTL or CTL operators. The OVA `istrue` construct maps into holding patterns, and their non-overlapping event clocks map into ours. Unlike the proposed language, however, OVA can express disjunction among sequences and negation of sequences. Our language does not support negation because negated sequences generally cannot be realized diagrammatically. Our language does, however, still support constructing deterministic weak automata for the negations of event sequences, as outlined at the end of Section 3.

### 5 Relationship to Deterministic Weak Automata

This section characterizes which sequences in our language map to deterministic weak automata; almost all do, with the exception of those with particular interactions between escape conditions and repeated clusters. We construct an automaton corresponding to the semantics, prove the construction sound, then characterize when the resulting machine is both weak and deterministic.
Given an event sequence $V$, we construct a Büchi automaton that accepts all words with a prefix that models $V$. Figure 5 illustrates the intuition behind the expansion. The construction recursively expands states corresponding to clusters until all states correspond to individual events. Holding patterns, escape conditions, and the ordering aspects of timing constraints are incorporated as this expansion proceeds. The durational aspects of timing constraints are handled in a final phase once all states correspond to individual events.

Each intermediate machine during the computation abstracts the final machine, in that if there is no edge from one abstract state to another, then there is no edge from any state in the expansion of the first to the expansion of the second in the final machine. For sake of space, we present the detailed algorithm only up through creating states for each event; this is sufficient for our theorems.

The construction creates edges between abstract states based on which clusters can precede or follow other clusters; it also relies on notions of the first and last subclusters that could be encountered in a cluster. These concepts match intuition. For sake of space, we defer all but the definition of next clusters to the full paper [9]; examples of all four notions follow the definition. The theorem in Section 5.2 also refers to first and last *events*, which are obtained by iterating the first and last computations on clusters until they contain only events.

**Definition 13** Let $C$ be a cluster immediately contained in a cluster $C^P$ (if $C$ has no enclosing cluster, next($C$) is empty). If $C^P = \langle C_1; \ldots; C_k \rangle$ and $C = C_i$ for $i < k$, then next($C$) is $C_{i+1}$ if $C_{i+1}$ is not an repeating-* cluster and $\{C_{i+1}\} \cup$ next($C_{i+1}$) if $C_{i+1}$ is an repeating-* cluster. If $C = C_k$, then next($C$) is next($C^P$).

If $C$ is an repeating-* cluster, next($C$) also includes $C$. The case for unordered clusters unions similar results over all possible schedules, and repeated clusters $C$ have next($C$) as $\{C\} \cup$ next($C^P$).

**Examples:** Given sequence $\langle C_1; C_2; C_3^*; C_3^+ \rangle$, next($C_2$)$=\text{next}(C_3)$=$\{C_3, C_4\}$ and prev($C_3$)$=\{C_2, C_3\}$. Given sequence $\langle C_1; \{C_{21}, C_{22}^*\}; \{C_{31}, C_{32}\}^*; C_4 \rangle$ with a timing constraint from $C_{21}$ to $C_{22}$, next($C_{21}$)$=\{C_{22}, C_{31}, C_{32}, C_4\}$. For the first and
last sets, first(\(\{C_{21}, C_{22}^*\}\))=\(\{C_{21}\}\), last(\(\{C_{21}, C_{22}^*\}\))=\(\{C_{22}\}\), and first(\(\{C_{31}, C_{32}^*\}\)) = last(\(\{C_{31}, C_{32}^*\}\)) = \(\{C_{31}, C_{32}^*\}\).

**Algorithm 1** To construct an automaton for event sequence \((C, T, H, S)\):

1. Create a state *Final* with a self loop and mark it fair.
2. Create a state for \(C\) and mark it initial, final, and unexpanded.
3. Repeatedly select an unexpanded state \(N\) for some non-event cluster \(C\) and
   - Add holding patterns and edges for the escape conditions for \(C\) to \(N\).
   - Expand \(N\) according to the type of \(C\) and remove \(N\).
   - If \(N\) was marked initial (resp. final), mark the new states for all first (resp. last) clusters of \(C\) initial (resp. final). Copy all other propositional annotations (including fair) from \(N\) to the new states from the expansion.
4. Add an edge from each state marked final to the state *Final*.

**Expand Repeated Clusters.** For a state for repeated cluster \(C^*\), add an edge from the state for each previous cluster of \(C^*\) to that for each next cluster of \(C^*\).

**Expand Unordered Clusters.** For a state \(N\) for unordered cluster \(C = \{C_1, \ldots, C_k\}\):

- For every schedule \(CO_1, \ldots, CO_h\) of \(C\), create a chain of abstract states \(CON_1, \ldots, CON_h\). For every non-self-loop edge coming into \(N\), add an edge from the same source to \(CON_1\). For every non-self-loop edge leaving \(N\), add an edge from \(CON_k\) to the target of the original edge.\(^2\)
- Eliminate unnecessary nondeterminism by merging states with the same incoming transitions and labels into single states (this shares common prefix states across the various permutations).
- If \(N\) had an edge to itself, add an edge from each sink state in the subgraph that expands \(N\) to each source state in the subgraph that expands \(N\).

**Handle Escape Conditions and Holding Patterns**

- For each escape condition \(E\) of the form “reject if see \(X\) in \(C\)” , create a new abstract state \(N_E\) for \(E\), label \(N_E\) with \(X\), add an edge from each abstract state corresponding to \(C\) to \(N_E\) and add a self-loop at \(N_E\).
- For each escape condition \(E\) of the form “accept if see \(X\) in \(C\)” , create a new abstract state \(N_E\) for \(E\), label \(N_E\) with \(X\), add an edge from each abstract state corresponding to \(C\) to \(N_E\), add a self-loop at \(N_E\), and mark \(N_E\) as fair (with a new fairness constraint).
- For each escape condition \(E\) of the form “accept if don’t complete \(C\)” , mark every abstract for \(C\) as fair (with a new fairness constraint).
- For each holding pattern \(h\) for cluster \(C\) and each abstract state \(N_C\) corresponding to or expanded from \(C\), add \(h\) as a propositional label to \(N_C\).

\(^2\)To reduce the machine size, we could perform a bisimilarity minimization on the subgraph of all states that expanded \(N\).
Following Algorithm 1, all states correspond to single events but the durations of timing constraints have not been enforced. We handle this using a similar algorithm to that in our prior work [8]. For sake of space, and since the expansion into events does not affect weakness or determinism by construction, we do not reproduce the details here. To handle the event clock $clk$ in a timing constraint over events $e_1$ and $e_2$, the construction adds a unique label for $clk$ to each state between $e_1$ and $e_2$, and creates an automaton that outputs this label whenever $clk$ is true. A final step cross-products the core machine with the clock machines; this does not affect weakness.

The results on determinism and weakness that follow apply to those event sequences that end with a concrete event rather than a repetition (for reasons motivated in Section 4.2). We call such sequences event chains.

**Definition 14** An event sequence $⟨C, T, H, S⟩$ is an event chain if the iterative expansion of last($C$) contains no repeated clusters.

### 5.1 Soundness

**Theorem 2.** Let $V$ be an event sequence and let $M$ be the automaton obtained for $V$ from Algorithm 1. Let $W$ be an infinite word. $M$ accepts $W$ iff $W \models V$.

**Proof Sketch:** Intuitively, the proof develops a correspondence between states in the abstract machines and the windows in the range of an index assignment for $W$ and $V$. The theorem follows from an argument that the windows occurring in accepted (resp. rejected) words correspond to accepting (resp. rejecting) paths through the automaton.

### 5.2 Characterization of Determinism

**Theorem 2.** Given an event chain, Algorithm 1 produces a deterministic automaton if all of the following conditions are satisfied:

- For every unordered cluster $\{C_1, \ldots, C_k\}$, the first events of each $C_i$ are pairwise logically inconsistent with those of each $C_j \neq C_i$ unless a timing constraint orders $C_i$ and $C_j$.
- For each repeated cluster $C^*$, the first events of $C$ are pairwise logically inconsistent with the first events of each next cluster of $C^*$ (other than $C$).
- For each “accept/reject when see $X$ in $C$” escape condition, $X$ is logically inconsistent with all holding patterns for $C$.

**Proof Sketch:** The machine is deterministic if the choice among multiple next states is deterministic. The construction yields multiple next states in four cases: possible transitions to the Final state, when choosing between schedules for an unordered cluster, possible skips of repeated clusters, and when invoking escape conditions. The restriction to event chains guarantees that states with transitions to Final have no other outgoing transitions. By construction, transitions into the states that expand clusters occur when a first event is recognized for that cluster. If these events are logically inconsistent, then the corresponding transitions must be deterministic. This covers the remaining cases.
5.3 Characterization of Weakness

We call a cluster \( C \) fair if there exists an escape condition of the form “accept if don’t complete \( C \)”. A cluster is all-fair if it is either fair or all of its sub-clusters are all-fair. A cluster is non-fair if neither it nor any of its sub-clusters is fair.

**Lemma 1.** If an event sequence contains no all-fair repeated clusters, then the automaton from Algorithm 1 requires only one fair set.

**Proof Sketch:** If no cycle contains states from more than one fair set, then a single fair set suffices. Cycles can contain states from multiple fair sets under two conditions. First, two “accept don’t complete” conditions could exist for clusters \( C_1 \) and \( C_2 \) where \( C_1 \) contains \( C_2 \). In this case, a cycle that satisfies \( C_2 \) satisfies \( C_1 \), so only one fairness constraint is required. Second, a repeated cluster could have all sub-clusters fair, thus creating a cycle that visits each sub-cluster then self-loops for the repeated cluster. The theorem statement rules out this case.

**Theorem 3.** Given an event chain, Algorithm 1 produces a weak automaton iff every repeated cluster in the chain is non-fair.

**Proof Sketch:** Non-trivial strongly-connected components (SCCs) arise from abstract states with self-loops, which in turn arise from expanding states for repeated clusters. With the exception of the Final state and the states for “accept/reject if see” escape conditions (which form their own SCCs), states are marked fair only if they correspond to or expand from clusters that have “accept if don’t complete” conditions. If a repeated cluster is non-fair, then it has no fair SCCs embedded within self-loops (other, larger SCCs). If a repeated cluster is all-fair, it requires multiple fair sets and is not weak by definition. All other repeated clusters contain cycles with both fair and non-fair states.

Our mapping to deterministic weak automata is not complete; in other words, our language does not logically characterize deterministic weak automata. Consider the regular expression \( ab^* + bc^* \): a deterministic weak automaton accepts it, but it is not expressible in our language due to the use of disjunction.

6 Related Work

We are unaware of logical characterizations of weak automata, much less ones that account for diagrammability or other forms of usability. The original work on the efficiency of verifying weak automata is due to Bloem, Ravi and Somenzi [5]. Other timing diagram formalizations have supported some of the language extensions discussed here [2,6,12], but none related the diagrammatic features of these languages to efficiency in verification.

Amla et al.’s work on modular timing diagrams has much in common with this work [3]. Their work makes timing diagrams more expressive by combining them through non-diagrammatic operators for conjunction, iteration, and
deterministic choice. Expressions in their language encompass several timing diagrams, while our work pushes the limits of a single timing diagram. Accordingly, they target efficiency through a different model of automata. The core differences between our works appear to be philosophical; ours focuses on understanding the interplay between diagrammability and efficiency, while theirs focuses on building a practical verification framework around timing diagrams. The full paper provides a more detailed comparison [9].

7 Conclusions and Future Work

The relationships between timing diagrams and deterministic weak automata suggest that there exist formal models of event sequences that simultaneously address both usability and efficiency. A traditional theoretical approach to designing languages towards efficiency would be to find a syntactic (logical) characterization of weak automata. This approach, however, fails to account for the usability of that logical characterization. This is perhaps justifiable, as "usability" is an inherently informal notion. If we refine our notion of usability to mean diagrammability, however, formal models become possible. Formal characterizations of diagrammability usually rely on topological or spatial arguments [11]; appropriate characterizations for discrete linear events remain an open problem.

The event sequence language proposed in this paper targets diagrammability by allowing only a restricted form of disjunction; in particular, disjunction governs the ordering of events, but not their occurrence. This is consistent with diagrams’ tendency to imply that all depicted items actually exist (maps, for example, indicate that all depicted features are actually there). Such nuances in the different uses of logical operations appear fundamental to formal models of diagrammability. This limited nature of disjunction also targets efficiency by supporting our criteria for deterministic automata. Restricted forms of iteration enable the mapping to weak automata. Single timing diagrams support limited forms of iteration, and hence satisfy the criteria for weakness. Overall, the generality of our language substantially enriches the set of features our timing diagrams can support while retaining efficiency for verification.

Several avenues remain open for future work. Given that the proposed language is more expressive than our current timing diagrams, characterizing diagrammability is an important next problem in this project. We expect restrictions on cluster nesting similar to those in timing diagrams to be key to such a characterization. We also plan to explore formal relationships between other event sequence languages and ours; this would help identify subsets of other languages that could be visualized and verified efficiently through a mapping to weak automata. Finally, many general questions remain regarding the nature of diagrammatic representations and their relationship to computational concerns such as efficiency and decidability that are so important in verification. We hope that our work will contribute to better understanding of these issues.
Acknowledgements. The author thanks the anonymous reviewer whose thoughtful and extensive comments led to significant improvements in this paper.

References

Executing the Formal Semantics of the Accellera Property Specification Language by Mechanised Theorem Proving

Mike Gordon\textsuperscript{1}, Joe Hurd\textsuperscript{1}, and Konrad Slind\textsuperscript{2}

\textsuperscript{1} University of Cambridge Computer Laboratory, William Gates Building, JJ Thomson Avenue, Cambridge CB3 0FD, U.K.,
\textsuperscript{2} University of Utah, School of Computing, 50 South Central Campus Drive, Salt Lake City, Utah, UT84112, USA

Abstract. The Accellera Property Specification Language (PSL) is designed for the formal specification of hardware. The Reference Manual contains a formal semantics, which we previously encoded in a machine readable version of higher order logic. In this paper we describe how to ‘execute’ the formal semantics using proof scripts coded in the HOL theorem prover’s metalanguage ML. The goal is to see if it is feasible to implement useful tools that work directly from the official semantics by mechanised proof. Such tools will have a high assurance of conforming to the standard. We have implemented two experimental tools: an interpreter that evaluates whether a finite trace $w$, which may be generated by a simulator, satisfies a PSL formula $f$ (i.e. $w \models f$), and a compiler that converts PSL formulas to checkers in an intermediate format suitable for translation to HDL for inclusion in simulation test-benches. Although our tools use logical deduction and are thus slower than hand-crafted implementations, they may be speedy enough for some applications. They can also provide a reference for more efficient implementations.

1 Introduction

We describe the implementation of two tools that work by applying theorem proving strategies to the formal semantics of the Accellera Property Specification Language (PSL \cite{3}). The implementation method guarantees that the results are compliant with the standard. Accellera \cite{2} is an industry consortium formed in 2000 by combining “Open Verilog International” and “VHDL International”. PSL is being developed as a standard property language for both dynamic verification (e.g. simulation) and static verification (e.g. model checking) \cite{8}. The design of PSL is based on IBM’s Sugar language.

Previously we constructed a deep embedding of the Sugar semantics in higher order logic. Using the HOL theorem proving system we proved various general meta-theorems (see Section 2) and were able to provide some feedback and bug reports to the language designers \cite{12,11}. As the semantics evolved into the current standard we tracked the changes and made sure that our proofs in HOL still went through. Our semantics is believed to correspond faithfully to the
official formal semantics in the PSL Manual, but we cannot be completely certain because the official semantics is expressed in a mixture of English and \LaTeX.

Not only can theorem provers like HOL be used to prove meta-theorems, they can also be programmed to dynamically generate theorems for particular models and formulas. This provides a way of implementing tools that work deductively. The approach of having tools with ‘HOL Proof Inside’ has been explored by the Prosper project [7] and it is our goal to apply Prosper ideas to build verification tools that work with ‘deduction from PSL semantics inside’. This paper describes some preliminary experiments.

PSL has four kinds of syntactic constructs: Boolean Expressions $b$, Sequential Extended Regular Expressions $r$ (SEREs), Foundation Language (FL) formulas $f$ and Optional Branching Extension (OBE) formulas.

The PSL Foundation Language (FL) contains standard future-time LTL formulas as well as less standard formulas that are composed out of regular expressions. Formula $\{ r \}(f)$ is true if $f$ holds at the last state of any sequence matching $r$; formula $\{ r_1 \} \rightarrow \{ r_2 \}$ is true if every sequence matching $r_1$ is followed by a sequence matching $r_2$. FL also has abort formulas $f \text{ abort } b$ that check $f$ but aborts the checking if a state in which $b$ is true is encountered, and clocking formulas $f@c$ that are true when $f$ is true of the sequence of states consisting of only those states for which clock $c$ holds.

The OBE is conventional branching time Computation Tree Logic (CTL). Hasan Amjad has built a symbolic model checker for OBE properties that uses BDD representation judgements applied to our semantics to calculate the truth-value of PSL properties with respect to Kripke structures. This is described elsewhere [4].

The semantics of SEREs specifies $w \models r$ to mean that a finite sequence of states $w$ matches the regular expression $r$. Then semantics of FL formulas specifies $w \models f$ to mean that formula $f$ holds of a path (i.e. a finite or infinite sequence of states). The detailed semantics is in Section 2. PSL also has a large number of operators that are defined in terms of the primitives. As we shall illustrate, they can be added by making definitions in HOL.

Using standard methods of semantic embedding, $w \models f$ can be viewed as a boolean term of higher order logic, and then automated proof by the HOL system can be applied. We have implemented a proof strategy to evaluate $w \models f$ where $w$ is a specific finite path and $f$ is a formula. Currently all formulas except aborts are covered (though a few special cases of $w \models f \text{ abort } b$ can be evaluated). This strategy implements a tool that is useful for sanity checking that a property expresses what one expects: one can directly evaluate it on example paths and the result is guaranteed to correspond to the official semantics. Example paths can either be input directly as a sequence of states (a state is a set of atomic propositions), or can be captured from a simulation run (see Section 3.3 for examples). Evaluation is fast enough to be used on simple examples and provides a pedagogically useful animation of the semantics.

Our second tool, inspired by the IBM FoCs system [1], compiles a formula $f$ (from a subset of PSL formulas) into a checker automaton that can be added
to a simulation test-bench to detect when a property is violated. The checker is initially represented in an HDL-neutral format but can be ‘pretty printed’ into the syntax of particular HDLs. We have implemented a simple converter that generates Verilog. This provides a way of prototyping tools similar to FoCs, but which are guaranteed by construction to conform to the Accellera standard. Although generating a checker can be slow (seconds to minutes), the resulting HDL code can be efficient, and it is guaranteed to be equivalent to the PSL property it was compiled from. We think this compiler might be useful for debugging other property generators. Also, since the compilation is driven by symbolic execution, it can be tuned just by adding new theorems into the set of rules that are used.

The rest of this paper is as follows: Section 2 describes the Accellera property Specification Language (PSL) and its semantics in higher order logic; Section 3 presents our first tool, which evaluates $w \models f$ for a given $w$ and $f$; Section 4 presents our second tool, a checker generator.

## 2 The Accellera Property Specification Language PSL

This section describes the semantics of the linear parts of PSL (boolean expressions, SEREs, FL formulas) and is a careful manual transcription of the official semantics in the Language Reference Manual [3] into the machine readable logic supported by the HOL system.

Boolean expressions are evaluated with respect to states. SEREs are evaluated with respect to finite sequences of states, and FL formulas with respect to finite or infinite sequences of states. A non-empty set $P$ of atomic propositions is assumed given. A state is a subset of $P$, i.e. the set of propositions that are true in the state. If $p$ ranges over $P$, then the syntax of boolean expressions $b$ is:

$$b ::= p \quad \text{(Atomic proposition)}$$

$$\mid \neg b \quad \text{(Negation)}$$

$$\mid b_1 \land b_2 \quad \text{(Conjunction)}$$

This is represented in higher order logic by defining a new type (using a data type definition mechanism), parameterised on $P$, whose elements are boolean expressions. The semantics of boolean expressions are specified by defining $s \models b$, where $s \subseteq P$, by structural induction over the type of boolean expressions:

$$(s \models p = (p \in s)) \land (s \models \neg b = \neg (s \models b)) \land (s \models b_1 \land b_2 = s \models b_1 \land s \models b_2)$$

Here, and in what follows, the operator “$\models$” binds tightly, so that, for example, $s \models b_1 \land s \models b_2$ means $(s \models b_1) \land (s \models b_2)$ not $s \models (b_1 \land s \models b_2)$. The symbols $\neg$ and $\land$ are overloaded: the occurrence of $\neg$ in $\neg b$ is part of the boolean expression syntax of PSL, but the occurrence in $\neg (s \models b)$ is negation in higher order logic. Similarly $\land$ is overloaded: the occurrence in $b_1 \land b_2$ is part of the boolean expression syntax, but the other occurrences are conjunction in higher order logic.
2.1 Semantics of Unclocked SEREs and FL Formulas

In this section we do not specify the semantics of clocked SEREs \( r@c \) and formulas \( f@c \). These are described in Section 2.2.

The syntax of SEREs is represented in higher order logic by defining a new type whose elements represent SEREs. If \( r, r_1, r_2 \) etc. range over Sequential Extended Regular Expressions (SEREs) and \( b \) and \( c \) range over boolean expressions, then the syntax of SEREs is:

\[
\begin{align*}
  r & ::= b \quad \text{(Boolean formula)} \\
  & | \{r_1\} \cup \{r_2\} \quad \text{(Disjunction)} \\
  & | r_1; r_2 \quad \text{(Concatenation)} \\
  & | r_1 : r_2 \quad \text{(Fusion: overlapping concatenation)} \\
  & | \{r_1\} \& \& \{r_2\} \quad \text{(Length matching conjunction)} \\
  & | r[^*] \quad \text{(Repeat)} \\
  & | r@c \quad \text{(Clocking – semantics in Section 2.2)}
\end{align*}
\]

The semantics of a SERE \( r \) is given by specifying \( w \models r \) for every finite sequence of states \( w \). This can be read as “word \( w \) is recognised by regular expression \( r \)”.

Words are represented as lists. A list containing elements \( e_0, \ldots, e_n \) is denoted by \([e_0; \ldots; e_n]\). Juxtaposition of words denotes concatenation (e.g. \( w[s]w' \) is the concatenation of \( w, [s] \) and \( w' \)). If \( wlist \) is a list of lists then \( \textbf{Every} p \ wlist \) applies the predicate \( p \) to every element of \( wlist \) and returns the conjunction of the result (e.g. in the semantics below \( \textbf{Every} (\lambda w. \ w \models r) \ wlist \) asserts \( w \models r \) for every \( w \in \ wlist \)) and \( \textbf{Concat} \ wlist \) denotes the concatenation of the lists in \( wlist \) (e.g. \( \text{Concat} \ [[a; b]; [c]; [d; e; f]] = [a; b; c; d; e; f] \)). The notation \( |w| \) denotes the length of \( w \) (empty words have length 0) and \( w_i \) denotes the \( i \)th element of \( w \) counting from 0, so \( w_0 \) is the first element (note that subscripts on symbols not denoting lists are just subscripts). The input and output to HOL shown in this paper has been typeset using a HOL-to-Latex translator implemented by Keith Wansbrough. Applying this translator to the HOL semantics of SEREs yields:

\[
\begin{align*}
  (w \models b) &= (|w| = 1) \land w_0 \models b) \land \\
  (w \models r_1; r_2) &= \exists w1 w2. (w = w1 w2) \land w1 \models r_1 \land w2 \models r_2) \land \\
  (w \models r_1 : r_2) &= \exists w1 w2l. (w = w1[l]w2) \land w1[l] \models r_1 \land [l]w2 \models r_2) \land \\
  (w \models \{r_1\} \cup \{r_2\}) &= w \models r_1 \lor w \models r_2) \land \\
  (w \models \{r_1\} \& \& \{r_2\}) &= w \models r_1 \land w \models r_2) \land \\
  (w \models r[^*]) &= \exists wlist. (w = \text{Concat} \ wlist) \land \textbf{Every}(\lambda w. \ w \models r) \ wlist)
\end{align*}
\]

It is hoped that this semantics requires no additional explanation. Interested readers can compare it to the semantics in the PSL Reference Manual [3, B.2.2.1].

The syntax of PSL Foundation Language Formulas (FL) is given below. The suffix “!” found on some constructs indicates that these are ‘strong’ (i.e. liveness-enforcing) operators. If the corresponding weak operator (which is written without the “!” suffix) can be defined in terms of FL formulas, then it is not included in the core and is regarded as an defined operator (e.g. \( Xf = \neg X! \neg f \) and}
\[ f @ c = \neg (\neg f @ c!)) \]. The distinction between strong and weak operators is discussed and motivated in the PSL Manual [3, Section 4.4.3].

The syntax is represented in higher order logic by defining a new type whose elements are formulas. The FL primitives listed below are redundant. For example, \( \{r_1\} \mapsto \{r_2\}! \) and \( X! f \) can be defined in terms of suffix implication.

\[
\begin{align*}
  f & ::= p \quad \text{(Atomic formula)} \\
         & | \neg f \quad \text{(Negation)} \\
         & | f_1 \land f_2 \quad \text{(Conjunction)} \\
         & | X! f \quad \text{(Successor)} \\
         & | \{f\}_U f_2 \quad \text{(Until)} \\
         & | \{r\}(f) \quad \text{(Suffix implication)} \\
         & | \{r_1\} \mapsto \{r_2\}! \quad \text{(Strong suffix implication)} \\
         & | \{r_1\} \mapsto \{r_2\} \quad \text{(Weak suffix implication)} \\
         & | f \text{ abort } b \quad \text{(Abort)} \\
         & | f @! \quad \text{(Clocking – semantics in Section 2.2)}
\end{align*}
\]

Paths can be either finite or infinite. The notation \( w^i \) denotes the \( i \)-th tail of \( w \), i.e. the path obtained by chopping \( i \) elements off the front of \( w \) (so \( w^0 = w \)). The notation \( w^{i,j} \) denotes the finite sequence of states from \( i \) to \( j \) in \( w \), i.e. \( w_i w_{i+1} \cdots w_j \). The juxtaposition \( w^{i,j} w' \) denotes the path obtained by concatenating the finite sequence \( w^{i,j} \) on to the front of the path \( w' \). The HOL semantics of FL formulas is:

\[
\begin{align*}
  (w \models b &= |w| > 0 \land w_0 \models b) \land \\
  (w \models \neg f &= \neg (w \models f)) \land \\
  (w \models f_1 \land f_2 &= w \models f_1 \land w \models f_2) \land \\
  (w \models X! f &= |w| > 1 \land w^1 \models f) \land \\
  (w \models [f \ U f_2] &= \exists k \in (0 .. |w|). w^k \models f_2 \land \forall j \in (0 .. k). w^j \models f_1) \land \\
  (w \models \{r\}(f) &= \forall j \in (0 .. |w|). w^{0,j} \models r \Rightarrow w^j \models f) \land \\
  (w \models \{r_1\} \mapsto \{r_2\}! &= \forall j \in (0 .. |w|). w^{0,j} \models r_1 \Rightarrow \exists k \in (j .. |w|). w^{j,k} \models r_2) \land \\
  (w \models \{r_1\} \mapsto \{r_2\} &= \forall j \in (0 .. |w|). w^{0,j} \models r_1 \Rightarrow (\exists k \in (j .. |w|). w^{j,k} \models r_2) \land \\
  \neg (w \models f \text{ abort } b) &= w \models f \lor w \models b \lor \exists j \in (1 .. |w|). \exists w'. w^j \models b \land w^{0,j-1} w' \models f)
\end{align*}
\]

This semantics is a careful formalisation of the official semantics, with the exception that \( |w| > 0 \) has been added to the definition of \( w \models b \). This addition ensures that formulas are defined for empty paths (the official semantics is undefined). The semantics for non-empty paths is unchanged.

### 2.2 Semantics of Clocked SEREs and FL Formulas

SEREs and formulas not containing “@” are called unclocked and the sets of unclocked SEREs and formulas the unclocked subsets. In the previous section only the semantics of the unclocked subsets were defined. This is called the unclocked semantics.
Clocked SEREs have the form \( r@c \) and strongly clocked formulas the form \( f@c! \), where \( c \) is a boolean expression that is true when the clock is asserted.

Weakly clocked formulas \( f@c \) are defined by \( f@c = \neg((\neg f)@c!) \). Intuitively, \( w \models r@c \) and \( w \models f@c! \) mean, respectively, that \( w|_c \models r \) and \( w|_c \models f \) where \( w|_c \) is obtained from \( w \) by removing (‘projecting out’) all states in which \( c \) is false (i.e. restricting \( w \) to states in which \( c \) is true).

The formal semantics in the Reference Manual doesn’t use projections, instead two separate semantics are given: the first one defines the semantics of all constructs (included clocked ones) directly, the second one provides a set of ‘rewrites’ that can be used to recursively eliminate all occurrences of “@”, i.e. translate into the unclocked subsets.

The direct semantics is specified by recursively defining \( w \models^c r \) and \( w \models^c f \) for an arbitrary clock \( c \), and then the semantics of a SERE \( r \) and formula \( f \) are \( w \models^T r \) and \( w \models^T f \), respectively, where \( T \) is the top-level clock which is always true. The top-level semantics with a clock \( c \) are \( w \models^T r@c \) and \( w \models^T f@c! \).

The rewrites semantics is formalised by first defining, for each clock \( c \), a function \( T^c \) that maps an arbitrary SERE or formula into the unclocked subset. Thus \( \lambda c. T^c \) is a function mapping a clock \( c \) to a translation function \( T^c \) which has \( c \) as the clock context. The top-level clock is \( T \), so the top-level translations of \( r \) and \( f \) are \( T^T(r) \) and \( T^T(f) \). The meanings of these can then be computed using the unclocked semantics in Section 2.1.

The definition of \( \models^c \) is much more complex than the definition of \( \models \), and we do not give it here. However, we have formalised it in higher order logic and proved [12,11] the sanity checking property that, if \( \text{ClockFree}(r) \) and \( \text{ClockFree}(f) \) mean that \( r \) and \( f \) are unclocked, then:

\[
\vdash \forall r. \text{ClockFree}(r) \Rightarrow (w \models^T r = w \models r) \\
\vdash \forall f. \text{ClockFree}(f) \Rightarrow (w \models^T f = w \models f)
\]

We have also proved using the HOL system that:

\[
\vdash \forall r. \ w \models^T r = w \models T^T(r) \\
\vdash \forall f. \ w \models^T f = w \models T^T(f)
\]

which allows us to evaluate the semantics of any construct by first applying these equations and then using the unclocked semantics.

The definition of \( T^c(r) \) and \( T^c(f) \) is by recursion over the structure of SERE \( r \) and formula \( f \). For SEREs:

\[
(T^c(b) = (\neg c[*]; c \land b)) \land \\
(T^c(r_1; r_2) = T^c(r_1); T^c(r_2)) \land \\
(T^c(r_1 : r_2) = T^c(r_1 : T^c(r_2)) \land \\
(T^c([r_1] \mid \{r_2\}) = (T^c([r_1]) \mid \{T^c(r_2)\}) \land \\
(T^c([r_1]\&\{r_2\}) = \{T^c(r_1)\}\&\{T^c(r_2)\}) \land \\
(T^c(r[\#]) = T^c(r[\#]) \land \\
(T^c(r@c) = (\neg c_1[\#]; c_1 : T^{c_1}(r)))
\]

and for formulas:
3 Executing the Formal Semantics

The HOL system has an ML function EVAL [5] which when applied to a term \( t \) proves a theorem \( \vdash t = t' \), where \( t' \) is the result of evaluating \( t \). EVAL performs call-by-value order rewriting efficiently using logic definitions that are in force in the context in which it is invoked. It can also invoke equations and decision procedures that have been explicitly added to the context.

3.1 Executing the Clock Removal Rewrites

The semantics of a formula \( f \) with respect to a path \( w \) is \( w \models_T f \). The first step in evaluating \( w \models_T f \) is to rewrite with the equations:

\[
\vdash \forall r \ w. \ w \models_T r = \ w \models_T T(r) \text{ and } \forall f \ w. \ w \models_T f = \ w \models_T T(f)
\]

The next step is to execute the definition of \( T \), and the final step is to evaluate the unclocked semantics (Section 3.2).

The clocking removal rules are directly executable, but the results are complicated. For example EVAL \( (T^c(T^c[c]; \{\neg rq@c1\} \& \{ak@c2\}; rq@c1)) \) evaluates to the almost completely inexpressible theorem:

\[
\vdash T^c(T[c]; \{\neg rq@c1\} \& \{ak@c2\}; rq@c1) =
\neg c[c]; c \land T[c]; \neg c[c]; c_1 : \neg c[c]; c_1 \land \neg rq \& \neg c[c]; c_2 : \neg c[c]; c_2 \land ak; \neg c[c]; c_1 : \neg c[c]; c_1 \land rq
\]

This illustrates how much more natural and high-level are properties expressed using the @c clocking construct. Note also that \( c_1 : \neg c[c] \) is equivalent to \( c_1 \), which shows the need to perform peephole optimisations on the output of naive evaluation with the rewrites. Executing the rewrites for formulas typically produces even more incomprehensible results than with SEREs! For example, consider the following (the operator before is defined in Section 3.3):

\[
\vdash T^c(T[c]; \neg ak_1; ak_1; T)((\neg ak_2 \land X! (ak_2) \text{ before } \neg ak_1 \land X! (ak_1))) =
\neg c[c]; c \land T[c]; \neg c[c]; c \land \neg ak_1; \neg c[c]; c \land ak_1;
\neg c[c]; c \land T[c]; ((\neg c \ U \ c \land \neg \neg \neg \neg c \land \neg \neg ak_1 \land X! ([\neg c \ U \ c \land ak_1])) \ U \ c \land \neg ak_2 \land
\]
Negations $\neg f$, Conjunctions $f_1 \land f_2$, and Next-State $X! f$

To evaluate formulas, first note that $w \models \neg f$, $w \models f_1 \land f_2$ and $w \models X! f$ can be rewritten directly using the semantics. For example, here are the results of invoking EVAL on $p \land X! f$ with three increasingly specific paths (in each case EVAL is applied to the term on the left hand side of the equation, and generates a theorem showing the evaluation of this term):

\[
\begin{align*}
\vdash w \models p \land X! f \quad &\iff (|w| > 0 \land w_0 \models p) \land |w| > 1 \land (w^1) \models f \\
\vdash ([s_0]w) \models p \land X! f \quad &\iff s_0 \models p \land |w| + 1 > 1 \land w \models f \\
\vdash s_0s_1s_2s_3s_4s_5s_6s_7s_8s_9 \models p \land X! f \quad &\iff s_0 \models p \land s_1s_2s_3s_4s_5s_6s_7s_8s_9 \models f
\end{align*}
\]

These illustrate symbolic evaluation: when laws apply they are used to reduce a term, but if no laws are applicable then the term is left unevaluated: $|w| + 1 > 0$ can be evaluated, since EVAL has been told $\vdash \forall n.\ n + 1 > 0 = T$, but $|w| + 1 > 1$ cannot be evaluated for an arbitrary variable $w$, but the more specific term $|s_0s_1s_2s_3s_4s_5s_6s_7s_8s_9| + 1 > 1$ can be evaluated: even though the states $s_i$ are left as variables, since the path has length 10, which is greater than 0. With a fully concrete path the truth of the formula is completely determined. To display concrete examples we write $\{\ldots\}\{\ldots\} \cdots \{\ldots\} \models f$ where $\{\ldots\}$ are sets of atomic propositions representing states. Note that in such examples braces are set brackets, not part of SERE syntax. For example:

\[
\vdash \{a\}\{a, b\}\{b\} \models a \land X! (b) = T
\]

Until Formulas $[f_1 U f_2]$

The semantics of the until-construct is:

\[
w \models [f_1 U f_2] = \exists k \in (0 .. |w|).\ w^k \models f_2 \land \forall j \in (0 .. k).\ w^j \models f_1
\]

which cannot be directly executed, but there is a standard recursive version of this definition that can easily be proved as a theorem and is directly executable:

\[
\vdash w \models [f_1 U f_2] = |w| > 0 \land (w \models f_2 \lor w \models f_1 \land w^1 \models [f_1 U f_2])
\]

The following example is from the Reference Manual [3, Example 2, page 45].
Define \( w_1 \) to be this path, namely:
\[
w_1 = \{c,clk2\}\{clk1\}\{\{clk1,a,clk2\}\{a\}\{clk1,a,b,c\}\{c,clk2\}
{c,clk2}\{clk1,b\}\{b\}\{clk1,clk2\}
\]

Recall that weak clocking is defined by: \( f@c = \neg(\neg f@c!) \). After making this definition we can evaluate examples like \( (w_1') \models ^\top (c \land X! ((\lambda a. b))@clk_1 \) and \( (w_1') \models ^\top (c \land X! (((\lambda a. b))@clk_1))@clk_2 \) for \( 0 \leq i < |w_1'| \) and confirm that the first is true only when \( i \) is 4 or 5, and the second only when \( i \) is 0. The semantics of multiple clocking is subtle (clocks do not accumulate: an inner clock ignores an outer one) and is still under discussion and may change. Our tools facilitate experiments on concrete scenarios to gain insight into the current semantics.

**Suffix Implication \( \{r\}(f) \)**

Suffix implication formulas \( \{r\}(f) \) are executed by generating a matcher for \( r \) and then invoking \texttt{EVAL} on \( f \) whenever a match is found. In detail, the SERE \( r \) is first lifted to an element of a HOL theory of regular expressions (based on Nipkow’s Isabelle work [13], but with many details adjusted for PSL), and then a proof procedure lazily constructs the state set, accepting states and transition table of an equivalent DFA. This DFA is run along a finite trace \( w \), and whenever it enters an accepting state \texttt{EVAL} is used to check \( x \models \neg f \) on the remaining trace \( x \).

The constants that do this lifting (\texttt{sere2regexp}) and DFA execution (\texttt{acheck}) are defined to be executed efficiently in the logic, but the following theorem shows that they also preserve the semantics of the original suffix implication formula:

\[ \forall w, r, f. \text{ClockFree}(r) \Rightarrow (w \models \{r\}(f) = \text{acheck}(\text{sere2regexp } r) (\lambda x. x \models f) w) \]

**Strong Suffix Implications \( \{r_1\} \Rightarrow \{r_2\}! \)**

Strong implications \( \{r_1\} \Rightarrow \{r_2\}! \) are reduced to suffix implications by:

\[ \forall w, r_1, r_2. w \models \{r_1\} \Rightarrow \{r_2\}! = w \models \{r_1\}(\neg\{r_2\}(F)) \]

**Weak Suffix Implications \( \{r_1\} \Rightarrow \{r_2\} \)**

Weak implications are executed by, if necessary, performing a reachability calculation inside HOL. We add a \texttt{Prefix} operator\(^2\) to the HOL regular expression theory, with the semantics that \texttt{Prefix}(\( r \)) matches a word \( w \) if it can be extended by \( w' \) such that \( r \) matches \( ww' \). We can now use our generic lifting and DFA execution constants to execute weak implication, and the following theorem guarantees that the semantics of the original formula are preserved:

\[ \forall w, r_1, r_2. \text{ClockFree}(r_1) \land \text{ClockFree}(r_2) \Rightarrow \]

\(^1\) This equivalence was first observed by Dana Fisman (private communication).

\(^2\) The prefix operators used for weak implication (\texttt{Prefix}) and abort (\texttt{FormPrefix}) are based on an idea from Dana Fisman (private communication).
\[ w \models \{ r_1 \} \rightarrow \{ r_2 \} = \]
\[ \text{achek} \ (\text{sere2regexp} \ r_1) \]
\[ (\lambda x. \ x \models \neg \{ r_2 \}(F) \lor \text{amatch} \ (\text{Prefix} \ (\text{sere2regexp} \ r_2)) \ x) \ w \]

The \text{amatch} constant checks whether a regular expression matches a word, by building an equivalent DFA, executing it along the word, and testing whether it is in an accepting state at the end. If the regular expression is \text{Prefix}(r), then the state \( s \) is accepting precisely when it is possible to reach an accepting state from \( s \) on the transition graph of (the DFA corresponding to) \( r \). To implement this, we defined a version of Dijkstra’s reachability algorithm, and proved it correct [6].

To summarise, we execute \( w \models \{ r_1 \} \rightarrow \{ r_2 \} \) solely by deductions in the logical kernel. We first use the above theorem to reduce the problem to executing a DFA. This involves performing many on-the-fly deductions to evaluate transitions and accepting states. The \text{Prefix} operator is the most complex of these on-the-fly deductions, requiring a reachability calculation on the transition graph. This reachability calculation can be reduced to an instance of Dijkstra’s algorithm, but to make that step we need the correctness proof of the algorithm. The end result of all this deduction is a HOL theorem of the form \( \vdash (w \models \{ r_1 \} \rightarrow \{ r_2 \}) = b \), where \( b \) is either \( T \), \( F \), or something more complex if the original term contained variables.

**Aborts** \( f \ abort \ b \)

We currently do not have a fully general method of executing \( w \models f \ abort \ b \), but evaluation in some cases is possible. First define a formula prefix function \text{FormPrefix} and an auxiliary function \text{AbortAux}.

\text{FormPrefix} \ w \ f = \exists w'. w w' \models f \]
\text{AbortAux} \ w \ f \ b \ n = \exists j \in n .. |w|.w^j \models b \land \text{FormPrefix} \ w^{0.j-1} f \]

then it is easy to prove:

\[ \vdash w \models f \ abort \ b = w \models f \lor w \models b \lor \text{AbortAux} \ w \ f \ b \ 1 \]
\[ \vdash \text{AbortAux} \ w \ f \ b \ n = \]
\[ n < |w| \land (w^n \models b \land \text{FormPrefix} \ w^{0.n-1} f \lor \text{AbortAux} \ w \ f \ b \ n + 1) \]

and adding these to the rewrites used by \text{EVAL} enables \( f \ abort \ b \) to be executed in the trivial cases when \( w \models f \) or \( w \models b \) evaluate to true. For a non-trivial concrete example, consider the following (c.f. [14, Fig. 8, page 22]):

<table>
<thead>
<tr>
<th>time</th>
<th>00</th>
<th>01</th>
<th>02</th>
<th>03</th>
<th>04</th>
<th>05</th>
<th>06</th>
<th>07</th>
<th>08</th>
<th>09</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>start</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>req</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>ack</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>interrupt</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

this corresponds to the finite path \( w_2 \) where

\[ w_2 = \{\}\{\text{start}\}\}\{\}\{\text{req}\}\}\{\}\{\}\{\}\{\text{interrupt}\}\}\{\}\{\}\{\}\{\}

If we define:

\[ \forall f. \ F f = \{ T \ U f \} , \quad \forall f. \ \text{eventually} ! f = F f \]
\[ \forall f. \ G f = \neg (F \ (\neg f)) , \quad \forall f. \ \text{always} f = G f \]

then \text{EVAL} will prove:
\( \vdash w2 \models always(start \rightarrow (always(req \rightarrow eventually!ack) \ abort \ interrupt)) \)

\[
= \text{FormPrefix } \{\{start\}\{\}\{\\{req\}\{\}\{\}\{\}\}\{T \ U \ \neg \neg \neg req \land \neg[T \ U \ ack]\}
\]

The right hand side of this equation is true if path \( \{\{start\}\{\\{\\{\\\}\\\}\\\}\{\\\}\}\{\\\}\{\\\}\}\) can be extended to make \( \neg[T \ U \ \neg \neg \neg req \land \neg[T \ U \ ack]] \) true. In this particular case it is sufficient to only consider extensions either by the empty path or by a singleton path consisting of one state of the form \( \{x\} \). The following easily proved theorem says that \( w \models f \) or \( \exists x . (w[x]) \models f \) is sufficient.

\( \vdash \text{FormPrefix } f \ w = w \models f \lor (\exists x . (w[x]) \models f) \lor \text{FormPrefix } w \ f \)

For the example above, adding this to the equations used by EVAL results in a term \( \exists x . \neg(\neg(ack = x) \lor (req = x) \land \neg(ack = x)) \) being generated. EVAL can be programmed to invoke a decision procedure on such terms. This is an ad hoc partial solution. We hope eventually to make our implementation complete.

### 3.3 More Examples

The first example below illustrates the utility of having an automatic semantics calculator. The second example shows how such a calculator can be used to analyse behaviours captured from simulation.

#### 3.3.1 An Example from an Accellera Online Discussion

A recent online discussion \([9,10]\) concerned the intervals for which the SERE \(((\{a; b\})@clk_1; c)@clk_2 \ \text{“hold tightly”} \ \text{within the behaviour:} \)

<table>
<thead>
<tr>
<th>time</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>clk1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>a</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>b</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>c</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>clk2</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

The Reference Manual introduces the terminology \( r \) “holds tightly” for \( w \) if and only if \( w \models T r \). To understand this example, note that clocks don’t accumulate: only the current, i.e. innermost, one is used to sample the path. To analyse this example, a simple ML function can easily be written that evaluates a SERE on all sub-intervals of a path and returns the results that correspond to intervals for which the SERE holds. Using this we can analyse the example and deduce that the only interval where the SERE holds tightly is:

\[ \vdash \{clk_2\}\{clk_1, a\}\{a\}\{clk_1, b, clk_2\}\{c\}\{clk_1\}\{c, clk_2\} \models a; b@clk_1; c@clk_2 = T \]

This resolves the discussion in favour of the Manual [(10) is correct, (9) is wrong]. Note that the other examples in the Manual can be (and have been) automatically checked. The fact that there is ongoing discussion about properties as simple as this suggests that our semantics calculator might be a useful tool for property writers.
3.3.2 An Example from the FoC Manual

Evaluation in HOL is nearly instantaneous for examples of the scale above. Whilst we would not claim our evaluator can handle ‘industrial scale’ problems, it can be applied to significantly more complex examples. In the IBM FoC Manual there is a Sender-Buffer-Receiver in which the Sender (S) communicates with the buffer (B) using four-phase handshakes with request signal StB_REQ and acknowledgement BtoS_ACK, and the Buffer communicates with the Receiver (R) with a four-phase handshake with request signal BtoR_REQ and acknowledgement RtoB_ACK.

We can define in HOL a function FourPhase such that FourPhase req ack is true if signals req and ack satisfy properties required of a four-phase handshake. First define:

\[ \forall r. \text{never}(r) = \{ T[*]; r \} \mapsto \{ F \} \]

then define:

**FourPhase** req ack = 
\[ \text{never}(T[*]; \neg \text{req} \land \text{ack}; \text{req}) \land \text{never}(T[*]; \text{req} \land \neg \text{ack}; \neg \text{req}) \land \text{never}(T[*]; \neg \text{ack} \land \neg \text{req}; \text{ack}) \land \text{never}(T[*]; \text{ack} \land \text{req}; \neg \text{ack}) \]

Definitions like FourPhase in HOL are analogous to definitions of verification units (vunits) in PSL.

We have written a Verilog model to generate paths. If SimRun is a 700 state Verilog generated path, our tool currently takes about a couple of minutes on a 1GHz machine to evaluate: \( \text{SimRun} \models \text{FourPhase} \text{ StbB_REQ BtoS_ACK} \) and \( \text{SimRun} \not\models \text{FourPhase} \text{ BtoR_REQ RtoB_ACK} \). Notice that both never and FourPhase have an initial \( T[*] \). If we remove the occurrences of \( T[*] \) in FourPhase then the checking is more than twice as fast. If we augmented the rewrites used by EVAL to include:

\[ \vdash \forall w \ r_1 \ r_2 \ r_3, \ w \models (r_1; r_2); r_3 = w \models r_1; (r_2; r_3) \]
\[ \vdash \forall w \ r. \ w \models r[*]; r[*] = w \models r[*] \]

then this optimisation could be made to happen automatically.

If, using the definition of \( G \ f \) given earlier, we define:

\[ [f_1 W f_2] = [f_1 U f_2] \lor G f_1, \quad f_1 \text{ before } f_2 = [\neg f_2 W f_1 \land \neg f_2] \]

then AckInterleave ack_1 ack_2 defined below states that ack_2 is asserted between any two ack_1 assertions:

**AckInterleave** ack_1 ack_2 = 
\[ \{ (T[*]; \neg \text{ack}_1; \text{ack}_1) \} (\neg \text{ack}_2 \land X! (\text{ack}_2) \text{ before } \neg \text{ack}_1 \land X! (\text{ack}_1)) \]

Checking that the conjunction below evaluates to \( T \) takes about 5 minutes.

\[ \text{SimRun} \models \text{AckInterleave } \text{BtoS_ACK } \text{RtoB_ACK} \land \text{SimRun} \not\models \text{AckInterleave } \text{RtoB_ACK } \text{BtoS_ACK} \]

This corresponds to the vunit **ack_interleaving** in the FoC Manual example.
4 Compiling the Formal Semantics

In the last section we saw how to execute the formal semantics by deduction in the theorem prover. In particular, SEREs are executed by constructing a provably equivalent DFA. In the same way, some PSL formulas are equivalent to DFAs, where a violation of the formula corresponds to the DFA entering an accepting state. In this section, we show how to safely compile a subset of such PSL formulas as ‘checker modules’ in a HDL. An off-the-shelf simulation tool is then used to simulate the circuit together with the checker, and any violations of the property are detected and reported to the user.

To illustrate the operation of the compiler, we will use part of the FourPhase property (introduced in Section 3.3).

\[ \text{never}(\neg \text{StoB}_\text{REQ} \land \text{BtoS}_\text{ACK}; \text{StoB}_\text{REQ}) \]

This says that whenever \(\text{StoB}_\text{REQ}\) is low and \(\text{BtoS}_\text{ACK}\) is high, it is never the case that \(\text{StoB}_\text{REQ}\) will go high before \(\text{BtoS}_\text{ACK}\) goes low. By the definition of \(\text{never}\) (also in Section 3.3), this property holds if and only if the following SERE does not hold for any initial segment of the trace:

\[ \text{T}[\text{r}]; \neg \text{StoB}_\text{REQ} \land \text{BtoS}_\text{ACK}; \text{StoB}_\text{REQ} \]

If we convert this SERE to an equivalent DFA, it is easy to check whether it accepts any initial segment of a trace. We simply advance the DFA along the trace according to its transition function, and if it ever reaches an accepting state we report that the \(\text{never}\) property has been violated.

To summarise, compiling the property \(\text{never}(r)\) reduces to generating an equivalent DFA to the SERE \(\text{T}[\text{r}]; r\), and replacing accepting states with an error message reporting that the property has been violated.

Let us now look more closely at the compilation process, to see how the semantics are preserved. We begin with the PSL formula \(\text{never}(r)\). We convert the SERE \(\text{T}[\text{r}]; r\) to an element of the HOL regular expression theory, and then to a DFA with a set of states, a subset of accepting states and a transition table. We intend to simulate this DFA concurrently with a circuit, and report an error whenever the DFA enters an accepting state. We can consider the circuit simulation to be producing an infinite trace, and the DFA effectively run on all initial segments of this. The following theorem shows that this mode of operation preserves the semantics of the original PSL formula \(\text{never}(r)\):

\[ \vdash \forall r \ w . \ \text{ClockFree}(r) \land (|w| = \infty) \Rightarrow \]
\[ (w \models \text{never}(r) = \forall n. \neg \text{amatch} (\text{sere2regexp} (\text{T}[\text{r}]; r))(w^{0,n})) \]

The next step is the extraction of the DFA from HOL to an ML data-type, ready for a compiler back end to output code for a particular HDL. We use proof as much as possible in this function, because it increases our confidence in the correctness of the extracted DFA while incurring relatively little cost. The ML function that performs the extraction takes as input a list of atomic propositions and a regular expression, and returns for each \textit{reachable} state of the DFA: (i) an integer state identifier, and the HOL term that represents the state, (ii) a boolean that is true for accepting states, and a HOL theorem proving this, and
(iii) a ‘condition’ data-type that encodes a series of tests on the truth value of atomic propositions followed by a transition to a new state, with HOL theorems proving the conditional transitions correct.

Shown below is the ML output from applying the DFA extraction function to our example: \( R := \text{sere2regexp}(T[\star]; \neg\text{StoB}_\text{REQ} \land \text{BtoS}_\text{ACK}; \text{StoB}_\text{REQ}). \)

\[
\begin{align*}
(0, \text{[6]}, \text{false}, \neg \text{eval_accepts } R [6] = F), \\
\text{Branch}(\text{StoB}_\text{REQ}), \\
\text{Leaf}(1, \neg \text{StoB}_\text{REQ} \land \text{BtoS}_\text{ACK} \land \text{StoB}_\text{REQ}), \\
\text{Leaf}(2, \neg \text{StoB}_\text{REQ} \land \text{BtoS}_\text{ACK} \land \text{StoB}_\text{REQ}), \\
\text{Leaf}(3, \text{true}, \neg \text{eval_accepts } R [0; 4] = T).
\end{align*}
\]

For reasons of space only the transition function for state 0 (the initial state) is shown. The term representing this state is \([6]^3\), the \text{false} indicates that this state is not accepting, and is followed by a theorem proving this.

The condition first tests the atomic proposition \text{StoB}_\text{REQ}, and if true moves to state 1 (which as we see is represented in HOL as \([4]\)). The conditional theorem at this leaf reflects this transition.

From this language independent description of a DFA, it is a simple matter to generate versions in a HDL. We have implemented a pretty-printer for Verilog syntax. The resulting Verilog module for our example property is shown in Fig. 1, and it has correctly reported errors during simulations of a buggy buffer circuit.

5 Conclusions and Future Work

The main point of this paper is that a formal semantics is not just documentation. Current theorem provers are powerful enough to be programmed to execute semantics in interesting ways, though a major challenge is to engineer the deductions to be fast enough to be useful. We have illustrated this with two prototype tools. The first one could be useful for property developers and teachers and learners of PSL. The second one illustrates a novel way of implementing an EDA tool that guarantees conformance to the standard. We think such semantics-based tools could eventually be made efficient enough for industrial scale use, but one needs to choose applications where semantic accuracy is more critical than performance. The times (minutes) quoted in Section 3.3 will not impress members of the model checking community, but this doesn’t necessarily mean they are unacceptable, given the correct-by-construction benefits of the implementation method.

3 The values of the HOL terms representing states are an artifact of the DFA subset construction, and should be treated as arbitrary terms.
module Checker (StoB_REQ, BtoS_ACK, BtoR_REQ, RtoB_ACK);

input StoB_REQ, BtoS_ACK, BtoR_REQ, RtoB_ACK;
reg [1:0] state;

initial state = 0;

always @ (StoB_REQ or BtoS_ACK or BtoR_REQ or RtoB_ACK)
begin
  $display("Checker: state = %0d", state);
  case (state)
  0: if (StoB_REQ) state = 1; else if (BtoS_ACK) state = 2; else state = 1;
  1: if (StoB_REQ) state = 1; else if (BtoS_ACK) state = 2; else state = 1;
  2: if (StoB_REQ) state = 3; else if (BtoS_ACK) state = 2; else state = 1;
  3: begin $display("Checker: property violated!"); $finish; end
  default: begin $display("Checker: unknown state"); $finish; end
  endcase
end
endmodule

Fig. 1. The Verilog state machine for the example property.

The work described here illustrates a convergence of computation and deduction, in which the execution of theorem proving strategies becomes a powerful method of implementation. We plan to extend, package and ruggedise our prototypes into standalone tools that automatically invoke HOL (currently they are invoked from HOL via ML functions). The interpreter is complete excepts for aborts, but the checker only handles a subset of formulas. Our goal is to cover the whole of PSL.

Acknowledgements. Thanks to Cindy Eisner, Dana Fisman and Hasan Amjad for help with our research, and Keith Wansbrough for help preparing the paper. Additional thanks to Cindy Eisner for comments on an earlier version of this paper.

References


On Combining Symmetry Reduction and Symbolic Representation for Efficient Model Checking

E. Allen Emerson and Thomas Wahl

Department of Computer Sciences and Computer Engineering Research Center
The University of Texas, Austin/TX 78712, USA
{emerson,wahl}@cs.utexas.edu

Abstract. BDDs allow succinct symbolic representation of digital circuits. Symmetry reduction factors out redundancy inherent in the regular organization of many systems. Both are successful techniques for combating state space explosion. It would be desirable to combine them into symbolic symmetry reduction. Unfortunately, the straight-forward approach to symmetry reduction requires the orbit relation, whose symbolic representation as a BDD is in general of exponential size. We investigate the use of generic representatives as a means of overcoming this problem for fully symmetric systems: instead of first representing the system as a BDD and then applying symmetry reduction, we translate the given program text into a symmetry-reduced version. The result can then be encoded using a BDD. We demonstrate that this method is superior not only to the traditional orbit-relation based symmetry reduction, but also to the approach using multiple representatives.

1 Introduction

Symbolic representation of systems, most notably in the form of binary decision diagrams (BDDs), is often more compact than explicit, enumerative representation. Symmetry reduction is a powerful technique to limit the state space explosion problem. In symmetric systems, two states are considered equivalent if they are identical up to certain permutations of the participating processes. This relation gives rise to equivalence classes of states, called orbits. The Kripke structure built over the orbits can be shown to be bisimulation-equivalent to the structure built over individual states.

The combination of symbolic representation with symmetry reduction was investigated in [CEFJ96]. The paper describes how the BDD for the representative function can be constructed, which maps a state to its unique orbit representative. Symbolic model checking in the presence of symmetry is then implemented by applying the representative function to the intermediate results during fixpoint evaluations.

Computing the representative function requires the orbit relation, which contains pairs of states that are permutations of each other. The orbit relation turned out to be the bottleneck of symbolic symmetry reduction, since its BDD is, for many underlying symmetry groups, of size exponential in the minimum of the number of components and

* This work was supported in part by NSF grants CCR-009-8141 and CCR-020-5483, and SRC contract 2002-TJ-1026.
the number of states per component. A partial remedy is to permit *multiple representa-
tives* per orbit, which might be computable without using the orbit relation. However,
choosing too many representatives per orbit defeats the purpose of symmetry reduction.

To overcome the limitations of the above approach [CEFJ96] to combining symmetry
reduction and symbolic representation using BDDs, we investigate, for *fully symmetric*
systems, the use of *generic representatives* [ET99] as a means of avoiding the problems
associated with picking representative states. Instead of first building a BDD for the
system and then implementing symmetry reduction via the orbit relation, the symmetry
is factored out at the source code level by compiling the original program into one
that operates on counter variables. To keep track of equivalence classes of states, it is
sufficient to store the number of processes in a given location, rather than their identities.

For example, in a system with process locations $N$, $T$ and $C$, the states $(N, N, T, C)$,
$(N, C, T, N)$, and $(T, N, N, C)$ are all symmetry-equivalent and can be represented
generically as $(2N, 1T, 1C)$.

In this paper we show how this idea can be applied to practical systems, where
processes communicate via shared variables. In many applications, a global variable is
used to point to one distinguished process, like one that possesses a token, or one that
is currently allowed to enter its critical section. Since generic representatives get rid of
process identities, such a variable must be adapted to a generic program. We show how
this can be done by replacing it with a new variable that keeps track of the location of the
distinguished process, rather than its identity. This method presents a slight challenge,
though: if the distinguished process executes a transition, then its identity remains the
same, but its location changes. This change must be reflected in the new variable. The
complexity of a transition might therefore grow when translated into its generic form,
although only by a small constant amount.

We place suitable restrictions on the use of those global variables containing process
indices in guards and actions in a program to ensure full symmetry. We also show
the details of the program translation. We then define Kripke structures derived from
the original and translated programs, respectively, and establish their bisimilarity. The
generic method preserves all of the symmetry reduction, is applicable to a large class
of fully symmetric systems and is efficient; in particular, it completely avoids the orbit
relation. We demonstrate its usefulness in symbolic symmetry reduction by presenting
experimental results for two systems with unique, multiple and generic representatives.

The remainder of this paper is organized as follows. In section 2, we review traditional
symmetry reduction with BDDs. In section 3 we illustrate, by means of an example, the
notion of generic representatives. Section 4 formally describes how to translate a program
given as a synchronization skeleton into an “equivalent” generic program. The translation
of this new program into BDDs is the topic of section 5. We compare our method against
other symbolic symmetry reduction techniques experimentally in section 6. Related and
future work are discussed in the concluding section 7.

## 2 Preliminaries

**Notation.** For $a, b \in \mathbb{N}$, we denote by $[a..b]$ the set $\{i \in \mathbb{N} : a \leq i \leq b\}$. For a
permutation $\pi$, the symbol $\pi^{-}$ stands for its inverse. In programs, we use an imperative
language style syntax. Block structure is indicated by indentation (instead of begin/end); comments go from “//” to the end of the line.

2.1 Permutations Acting upon States

The ideas presented in this paper apply to process symmetries, which describe the phenomenon that in a system of replicated process components, processes can be rearranged in certain ways simultaneously in the source and target state of all transitions of the system, without changing the overall transition relation. This can be formalized as follows. The systems under consideration are similar to shared variable programs [ES96]. We assume there are \( n \) concurrently executing processes, following an interleaved model of computation, which share some global variables. The possible local states of a process are given by a set of process locations \( L \). A system state can therefore be written as \( s = (v, l_1, \ldots, l_n) \), where \( v \) is a (possibly tuple-valued) global variable and \( l_i \in L \) is the location of process \( i \). For \( L \in L \), we use \( L_i \) as a shorthand for the expression \( l_i = L \). The rearrangement of processes in a state is formalized in terms of a permutation \( \pi : [1..n] \rightarrow [1..n] \) acting upon process indices. The mapping \( \pi \) can be extended to act on system states by defining \( \pi(s) = (v^\pi, l_{\pi(1)}, \ldots, l_{\pi(n)}) \), where \( v^\pi \) describes the result of \( \pi \) acting on \( v \). The definition of \( v^\pi \) depends on the character of \( v \): some global variables, like a binary semaphore, are invariant under permutations, such that \( v^\pi = v \). On the other hand, a global token variable pointing to (the id of) some process is directly affected by \( \pi \), as we shall see in section 3. In this case, one defines \( v^\pi = \pi(v) \).

2.2 Symmetry Reduction in Theory

Given a set of permutations \( G \) acting on \([1..n]\), a Kripke structure \( M = (S, R, s_0) \), and a definition of \( \pi(s) \) for \( s \in S \), we say that \( M \) is symmetric with respect to \( G \) if, for all \( \pi \in G \), \( \pi(R) := \{ (\pi(s), \pi(t)) : (s, t) \in R \} \) satisfies \( \pi(R) \subset R \). In this case, it can be shown that in fact \( \pi(R) = R \), and that \( G \) is a group with function composition as the group operation. If \( G \) contains all permutations over \([1..n]\), \( M \) is called fully symmetric. This paper focuses on fully symmetric systems. \( G \), the full symmetry group, is therefore henceforth omitted.

The orbit relation \( \theta(s, t) := \exists \pi : \pi(s) = t \) defines an equivalence between states; the equivalence classes it entails are called orbits. Instead of considering all states in \( S \), it suffices now to choose a small set \( \text{Rep} \) of representatives. This choice is reflected by the representative relation \( \xi \subset S \times \text{Rep} \), which assigns to every state in \( S \) elements of \( \text{Rep} \) such that:

- **soundness:** for all \( (s, r) \in \xi \), there exists \( \pi \) such that \( \pi(s) = r \) (i.e. \( \xi \subset \theta \)), and
- **totality:** for all \( s \in S \), there exists \( r \in \text{Rep} \) such that \( (s, r) \in \xi \).

The symmetry-reduced transition relation is obtained by replacing source and target of edges in \( R \) by representatives:

\[
\bar{R} = \{ (\bar{s}, \bar{t}) \in \text{Rep} \times \text{Rep} : \exists s, t \in S : (s, \bar{s}) \in \xi, (t, \bar{t}) \in \xi \land (s, t) \in R \}. \tag{1}
\]
The structure $\overline{M} := (\overline{\text{Rep}}, \overline{R}, \overline{s}_0)$, for any $\overline{s}_0$ with $(s_0, \overline{s}_0) \in \xi$, is called the \textit{quotient model} of $M$. For suitable choices of $\text{Rep}$ and $\xi$ it can be shown that $\overline{M}$ is bisimulation-equivalent to $M$, and therefore

$$M, s \models f \iff \overline{M}, \overline{s} \models f$$

for any $\overline{s}$ such that $(s, \overline{s}) \in \xi$ and every symmetric formula $f$: for all $\pi$, every $s \in S$ and every maximal propositional subformula $p$ appearing in $f$, $M, s \models p \iff M, \pi(s) \models p$.

The “suitable choices” for $\text{Rep}$ and $\xi$ turn out to be crucial for efficiency.

### 2.3 Unique Representatives

It seems natural to pick exactly one representative from each orbit, such that the relation $\xi$ becomes a function. For instance, given a system state as an $n$-tuple over the process locations $L$, $\xi$ could return the tuple with the locations sorted according to some ordering within $L$ [LN91]. This mapping is sound, since sorting amounts to applying a permutation. It is also total, since every system state can be sorted in this way. Finally, the structure $\overline{M}$ derived from this choice of $\text{Rep}$ is indeed bisimulation-equivalent to $M$.

The only currently known way to construct a BDD for $\xi$ with unique representatives is by first building the BDD for the orbit relation $\theta$ and then projecting the second component of $\theta$ onto $\text{Rep}$: $\xi = \{(s, r) \in \theta : r \in \text{Rep}\}$. Unfortunately, this approach is generally problematic in terms of both time and space [CEFJ96]: The orbit problem—are two states related by $\theta$?—is at least as hard as the graph isomorphism problem, for which no polynomial-time algorithm is known. Making it worse for symbolic representations, the BDD of the orbit relation is, for many common symmetry groups, of size at least $\min\{2^n, 2^{|L|}\}$.

### 2.4 Multiple Representatives

A computationally less expensive choice of $\text{Rep}$ and $\xi$ is possible if the uniqueness requirement for the representatives is dropped. This approach imposes a few weaker constraints on $\text{Rep}$ and $\xi$, which we sketch here briefly; for details see [CEFJ96].

**Definition 1** ([CEFJ96]) Let $\text{Rep}$ be a set of representatives and $\xi$ a sound and total representative relation. A set $C$ of permutations is complete if:

- for all $(s, r) \in \xi$, there exists $\pi \in C$ such that $\pi(s) = r$, and
- for all $\pi \in C$ and $r \in \text{Rep}$, $(\pi(r), r) \in \xi$.

Notice that if the representatives are unique as in 2.3, the full symmetry group $G$ is a complete set. We hope, however, to find a small complete subset $C$. Intuitively, we can then restrict our attention to permutations from $C$ in the search for representatives of a given state.

**Theorem 2** ([CEFJ96]) Let $\text{Rep}$ and $\xi$ be as in definition 1. If there exists a complete set $C$, then $M, s \models f \iff \overline{M}, \overline{s} \models f$ with $\overline{M}, \overline{s}$ and $f$ as in (2).
In practice, it is the programmer’s responsibility to first define a set $Rep$ representable by a small BDD. In [CEFJ96], it is described how a suitable set $C$ can be derived. By finally defining $\xi$ as
\[(s, r) \in \xi \iff r \in Rep \land \exists \pi \in C : \pi(s) = r,\]
$C$ is a complete set for $Rep$ and $\xi$. If the expression $\exists \pi \in C : \pi(s) = r$ and $R$ can also be encoded succinctly, then the BDD for $\bar{R}$ as in (1) is small; the orbit relation is nowhere used. By theorem 2, we can now perform model checking on $\bar{M}$.

The symmetry reduction effect is negatively impacted by choosing several representatives per orbit. While this could still be advantageous when using BDDs, it is not clear that $Rep$ can always be chosen to allow a small BDD for $\xi$ and $\bar{R}$. In the remainder of this paper, we argue that in the case of full symmetry, a solution exists that avoids all these problems altogether.

3 Generic Representatives – A Case Study

Full symmetries occur frequently in practice, whenever a system is composed of unordered, pairwise interchangeable components. This is the case for clique networks of processes, but also for bus and star topologies, where components communicate via a centralized hub (such as in cache coherence protocols). In the latter cases, the bus or hub can be “factored out”, while full symmetry reduction can be applied to the remaining processes.

A fully symmetric system is concisely specified by the number $n$ of processes, possible global variables with initial values, and the common program executed by all processes. As an abstraction of this program, we assume, for the purpose of describing the formal translation into BDDs, the input model of synchronization skeletons. These skeletons are appropriate and powerful enough to describe most control-intensive synchronization problems over finite domains. Combinations of values for the local variables of a process are abstracted into a local state; assignments to those variables are represented as local state changes. Sequential code executed by a process in an atomic action is abstracted into a single transition.

As an example, consider a token-based solution to the $n$-process Mutual Exclusion problem with a global variable $tok \in [1..n]$, and the skeleton in figure 1. A skeleton’s arcs can be labeled with guards (shown in the diagram above the arc) and actions (shown below it, executed after the transition). The skeleton in the figure allows a process to
Variables:

\[ n_N, n_T, n_C : [0..n] \]

\[ TOK : \{ N, T, C \} \]

Initial values:

\[ (n_N, n_T, n_C) := (n, 0, 0) \]

\[ TOK := N \]

// from transition
// N → T:
if \( n_N > 0 \)
  if \( TOK = N \)
    if \( n_N = 1 \)
      \( TOK := T \)
  else
    \( TOK := \{ N, T \} \)

\[ n_N := n_N - 1 \]
\[ n_T := n_T + 1 \]

Fig. 2. Generic version of the token-based Mutual Exclusion solution

enter its critical section \( C \) if it currently possesses the token (\( tok = self \)). Upon leaving \( C \), it sets \( tok \) to a nondeterministic value in \([1..n]\). The skeleton gives rise to a fully symmetric structure, as we will see in the next section for skeletons written in a specific input syntax.

We now want to construct a new program based on counters that yields a bisimulation-equivalent structure. Instead of a local state variable for each process, we somewhat conversely declare global counter variables for each local state, calling them \( n_N, n_T, n_C \). A slight challenge is provided by the \( tok \) variable with range \([1..n]\). Since the counter variables deliberately ignore process identities, we cannot check a guard like \( tok = self \) any more. However, assume there are several processes in location \( T \). Since they are indistinguishable, it does not matter which of them has the token (if any). Rather, it suffices to remember, in a new variable \( TOK \), the location of the process possessing the token. Thus, \( TOK \) ranges over \( \{ N, T, C \} \).

The translated program consists of the variables and statements shown in figure 2. The values of the counter variables range from 0 to the number of processes, \( n \). The initial values of all four variables follow from the fact that all processes start out in location \( N \). All transitions in the new program require that the counter of the source state is positive, since the transition can be taken only if there is a process in that state.

The first transition, \( N \rightarrow T \), has apparently nothing to do with the token, since \( tok \) does not explicitly appear in it. However, the process executing it might be the one possessing the token, in which case the new variable \( TOK \) must be updated from \( N \) to \( T \). If \( TOK = N \) and \( n_N = 1 \), then the executing process has the token, and we set \( TOK \) to \( T \). If \( TOK = N \) and \( n_N > 1 \), then the process executing the transition may or may not be the one possessing the token, so we must set \( TOK \) to \( T \), or \( TOK \) must remain at \( N \), respectively. Hence, the new program has two transitions in this case, which we abbreviate by a nondeterministic assignment \( TOK := \{ N, T \} \). Finally, the actual location change is reflected by decreasing \( n_N \) and increasing \( n_T \). A similar, but simpler reasoning motivates the translation of the other two transitions; in particular, the condition \( n_L > 0 \) in the assignment to \( TOK \) in the last statement ensures that only locations in which there is at least one process are nondeterministically chosen.
The property to be verified also needs to be translated into counters. As an example, compare the mutual exclusion (safety) and communal progress (liveness) requirements in specific and generic notation:

<table>
<thead>
<tr>
<th>Specific</th>
<th>Generic</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Safety:</strong></td>
<td>AG $\forall i, j : i \neq j : - (C_i \land C_j)$</td>
</tr>
<tr>
<td><strong>Liveness:</strong></td>
<td>AG($\exists i T_i \Rightarrow AF \exists j C_j$), AG($n_T &gt; 0 \Rightarrow AF n_C &gt; 0$).</td>
</tr>
</tbody>
</table>

The liveness property states that if there is some process in its trying region, then in any possible future, there should eventually be some process entering its critical section. This property is weaker than progress of an individual process, formally AG $\forall i : (T_i \Rightarrow AF C_i)$. The latter formula, however, is not symmetric, since the maximal propositional subformula $T_i$ is not invariant under permutations. It can therefore not directly be verified over a symmetry reduced structure (whether specific or generic). One approach to overcoming this problem is to “factor out” one of the processes and treat its local variables as global. The progress property is formulated for this process, and symmetry reduction is applied to the remaining ones. This approach is described in more detail by Pnueli, Xu, and Zuck [PXZ02], incidentally for counter-abstracted programs.

To see that implementing the above translation is tantamount to performing symmetry reduction on the program text, notice that all states from one equivalence class of the original system are mapped by the translation to the same counter tuple $(TOK, n_N, n_T, n_C)$. This tuple can therefore be viewed as an “unusual notation” for the representative of the orbit—we call it a generic representative. The new program can now be transformed into a Kripke structure, represented by BDDs, and model checked.

### 4 Translating Symmetric Programs into Generic Form

The global variable $tok$ in the previous section contains a process index, which is lost after the introduction of counters. Such variables require special treatment during the translation process. We call them id-sensitive. Global variables independent of process identities, for example a boolean semaphore, are, as we shall see, much simpler to handle. We refer to them as id-independent variables.

We assume a program $P$ in the form of the following parameters: (1) the number $n$ of processes, (2) any number of id-independent global variables, given as a single vector $v$ with range $V$ (cross product of individual ranges), along with initial value $x_0$, (3) any number $z$ of id-sensitive global variables, given as $d = (d_1, \ldots, d_z)$ with range $[1..n]^2$, along with initial value $k_0$, and (4) a synchronization skeleton. The latter is a finite directed graph, each node of which represents (and is identified with) a process location; call their number $l$. One of the nodes, $I_0$, is the distinguished initial location of every process. The edges may be labeled with a guard and an action (which default to true and no-op, respectively).

**Syntax of Guards.** Guards are arbitrary propositional combinations of boolean-valued basic guards, the latter being conditions on process locations and expressions over global variables. In order to ensure full symmetry of the structure entailed by the program, basic guards must meet certain criteria.
### Definition 3
For a quantified boolean formula $h$ over atoms of the form $L_i$, $i \in [1..n]$, and a permutation $\pi$ on $[1..n]$, define $\pi(h)$ by $\pi$ acting upon the indices. Formula $h$ is fully symmetric if $h \iff \pi(h)$ is valid.

Some basic guards satisfying this definition are listed in Table 1. As an example, the guard, exactly one process is in location $L$, formally $(\forall i : L_i) \land (\forall i, j : L_i \land L_j \Rightarrow i = j)$, is equivalent to $-0 \land -2$, where 0 and 2 are two basic guards from the table. It is more succinctly written as $n_L = 1$ in generic terms.

Any (syntactically valid) expression over id-independent global variables is “by nature” fully symmetric and thus a legal basic guard. As for an id-sensitive variable $d$, we allow the expressions $d = \text{self}$ and $d \neq \text{self}$ as basic guards.

### Syntax of Actions.
An action consists of at most one assignment to each of the global variables. The execution model for the assignments—e.g. parallel or sequential—is left to the implementation, since it is irrelevant for the translation of the source program into generic representatives.

As with guards, to ensure full symmetry the syntax of actions must be restricted. Any (syntactically valid) assignment to the id-independent variables is legal, since it is irrelevant for the translation of the source program into generic representatives.

### Definition 4
A program specified in the above syntax defines a Kripke structure $M = (S, R, s_0)$ with $S = V \times [1..n]^2 \times [1..l]^n$, $s_0 = (x_0, k_0, l_0, \ldots, l_0)$, and $R$ containing all pairs $(s, t)$ with

$$s = (x, k, l_1, \ldots, l_{i-1}, A, l_{i+1}, \ldots, l_n), \quad t = (x', k', l_1, \ldots, l_{i-1}, B, l_{i+1}, \ldots, l_n)$$

such that there is an edge $e : A \rightarrow B$ in the skeleton with a guard that evaluates to true for $v = x$, $d = k$, $\text{self} = i$ and process locations as in $s$, and $e$’s action $A$ satisfies the Hoare triple $\langle v = x \rangle A(v = x')$, and for each id-sensitive variable $d$ with values $k$ and $k'$ in $s$ and $t$, resp., $A$ has an assignment $d := \text{self}$ and $k' = i$, or an assignment $d := \text{ndet}Z$ for some $Z$ with $k' \in Z$, or $A$ has no assignment to $d$ and $k' = k$.

### Theorem 5
For $s = (x, k_1, \ldots, k_z, l_1, \ldots, l_n)$, let $\pi(s) = (x, \pi^{-}(k_1), \ldots, \pi^{-}(k_z), l_{\pi(1)}, \ldots, l_{\pi(n)})$ and $\pi(R)$ as in section 2.2. $M$ is fully symmetric, that is, $\pi(R) \subset R$ for all permutations $\pi$. 

---

### Table 1. Fully symmetric basic guards on process locations

<table>
<thead>
<tr>
<th>no.</th>
<th>Basic Guard</th>
<th>Generic version</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$\forall i : \neg L_i$</td>
<td>$n_L = 0$</td>
<td>none</td>
</tr>
<tr>
<td>1</td>
<td>$\forall i : L_i$</td>
<td>$n_L = n$</td>
<td>all</td>
</tr>
<tr>
<td>2</td>
<td>$\exists i, j : i \neq j : L_i \land L_j$</td>
<td>$n_L \geq 2$</td>
<td>at least two</td>
</tr>
</tbody>
</table>
We are now ready to describe the translation of program $\hat{P}$ from its components (1) through (4) (beginning of this section): The new program $\hat{P}$ consists of the same variable $v$ with initial value $x_0$, further variables $\hat{d}_j$, $j \in [1..z]$, with range $[1..l]$ and common initial value $I_0$, and variables $n_1, \ldots, n_l$ with range $[0..n]$ and initial values $n_{I_0} = n$, $n_L = 0$ for $L \neq I_0$. Every edge of the skeleton is translated into a statement as follows:

\[
\begin{align*}
\text{if } n_A > 0 & \land \text{gen}(\text{guard}) \\
update1(\text{guard}) & \\
n_A := n_A - 1 \\
n_B := n_B + 1 \\
update2(\text{action})
\end{align*}
\]

The condition $n_A > 0$ ensures that there is a process in location $A$. The guard is translated by a function $\text{gen}$ as follows: each basic guard on process locations is replaced according to table 1. For an id-sensitive variable $d_j$, guard $d_j = \text{self}$ is replaced by $\hat{d}_j = A$, guard $d_j \neq \text{self}$ by $\hat{d}_j \neq A \lor n_A \geq 2$ (if $n_A \geq 2$, there is a process $i$ in location $A$ with $d_j \neq i$; hence $d_j \neq \text{self}$ is true for that process). Expressions over $v$ are unchanged.

Definition 6 Program $\hat{P}$ defines a Kripke structure $\hat{M} = (\hat{S}, \hat{R}, \hat{s}_0)$ with $\hat{S} = V \times [1..l]^z \times [1..n]^l$, $\hat{s}_0 = (x_0, I_0, \ldots, I_0, n_1, \ldots, n_l)$ such that $n_{I_0} = n$, $n_L = 0$ for all $L \neq I_0$, and $\hat{R}$ containing all pairs $(\hat{s}, \hat{t})$ such that there is a (nondeterministic) statement in $\hat{P}$ whose top-level condition $n_A > 0 \land \text{gen}(\text{guard})$ evaluates to true and that contains an execution that, applied to $\hat{s}$, results in $\hat{t}$.
Theorem 7  Structures $M$ (definition 4) and $\hat{M}$ are bisimulation-equivalent via
\[ b: S \to \hat{S}, \quad b(\mathbf{x}, k_1, \ldots, k_z, l_1, \ldots, l_n) = (\mathbf{x}, l_{k_1}, \ldots, l_{k_z}, n_1, \ldots, n_l) \]
with $n_L := |\{ j \in [1..n] : l_j = L \}|$. Function $b$ maps every state to its unique generic representative. The following theorem shows that although generic representatives are not based on permutations, they define the same equivalence classes as the orbit relation:

Theorem 8  For any $r, s \in S$, $b(r) = b(s)$ if and only if $\exists \pi : \pi(r) = s$.

In order to model check over structure $\hat{M}$, the specification must be rewritten in generic notation. We assume it is a CTL formula whose atomic propositions are fully symmetric expressions on local state variables (translated like the examples in table 1) and expressions on the id-independent global variable (unchanged). Such a formula is symmetric in the sense defined right below (2).

Note that the translation of the program as well as of the formula can be done fully automatically, in time linear in the size of the program text.

5 Translating Generic Programs into BDDs

In this section, we show how the statements of the generic program, obtained in section 4, can be encoded in a BDD efficiently. We will also estimate the sizes of those BDDs, depending on $n, l$ and the size of the input synchronization skeleton. In this section we ignore the existence of the id-independent variable $v$: since expressions involving it are subject to no restrictions, BDD sizes cannot be estimated. However, those expressions are not altered during the translation; hence they do not contribute any change in BDD size.

The generic structure $\hat{M} = (\hat{S}, \hat{R}, \hat{s}_0)$ is the disjunction of statements of the form in (4) in section 4. BDDs implementing those statements can be obtained as follows:

- $n_A > 0$ iff there is at least one true bit among the $\lceil \log(n + 1) \rceil$ bits representing $n_A$. This can be implemented as a disjunction over all those bits. The resulting BDD size is linear in the number of participating bits: $O(\log n)$.

- $\text{gen}(\text{guard})$ is a propositional combination of basic generic guards. Guards from table 1 can be realized as above with a BDD that compares the constant bit-wise against the counter variable; size $O(\log n)$. Basic generic guards involving the id-sensitive variable have the form $\hat{d} = A$ or $\hat{d} \neq A \lor n_A \geq 2$, which can again be verified bit-wise; these BDDs thus have maximum size $O(\log l \log n)$ ($\hat{d} \in [1..l], n_A \in [0..n]$). Let $F$ denote the number of basic guards appearing in $\text{guard}$. The total BDD size for this part of a transition is then no more than $O((\log l \log n)^F)$. Since $F$ is typically a small constant, this bound is usually polynomial in practice.

- $\text{update1}(\text{guard})$: an if-then-else statement can be implemented using the common ITE operation for BDDs. Since the expressions contained inside the if-then-else are again comparisons against constants, the entire statement can be encoded in a BDD of size $O(\log^{\alpha} l \cdot \log^{\beta} n)$, for small constants $\alpha$ and $\beta$. 

\( n_L := n_L \pm 1 \): since the right-hand side is not a constant, a bit-wise comparison is not possible. The increment can be implemented by searching (using existential quantification) for a bit position \( i \) at which \( n_L \) is 0, \( n_L' \) (the next-state value) is 1, for all preceding bits \( n_L \) and \( n_L' \) are identical, and for all succeeding bits \( n_L \) is 1 and \( n_L' \) is 0. The worst-case BDD size over two variables of \( \lceil \log(n+1) \rceil \) input bits is \( 2^{\lceil \log(n+1) \rceil} = O(n^2) \).

update2 (action): assignment \( \hat{d} := \text{ndet}\{L : n_L > 0\} \) can be realized with a BDD for the expression \( \bigvee_{L \in [1..l]}(n_L > 0 \land \hat{d}' = L) \) of size \( O((\log n \log l)^l) \). The BDD for the if-then-else statement then has size \( O(\log^2 n \cdot (\log n \log l)^{2l}) \).

Assuming (very defendably) that \( F \) is a small constant, we can see that all parts of the translation of an edge can be expressed with a BDD that is low-degree polynomial in \( n \), although, with respect to \( l \), it can be of order \( (\log l)^{2l} \) (caused by the \( d := \text{ndet}\{L : n_L > 0\} \) statement). The complexity of the overall transition relation depends on the way the individual statements are combined, but it is guaranteed to be polynomial in \( n \) as well.

It is interesting to investigate how the relative sizes of \( n \) and \( l \) influence the benefit of generic representatives. Because of the \( n \log l \) input variables of the BDDs for the specific representatives algorithm (\( n \) variables of range \([1..l]\) for \( \theta, \xi \) and \( \bar{R} \)), hence a maximum specific BDD size of roughly \( l^n \), it can be assumed that the generic method is most useful if \( n \) is larger than \( l \). Asymptotically, this is the case if \( l \) is a constant and \( n \) is considered variable. This situation occurs frequently in practice, since, for a given application, the number \( l \) of local states is often fixed. Our second experimental case, presented in the next section, is such an instance.

6 Experimental Results

We compare traditional to generic symmetry reduction using two examples:

The first is an artificial Mutual Exclusion scenario that allows us to show how the generic method scales for varying values for \( n \) and \( l \). Each process can be in one of the local states \( L^1, \ldots, L^l \), where \( L^{l-1} \) and \( L^l \) take the rôles of the trying region and critical section, respectively. The process must go through \( L^1 \) to \( L^{l-1} \) in this order before proceeding into \( L^l \). In addition, the transition into \( L^l \) is protected by a binary semaphore, which is released again upon the process’ return to \( L^1 \):

<table>
<thead>
<tr>
<th>Transition</th>
<th>Guard</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>( L^i \rightarrow L^{i+1} ) for ( 1 \leq i \leq l-2 )</td>
<td>true</td>
<td>no-op</td>
</tr>
<tr>
<td>( L^{l-1} \rightarrow L^l )</td>
<td>!sem</td>
<td>sem := 1</td>
</tr>
<tr>
<td>( L^l \rightarrow L^1 )</td>
<td>true</td>
<td>sem := 0</td>
</tr>
</tbody>
</table>

As a second example, we chose a variant of the MCS list-based queuing lock with atomic compare_and_swap instruction [MCS91, also used in ID96]. The algorithm consists of an acquire and a release operation for a lock with the property that a process waiting for the lock spins only on process-local variables, instead of spinning on a shared variable (like a semaphore). According to [MCS91], spins on shared variables can cause memory detention and severe system performance degradation.
For the second example, the input was not a synchronization skeleton, but the program text for the two operations. In order to perform counter abstraction on this symmetric system, the number of local states needs to be determined. The acquire operation forces processes to line up for the lock in a queue. Each process remembers its successor, which can be any of the \( n - 1 \) other processes, such that the number of local states of a process is not constant. While forming a queue is a valuable property for enforcing a special type of liveness on the processes, it is less relevant for the verification of safety properties. We therefore generalized the system so as to allow any process that is “ready” to obtain the lock to do so. Since the safety property—no two processes can acquire the lock at the same time—turned out to be true for this conservative abstraction with a constant number of 28 local states, we conclude that it holds in a system that enforces FIFO order.

For both problems, we experimented with unique, multiple and generic representatives. For multiple ones, we chose the set \( \text{Rep} \) as follows:

\[
r \in \text{Rep} \iff \exists i: 1 \leq i \leq l : \text{process 1 is in location } L^i \land \text{locations } L^j \text{ with } j < i \text{ do not appear in } r.
\]

For example, using \( l = 3 \), the states \((L^1, L^2, L^1), (L^3, L^3, L^1), (L^1, L^3, L^2)\) and \((L^2, L^3, L^2)\) belong to the set \( \text{Rep} \), but are not unique representatives, in which the superscripts have to be in order. It turns out that the BDD for the representative relation \( \xi \) derived from \( \text{Rep} \) can be computed much more efficiently than that for the function \( \xi \) for unique representatives. Looking back at definition 1, the complete set \( C \) to be chosen contains the \( n \) permutations that swap index 1 with index \( i \), for \( 1 \leq i \leq n \). It can be shown that \( C \) indeed satisfies the two properties required in definition 1. \( C \) is exponentially smaller than the full symmetry group.

For the first example, we verified the standard safety property: \( \text{AG} \forall i, j : i \neq j : \neg (L^i_l \land L^j_l) \) (generically: \( \text{AG} n_l < 2 \)). For the second example, we verified that no two processes can acquire the lock at the same time, and also that there is no deadlock in the system. The latter means that it is never the case that all processes are simultaneously spinning in one of the two busy-waits that are present in the operations. Such a situation would cause a deadlock since a process can not free itself from a busy-wait, but can only be unlocked by another processes.

These properties were verified using the CUDD BDD package [S01] for the standard symbolic fixpoint characterization of \( \text{EF bad} \). Table 2 shows how the space requirements and running times of the three methods of symmetry reduction compare.

**Discussion.** First, for multiple and generic representatives, it can be seen that there is still room to grow memory-wise, but not necessarily so for unique representatives. Indeed, the main motivation for research on alternatives to unique representatives was the impractical BDD size of the orbit relation.

Further, the unique representatives approach spends nearly all of its time on the orbit relation construction. The use of multiple representatives clearly reduces memory and time requirements. The generic representatives solution outperforms, by several orders of magnitude, the other two both in terms of memory and time, and hence in the size of problems it can handle. According to the table, although multiple representatives do remedy the major disadvantage of an orbit relation based solution somewhat, generic representatives have in turn an equally impressive benefit over multiple ones.
Table 2. Space and run time comparisons (i686/1400 Mhz PC, 256MB memory)

<table>
<thead>
<tr>
<th>Choice of n, l</th>
<th>Unique Specific Representatives</th>
<th>Multiple Specific Representatives</th>
<th>(Unique) Generic Representatives</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>no. of live BDD nodes</td>
<td>time in sec. (% orbit rel.)</td>
<td>no. of live BDD nodes</td>
</tr>
<tr>
<td>M 8 4</td>
<td>114,894</td>
<td>8.2 (97%)</td>
<td>2,211</td>
</tr>
<tr>
<td>U 6 5</td>
<td>2,152,710</td>
<td>137.3 (97%)</td>
<td>6,612</td>
</tr>
<tr>
<td>T 16 16</td>
<td>? &gt;15h (100%)</td>
<td>132,377</td>
<td>6.6</td>
</tr>
<tr>
<td>E 64 16</td>
<td>— —</td>
<td>599,561</td>
<td>198.8</td>
</tr>
<tr>
<td>X 128 128</td>
<td>— —</td>
<td>? &gt;15h</td>
<td>69,060</td>
</tr>
<tr>
<td>256 128</td>
<td>— —</td>
<td>— —</td>
<td>78,060</td>
</tr>
<tr>
<td>M 3 28</td>
<td>113,188</td>
<td>2.4 (79%)</td>
<td>30,614</td>
</tr>
<tr>
<td>C 4 28</td>
<td>9,478,195</td>
<td>43867.7 (95%)</td>
<td>75,604</td>
</tr>
<tr>
<td>S 8 28</td>
<td>? &gt;15h (100%)</td>
<td>272,080</td>
<td>15.4</td>
</tr>
<tr>
<td>L 16 28</td>
<td>— —</td>
<td>2,417,477</td>
<td>5055.3</td>
</tr>
<tr>
<td>K 60 28</td>
<td>— —</td>
<td>? &gt;15h</td>
<td>34,170</td>
</tr>
<tr>
<td>— —</td>
<td>293,981</td>
<td>266.8</td>
<td></td>
</tr>
</tbody>
</table>

7 Conclusion

In this paper, we investigated the use of generic representatives in symbolic model checking of fully symmetric systems. Compared to unique representatives, with generic ones there is no need to construct the orbit relation. Compared to multiple representatives, the generic ones maintain full symmetry reduction. The BDD derived from the generic structure $\hat{M}$ turned out to be small for the examples we experimented with. For the class of programs presented here, the translation into generic representatives can be done automatically and in negligible time.

Generic representatives seem to prove useful outside the symbolic domain as well: we translated some of the fully symmetric example programs coming with the Murϕ explicit state verifier [DDHY92] into generic representatives. For some examples, we obtained savings in terms of both time and space of several orders of magnitude over Murϕ’s symmetry reduction algorithms (using unique or multiple representatives).

Related and Future Work. Barner and Grumberg [BG02] considered combining symmetry and symbolic representation using BDDs mainly for falsification. They perform reachability analysis by discarding states symmetric to previously seen states. However, due to orbit complexity problems, the algorithm uses multiple representatives and therefore forgoes some of the symmetry reduction possible. Also, according to [BG02], computation costs often incur the use of under-approximations of the set of reached representatives, which renders the algorithm inexact.

Finite counters have been used previously to abstractly represent states of systems with many processes. Pnueli, Xu and Zuck [PXZ02] used truncated counters with values 0, 1, or 2 to approximate the number of processes in certain locations in reasoning about symmetric parameterized systems. Emerson and Trefler [ET99] used counters in the form of generic representatives in connection with fully symmetric programs. Other
examples can be found in the work by Emerson and Srinivasan [ES90] on synthesis of parameterized programs and in the work by Pong and Dubois [PD95] on cache protocol verification.

Several years ago, Ip and Dill [ID96] introduced scalar sets in the description of the input program to enforce full symmetry. The Murϕ verifier is an explicit-state implementation of this approach. Since Murϕ was originally not designed to exclusively target symmetric systems, Murϕ’s input language is more general. In addition to non-symmetric programs, it allows one to write programs exhibiting symmetry other than process symmetry, which is discussed in this paper. To make our approach more readily applicable, we would like to allow a more convenient input language than synchronization skeletons, perhaps similar to that of Murϕ.

The present formulation of generic representatives is directly only applicable to (the common case of) fully symmetric systems. We would like to do research on systems whose symmetry group is the product of full symmetry groups of subsystems [CEJS98, section 5.1], and systems that are almost, but not fully, symmetric [ET99]. The ultimate goal is to apply the generic method to some larger, perhaps industrial-size examples.

References


On the Correctness of an Intrusion-Tolerant Group Communication Protocol

Mohamed Layouni1, Jozef Hooman2, and Sofiène Tahar1

1 Department of Electrical and Computer Engineering
Concordia University, Montreal, Canada
{layouni,tahar}@ece.concordia.ca
2 Computing Science Department
University of Nijmegen, Nijmegen, The Netherlands
hooman@cs.kun.nl

Abstract. Intrusion-tolerance is the technique of using fault-tolerance to achieve security properties. Assuming that faults, both benign and Byzantine, are unavoidable, the main goal of Intrusion-tolerance is to preserve an acceptable, though possibly degraded, service of the overall system despite intrusions at some of its sub-parts. In this paper, we present a correctness proof of the Intrusion-tolerant Enclaves protocol [1] via an adaptive combination of techniques, namely model checking, theorem proving and analytical mathematics. We use Murphi to verify authentication, then PVS to formally specify and prove proper Byzantine Agreement, Agreement Termination and Integrity, and finally we mathematically prove robustness of the group key management module.

1 Introduction

A substantial progress in the formal verification of cryptographic protocols has been achieved during the last decade. A wide variety of techniques has been developed to verify a number of key security properties ranging from confidentiality and authentication to atomic transactions and non-repudiation [2,3]. Nevertheless, all the focus was either on two-party protocols (i.e., involving only a pair of users) or, in the best cases, on group protocols with centralized leadership (i.e., a presumably trusted fault-free server managing a group of users). In the present work, we are concerned with the verification of the intrusion-tolerant Enclaves [1]: a group-membership protocol with a distributed leadership architecture, where the authority of the traditional single server is shared among a set of n independent elementary servers, of which at most f could fail at the same time. The protocol has a maximum resilience of one third (i.e., $f \leq \left\lfloor \frac{n-1}{3} \right\rfloor$) and uses an algorithm similar to the consistent broadcast of Bracha and Toueg [4].

The primary goal of Enclaves is to preserve an acceptable group-membership service of the overall system despite intrusions at some of its sub-parts. For instance, an authorized user $u$ who requests to join an active group of users should be eventually accepted, despite the fact that faulty leaders may coordinate their messages in such a way as to mislead non-faulty leaders (the majority)
into disagreement, and thus into rejecting user $u$. Moreover, in order to prevent malicious leaders from leaking sensitive information (e.g., group keys) or providing clients with fake group keys, Enclaves uses a verifiably secure secret sharing scheme.

To achieve its intrusion-tolerant capabilities, Enclaves relies on the combination of a cryptographic authentication protocol, a Byzantine fault-tolerant leader agreement protocol and a secret sharing scheme. Although we assume the underlying cryptographic primitives and fault-tolerant components to be perfect, one cannot easily guarantee security of the whole protocol. In fact, several protocols had been long thought to be secure until a simple attack was found (see [20] for a survey). Therefore, the question of whether or not a protocol actually achieves its security goals becomes paramount. To date, most of the research in protocol analysis has been devoted to finding attacks on known, either two-party or centralized protocols. In this paper we are concerned with the verification of a distributed multi-leader group communication protocol.

An important issue that arises in formal verification of Byzantine fault-tolerant protocols, is the modeling of Byzantine behavior. How much power should be given to a Byzantine fault and how general should the model be to capture the arbitrary nature of a Byzantine fault behavior? These questions have been extensively studied [7,9,10] and continue to be a center of focus. In this paper, faults are only limited by cryptographic constraints. For instance, faulty leaders can arbitrarily send random messages, reset their local clocks and perform any action without satisfying its precondition. They cannot, however, decrypt a message without having the appropriate key, or impersonate other participants by forging cryptographic signatures. More details about our fault assumptions are discussed in Section 2.

In this work, we discuss a formal analysis of the overall Byzantine fault-tolerant Enclaves protocol. We experiment with an adaptive combination of techniques, chosen according to the nature of the correctness arguments in each module, the environment assumptions, and the easiness of performing verification. For instance, we found it more profitable to model-check the authentication module by taking advantage of the reduction techniques available in Murphi [15]. The Byzantine leaders agreement module, however, was a little trickier. In fact, the latter relies, to a large extent, on the timing and the coordination of a set of distributed actions, possibly performed by Byzantine faulty processes whose behavior is hard to represent in a model-checker. Instead, we use PVS [21] and formalize the protocol in the style of Timed-Automata [5]. This formalism makes it easy to express timing constraints on transitions. It also captures several useful aspects of real-time systems such as liveness, periodicity and bounded timing delays. Using this formalism, we specified the protocol for any number of leaders, and we proved safety and liveness properties such as Proper Agreement, Agreement Termination and Integrity. Finally, the group-key management module is based on a secret sharing scheme whose security relies fundamentally on the hardness of computing discrete logarithms in groups of large prime order. Due to the hardness of expressing the latter
correctness arguments in a formal language, we found it more convenient to
give a manual proof of the module’s robustness and unpredictability properties,
using the Random Oracle model [19].

The remainder of this paper is organized as follows. In Section 2, we give an
overview of the architecture and design goals of Enclaves, and we explicitly state
our system model assumptions. In Section 3, we describe the model checking of
the authentication module in Murphi. In Section 4, we present how we model
the elementary components of the Byzantine leader agreement module in PVS
and how we build the final protocol model out of these ingredients. In Section 5,
we formulate and prove our correctness theorems. In Section 6, we briefly give
the mathematical proof of robustness and unpredictability of the group key
management module. In Section 7, we discuss some related work. Finally in
Section 8, we conclude the paper by commenting on our results and stating
some perspectives for future work.

2 The Enclaves Protocol

Enclaves [1] is a protocol that enables users to share information and collabo-
rerate securely through insecure networks such as the Internet. Enclaves provides
services for building and managing groups of users. Access to a given group is
granted only to sets of users who have the right credentials to do so. Authorized
users can dynamically, and at their will, join, leave, and rejoin, an active group.
The group communication service relies on a secure multicasting channel that
ensures integrity and confidentiality of group communication. All messages sent
by a group member are encrypted and delivered to all other group members.

The group-management service consists of user authentication, access con-
trol, and group-key distribution. Figure 1 shows the different phases of the pro-
tocol execution. Initially at time $t_0$, user $u$ sends requests to join the group to a
set of leaders. These leaders locally authenticate $u$ within time interval $[t_1,t_2]$.
When done, the agreement procedure starts and terminates at time $t_4$ by reach-
ing a consensus as whether or not to accept user $u$. Finally on acceptance, user
$u$ is provided with the current group composition, as well as information to re-
construct the group-key. Once in the group, each member is notified when a new
user joins or a member leaves the group in such a way that all members are in
possession of a consistent image of the current group-key holders.

In summary, Enclaves should guarantee the following properties, even in the
presence of up to $f$ corrupted leaders:

- **Proper authentication and access control**: Only authorized users can join the
group and an authorized user cannot be prevented from joining the group.
- **Confidentiality of group communication**: Messages from a member $u$ can be
read only by the users who were in $u$’s image of the group at the time the
message was sent.
Given the above assumptions, we prove that the *Proper authentication and access control* requirement holds through (1) the model checking of the Proper Authentication invariant in Murphi (cf. Section 2), and (2) the proofs of Proper Agreement, Agreement Termination and Agreement Integrity theorems in PVS (cf. Sections 3 and 4). In addition, we prove the *Confidentiality of group communication* requirement via a mathematical analysis of the Robustness and Unpredictability properties of the group key management module of Enclaves (cf. Section 6).
3 Model Checking Authentication in Murphi

Murphi has a language that supports scalable models. In a scalable model one typically starts with a small protocol configuration and gradually increases the protocol size. In many cases, errors in the general protocol (possibly infinite state) will also show up in down-scaled (finite state) version of the protocol. The Murphi tool is based on explicit state enumeration and supports a number of reduction techniques such as symmetry and data independency [16,17]. The desired properties of a protocol can be specified in Murphi by invariants. If a state is reached where some invariant is violated, Murphi prints an error trace exhibiting the problem.

Our verification has been conducted as follows. First, we formulated the protocol by identifying the protocol participants, the state variable and messages, and the key actions to be taken. Then we added an intruder to the system. In our model, the intruder is a participant in the protocol, capable of eavesdropping messages in transit, decrypting cipher-text when it has the appropriate keys, and generating new messages using any combination of previously gained knowledge. Finally, we stated the desired correctness conditions and ran the protocol for some specific size parameters.

The main property we are concerned about in this paper is mutual authentication between a given pair of leader and client. More precisely, at the end of a protocol execution between a leader \( L_i \) and a client \( C \), \( L_i \) should be able to assert that it has been talking, indeed, to client \( C \), and vice-versa. The verification has been done by means of invariant checking under the above mentioned assumptions. The client proper authentication invariant is given below. It basically states that for each leader \( i \), if it committed to a session with a client, this client (whose identifier is stored in \( \text{lead}[i].\text{client} \)), must have started the protocol with leader \( i \), i.e., have stored \( i \) in its field leader and be awaiting for acknowledgment (i.e., in state \( C_{\text{ACK}} \)).

\[
\text{invariant "client proper authentication"}
\begin{align*}
\text{forall } i: \text{LeaderId do} \\
& \text{lead}[i].\text{state} = \text{L\_COMMIT} & \\
& \text{ismember(lead}[i].\text{client}, \text{ClientId}) \\
& \text{->} \\
& \text{clnt[lead}[i].\text{client}].\text{leader} = i & \\
& \text{clnt[lead}[i].\text{client}].\text{state} = \text{C\_ACK} \\
\end{align*}
\]

In addition to the above invariant, we have checked a similar one for leaders proper authentication (i.e., the clients are sure about the identity of the leaders they are communicating with). Table 1 shows the number of reached states and CPU run times taken on a 440 Mhz Sparc machine with 256 MB of memory for different sizes of the protocol. The instances we consider, have been chosen to emphasize the weight of each size parameter. For example, the intruder is modeled to be very powerful (intercepts, replays, and generates messages), so adding a second intruder does not increase the intrusion power, it
Table 1. Model checking experimental results

<table>
<thead>
<tr>
<th>Number of Clients</th>
<th>Leaders</th>
<th>Intruders</th>
<th>Network size</th>
<th>States</th>
<th>CPU time</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>4</td>
<td>1</td>
<td>1</td>
<td>4591</td>
<td>13.25 s</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>1</td>
<td>3</td>
<td>125793</td>
<td>331.00 s</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>2</td>
<td>3</td>
<td>277176</td>
<td>1481.35 s</td>
</tr>
<tr>
<td>4</td>
<td>10</td>
<td>1</td>
<td>3</td>
<td>797000</td>
<td>–</td>
</tr>
</tbody>
</table>

just multiplies the complexity. Also, the last row in Table 1, shows a non conclusive result, where Murphi runs out of memory before reaching all possible states.

4 Modeling Byzantine Agreement in PVS

Most group communication protocols, including Enclaves, can be modeled by an automaton whose initial state is modified by the participants’ actions as the group mutates (new members join). Because Enclaves depends also on time (participants timeout, timestamp group views, etc.), it was convenient to model it as a timed automaton. In the current verification, timing is used only to ensure actions progress. Timing, however, is essential to prove upper bounds on agreement delays (e.g., a maximum join delay), but this is beyond the scope of this paper. Participants in a typical run of Enclaves consist of a set of $n$ leaders ($f$ of which are faulty), a group of members, and one or more users requiring to join the group.

In the remainder of this section, we first explain our general PVS theory about timed automata. The parameters of this theory are used here to formalize Enclaves by defining the actions, the states, and the precondition and effect of each action. Finally, the resulting executions of the protocol and fault assumptions are described.

4.1 Timed Automata

We present a general, protocol-independent, theory called TimedAutomata. Given a number of parameters, it defines all possible executions of the protocol as a set of Runs. A run is a sequence of the form $s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} s_2 \xrightarrow{a_2} s_3 \xrightarrow{a_3} \ldots$ where the $s_i$ are states, representing a snapshot of the system during execution and the $a_i$ are the executed actions. A particular protocol (an instance of the timed automaton) is characterized by sets of possible States and Actions, a condition Init on the initial state, the precondition Pre of each action, expressing in which states that action can be executed, the effect Effect of each action, expressing the possible state changes by the action, and a function now which gives the current time in each state. In a typical application, there is a special delay action which models the passage of time and increases the value of now. All other actions do not change time\(^1\).

\(^1\) For more details about the PVS theories and proofs, we refer the reader to the webpage: http://hvg.ece.concordia.ca/Research/CRYPTO/Enclaves.html
4.2 Leaders Actions

To define the actions of the leaders, we first state a few preliminary definitions. Let \( n \) be the number of leaders and let \( f \) be such that \( 3f + 1 \leq n \) (the maximum number of faulty leaders). For simplicity, leaders are identified by an element of \( \{0, 1, \ldots, n - 1\} \). Users are represented by some uninterpreted non-empty type, and time is modeled by the set of non-negative real numbers.

The actions of the protocol are represented in PVS as a data type, which ensures, e.g., that all actions are syntactically different. Thereafter, we define the following actions:

- A general delay action which occurs in all our timed models; it increases the current time (now), and all other clocks that may be defined in the system, with the amount specified by a delay parameter \( del \).
- An announce action is used to send announcement messages of new locally authenticated users to the other leaders of the protocol.
- A trypropagate action allows a user announcement to be further spread among leaders. This action is executed periodically, but it only changes the state of the system if enough announcements \((f + 1)\) have been received for the considered user and it has not already been announced or propagated by the leader in question before.
- An action Tryaccept used to let leaders periodically check whether they have received enough announcements and/or propagation messages for a given user. Once this condition is satisfied, the user is accepted to join the group.
- A receive action allows a leader to receive messages; it removes a received message from the network and adds corresponding data to the local buffer of the leader.
- A crash action models the failure of a leader. After a crash, a leader may still perform all the actions mentioned above, but in addition it may perform a misbehave action.
- An action misbehave models the Byzantine mode of failure and can only be performed by a faulty (crashed) leader.

Besides, we define three time constants for the maximum delay of messages in the network, the maximum delay between trypropagate actions and the maximum delay between tryaccept actions.

4.3 States

In order to properly capture the distributed nature of the network, it is suitable to model two kinds of states: a local state for each leader, accessible only to the particular leader, and a global state to represent global system behavior which includes the local state of each leader, the representation of the network and a global notion of time.

An important part of the local state is the group view, which is a set of users in the current group. In fact, the ultimate goal of Enclaves is to assure consistency of the group views. Moreover, we use a Boolean flag (faulty) marking the leader...
status as faulty or not, some local timers (\textit{clockp} and \textit{clocka}) to enforce upper bounds on the occurrence of \textit{trypropagate} and \textit{tryaccept} actions, and finally a list (\textit{received}) of the leaders from which the local leader received proposals for a given user.

\textbf{Views} : \textbf{TYPE} = \text{setof[UserIds]}

\textbf{LeaderStates} : \textbf{TYPE} =

\begin{itemize}
  \item \text{[view]} : \text{Views},
  \item \text{faulty} : \text{bool},
  \item \text{clockp} : \text{Time}, \% \text{clock for the trypropagate action}
  \item \text{clocka} : \text{Time}, \% \text{clock for the tryaccept action}
  \item \text{received} : \text{[UserIds -> list[LeaderIds]]} \#]
\end{itemize}

We model \textit{Messages} as quadruples containing a source, a destination, a proposed user and a timestamp indicating an upper bound on the delivery time, i.e., the message must be received before the \textit{tmout} value.


In the \textit{global states}, the network is modeled as a set of messages. Messages that are broadcast by leaders are added to this set, with a particular time-out value, and they are eventually received, possibly with different delays and at a different order at recipient ends. The global state also contains the local state of each leader and a global notion of time, represented by \textit{now}.

\textbf{GlobalStates} : \textbf{TYPE} = \text{[# ls : [LeaderIds -> LeaderStates], now : Time, network : setof[Messages] #]}

s, s0, s1 : VAR \text{GlobalStates}

Furthermore, we define a predicate \textit{Init} that expresses conditions on the initial state, requiring that all views, received sets and the network are empty, and all clocks and \textit{now} are set to zero.

\subsection{Precondition and Effect}

For each action \(A\), we define its precondition, expressing when the action is enabled, and its effect. An \textit{announce} action may always occur and hence has precondition \textit{true}. Similarly for \textit{trypropagate} and \textit{tryaccept}, which should occur periodically. Action \textit{receive(i)} is only allowed when there exists a message in the network with destination \(i\). For simplicity, a \textit{crash} action is only allowed if the leader is not faulty (alternatively, we could take precondition \textit{true}). A \textit{misbehave} action may only occur for faulty leaders.
Most interesting is the precondition of the \textit{delay}(t) action. This action increases \textit{now} and all local timers (\textit{clockp} and \textit{clocka}) by \( t \). To ensure that messages are delivered before their time-out value, we require that the condition \textit{prenetwork}, defined below, holds in the state before any \textit{delay}(t) action is taken, which fits our informal assumptions about network reliability.

\textit{prenetwork}(s, t) : \textbf{bool} = \textbf{FORALL} msg :

\[ \text{member}(msg, \text{network}(s)) \text{ IMPLIES } \text{now}(s) + t \leq \text{tmout}(msg) \]

Similarly, there is a condition \textit{preclock} which requires that all timers (\textit{clockp} and \textit{clocka}) are not larger than \textit{MaxTryPropagate} and \textit{MaxTryAccept}, respectively. Since the \textit{trypropagate} and \textit{tryaccept} actions reset their local timers to zero, this may enforce the occurrence of such an action before a time delay is possible.

Next we define the effect of each action, relating a state \( s_0 \) immediately before the action and a state \( s_1 \) immediately afterwards.

- \textit{delay}(t) increments \textit{now} and all local timers by \( t \), as defined by \( s_0 + t \).
- \textit{announce}(i, u) adds, for each leader \( j \) a message to the network, with source \( i \), time-out \( \text{now}(s_0) + \text{MaxMessageDelay} \), proposal \( u \), and destination \( j \).
- \textit{trypropagate}(i) resets \textit{clockp} to zero and adds to the network messages, to all leaders, containing proposals for each user for which at least \( f + 1 \) messages have been received.
- \textit{tryaccept}(i) resets \textit{clocka} to zero and adds to its local view all users for which at least \( n - f \) messages have been received.
- \textit{receive}(i) removes a message with destination \( i \) from the network, say with source \( j \) and proposal \( u \), and adds \( j \) to the list of received leaders for \( u \), provided it is not in this list already.
- \textit{crash}(i) sets the flag \textit{faulty} of \( i \) to \textit{true}.
- \textit{misbehave}(i) may just reset the local timers \textit{clockp} and \textit{clocka} of \( i \) to zero, as expressed by \textit{ResetClock}(\( s_0, i, s_1 \)), or it may add randomly as well as maliciously chosen messages to the network (provided that timeouts are not violated). A misbehaving leader, however, cannot impersonate other protocol participants, i.e., any message sent on the network has the identifier of its actual sender.

### 4.5 Protocol Runs and Fault Assumption

Runs of this timed automata model of Enclaves are obtained by importing the general timed automata theory. This leads to type \textbf{Runs}, with typical variable \( r \). Let \textit{Faulty}(\( r, i \)) be a predicate expressing that leader \( i \) has a state in which it is faulty. It is easy to check in PVS that once a leader becomes faulty, it remains faulty forever. Let \textit{FaultyNumber}(\( r \)) be the number of faulty leaders in run \( r \) (it can be defined recursively in PVS). Then we postulate by an axiom that the maximum number of faults is \( f \) (\textbf{MaxFaults} : \textbf{AXIOM} \textit{FaultyNumber}(\( r \)) \leq f).
5 Proving Byzantine Agreement in PVS

We are interested in verifying the following properties of the Enclaves protocol:

- **Termination**: if user $u$ wants to join an active group and has been announced by enough non-faulty leaders, then eventually user $u$ will be accepted by all non-faulty leaders and become a member of the group.

- **Integrity**: a user that has been accepted in the group should have been announced by a non-faulty leader earlier during the protocol execution.

- **Proper Agreement**: if a non-faulty leader decides to accept user $u$, then all non-faulty leaders accept user $u$ too.

In the remainder of this section, we briefly outline proofs of the above theorems.

**Theorem 1 (Termination)**
For all $r$ and $u$, \texttt{announced\_by\_many}(r, u) implies \texttt{accepted\_by\_all}(r, u)
where
- \texttt{announced\_by\_many}(r, u) expresses that at least $(f + 1)$ non-faulty leaders announced user $u$ during run $r$;
- \texttt{accepted\_by\_all}(r, u) asserts that eventually all non-faulty leaders have user $u$ in their view during run $r$.

**Proof.** Assume \texttt{announced\_by\_many}(r, u), which implies that at least $(f + 1)$ non-faulty leaders broadcast a proposal for $u$. Because of the reliability of the network, eventually these messages will be delivered to their destination, and in particular to the $(n - f)$ non-faulty leaders of the network. They all receive $(f + 1)$ announcement messages for user $u$, which is enough to trigger the propagation procedure (for $u$) for all non-faulty leaders who did not participate in the announcement phase. Now because of the network reliability, we conclude that eventually all non-faulty leaders will receive at least $(n - f)$ approvals for user $u$, enough to make a majority, since $(n - f) > f$ follows from $n > 3f$.

**Theorem 2 (Integrity)**
For all $r$ and $u$, \texttt{accepted\_by\_one}(r, u) implies \texttt{announced\_by\_one}(r, u)
where
- \texttt{accepted\_by\_one}(r, u) holds if at least one leader eventually included $u$ in its view during run $r$;
- \texttt{announced\_by\_one}(r, u) expresses that at least one non-faulty leader announced user $u$ during run $r$.

**Proof.** We proceed by contrapositive and use the non-impersonation property. We assume that for all non-faulty leaders no announcement for user $u$ has been done during run $r$. Now because of non-impersonation, faulty leaders cannot send more than $f$ different announcements. This implies that the leaders would receive no more than $f$ announcements for user $u$, which is not enough to trigger propagation actions. This yields that $u$ will never be proposed by any of the non-faulty leaders, and hence none of them will receive as much as $(n - f)$ messages for $u$ (recall $(n - f) > f$). As a result, user $u$ will never be accepted by any of the non-faulty leaders.
Theorem 3 (Proper Agreement)
For all \( r \) and \( u \), \( \text{accepted\_by\_one}(r, u) \) implies \( \text{accepted\_by\_all}(r, u) \)

Proof. \( \text{accepted\_by\_one}(r, u) \) implies that there exists a non-faulty leader that received at least \((n - f)\) approvals (i.e., announcements or propagation messages) for user \( u \). Among these approvals, at least \((n - 2f)\) come from non-faulty leaders (by non-impersonation). Now because these leaders are non-faulty, they broadcast the same approval to all the other leaders. In addition, because of the network reliability, these messages are eventually delivered to destination. This implies that all \((n - f)\) non-faulty leaders receive eventually the above \((n - 2f)\) approvals. Since \((n - 2f) \geq (f + 1)\), all \((n - f)\) non-faulty leaders have received at least \((f + 1)\) messages for \( u \). Similar to the proof of Termination, the latter implies the start of the propagation procedure, then the reception of at least \((n - f)\) approvals for user \( u \), and finally the acceptance of \( u \) by all non-faulty leaders. □

The above proofs were conducted successfully in PVS and required over 40 lemmas. Integrity and Termination were the most challenging to prove and they helped deduce Proper Agreement.

6 Group Key Management: Mathematical Proof

In the previous sections we discussed authentication and leaders agreement. We saw also that once the leaders agree on accepting a client \( C \), they proceed with providing it with a group key. We direct our focus here to the Enclaves group key management module [1]. This module is based on a secret sharing scheme which ensures that (1) the \( f \) dishonest leaders cannot obtain the group key even if they conspire altogether (at least \((f + 1)\) shares are needed to reconstruct the secret); (2) the group key is renewed every time the group changes (new join or leave); and (3) the clients are able to discern valid key shares from fake ones (possibly issued by malicious leaders).

The group key management protocol of Enclaves is based on previous work of Cachin et al. [19]. The security property of the protocol relies on the hardness of computing discrete logarithms in a group of large prime order. Such a group \( G_q \) can be constructed by selecting two large prime numbers \( p \) and \( q \) such that \( p = 2q + 1 \) and defining \( G_q \) as the unique subgroup of order \( q \) in \( \mathbb{Z}_p^* \). The protocol works as follows. Initially, we assume that a dealer chooses a generator \( g \) of \( G_q \) and a random secret integer \( x \in \mathbb{Z}_q \). The dealer then generates \( n \) shares \( x_1, \cdots, x_n \in \mathbb{Z}_q \) using an \( f \)-threshold \(^2\) Shamir's secret sharing scheme [18]. The dealer secretly transmits the shares \( x_i \) to their corresponding leaders and makes public \( h_i = g^{x_i} \) for all leaders \( \{L_i\}_{i \leq n} \). We denote by \( \tilde{g} = H(G) \) the output of a hash function \( H \) applied to the most recent set of clients forming the group \( G \). In this scheme, the secret group key to be reconstructed by the clients is \( \tilde{g}^x \).

In addition to \( p, q \) and \( g \), we assume that \( H \) is also known to all the participating leaders. Given the above assumptions, the protocol works as follows:

\(^2\) The secret cannot be reconstructed unless \((f + 1)\) shares are available.
1. Leader $L_i$ picks randomly $s \in \mathbb{Z}_q$ and computes $(a, b) = (g^s, \tilde{g}^s)$.
2. Leader $L_i$, then, computes $c = H'(y_i, \tilde{g}, a, b)$, where $y_i = \tilde{g}^{x_i}$, and with $H': G_q^4 \rightarrow \mathbb{Z}_q$ a public hash function.
3. Now leader $L_i$ computes $r = s + cx_i$ and sends each client the quadruple $(y_i, a, b, r)$, that is the share $y_i$ and the proof of validity $(a, b, r)$.
4. Now the client computes $c' = H'(y_i, \tilde{g}, a, b)$, supposed to be equal to $c$, and accepts the share $y_i$ only if the following equations hold:

\[
\tilde{g}^r = a \ y_i^{c'} \quad \text{(1)}
\]

\[
\tilde{g}^r = b \ y_i^{c'} \quad \text{(2)}
\]

Let $S$ be any set of $f + 1$ (or more) shares $y_i$ that a given client has received. For simplicity, assume $S = \{y_1, y_2, ..., y_{f+1}\}$. We denote by $(a_i)_{1 \leq i \leq f+1}$ the Lagrange interpolation coefficients, such that $\sum_{i=1}^{f+1} a_i x_i = x$, where $a_i = \prod_{j \neq i} \frac{j}{j-1}$.

Given the above shares, the clients recover the secret group key as follows:

\[
\tilde{g}^x = \tilde{g}^{(\sum_{i=1}^{f+1} a_i x_i)} = \prod_{i=1}^{f+1} (\tilde{g}^{x_i})^{a_i} = \prod_{i=1}^{f+1} y_i^{a_i}.
\]

### 6.1 Security Analysis: Manual Proof

We sketch proofs of two key properties, namely, robustness and unpredictability.

**Theorem 4 (Robustness)** In the random oracle model, a dishonest leader cannot forge, with a non-negligible probability, a valid proof for a non valid share.

**Proof sketch:** Let $y_i$ be the share provided by leader $L_i$ and $(a, b, r)$ be the corresponding correctness proof. $y_i, a, b$ and $r$ should then satisfy the following equations:

\[
g^r = a \ h_{i}^{c} \quad \text{(3)}
\]

\[
\tilde{g}^r = b \ y_{i}^{c} \quad \text{(4)}
\]

where $c = H'(y_i, a, b, \tilde{g})$. Equation (3) yields $a \in G_q$, since $h_{i}^{c}$ and $g^r$ are both in $G_q$ (Closure of $G_q$ under multiplication). The latter implies that it exists $\gamma \in \mathbb{Z}_q$ such that $a = g^\gamma$. Equation (3) gives: $g^r = g^\gamma g^{cx_i}$, which implies: $r = \gamma + cx_i$. Now equation (4) becomes:

\[
\tilde{g}^r = b \ y_{i}^{c} \iff \tilde{g}^{(\gamma + cx_i)} = b \ y_{i}^{c} \iff \tilde{g}^\gamma b^{-1} = (\tilde{g}^{-x_i} y_i)^c
\]

This yields two possible cases:

---

3. The $a_i$ depend only on the leaders indexes and hence are publicly known.

4. In this model, the hash function can be seen as an oracle producing a random value at each query. If the same query is asked twice, an identical answer is given [19].
1. \(y_i = \tilde{g}^{x_i}\). In this case, the share is correct. \(b = \tilde{g}^Y\) and for all \(c \in \mathbb{Z}_q\) the verifier equations trivially hold.
2. \(y_i \neq \tilde{g}^{x_i}\). In this case, we must have \(c = \log_{\tilde{g}}^{-x_i}(\tilde{g}^Y b^{-1})\).

Once the triplet \((y_i, a, b)\) is chosen, if \(y_i\) is not a valid share, then there exists a unique \(c \in \mathbb{Z}_q\) that satisfies the verifier equations. In the random oracle model, the hash function \(H'\) is assumed to be perfectly random. Therefore, the probability that \(H'(y_i, a, b, \tilde{g})\) equals \(c\), once \((y_i, a, b)\) fixed, is \(\frac{1}{q}\). On the other hand, if the attacker performs an adaptively chosen message attack by querying an oracle \(N\) times, the probability for the attacker to find a triplet \((y_i, a, b)\), such that \(c = H'(y_i, a, b, \tilde{g})\), is \(P_{success} = 1 - (1 - \frac{1}{q})^N \approx \frac{N}{q}\) for large \(q\) and \(N\). Now if \(k\) is the number of bits in the binary representation of \(q\), then \(P_{success} \leq \frac{N}{2^k}\). Since a computationally bounded leader can only try a polynomial number of triplets, then when \(k\) is large, the probability of success is negligible \((P_{success} = \frac{N}{2^k} < 1)\).

**Theorem 5 (Unpredictability)** An attacker that corrupts up to \(f\) leaders cannot, with a non-negligible probability, learn the secret group key \(\tilde{g}^x\).

This has been proved by Cachin et al. [19] and relies on both:

- The perfect cryptography assumption (i.e., conditional entropy is no greater than simple entropy)
  \[S(y_{i_{f+1}} \mid y_{i_1}, y_{i_2}, \ldots, y_{i_j}) = S(y_{i_{f+1}})\] for all \(j \leq f\)

- The Computational Diffie-Hellman assumption [22], which states that there is no polynomial time probabilistic algorithm that computes \(y_i = \tilde{g}^{x_i}\) given \(g, \tilde{g}\), and \(h_i = g^{x_i}\), with a non-negligible probability of error.

As a result, the knowledge of up to \(f\) shares does not help the attacker to predict any extra valid shares. Therefore, the data to which an attacker might have access is not sufficient to reconstruct the group key with a non-negligible probability of error.

## 7 Related Work

Much work has been done to formally verify fault-tolerance in distributed protocols. Some of these verifications deal with the Byzantine failure model [7], while others remain limited to the benign form [8]. A variety of automata formalisms has been adopted to specify such protocols.

Castro and Liskov [7] specified their Byzantine fault-tolerant replication algorithm using the I/O automata of Tuttle and Lynch [6]. They have manually proved their algorithm’s safety, but not its liveness, using invariant assertions and simulation relations. This work, although similar to our Byzantine agreement module, has never been mechanized in any theorem prover.
Kwiatkowska and Norman [9] analyzed the Asynchronous Binary Byzantine Agreement [19] (based on a concept similar to our key management module) using a combination of mechanical inductive proofs (for non-probabilistic properties) and finite state checks (probabilistic properties) plus one high-level manual proof. Our approach, too, takes advantage of the easiness and performance of the different earlier mentioned techniques to prove the overall Enclaves protocol.

Timed automata were also used to model the fault-tolerant protocols PAXOS [11] and Ensemble [14]. The authors assume a partially synchronous network and support only benign failures. This bears some similarities with our Enclaves verification in the sense that we assume some bounds on timing, but unlike the work in [11,14] we are dealing with the more subtle Byzantine kind of failure.

In [13], Archer et al. presented the formal verification of some distributed protocols using the Timed Automata Modeling Environment (TAME). TAME provides a set of theory templates to specify and prove I/O automata similar to those we use in our specification.

8 Conclusion and Future Work

This paper reports results about the formal verification of an Intrusion-Tolerant protocol. We experimented with an adaptively chosen combination of techniques based on the nature of the correctness arguments in each module of the protocol, the environment assumptions and the easiness of performing verification.

We believe to have achieved a promising success in verifying a complex protocol such as Enclaves. Nevertheless, our results could be improved further in various aspects. For instance, the feasibility of model checking is always limited to instances with a finite number of states, which may, in some cases, prevent from discovering security flaws in realistic implementations of the protocols. This can be improved by the use of rank functions [2]. We believe that using rank functions is a very efficient way to mechanically prove authentication properties and we are considering it among our future work plans.

Thanks to the high level of expressiveness of the Timed-Automata formalism, as well as the rich datatype package of PVS, we succeeded to formalize the Byzantine agreement module for any number of leaders, in a way that thoroughly captures the many subtleties on which the correctness arguments of Enclaves rely. We have proved the protocol to satisfy its requirements of Termination, Integrity and Proper Agreement. Yet, we have not proved the consistency of group membership when members leave the group. This is also among our future work. Finally, one promising direction for further development would be to perform the mathematical analysis mechanically in PVS. This requires the elaboration of some general purpose theories (e.g., probabilities) not yet available in PVS. The current specification can be further extended by widening the Byzantine faults capabilities and by introducing the joint cryptographic layers that have been abstracted away. Also results about an upper bound on Agreement establishment delays can be further investigated.
Acknowledgments. The formal specification and analysis of Enclaves benefited from fruitful discussions with Adriaan de Groot of the University of Nijmegen.

References

Abstract. We propose new, tractably (in some cases provably) efficient algorithmic methods for exact (sound and complete) parameterized reasoning about cache coherence protocols. For reasoning about general snoopy cache protocols, we introduce the guarded broadcast protocols model and show how an abstract history graph construction can be used to reason about safety properties for this framework. Although the worst case size of the abstract history graph can be exponential in the size of the transition diagram of the given protocol, the actual size is small for standard cache protocols as is evidenced by our experimental results. The framework can handle all 8 of the cache protocols in [19] as well as their split-transaction versions. We next identify a framework called initialized broadcast protocols suitable for reasoning about invalidation-based snoopy cache protocols and show how to reduce reasoning about such systems with an arbitrary number of caches to a system with at most 7 caches. This yields a provably polynomial time algorithm for the parameterized verification of invalidation based snoopy protocols. Our results apply to both safety and liveness properties. Finally, we present a methodology for reducing parameterized reasoning about directory based protocols to snoopy protocols, thus leveraging techniques developed for verifying snoopy protocols to directory based ones, which are typically much harder to reason about. We demonstrate by reducing reasoning about a directory based protocol suggested by German [17] to the ESI snoopy protocol, a modification of the MSI snoopy protocol.

1 Introduction

Cache protocols provide a vital buffer between the ever growing performance of processors and lagging memory speeds making them indispensable for applications such as shared memory multi-processors. Unfortunately, cache protocols are behaviorally complex. Ensuring their correct operation, in particular that they maintain the fundamental safety property of coherence so that different processes agree on their view of shared data items, can be subtle. The difficulty of the problem is often magnified as the number $n$ of coordinating caches increases. Moreover, it is highly desirable that a cache protocol be correct independent of the magnitude of $n$. There is thus great practical as well as theoretical interest in uniform parameterized reasoning about systems comprised of $n$
homogeneous cache protocols so as to ensure correctness for systems of all sizes \( n \). This general problem is known in the literature as the Parameterized Model Checking Problem (PMCP). It is, in general, algorithmically undecidable, but of great practical importance, which has led to many heuristics and algorithms for particular cases. In this paper, we present new, tractably (in some cases provably) efficient algorithmic methods for exact parameterized reasoning about cache coherence protocols.

First, for reasoning about general snoopy cache protocols, we introduce the guarded broadcast protocols model wherein processes coordinate using broadcast primitives plus boolean guards. A broadcast transmission corresponds to a cache putting a message on the bus; reception of such a message corresponds to snooping the bus and taking appropriate action. Boolean guards make it possible to model protocols (e.g., Illinois-MESI, Firefly, Dragon) that need to determine the presence or absence of the required memory block in other caches. We show how an abstract history graph construction can be used to reason about safety properties of guarded broadcasts. In the construction, a path \( x \) leading to global state \( s \) is represented as a tuple of the form \((a, A) \in S \times 2^S\), where \( S \) is the set of local states of the given cache protocol, that reflects not merely the local states present in \( s \) but also takes into account the local transitions that were fired along \( x \) to get to \( s \), viz., the history of \( s \) along \( x \). The extra historical information, that our construction stores, permits us to reason about safety properties for an arbitrary number of caches in an exact fashion as opposed to the standard abstract graph construction [24] that only takes into account the set of local states present in \( s \) and is thus sound but not guaranteed complete. We establish a path correspondence between concrete computations of the original system and paths in the abstract graph which also allows us to automatically generate error traces once an erroneous ‘abstract state’ is detected. In the worst case, the size of the abstract graph may be exponential in the size of the state diagram of the given cache protocol, thus enabling us to reason about the more expressive framework of guarded broadcast for the same worst case time complexity as ordered broadcasts. In practice, however, the abstract graph tends to be small as is documented by our empirical results.

Next we consider the PMCP for invalidation based snoopy protocols, viz., protocols that on a write operation invalidate the memory block being written to in all other caches of the given system [7]. We model such protocols using the new framework of initialized broadcast protocols. For this model, we consider the PMCP for formulae of the form \( \bigwedge_{i \neq j} A h(i, j) \) and \( \bigwedge_{i \neq j} E h(i, j) \), where \( h(i, j) \) is a LTL\( \backslash X \) formula over a pair of distinct processes. For such formulae, we show how to reduce reasoning for a system with an arbitrary number of processes to systems with at most a cutoff (in fact 7) number of processes. This yields a provably polynomial time algorithm (in the size of the state diagram of a single cache unit) for reasoning about the PMCP for a broad class of linear time properties of invalidation-based protocols, not just safety. Also the use of cutoffs has the important advantage that the large system with, say 100 caches, is very much like the small system with 7 caches. This provides a simple reduction from \( n \) to 7 processes that automatically caters to error recovery.

Finally, we consider the PMCP for directory-based protocols wherein information regarding cache states of individual memory blocks is stored in a centralized directory and all transaction regarding cache state lookup, invalidations, updates etc., take place
across a network. We use the observation that for most directory based protocols there exists a snoopy protocol with exactly the same states [7] and running essentially the same protocol except that the implementation of each snoopy broadcast transition is broken up into several steps. Since the executions of steps corresponding to different snoopy broadcasts can interleave among themselves, it makes directory-based protocols behaviorally more complex and thus seemingly harder to verify than their snoopy counterparts. However, we demonstrate, using a directory based protocol suggested by German [17], that since all transactions are serviced via the centralized directory, it leads to a serialization of steps of snoopy broadcasts in a way that there is limited overlap among steps corresponding to different snoopy broadcasts. We can then establish path correspondences between computation paths of directory based protocols and their snoopy counterparts thereby allowing us to reduce the PMCP for linear time properties from directory based protocols to snoopy ones. Thus techniques developed for reasoning about parameterized snoopy broadcasts can now be leveraged. As an example, we show how to reduce reasoning about this directory based protocol to the ESI snoopy protocol, a modification of the MSI protocol, which was verified using the abstract history graph construction in less than 0.01 secs.

The rest of the paper is organized as follows. We begin by introducing the system model in section 2. In section 3, we present the abstract history graph construction for verifying safety properties of guarded broadcast protocols while cutoff results for initialized broadcast protocols are given in section 4. In section 5, we demonstrate, using the protocol suggested by German, how to reduce reasoning about directory based protocols to snoopy protocols. Applications and experimental results are given in section 6 and we end with some concluding remarks in section 7.

2 The System Model

We consider systems of the form, \( U^n \), comprised of finite, but arbitrarily many, copies of a process template, \( U \), executing asynchronously with interleaving semantics. The template \( U \) is formally defined by the 4-tuple \( U = (S, \Sigma, R, i) \), where \( S \) is a finite, non-empty set of states; \( \Sigma \) is a finite set of labels including the internal transition label \( \tau \), and broadcast send and receive labels of the form \( l!! \) and \( l?? \), respectively; \( R \) is the transition relation; and \( i \) the initial state. Each transition of \( R \) is either an internal transition of the form \( a \xrightarrow{g:\tau} b \), a broadcast send of the form \( a \xrightarrow{g!!} b \), or a broadcast receive of the form \( a \xrightarrow{g??} b \), where \( g \) is a boolean guard.

We assume that receives are deterministic, viz., for each label \( l!! \) appearing in some broadcast send and for each state \( a \) in \( S \), there is a unique corresponding receive transition on \( l?? \) out of \( a \). The guard \( g \) labeling a transition \( tr \) of \( R \) is either the boolean expression \( true \) or the specialized conjunctive guard \( \land (i) \), or the specialized disjunctive guard \( \lor \neg (i) \), where \( i \) is the initial state of \( U \). We assume that the guard is \( true \) for receive transitions. In practice, the above mentioned guards suffice in modeling cache coherence protocols as each cache only needs to know whether another cache has the memory block it requires, expressed using the specialized disjunctive guard, or whether no other cache has it, expressed using the specialized conjunctive guard.
To capture block replacement behavior, we also require that templates be *initializable*.\(^1\) This means that from each state \(a\) of a protocol, there is an unguarded, internal transition of the form \(a \xrightarrow{\tau} i\). Such initializations model block replacement behavior, where a cache is non-deterministically pushed into its invalid state, irrespective of the current state of the block. For simplicity, re-initialization transitions and self-loop receptions are not drawn in state transition diagrams of cache protocols (cf. [7]).

We now introduce the following frameworks (a) *Initialized Broadcast Protocols* for dealing with invalidation based snoopy protocols, and (b) *Guarded Broadcast Protocols* for dealing with general snoopy cache protocols, by specifying the types of broadcast transition allowed. The two frameworks are incomparable in that each framework can model a protocol that the other cannot.

**Initialized Broadcast Protocols.** There are two major classes of snoopy cache protocols: *update based* and *invalidation based*. In update based protocols, e.g., *Dragon* and *Firefly*, whenever a shared location is written to by a processor, its value is updated in the caches of all other processors holding that memory block without invalidating the block. In contrast, with invalidation based protocols, e.g., *MESI* and *Berkeley*, on a write operation the memory block being written to is invalidated in all other caches [7]. In this paper, we model invalidation-based protocols using the framework of *Initialized Broadcast Protocols* wherein, each broadcast transition of \(U\) is either an (a) *i-flush*: transition \(a \xrightarrow{\Pi} b\) is called an i-flush iff from each state \(c\) of \(U\) there is the (unique) matching receive \(c \xrightarrow{\Pi} i\), or (b) *initialized-broadcast*: transition \(tr = a \xrightarrow{\Pi} b\) is an initialized broadcast send transition provided that \(a = i\) and every matching reception transition for \(tr\) is of the form \(c \xrightarrow{\Pi} d\), where either both \(c,d \neq i\) or both \(c,d = i\).

**Guarded Broadcasts.** In *Guarded Broadcasts*, each broadcast transition \(tr\) is of either of the two forms (a) *Flush*: Given state \(a\) of \(U\), transition \(b \xrightarrow{\Pi} c \in R\), where \(c \neq i\), is called an \(a\)-flush transition provided that there exists the matching receive transition \(i \xrightarrow{\Pi} i\) in \(R\) and for each state \(d \neq i\) of \(U\), there is a matching receive transition of the form \(d \xrightarrow{\Pi} a\) in \(R\); a flush transition is an \(a\)-flush for some \(a\). (b) *Push*: Transition \(a \xrightarrow{\Pi} b\), where \(b \neq i\), is a push transition provided that there exists the matching receive transitions \(i \xrightarrow{\Pi} i\), \(a \xrightarrow{\Pi} a\) and \(b \xrightarrow{\Pi} b\) in \(R\) and for every path \(c \xrightarrow{\Pi} d \xrightarrow{\Pi} e\), we have \(d = e\).

In either framework, given \(U\), the state transition diagram for \(U^n = (S^n, \Sigma, R^n, i^n)\), the system with \(n\) copies of \(U\), is based on interleaving semantics in the standard way. We write \(x.s \in U^n\) to mean that finite computation path \(x\) of \(U^n\) ends in global state \(s\). For local state \(a\), \(\text{num}(a, s)\), denotes the number of copies of local state \(a\) in \(s\).

The template \(U\) for a protocol, such as *MSI* (figure 1), is obtained from its state transition diagram through a simple abstraction, treating the behavior of the processors as purely nondeterministic. The transformation is straightforward, syntactic, and mechanical and tantamounts to relabeling the transitions of the given template to illustrate the link between broadcast sends and their matching receives.

\(^1\) Initializability is not needed for the results in section 3.1.
Safety Properties. For cache coherence protocols, we are typically interested in pairwise reachability, viz., given a pair \((a, b)\) of local states \(a\) and \(b\) of template \(U\), deciding whether for some \(n\), there exists a reachable global state of \(U^n\), with a process in each of the local states \(a\) and \(b\), viz., \(U^n \models \bigvee_{i \neq j} \text{EF}(a_i \land b_j)\). For instance, in the case of the MSI protocol, we are interested in showing that none of the pairs in the set \{\((M, M)\), \((M, S)\)\} is pairwise reachable.

3 Model Checking Guarded Broadcasts for Safety Properties

In a split-transaction bus, each transaction is split into two independent sub-transactions: a request transaction and a response transaction. Other transactions (or sub-transactions) are allowed to interfere (interleave) between them so that the bus can be used while response to the original request is being generated. The advantage is a more effective utilization of the bus. To deal with the non-atomic nature of bus transactions, extra states called transient states are introduced in the state transition diagram of split-transaction based protocols to indicate outstanding bus requests. This however makes snoopy split transaction bus protocols harder to reason about than their ‘non-split’ counterparts. We now show how to reason about guarded broadcasts, which can model all snoopy protocols in [19] and their split transaction bus versions, using an abstract history graph construction.

3.1 Protocols without Conjunctive Guards

In this section, we consider guarded broadcasts wherein template \(U\) does not have conjunctive guards; but guards of the form \(\text{true}\) or \(\bigvee \neg (i)\) are permitted. This allows us to handle the MSI, MOESI, MESI (not the Illinois version which is handled in the next section), Berkeley and N+1 protocols, and their split-transaction versions.

We motivate our technique with the help of an example. Consider the computation \(x = (I, I) \rightarrow (I, S)\) of the system, \(U^2\), comprised of two caches running the MSI protocol.
We exploit the observation that we can pump up the multiplicity of each of the local states $I, S$ to be greater than or equal to any arbitrary number $n$, by firing the transition $1 \xrightarrow{PrRd!!} S$ successively $n$ times as shown $(1, \ldots, 1) \xrightarrow{PrRd_1} \ldots \xrightarrow{PrRd_n} (S, \ldots, S, 1, \ldots, 1)$. On the other hand, consider the computation $y = (1, 1) \rightarrow (1, M)$. We cannot pump up the multiplicity of local state $M$, because in order for that to happen, we need to fire the transition $tr = 1 \xrightarrow{PrWp!!} M$ repeatedly. But a process firing $tr$, a flush transition, clobbers every other process by forcing it into its initial state. Thus we can have at most one copy of $M$ in any global state.

**Definition (representative).** Given template $U = (S, \Sigma, R, i)$, and a finite computation $x.s$ of $U^n$, we define $rep(x.s)$ to be the tuple $(a, A) \in S \times 2^S$, where, if no flush transition was fired along $x$, then $a = i$ and $A = \{s[j] | j \in [1 : n]\}$; and if $U_i$ is the process to last fire a flush transition along $x$, then $s[i] = a$ and $A = \{s[j] | j \in [1 : n] \land j \neq i\}$.

Then the above discussion can be formalized as the following *unbounded pumping property* implicitly shown in the proof of proposition 3.1. Let computation path $x.s \in U^n$ be such that $rep(x.s) = (a, A)$. Then given a positive integer $p$, there exists $y.t \in U^m$, for some $m$, such that $rep(y.t) = (a, A')$, where $A \subseteq A'$ and for each $a' \in A'$, $num(a', t) \geq p$. Thus we can represent $x.s$ by the tuple $(a, A') \in S \times 2^S$, representing a *formal* state with (at least) one copy of $a$ and arbitrarily many copies of each state in $A'$. Given template $U$, we now define the abstract history graph $A_U = (S_U, \mathcal{R}_U, (i, \{i\}))$, as a transition diagram over tuples in $S_U = S \times 2^S$ that captures the behaviour of a system instance of arbitrary size. To define the transition relation $\mathcal{R}_U$, given a tuple $(a, A)$ and an internal or a broadcast send transition $tr = c \rightarrow d$, we introduce the notion of the successor of $(a, A)$ via $tr$ as either the *1-successor*, which covers the scenario when a process in local state $a$ (that possibly) has multiplicity one fires $tr$; or the *2-successor* of $(a, A)$, covering the scenario when a process in one of the states in $A$ each of which can be thought of as having arbitrarily large multiplicity fires $tr$.

**Definition (1-successor).** Let $(a, A) \in S \times 2^S$ and let transition $tr = a \rightarrow b \in R$ labeled by guard $g$, be enabled in $(a, A)$, viz., if $g = \bigvee -i(i)$, then $\exists a' \in A : a' \neq i$. Then $succ_1((a, A), tr) = (b, B)$, where if $tr$ is an internal transition then $B = A$ and if $tr$ is a broadcast send transition then $B = \{b | \exists a' \in A : \exists a' \rightarrow b' \in R$ that is a matching receive for $tr \}$. 

**Definition (2-successor).** Let $(a, A) \in S \times 2^S$ and let transition $tr = b \rightarrow c \in R$, where $b \in A$, be such that if $tr$ is labeled by guard $g$ then it is enabled in $(a, A)$, viz., if $g = \bigvee -i(i)$, then for some $a' \in \{a\} \cup A: a' \neq i$. Then, $succ_2((a, A), tr)$, is defined as the tuple

- $(c, \{c\} \cup \{i\})$ if $tr$ is a $c'$-flush transition
- $(a, A \cup \{c\})$ if $tr$ is an internal transition. Note that since we had arbitrarily many copies of $b$ to start with so even after firing internal transition $tr$ we are guaranteed arbitrarily many processes in local state $b$ which is therefore not excluded from the second component of the resulting tuple.
Fig. 2. The abstract history graph for the MSI Cache Coherence Protocol

- \((d, B)\) if \(tr\) is a push broadcast transition, where \(a \rightarrow d\) is the (unique) matching receive for \(tr\) from \(a\) and \(B = \{c\} \cup \{b' \mid \exists a' \in A : \exists a' \rightarrow b' \in R\}\) that is a matching receive for \(tr\). Since we have arbitrarily many copies of \(b\) so in \(B\) we include the local state that results from firing the matching receive for \(tr\) from \(b\) which by definition of a push transition is \(b\) itself.

Definition (Abstract History Graph). Given template \(U = (S, \Sigma, R, i)\), the abstract history graph of \(U\), is defined as \(A_U = (S_U, R_U, (i, \{i\}))\), where \(S_U = S \times 2^S\) and \(R_U = \{(\sigma, (b, B)) \mid (b, B) = \text{succ}_1((a, A), tr)) \text{ or } (b, B) = \text{succ}_2((a, A), tr))\) for some internal or broadcast send transition \(tr\) of \(U\).

As an example, the abstract history graph for the MSI protocol is shown in figure 2. Self loops are omitted for the sake of simplicity. For convenience, we have labeled each transition of the graph by the label of the transition responsible for ‘firing’ it. We now establish a ‘path correspondence’ between finite computations of \(U^n\) and finite paths of \(A_U\) starting at \((i, \{i\})\). Let \((a, A) \geq (b, B)\) denote \(a = b\) and \(B \subseteq A\).

Proposition 3.1 (Covering Projection). For any \(n\) and any finite path \(x.s\) in \(U^n\), there exists a finite path \(y.t\) in \(A_U\) starting at \((i, \{i\})\) such that \(t \geq \text{rep}(x.s)\).

The tuple \(t\) not only stores the set of local states present in \(s\), but also the states that could potentially be present in a global state of a system with sufficiently many copies of \(U\) that results by firing (a stuttering) of the same sequence of transitions as were fired along \(x\) to get to \(s\). Thus \(t\) drags along some ‘history’ of computation \(x\) leading to \(s\) and thereby stores more information than \(\text{rep}(x.s)\).

Proposition 3.2 (Lifting). Let \(x\) be a path of \(A_U\) starting at \((i, \{i\})\) and leading to tuple \((a, A)\) of \(A_U\). Then, given \(p \geq 1\), there exists \(y.t \in U^n\), for some \(n\), such that \(\text{rep}(y.t) = (a, A)\) and \(t\) has at least \(p\) copies of each state in \(A\) plus a copy of \(a\).

Combining the previous two results, we have

Theorem 3.3 (Decidability Result). Pair \((a, b)\) \(\in S \times S\) is pairwise reachable iff there exists a path in \(A_U\) starting at \((i, \{i\})\) to a tuple of the form \((c, C)\) where either \(a = c\) and \(b \in C\); or \(b = c\) and \(a \in C\); or \(a \in C\) and \(b \in C\).
Thus we have reduced the problem of pairwise reachability for a pair of local states of a given template $U$ to the problem of reachability in $A_U$. In the worst case, the size of the abstract graph is $O(|U|^2)$, however, we need only consider the set of tuples reachable from $(i, \{i\})$ which, in practice, is much smaller (cf. section 6).

**Corollary 3.4.** The pairwise reachability problem for a pair of local states of a given template $U$ can be solved in time $O(|U|^2)$, where $|U|$ is the size of template $U$ as measured by the number of states and transitions in $U$.

### 3.2 Adding the Specialized Conjunctive Guard

To reason about systems wherein the templates are augmented with the specialized conjunctive guard along with the assumption of initializability, we modify the abstract history graph by adding for every tuple $(a, A)$, a transition of the form $(a, A) \rightarrow (a', \{i\})$, where either $a' = a$ or $a' \in A$, to $A_U$. Broadly speaking, the intuition behind the modification is that we can make the specialized conjunctive guard of a process evaluate to true starting at any global state by driving all the other processes into their respective initial states by making use of the initializing internal transition. Then, path correspondences as in section 3.1 can be shown and so, pairwise reachability can be decided in time $O(|U|^2)$, where $|U|$ is the size of $U$. Examples include the Illinois-MESI, Dragon and Firefly protocols and their split-transaction versions.

### 4 Reasoning about Invalidation Based Protocols Using Cutoffs

In this section, we consider the PMCP for formulae of the form $\bigwedge_{i \neq j} A h(i, j)$ and $\bigwedge_{i \neq j} E h(i, j)$, where $h(i, j)$ is a LTL $\setminus X$ formula over the local states of $U_i$ and $U_j$. We show how to reduce reasoning about a system with an arbitrary number of processes (caches) to a system with up to a cutoff (in fact 7) number of processes. This immediately yields a polynomial time algorithm for the PMCP at hand. The use of cutoffs has several advantages. First, the small system with a cutoff number of processes is identical to the large system, but with a fewer number of processes, and thus there is no need to construct, for instance, an abstract graph that may have a complex, non-obvious structure. Secondly, it automatically caters to error trace recovery. We later show how to reduce reasoning about LTL $\setminus X$ properties from directory-based to snoopy protocols for which these results can be leveraged.

We now present the cutoff result for properties of the form $\bigwedge_{i \neq j} E h(i, j)$. Since all processes in the systems we consider are copies of a single template $U$, they are all isomorphic up to renaming. Therefore symmetry considerations dictate that $U^n \models E h(1, 2)$ iff for each pair $i, j$, where $i \neq j, U^n \models E h(i, j)$. We shall therefore concentrate only on the formulae $A h(1, 2)$ and $E h(1, 2)$.

**Proposition 4.1 (Cutoff Result for Finite Paths).** For all $n \geq 7$, $U^n \models E_{\text{fin}} h(1, 2)$ iff $U^7 \models E_{\text{fin}} h(1, 2)$, where $E_{\text{fin}}$ quantities over finite paths only.

**Proof Sketch.** We present the main ideas behind the proof. The proof of the cutoff result proceeds by establishing a stuttering path correspondence between $U^n$, where $n \geq 7$,
and $U^7$, viz., constructing a finite stuttering computation path $y$ of $U^7$ corresponding to a given finite path $x$ of $U^n$ that preserves the local computation paths of processes $U_1$ and $U_2$, modulo stuttering, and vice versa.

($\Rightarrow$) Given a finite computation $x$ of $U^n$, where $n \geq 7$, we show how to construct a finite computation $y$ of $U^7$ that preserves the local computations of processes $U_1$ and $U_2$, modulo stuttering. Towards that end, we parse (the transitions of) $x$ as $x = N_0I_0...I_mN_{m+1}$, where $I_i$ is the $i$th global transition to be executed along $x$ that results by firing either an i-flush or a transition labeled with $\wedge(i)$. Thus $N_i$s are strings of transitions whereas $I_i$s are single transitions. The construction of $y$ proceeds by constructing for each subsequence $N_iI_i$, a corresponding subsequence $N'_iI'_i$ by projecting onto the local subsequences of $N_iI_i$ of a set $P_i$ of process indices defined below.

In defining $P_i$, there are two main considerations (a) every projected broadcast receive has a matching send, and (b) the specialized disjunctive guard is true for every projected local transition (the conjunctive guard $\wedge(i)$ is automatically true for all projected transitions). Clearly, we need to project on to process indices 1 and 2 as we have to preserve the local computation sequences of $U_1$ and $U_2$ modulo stuttering. Also, we need to project onto indices $p_3$ and $p_4$ of the processes responsible for firing the solitary global transitions in $I_{i-1}$ and $I_i$, respectively. Projection on to index $p_3$ ensures ‘continuity’ of the local computation of the process responsible for firing the global transition constituting $I'_{i-1}$, while projection on to index $p_4$ guarantees that every projected receive transition in $I'_i$ has a matching send in $I'_i$. Finally, let $N_i = x_{i'...i'+1}$ and let $a$ and $b$ be, respectively, the least and second least among all integers $c \in [0 : l]$ having the property that $x_{i'+c}[p] \neq i$, for some $p \in [1 : n] \setminus \{(1, 2) \cup \{p_3\} \cup \{p_4\}\}$. To ensure that the specialized disjunctive guard is true for the projected transitions, we include the indices $p_5$ and $p_6$ in $P_i$, where $x_{i'+a}[p_5] \neq i$ and $x_{i'+b}[p_6] \neq i$. Then, we let $P_i = \{1, 2\} \cup \{p_3\} \cup \{p_4\} \cup \{p_5\} \cup \{p_6\}$. A seventh process with index $p_7$, say, is required to ensure that in $N'_i$, every projected initialized broadcast receive transition has a matching broadcast send. Since, by definition, an initialized broadcast send is fired only from the initial state, we use this process, which we (try to) maintain in its initial state $i$, to fire the required send transition and then ‘recycle’ it by firing the initializing internal transition to make it transit back to $i$. The computation $y$, then results by ‘sewing’ up the subsequences $N'_iI'_i$ appropriately, in the same relative order as the original subsequences $N_iI_i$ along $x$. Note that the sets $P_i$ may be different for different $i$; however, since all processes in our system are isomorphic up to renaming, for each $i$, $U^7$ can mimic the local sub-computations of $N'_iI'_i$.

($\Leftarrow$) The lifting part is simpler. Given a computation $y$ of $U^7$, we can construct a valid computation $x$ of $U^n$, where $n \geq 7$, by letting processes $U_1, ..., U_7$ execute exactly the same local computations as in $y$ while the rest of the processes just stutter in their initial states without executing any non-receive transition at all (all receives from $l$ loop back to $i$).

The proof technique of proposition 4.1, extends to the case where we consider full paths (and full paths under the assumption of unconditional fairness). We then have the following.

**Proposition 4.2 (Cutoff Result for Full Paths).** For all $n \geq 7$, $U^n \models \mathsf{Eh}(1, 2)$ iff $U^7 \models \mathsf{Eh}(1, 2)$, where $h(i, j)$ is a LTL/X formula over processes $U_i$ and $U_j$. 


As a corollary to propositions 4.1 and 4.2, we have the following.

**Proposition 4.3 (Efficient Decidability Result).** For initialized broadcast protocols, the PMCP for formulae of the types \( \bigwedge_{i \neq j} E_{\text{fin}} h(i, j) \), \( \bigwedge_{i \neq j} A_{\text{fin}} h(i, j) \), \( \bigwedge_{i \neq j} E h(i, j) \) and \( \bigwedge_{i \neq j} A h(i, j) \) is decidable in polynomial time in the size of the template \( U \) specifying the parameterized family.

## 5 Reducing PMCP for Directory Based to Snoopy Protocols

In this section, we present a methodology for reducing the PMCP for (stuttering insensitive) LTL\( \setminus X \) properties for directory based to snoopy cache protocols thereby enabling us to leverage the techniques developed for snoopy protocols. We exploit the observation that with most directory based protocols one can associate a snoopy protocol with exactly the same local states [7] and executing essentially the same protocol except that the implementation of each snoopy broadcast transition is broken down into several smaller steps that execute asynchronously. We call such transitions *distributed broadcasts*. The interleavings of the steps of different distributed broadcasts makes directory based protocols behaviorally more complex than their snoopy counterparts and thus seemingly harder to reason about. However, the central directory can service only one distributed broadcast at a time, and so in a given computation, \( x \), of the system, \( U^n_{\text{Directory}} \), comprised of \( n \) caches running the directory based protocol Directory, there is a unique serial order on the way distributed broadcasts are serviced along \( x \). This allows us to construct a computation \( y \) of \( U^n_{\text{Snoop}} \), where Snoop is the snoopy protocol corresponding to Directory, by letting the snoopy broadcast transitions fire in the same linear order as their distributed counterparts were serviced along \( x \). This path correspondence allows us to reduce reasoning about linear time properties from directory based to snoopy based protocols. We demonstrate our technique using a directory based protocol suggested by German [17], which we denote by DIR.

**Reasoning about the DIR Directory Based Protocol.** In the DIR protocol, each cache is represented as a client process with the directory being represented as the Home process. The variables used in DIR are given below.

```plaintext
  type message = {empty, req_shared, req_exclusive, invalidate, invalidate_ack, grant_shared, grant_exclusive}
  type cache_state = {invalid, shared, exclusive}
  channel11, channel12, channel13 : array[1:n] of message
  home_sharer_list, home_invalidate_list: array[1:n] of boolean
  home-exclusive-granted: boolean
  home_current_command: message
  home_current_client: [1:n]
  cache: array[1:n] of cache_state
```

Each client has three possible local states, viz., invalid, shared and exclusive, represented by the variable cache_state. Communication between client[i], the process representing the \( i \)th cache, and Home, the process representing the directory, takes place via the following variables that are shared pairwise between client[i] and Home.
The ESI Snoopy Cache Protocol.

The template for the ESI protocol is defined as $U = (\{I, S, E\}, \{PrRd, PrWr\}, R, I)$, where the transition relation $R$ consists of the broadcast send transition $I \xrightarrow{PrRd!!} S$ with the matching receives $E \xrightarrow{PrRd??} I, S \xrightarrow{PrRd??} S$.
and $I^{PrRd!!} \rightarrow I$; and the I-flush broadcast $I^{PrWr!!} \rightarrow M$. The symbols E, S and I denote, respectively, the exclusive, shared and invalid states.

**Establishing the Stuttering Path Correspondence.** Let $U^{n}_{\text{DIR}}$ represent a system with $n$ clients running the directory based protocol $\text{DIR}$. We begin by showing how the variables used in the $\text{DIR}$ protocol impose a relative ordering on the execution of the transitions of the protocol. For transitions (numbered) $j, k$, we say that $j$ pre-empt $k$, denoted by $jPk$, to denote the fact that along any global computation of $U^{n}_{\text{DIR}}$, between any two firings of $k$ (possibly by different clients), there must be at least one firing of $j$. We write $(j + k)Pm$ to mean that either $jPm$ or $kPm$, and $j_{0}P...P_{j}{k}$ to mean that for all $l \in [1 : k]$, $j_{l-1}P_{j}$. For transition $j$ and index $i \in [1 : n]$, we write $j_{i}$ to indicate that the execution of transition $j$ modifies the local variables of $U_{i}$, the process representing the $i$th client, or the communication variables, $\text{channel11}[i], \text{channel12.4}[i]$ and $\text{channel13}[i]$, shared pairwise between $U_{i}$ and $\text{Home}$.

We first show that $(9 + 10)P3$. Note that variable $\text{home.current.command}$ must be set to empty for transition 3 to be enabled and that can be done only by firing transitions 9 or 10. Thus one of 9 or 10 has to be fired for 3 to be fired (except for the first time).

Also, every time 3 is fired it sets home_current_command to a non-empty value thus disabling itself and so again one of 9 or 10 has to be fired for 3 to fire again. Similarly, the firing of both 3 and (9+10) the firing of transitions 3 and (9+10). We call these transitions, including transitions 3 and (9+10), the body of the distributed transition. The crucial observation is that the bodies of different distributed transitions do not overlap as once 3 is executed by a process, one...
of 9 or 10 has to be executed by the same process for 3 to be executed again possibly, by a different process, to begin executing the body of another transition. Thus given computation $x$ of $U^n_{DIR}$, we can arrange all the distributed broadcast transitions fired along $x$ in a sequence $d-tr_0, d-tr_1, ...$ based on the order in which their bodies were executed. We say that a distributed transition $d-tr$ is fired by process $U_k$ of $U^n_{DIR}$ iff the entry transition of $d-tr$ sets the value of home_current_client to $k$. Let transition $d-tr_j$ be fired by process $U_{ij}$ of $U^n_{DIR}$. Let $y$ be the computation sequence of $U^n_{ESI}$ that results by firing the snoopy broadcasts $tr_0, tr_1, ...$ in the order listed with transition $tr_j$ being fired by process $U_{ij}$ of $U^n_{ESI}$. Conversely, given a computation path $y$ of $U^n_{ESI}$, we can construct a computation path $x$ of $U^n_{DIR}$ by replacing the firing of each snoopy broadcast $tr_j$ by process $U_{ij}$ of $U^n_{ESI}$ by the firing of all steps of $d-tr_j$ successively back to back by process $U_{ij}$ of $U^n_{DIR}$. This establishes the desired path correspondence.

For the DIR protocol, we are required to verify that in any global state $u$ of $U^n_{DIR}$, $(u[1] \neq u[2] \wedge u[1] = \text{exclusive}) \Rightarrow u[2] = \text{invalid}$. Towards that end, it suffices to check the following: $\forall n : U^n_{DIR} \models \neg EF(a_1 \wedge b_2)$, where $(a, b) \in \{(\text{exclusive}, \text{exclusive}), (\text{exclusive}, \text{shared})\}$, viz., none of the pairs (exclusive, exclusive), (exclusive, shared) of the DIR protocol is pairwise reachable. The next result reduces reasoning about pairwise reachability for the DIR to the ESI protocol.

**Proposition 5.1 (Reduction for Safety).** For $a, b \neq \text{invalid}, U^n_{DIR} \models EF(a_1 \wedge b_2)$ iff $U^n_{ESI} \models EF(a_1 \wedge b_2)$.

Thus it suffices to check that none of the pairs (E, E), (E, S) is pairwise reachable for the ESI protocol. This took 0.01 secs using the abstract history graph technique, and 0.02 secs using the cutoff technique.

The above technique of establishing stuttering path correspondences also works, in general, for LTL $\setminus X$ formulae. In [4], it was shown that the property $A(G(\text{channel11}[1] = \text{request_shared} \Rightarrow F(\text{channel12_4[1]} = \text{grant_shared})))$, viz., once a block is requested in the shared state by a cache then it is eventually granted shared access, fails. However, if we assume unconditional fairness, viz., every process fires infinitely often, then the property holds. We now modify the ESI protocol by introducing the intermediate local states $rS$ and $rE$, standing for request_shared and request_exclusive, respectively. Before executing a broadcast send to the exclusive (shared) state, we first transit via an internal transition to $rE$ ($rS$) and then fire the broadcast send labeled with PrWr!! ($PrRd!!$) to transit to the exclusive (shared) state. Then the above liveness property can be reduced to the PMCP for $A(G(rS \Rightarrow F(S)))$ for the modified ESI. This property has a cutoff of 7 and was verified to hold under assumption of unconditional fairness in 0.02 secs$^2$. Note that the property fails if we do not assume fairness. In that case an error trace is automatically generated for the 7 process instance. No manual effort as in [4] is required to validate the erroneous path in the abstraction, an advantage of using cutoffs.

$^2$ Technically, we verify the LTL $\setminus X$ expressible assertion $fair \Rightarrow G(rS \Rightarrow F(S))$. 

---

**Exact and Efficient Verification of Parameterized Cache Coherence Protocols**

259
6 Applications and Experimental Results

We consider PMCP for all the snoop based cache protocols presented in [19] (MSI, MESI, Illinois-MESI, MOESI, Berkeley, Synapse N+1, Dragon, Firefly) and the split-transaction version of the MESI protocol. Using the abstract history graph, each of the above protocols was verified in at most 0.01 secs. Although in the worst case the number of reachable abstract states in the modified abstract history graph for template $U = (S, R, S, i)$ could be as large as $|S|2^{|S|}$, in practice it typically turns out to be much smaller. For instance in the MESI protocol, the number of reachable abstract states was 6, against a worst case possibility of $4 \times 2^4 = 64$ states. In conclusion, the abstract history graph construction seems to work well in practice. In fact, it seems to work even better than the polynomial time cutoff method which too is very efficient requiring only a fraction of a second to verify each invalidation based protocol. This, however, may be due to the fact that whereas the abstract history graph was built directly from the description of the protocol using a separately written code, for the cutoff method we used SMV, possibly resulting in extra overheads from compilation of the protocol specifications, building BDDs etc. The experiments were carried out on a machine with a 797MHz Intel Pentium III processor and 256 Mb RAM.

<table>
<thead>
<tr>
<th>Protocol</th>
<th>Abstract History Graph</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td># of Abstract States</td>
</tr>
<tr>
<td></td>
<td>user time (secs.)</td>
</tr>
<tr>
<td>MSI</td>
<td>5</td>
</tr>
<tr>
<td>MESI</td>
<td>6</td>
</tr>
<tr>
<td>Illinois</td>
<td>6</td>
</tr>
<tr>
<td>MOESI</td>
<td>7</td>
</tr>
<tr>
<td>N+1</td>
<td>5</td>
</tr>
<tr>
<td>Berkeley</td>
<td>5</td>
</tr>
<tr>
<td>Firefly</td>
<td>6</td>
</tr>
<tr>
<td>Dragon</td>
<td>8</td>
</tr>
<tr>
<td>Split MESI</td>
<td>82</td>
</tr>
</tbody>
</table>

7 Concluding Remarks

The generally undecidable PMCP has received a good deal of attention in the literature. A number of interesting proposals have been put forth, and successfully applied to certain examples (e.g, [2,3,5,20]). Most of these works, however, suffer from the drawbacks of being either only partially automated or being sound but not guaranteed complete. Much human ingenuity may be required to develop, e.g., network invariants; the method may not terminate; the complexity may be intractably high; and the underlying abstraction may only be conservative, rather than exact.\(^3\)

Similar limitations apply to prior work on PMCP for cache protocols. Some concrete examples of verification of cache protocols can be found in [6,22]. Pong and Dubois [24] described general methods that were sound but not complete, as they were based on

\(^3\) However for frameworks that handle specialized domains, sound and complete, fully automatic and, in some cases, efficient decision procedures can be given ([9,10,13,15,23]).
conservative, inexact abstractions. In [16], it was shown that the PMCP for safety over broadcast protocols [14] is decidable using the general backward reachability procedure of [1]. In [21], Maidl, using a proof tree based construction, shows decidability of the PMCP for a broad class of systems including broadcast protocols, but the decision procedure is not known to be primitive recursive. Moreover [14,16,21] do not report experimental results for cache protocols. In [8], Delzanno uses arithmetical constraints to model global states of systems with many identical caches. His method uses invariant checking via backward reachability analysis of [1] and provides a broad framework for reasoning about cache coherence protocols but his procedure does not terminate on some examples. More recently, a decision procedure based on a modification of the backward reachability algorithm that guarantees termination for all snoopy cache protocols has been given in [12]. However, the backward reachability algorithm of [1] that [8,12,16], make use of, although general, suffers from the handicap that the best known bound for its running time is not known to be primitive recursive. Furthermore, this technique does not provide a way to generate error traces when a bug is detected. An elegant cutoff method that can verify the DIR protocol was given in [23], but it was sound and not complete and worked only for safety properties. Also in [4], a broad technique was proposed for the verification of WS1S systems that can handle the DIR protocol as an example, but again the resulting technique was sound but not complete.

In this paper, we made three distinct contributions to the parameterized model checking of cache coherence protocols.

First, to reason about general snoopy broadcast protocols, we introduced the framework of Guarded Broadcast Protocols. It is both a generalization and a significant simplification of ordered broadcast protocols [11] which required identification of a pre-order on the set of local states of the protocol. The extra transient states found in split-transaction bus protocols prevent the imposition of the necessary pre-order. Our new guarded protocol framework eliminates the need to impose a pre-order on protocol states and thereby caters readily for split transactions. This framework is broadly applicable, handling safety properties, and catering for all 8 snoopy protocols in Handy [19], even in their split transaction formulations.

Second, we presented the framework of Initialized Broadcast Protocols, establishing provably efficient reasoning about safety and liveness of invalidation based snoopy protocols. We showed that a system with an arbitrary number of caches could be reduced to a system with at most 7 caches. This yields a fully automatic and provably efficient polynomial time algorithm for verifying parameterized invalidation based snoopy cache protocols. Cutoffs have the added important advantage that the small system with 7 caches is a precise replica of large system with \( n \) caches, up to size. This not only makes the reduction simple but also caters automatically for error recovery as there is an error in a large system if and only if there is one in the system with the cutoff number of processes.

Third and last, we described a method for reducing parameterized reasoning about directory based protocols to reasoning about snoopy protocols. We have illustrated the method using the DIR directory based protocol as an example. We then leverage the above cutoff and abstract history graph techniques developed for snoopy protocols to reason about linear time properties of parameterized directory based protocols, which typically are much harder to reason about, in an exact fashion.
References

17. S.M. German. Private communication.
Design and Implementation of an Abstract Interpreter for VHDL

Charles Hymans

STIX, École Polytechnique, 91128 Palaiseau, France
charles.hymans@polytechnique.fr

Abstract. We describe the design by abstract interpretation of a static analysis for the popular hardware language VHDL. From a VHDL description, the analysis computes a superset of the states reachable during any simulation run. This information is useful in the validation of safety properties of hardware components. The construction of the analysis is based on the formal definition of a semantics for VHDL. Soundness with respect to this semantics is shown. Various techniques allow a compromise between the desired accuracy and the cost of the final algorithm. We present a few examples and detail the essential implementation choices.

1 Introduction

We present the design of a static analysis for VHDL. It computes a superset of the states that may be encountered during any simulation run of a description. Following the methodology of abstract interpretation [2], we first define the semantics of a subset of VHDL. A sound static analysis is then obtained from this formalization by abstraction. We make our construction generic in the underlying symbolic domain used to represent the possible values that signals may take. That way, it is possible to plug in various back-ends so as to attain the best compromise between precision and efficiency. This work extends [5]. Arrays, variables, for-loops and until clause in wait statements were previously not considered. A finer abstraction of the state-space, which keeps track of the history of computation, is proposed. All implementation details are new.

Motivating example. We consider a component which performs the multiplication of an input matrix by a constant matrix. The input matrix is fed one coefficient at a time through a wire DI on rising edges of the clock CLK. New coefficients are signaled by setting a flag DSI high and need not be given in consecutive cycles. Similarly, the result is produced on DO while the flag DSO is set. We write a test-bench made up of the input generator of Fig. 1 and the checker of Fig. 2. The generator stimulates the design to do the multiplication of a unique matrix INPUT. It does this an unbounded number of times and waits arbitrarily long between each coefficient. The checker simply asserts the values read on DO when DSO is high are the correct results of the multiplication. Our prototype implementation is able to determine, without any human intervention, that the
initial INPUT := (1,1,0,1);
process
for I in 0 to 3 loop
    wait on CLK until CLK;
    DSI <= FALSE;
    while random loop
        wait on CLK until CLK;
    end loop;
    DSI <= TRUE; DI <= INPUT(I);
end loop;
end process;

Fig. 1. Input driver

initial RESULT := (-4,17,-9,10);
process
for J in 0 to 3 loop
    wait on CLK until CLK;
    while (not DSO) loop
        wait on CLK until CLK;
    end loop;
    assert DO = RESULT(J);
end loop;
end process;

Fig. 2. Output Checker

Fig. 3. Syntax

assertion DO = RESULT(J) is never broken. Note that this is not practicable by conventional simulation.

2 An Operational Semantics for VHDL

To be able to reason about VHDL descriptions, we first formally define their semantics. Formalizations close to ours can be found in [3,4,6]. We suppose an elaboration phase – similar to the one presented in the standard [1] – compiles the description into a program of the kernel language of Fig. 3. Programs manipulate integers, booleans and statically allocated arrays. Note we deliberately ban delayed signal assignments (signal assignments with an after clause). They do not appear in the designs we wish to validate and add much complexity since, in their presence, the precise layout of the memory used by a program is not known statically.

We express the execution of a program P as a small-step operational semantics. Program statements \( C \) are uniquely tagged with labels \( l \) that are taken
\[
\begin{align*}
\text{var} & \quad \ell_v := e \quad \rho \vdash e \Rightarrow v \\
& \quad (l, \rho) \to (\text{next}(l), \rho[\varnothing \leftarrow v]) \\
\text{suspend} & \quad \ell\text{wait on } W \text{ until } b \text{ for } t \\
& \quad (l, \rho) \to (c, \rho) \\
\text{while } b \text{ do } & \quad \ell C; P \text{ end} \\
\rho \vdash b & \Rightarrow \text{true} \\
& \quad (l, \rho) \to (l', \rho) \\
\text{enter} & \quad \ell \text{while } b \text{ do } P \text{ end} \\
\rho \vdash b & \Rightarrow \text{false} \\
& \quad (l, \rho) \to (\text{next}(l), \rho)
\end{align*}
\]

Fig. 4. Sequential execution

\[
\begin{align*}
\Pi & \quad \forall j < i : c_j \notin \mathcal{L} \quad c_i \in \mathcal{L} \quad (c_i, \rho) \vdash (c_i', \rho') \\
& \quad (c, \rho) \to ((c_1, \ldots, c_i', \ldots, c_n), \rho') \\
\forall j & : c_j = (l_j, W_j, b_j, t_j) \quad \rho' = \text{update}(\rho) \\
& \quad \exists j : \text{wake}(W_j, b_j, \rho, \rho') \\
& \quad c_i' = \begin{cases} l_i & \text{if wake}(W_i, b_i, \rho, \rho') \\
& c_i & \text{otherwise} \end{cases} \\
\Delta & \quad (c, \rho) \to (c', \rho') \\
\forall j & : c_j = (l_j, W_j, b_j, t_j) \\
& \quad \rho' = \text{update}(\rho) \\
& \quad \forall j : \neg \text{wake}(W_j, b_j, \rho, \rho') \\
& \quad \exists j : t_j \neq \infty \\
& \quad t = \min\{t_i \neq \infty\} \\
& \quad c_i' = \begin{cases} l_i & \text{if } t_i = t \\
& (l_i, W_i, b_i, t_i - t) & \text{if } t_i \neq \infty \\
& c_i & \text{otherwise} \end{cases}
\end{align*}
\]

Fig. 5. Simulation algorithm

from a set \(\mathcal{L}\). The label of the unique statement which follows \(\ell C\) in the control flow graph of the enclosing process is fetched with \(\text{next}(l)\). The point of execution in a process is determined by the label of the statement that is to be executed next. The control point of a suspended process is augmented with a list of signals \(W\), a condition \(b\) and a duration \(t\). The duration is either a strictly positive integer or \(\infty\) to indicate the absence of a timeout. A global environment \(\rho\) stores values of variables and signals. We denote by \(\mathbb{X}\) the location where the future value of a signal \(x\) lies. We impose the syntactic restriction that no signal is assigned by more than one process. Hence, it is sufficient to remember only one future value for every signal.

An expression \(e\) evaluates to a value \(v\) in an environment \(\rho\), which we express by the judgment \(\rho \vdash e \Rightarrow v\). The meaning of expressions is defined by structural induction in the classical way. Figure 4 shows the sequential execution of an individual process. Paraphrasing the sig rule: the right-hand side expression is evaluated in the current environment; the resulting value is then scheduled for
the next cycle at location \(\tau\); and control is transferred to the next statement. The three rules of Fig. 5 are enough to completely characterize the simulation algorithm of VHDL. Processes are run concurrently as long as possible thanks to the first rule. Once all processes are suspended, the global environment is updated so that signal assignments encountered during the last simulation cycle take effect:

\[
update(\rho)(x) = \begin{cases} 
\rho(\overline{x}) & \text{if } x \text{ is a signal,} \\
\rho(x) & \text{otherwise.} 
\end{cases}
\]

The \(\Delta\) rule reactivates any process for which the value of some signal in the sensitivity list \(W\) was changed during the last cycle, and the condition \(b\) is met:

\[
\text{wake}(W, b, \rho, \rho') = (\exists x \in W : \rho(x) \neq \rho'(x)) \land (\rho' \vdash b \implies \text{true}).
\]

Finally, if no process activity can be resumed by \(\Delta\) then the final rule advances simulation time by the smallest timeout.

3 The Abstract Interpreter

The set \(\mathcal{O}\) of all prefixes of execution traces from some initial state \(s_0\) can be constructively expressed as the least fixpoint of the continuous operator:

\[
\mathcal{F}(X) = \{s_0\} \cup \{s_0 \ldots s_k s_{k+1} \mid \exists s_0 \ldots s_k \in X : s_k \rightarrow s_{k+1}\}.
\]

This fixpoint is not effectively computable or even finitely representable. So we adopt the methodology of abstract interpretation [2] to obtain a decidable approximation. We proceed in two steps.

**Generic Abstract Domain.** We build an abstract domain to encode sets of traces. We collect environments and group them according to the history of computations that led to their creation. Collections of environments are further abstracted thanks to an abstract numerical domain \(\mathcal{N}\). Numerical domains provide finite descriptions for sets of tuples of scalar values. We call \(\gamma_{\mathcal{N}}\) the concretization function on the numerical domain. The way environments are grouped depends on a function \(\kappa\) which creates a token \(h\) from an execution trace. Formally, a collection of abstract environments \(X\) represents the traces:

\[
\gamma(X) = \{s_0 \ldots s_k \mid h = \kappa(s_0 \ldots s_k) \land (c, \rho) = s_k \land R = X(c, h) \land \rho \in \gamma_{\mathcal{N}}(R)\}.
\]

Both the numerical domain \(\mathcal{N}\) and the grouping function \(\kappa\) are left as parameters of our construction. Hence, we have two orthogonal means to adjust the precision and efficiency of our analyzer.
Finally, our algorithm consists in computing the least fixpoint of the following monotonic function:

\[ \mathbb{F}(X)(c', h') = X_0(c', h') \uplus \bigcup \{ R' \mid \exists (c, h) : R = X(c, h) \land (c, h, R) \leadsto (c', h', R') \} . \]
The static analysis is correct. Indeed, thanks to the properties enforced on the
basic numerical operators, one can prove that we have:

\[ \mathcal{O} \subseteq \gamma(\text{lfp } \mathbb{F}) . \]

**Implementation** We implemented the abstract interpreter in OCaml. Execu-
tions that went through distinct branches of if-statements are distinguished
and for-loops are unrolled. For the back-end, we chose the domain of constants
which we encode with balanced binary trees. The major advantage is to improve
sharing, which in turn speeds up many operations. All abstract environments
computed during the analysis are placed in a hashtable. It is not necessary to
keep them all in memory, rather we store only the ones at the entry point of
loops. Once the fixpoint has been reached, we can rebuild the missing environ-
ments in a single last pass. This dramatically reduces memory consumption. The
fixpoint is computed with a standard worklist algorithm. The analysis was able
to automatically verify various instances of the introductory example.

4 Conclusion

We have shown the staged design of an abstract interpreter for a subset of VHDL.
It is based on a formalization of the simulation algorithm. As such, it has the
ability to handle non-synthesizable descriptions. This permits its early integra-
tion in the design cycle. With a first implementation, we successfully verified
non-trivial properties on a VHDL component. We hope to have demonstrated
the adequacy of the approach as an automatic means to validate fairly complex
safety properties. We were careful to separate concerns as much as possible so
that our analyzer can be easily improved by local modifications. In fact, we can
now focus on more efficient numerical domains tailored to prove specific classes
of properties. We need no longer concern ourselves with the idiosyncrasies of the
VHDL dialect.

**Acknowledgments.** We are grateful to P. Cousot, R. Cousot, F. Logozzo, X.
Rival and E. Upton for help, comments and discussions.

**References**

2. P. Cousot and R. Cousot. Abstract interpretation and application to logic pro-
   Journal of Logic Programming has mistakenly published the unreadable galley
   proof. For a correct version of this paper, see http://www.di.ens.fr/~cousot/).
3. K. Goossens. Reasoning about VHDL using operational and observational seman-
tics. In *Correct Hardware Design and Verification Methods*, volume 987 of *Lecture


A Programming Language Based Analysis of Operand Forwarding

Lennart Beringer

Laboratory for Foundations of Computer Science
School of Informatics, University of Edinburgh
Mayfield Road, Edinburgh EH3 9JZ, UK

Abstract. We outline a programming language based analysis of forwarding. Abstractions of processor behaviour are modelled as operational semantics for a language which captures the hardware resources for forwarding explicitly. Unsafe usage of the forwarding mechanism is eliminated by static semantics. These type systems may be linked to static program analysis frameworks but also characterise the instruction stream entering the datapath from other processor components.

1 Introduction

The forwarding (register-bypassing) of operands is a technique implemented in many modern microprocessors [7][5]. The formal correctness of forwarding mechanisms such as Tomasulo’s algorithm [16], and their interaction with other elements of processor architecture have been widely studied [1][14][10]. These verification efforts follow the well-known approach of relating processor implementations to the instruction set architecture [4][9], using model checking and theorem proving. In this paper, we present a more conceptual analysis of forwarding using an abstract model of computation comprising named operand queues, registers and functional units. We demonstrate that using programming language notation yields an analysis which separates structural and implementational aspects of forwarding. As a consequence, we may reason about constraints on the allocation of operand queues to operands which are imposed by functionality considerations without committing ourselves to a particular allocation algorithm.

Elements of a Programming Language Based Approach. Our approach is programming language based as it builds on the methodology of modern programming language design, in particular on the separation into static and dynamic semantics. We use

- syntax for reflecting architectural entities or the fine-grained structure of instructions. We present a language in which the forwarding resources are explicit: the names of operand queues appear in the syntax of instructions, much like those of registers.
- structural operational semantics (SOS, [13]) for defining processor behaviour. By giving a language several operational semantics one may compare processor behaviour at various levels of abstraction. In this paper, we outline a processor model for sequential execution (similar to the ISA), while referring the reader to [3] for models for distributed (out-of-order) execution and execution with finite operand queues.
– static semantics for formally expressing properties of the instruction stream which cannot easily be captured syntactically. In particular we employ type systems based on linear logic to characterise properties of the allocation of operand queues.

As program properties (static semantics) and processor behaviour (dynamic semantics) are tied together by syntax, proof techniques which are guided by the syntactic structure may be used for reasoning about properties of instruction streams in a particular processor model. The allocation of operand queues is thus validated according to the slogan *well typed programs can’t go wrong:* an instruction stream which has been accepted by the static semantics will not experience specific runtime hazards. The main such technique is *structural induction*, where the proof of a property of a certain phrase relies on related properties of syntactically constituent phrases.

Static semantics also allows system properties to be related to compile-time analysis. A system designer may hence explore whether properties of the instruction stream can better be ensured by the hardware or the compiler. For example, the decision whether a value is forwarded is often made in the control unit. Based on the static semantics, we present an alternative analysis using dataflow equations. This allows us to study the amount of forwardable values in application programs, but also indicates the type of analysis a hardware implementation must perform in order to exploit these forwarding opportunities.

**Related Work.** To our knowledge, no structural analysis of forwarding has been published yet, despite the plethora of verification exercises of processors with register bypassing.

The application of flat term rewriting systems (TRS) for describing and relating processor models is advocated in [2]. This formalism captures the structure of the processor in a similar way as our approach, and aspects of our computational model may be seen as a refinement of [2]’s substitution-based communication of operands. However, the operational model is not complemented by static semantics, hence program and processor may not be treated in combination and no link to compile time analysis may be made. On the other hand, [8] reports how hardware implementations may be directly generated from the TRS descriptions. We have not attempted this task but are confident that one could develop a corresponding translation from SOS-based descriptions.

Mountjoy et. al. [11] present a SOS description of transport-triggered architectures where the structure of the semantics follows the structure of the architecture. A single dynamic semantics is given which models the (synchronous) execution of a family of move-instructions in two phases. The authors observe that the legality of code relies on the ability of the compiler to structure code in a way which avoids output-conflicts. While static semantics is mentioned as a means to enforce this property, no details are given in [11] and the topic was apparently not pursued any further.

This paper represents a brief summary of the author’s PhD thesis [3]. The reader is referred to [3] for a more in-depth presentation which includes formal proofs, a description of the experimental results, and more motivating discussion.
2 Syntax and Operational Semantics

We consider a simplified model where a processor core consists of a number of typed functional units which are located in parallel to each other and are fed by instruction queues. Operands are communicated through registers or operand queues, and the syntax of our language treats both mechanisms identically. We have instructions like $\text{add } o_1\ op_2\ op_3$,$\text{dupl}_u\ op_1\ op_2\ op_3$ and $\text{if } o_1\ n_1\ n_2$ where the $o_i$ denote registers or operand queues, $fu$ represents a functional unit and the $n_i$ denote program labels.

Specific processor models are defined by giving dynamic semantics for the language. The sequential model of operation is defined by a relation $C \xrightarrow{t} D$ between configurations $C$ and $D$ and an instruction sequence $t$. Configurations consist of a register bank, a component describing the content of all operand queues (technically a map from operand queue identifiers to sequences of values), and a memory component. On arrival at a functional unit, an instruction awaits its operands in the queues or registers as indicated in its opcode. Whenever its functional unit becomes available, the instruction executes which involves consuming operands from (and sending results to) registers and operand queues. The relation $C \xrightarrow{t} D$ is defined along the syntactic structure. Rules for individual instructions employ micro-instructions for read and write access to operand queues, registers and memory. For example, executing the sequence $[1]\text{ldc}\ 4\ q_1\ [2]\text{dupl}_u\ q_1\ q_1\ q_1\ [3]\text{add}\ q_1\ q_2\ q_2$ in an initially empty state leads to $q_2$ containing the single value 8 and $q_1$ being empty.

The dynamic semantics may be used to inspect the execution of programs by unfolding the derivation tree for judgements $C \xrightarrow{t} D$. Properties of the dynamic semantics such as determinism may be proven using structural induction.

Alternative Dynamic Semantics. In addition to the sequential semantics, [3] defines a semantics for distributed (super-scalar) execution where instructions interleave. No assumptions are made regarding the delays inside functional units. We also consider operand queues of finite length. The relationships between these semantics correspond to the verification conditions in traditional processor verification, but may be proven by structural induction. For example, the distributed model subsumes in-order execution but admits additional interleavings, governed by the availability of operands in the operand queues.

3 Static Semantics

In general, each dynamic model of execution gives rise to specific run time hazards, i.e. conditions under which some programs do not execute correctly. Static semantics allows one to detect many classes of hazards syntactically.

For the sequential model of operation, the typical hazard occurs if an instruction fails to execute due to the lack of operands in the appropriate operand queues. This condition depends on the initial configuration, but also on contextual instructions. In the case of a loop, such a deadlock may only become manifest after a number of iterations.
The type system we present employs a fragment of linear logic [6]. Referring the reader to [3] for formal details, we consider types which are linear products over the set of operand queues and registers, where registers are modelled by exponentials “!”. We thus abstract from the particular values of operands and from the order of items in each queue. Our type system contains one axiom for each instruction form. To each instruction we associate a pair of types which relate configurations prior to the execution to the shape after the instruction has been executed. Typed instructions are composed to instruction sequences using a cut rule. Each straight-line sequence of code is again associated a pair of pre- and post-types. At branch points, we require the net effect of each loop body on the number of elements in each queue to be neutral, similar to work by Stata-Abadi [15]. Operands expected by a loop body must be provided by earlier basic blocks and all operands created in the body must either be consumed immediately or be passed on to successor basic blocks.

The inference of a typing derivation proceeds by weakening minimal typings of basic blocks until a unification of types at basic block boundaries is obtained. Failure of unification indicates the presence of a loop where each iteration consumes more values from the operand queues than it produces, or vice versa.

The soundness of the type system guarantees that well-typed code will not get stuck due to insufficiently many operands. The proof of this result proceeds by structural induction: first single instructions are considered by proving the soundness of the axioms, then straight-line code is considered by proving the soundness of the cut rule, and finally full programs are considered using the rule for combining basic blocks. Thus, well-typed programs will either diverge or will successfully complete, irrespectively of the number of loop iterations. The size of intermediate configurations is statically bound.

**Alternative Models of Execution.** In [3] we generalise our analysis to the alternative dynamic semantics. The typical hazards for distributed execution are race conditions and functional non-determinism as the execution of each instruction is triggered purely by the presence of operands. In our approach, these hazards are seen as a joint property of program and processor. Instead of immediately introducing hardware mechanisms for synchronisation we employ static semantics to identify non-deterministic programs. We extend the type system to detect race conditions and consider various techniques for guaranteeing that the corresponding serialisation requirements are met. Indeed, many programs may be serialised without additional synchronisation hardware.

A particular advantage of our analysis is observed for the model with operand queues of finite length. Here, the characteristic error condition consists of a deadlock due to an operand queue overflow. Our analysis shows that the absence of deadlock is preserved for deterministic programs when the length restrictions are relaxed, while for other programs this is in general not the case.

## 4 Program Analysis

The third aspect of a programming language based approach consists of the ability to formally relate low-level properties to program analysis frameworks [12]. We present a dataflow analysis for a labelled intermediate language for detecting when an intermediate
value is used exactly once. These read-once values are candidates for forwarding, as their single usage corresponds to the deletion from the operand queue during a read access.

The analysis targets the dynamic number of uses of an intermediate variable: any two assignments must be separated by exactly one read-access and no values should be left over at the end of a program run. We generalise the dataflow equations for liveness [12] by using a four-element lattice \( \mathcal{L} \) and say that a pair of functions \( \text{fwd}_{\text{entry}}, \text{fwd}_{\text{exit}} : \text{Lab}_P \to \text{Var}_P \to \mathcal{L} \) is a solution if

\[
\text{fwd}_{\text{exit}}(\ell)(x) = \begin{cases} 0 & \text{if } \ell \in \text{final}(P) \\ \bigcup_{(\ell, \ell') \in \text{flow}(P)} \text{fwd}_{\text{entry}}(\ell')(x) & \text{otherwise} \end{cases}
\]

(1)

\[
\text{fwd}_{\text{entry}}(\ell)(x) = \begin{cases} \text{uses}(\ell)(x) & \text{if } x \in \text{kill}(\ell) \\ \text{uses}(\ell)(x) \oplus \text{fwd}_{\text{exit}}(\ell)(x) & \text{otherwise} \end{cases}
\]

(2)

where \( \text{kill} \) and \( \text{uses} \) are again generalisations of the corresponding functions in the analysis of liveness. The forwardability information of a solution is contained in the component \( \text{fwd}_{\text{exit}} \). In [3] we show that a value assigned to a variable \( x \) at a program point \( \ell \) is read exactly once if \( \text{fwd}_{\text{exit}}(\ell)(x) = 1 \) holds. Variables \( x \) for which \( x \in \text{kill}(\ell) \) implies \( \text{fwd}_{\text{exit}}(\ell)(x) = 1 \) for all \( \ell \) may thus be deleted after any read access. Notice the similarity to the characterisation of useless variables by liveness analysis. The proof of this characterisation formally relates (1) and (2) to a dynamic semantics of the intermediate language.

**Compilation Based on Dataflow Solutions.** Based on dataflow analysis, a compiler may convert intermediate programs into assembly code. The allocation of operand queues to read-once variables differs from register allocation as the order of writing must coincide with the order of reading. In [3] we demonstrate how conflict graphs between read-once variables may be obtained similarly to conflict graphs for register allocation and prove the functional correctness of a translation which maps adjacent variables to different operand queues. We also show that the resulting code is well-typed and thus structurally correct with respect to the underlying hardware. The existence of weakenings for satisfying the typing condition at basic block boundaries is guaranteed by the dataflow solutions, and loops are of neutral net effect. Indeed, the typing judgements may be formally obtained from the dataflow solutions, eliminating the need for an assembly level type inference.

**Experimental Results.** The dataflow analysis was implemented for two conversions of JVM code into the intermediate language and exercised on the Linpack benchmark suite. We observed that nearly all usage of the operand stack may be translated into forwarding if an SSA-like conversion scheme is used. Furthermore, the number of allocated registers decreased by up to 50%, even if each operand queue may only be used for operands sent to a specific functional unit. More significant than these static measures are dynamic measures: our analysis shows that on average 65% of the (central) register read operations turn into (local) operand queue reads, while the corresponding number for write operations is 62%.
5 Discussion

We presented an analysis of forwarding based on dynamic and static semantics of a language with explicit forwarding. We demonstrated the ability of programming language technology to eliminate important classes of error conditions (deadlocks and race conditions) and to analyse the forwarding potential of programs. Interpreting our language as the compiler-visible definition of a processor leads to a verification approach which emphasises that overall system correctness depends as much on program properties as on the correctness of processor implementations. On the other hand, it may be undesirable to expose operand queues explicitly to the programmer. Under this perspective, our analysis demonstrates how a separation between functional and implementational aspects of forwarding may be achieved. Future work is needed to identify how the dataflow-based compilation may be related to hardware implementations. Although the technical results apply only to the specific model of computation considered in this paper, we thus argue that type systems and other syntax-directed formalisms provide a solid basis for structured reasoning about interactions between processor architecture and compilation.

Acknowledgements. The author is grateful to Colin Stirling and Ian Stark for supervising the work described in this paper and for suggesting numerous presentational improvements.

References


Integrating RAM and Disk Based Verification within the Murφ Verifier

Giuseppe Della Penna¹, Benedetto Intrigila¹, Igor Melatti¹, Enrico Tronci², and Marisa Venturini Zilli²

¹ Dip. di Informatica, Università di L’Aquila, Coppito 67100, L’Aquila, Italy
{dellapenna,intrigila,melatti}@di.univaq.it
² Dip. di Informatica Università di Roma “La Sapienza”,
Via Salaria 113, 00198 Roma, Italy
{tronci,zilli}@dsi.uniroma1.it

Abstract. We present a verification algorithm that can automatically switch from RAM based verification to disk based verification without discarding the work done during the RAM based verification phase. This avoids having to choose beforehand the proper verification algorithm.

Our experimental results show that typically our integrated algorithm is as fast as (sometime faster than) the fastest of the two base (i.e. RAM based and disk based) verification algorithms.

1 Introduction

Disk based verification algorithms [4,5,8,3,2] turn out to be very useful to counteract state explosion (i.e. the huge amount of memory required to complete state space exploration). However, using a disk based verification algorithm for a task that could have been completed just using a RAM based verification algorithm results in a waste of time. Unfortunately it is hard to predict beforehand the size of the set of reachable states so as to use the proper (RAM based or disk based) verification algorithm.

In this paper we present an explicit verification algorithm that can automatically switch from RAM based verification to disk based verification without discarding the work done during the RAM based verification phase. This avoids having to choose beforehand the kind of verification algorithm, thus saving on the verification time.

Our main contributions can be summarized as follows.

– We present (Section 3) an integration scheme (we call it serialization scheme) for the RAM based verification algorithm presented in [9] and the disk based verification algorithm presented in [2].
– We present (Section 4) experimental results on using our serialization scheme implemented within the Murφ verifier. Our experimental results show that

© Springer-Verlag Berlin Heidelberg 2003
typically our integrated algorithm is as fast as (sometime faster than) the fastest of the two base (i.e. RAM based and disk based) verification algorithms. This means that on a single machine we are able to run two verification attempts (RAM based and then disk based) within the time taken by the first terminating verification attempt.

2 State Space Exploration Algorithms

Our goal is to devise a serialization scheme for the RAM based state exploration algorithm presented in [9] (CBF, for Cached Breadth First visit in the following) and the disk based state exploration algorithm presented in [2] (DBF, for Disk Breadth First visit in the following).

Figure 1 shows the algorithm and data structures used by a Breadth First (BF) visit. Both the Enqueue() operation on BF queue Q as well as the Insert() operation on the visited states hash table T in Figure 1 may fail because of lack of memory. In such cases the BF visit stops with an out of memory message.

Algorithm CBF [9] implements BF queue Q on disk and, most importantly, replaces with a cache table the hash table T used by the standard BF visit in Figure 1. Using a cache table rather than a hash table means that, upon a collision, CBF may forget visited states and, as a result, it may revisit states. To prevent nontermination due to revisiting states, CBF terminates when the collision rate (i.e. the ratio between the number of collisions and the number of insertions) is above a user given threshold.

Algorithm DBF [2] is a disk based version of CBF. DBF uses a hash table M to store signatures (e.g. see [6]) of recently visited states, a file D to store signatures of all visited states (old states) and splits the BF queue Q of CBF into two queues: Q_ck and Q_unpack. DBF uses the checked queue Q_ck to store the states in the currently explored BF level and uses the unchecked queue Q_unpack to store the states that are candidates to be on the next BF level. At the end of each BF level, DBF uses file D to remove old states from Q_unpack.

Note that with DBF all data structures that grow with the state space size (namely: D, Q_ck, Q_unpack) are on disk, thus DBF bottleneck is computation time, rather than memory space.
3 Serializing CBF and DBF

In our context, a serialization scheme is an algorithm that allows us to stop the current verification task and to resume it possibly using a different algorithm without losing the work previously done.

Let $S$ be a FSS and $\text{Time}(A, S)$ the time needed by algorithm $A$ to complete state space exploration of $S$. A serialization scheme for state space exploration algorithms $A$ and $B$ is a state space exploration algorithm $[A, B]$ s.t. there exist time instants $0 \leq t_1 < t_2 \leq \text{Time}([A, B], S)$ s.t. for all $t \leq t_1$, $[A, B]$ behaves as $A$ and for all $t \geq t_2$, $[A, B]$ behaves as $B$.

Of course a serialization scheme for algorithms $A$ and $B$ is interesting only if the ratio $\text{Time}([A, B], S)/\min(\text{Time}(A, S), \text{Time}(B, S))$ (serialization ratio) is close to 1 for most FSS $S$. This means that on a single machine we are able to run two verification attempts (namely $A$ and $B$) within the time taken by the first terminating verification attempt among the two.

In this section we present a serialization scheme for the RAM based state space exploration algorithm CBF [9] and the disk based state space exploration algorithm DBF [2].

To switch from CBF to DBF we have to save on disk the current status of CBF in such a way that CBF status disk image can then be used to initialize DBF data structures. Figure 2 summarizes our serialization scheme.

CBF status disk image includes the following elements:
1. A file (queue file in Figure 2) containing BF queue $Q$ of Figure 1.
2. A file (state space file in Figure 2) containing the visited states (namely, cache table $T$ of Figure 1).
3. A file (administrative file in Figure 2) containing administrative information about the verification process. For example, such a file may contain: compression options with which CBF has been started (e.g. bit compression [1], hash compaction [6]); random seeds used in various hashing functions (e.g. in the...
computation of state signatures [6]), the BF level reached in the BF visit, the number of states visited so far, etc. In our serialization scheme switching from CBF to DBF is normally requested by the serialization controller (Figure 2) when CBF collision rate becomes greater than a user given threshold.

Serialization is requested by sending a signal to (the suitably modified) CBF. Indeed, to keep easy and efficient our serialization scheme, we only allow CBF to be stopped when it is easy to dump CBF current status to disk. Namely, before a new state is dequeued from the verification queue \( Q \). The CBF queue \( Q \), the cache \( T \) and the parameters are saved to disk in the respective files (Figure 2).

To initialize DBF using the disk image of CBF, first DBF parameters defining state format are overridden by CBF parameters saved in the administrative file on disk. This is needed to ensure compatibility between the data format saved on disk and DBF data format.

CBF queue \( Q \) stored on disk is then loaded and connected to the DBF checked queue \( Q_{\text{ck}} \). This is the best choice since \( Q \) has already been checked w.r.t \( T \). DBF unchecked queue \( Q_{\text{unck}} \) and DBF hash table \( M \) are left empty. DBF history file \( D \) is initialized with the set of visited states in \( T \) (Figure 2). After the above steps DBF can start normally.

4 Experimental Results

We implemented both algorithms CBF and DBF within the Mur\( \varphi \) verifier. This was done as illustrated, respectively, in [9] and [2]. The resulting verifiers are called, respectively, Cached–Mur\( \varphi \) and Disk–Mur\( \varphi \). Thus, not surprisingly, we implemented the serialization scheme outlined in Section 3 within the Mur\( \varphi \) verifier. We call Serial–Mur\( \varphi \) the resulting verifier. Unless otherwise stated, in this Section CBF denotes Cached–Mur\( \varphi \), DBF denotes Disk–Mur\( \varphi \) and [CBF, DBF] denotes Serial–Mur\( \varphi \).

Serial–Mur\( \varphi \) runs first Cached–Mur\( \varphi \) until it completes the verification or the collision rate hits a user given threshold \( \gamma \) (set to 0.1 in our experiments). If the collision rate is greater than or equal to \( \gamma \), Serial–Mur\( \varphi \) switches to Disk–Mur\( \varphi \).

Note that, from [9], we know that Cached–Mur\( \varphi \) behaves as standard Mur\( \varphi \) (both for explored states and verification time) if the collision rate is low. The limitation to 10% of collision rate used in our experiments makes Cached–Mur\( \varphi \) very similar to standard Mur\( \varphi \) in terms of performance.

In this Section we report the experimental results we obtained using [CBF, DBF]. Our goal is of course to assess effectiveness of our serialization scheme. Let \( \mathcal{S} \) be the FSS to be verified. Effectiveness in our case means that the serialization ratio (Section 3) \( (\text{Time}([\text{CBF, DBF}], \mathcal{S})/\min(\text{Time}(\text{CBF}, \mathcal{S}), \text{Time}(\text{DBF}, \mathcal{S}))) \approx 1 \).

We know [2] that if CBF has enough RAM then \( \text{Time}(\text{CBF}, \mathcal{S}) < \text{Time}(\text{DBF}, \mathcal{S}) \). In such cases [CBF, DBF] never switches to DBF and thus behaves as CBF. Thus in such cases \( \text{Time}([\text{CBF, DBF}], \mathcal{S})/\min(\text{Time}(\text{CBF}, \mathcal{S}), \text{Time}(\text{DBF}, \mathcal{S})) \approx 1 \) holds.
Table 1. Serial–Murϕ versus Disk–Murϕ.

<table>
<thead>
<tr>
<th>Protocol</th>
<th>Mem</th>
<th>0.5</th>
<th>0.4</th>
<th>0.3</th>
<th>Mem</th>
<th>0.5</th>
<th>0.4</th>
<th>0.3</th>
</tr>
</thead>
<tbody>
<tr>
<td>mcslock1</td>
<td>Rules</td>
<td>0.594</td>
<td>0.587</td>
<td>0.510</td>
<td>mcslck2</td>
<td>Rules</td>
<td>0.673</td>
<td>0.760</td>
</tr>
<tr>
<td>states</td>
<td>0.820</td>
<td>0.770</td>
<td>0.632</td>
<td>States</td>
<td>1.006</td>
<td>1.030</td>
<td>1.050</td>
<td></td>
</tr>
<tr>
<td>Time</td>
<td>0.837</td>
<td>1.109</td>
<td>0.836</td>
<td>Time</td>
<td>0.984</td>
<td>1.169</td>
<td>1.040</td>
<td></td>
</tr>
<tr>
<td>n_pertson</td>
<td>Rules</td>
<td>0.635</td>
<td>0.689</td>
<td>0.739</td>
<td>sci</td>
<td>Rules</td>
<td>0.607</td>
<td>0.709</td>
</tr>
<tr>
<td>states</td>
<td>1.002</td>
<td>0.984</td>
<td>0.951</td>
<td>States</td>
<td>0.867</td>
<td>0.975</td>
<td>0.694</td>
<td></td>
</tr>
<tr>
<td>Time</td>
<td>0.959</td>
<td>0.952</td>
<td>1.009</td>
<td>Time</td>
<td>1.013</td>
<td>0.976</td>
<td>0.971</td>
<td></td>
</tr>
<tr>
<td>sym cache3</td>
<td>Rules</td>
<td>0.709</td>
<td>0.568</td>
<td>0.654</td>
<td>States</td>
<td>0.966</td>
<td>0.733</td>
<td>0.767</td>
</tr>
<tr>
<td>states</td>
<td>1.029</td>
<td>0.988</td>
<td>1.057</td>
<td>States</td>
<td>1.029</td>
<td>0.988</td>
<td>1.057</td>
<td></td>
</tr>
</tbody>
</table>

Hence the interesting cases for us are those in which CBF does not have enough RAM to complete the verification task. In such cases $\text{Time}(\text{CBF}, S) = \infty$, thus $\min(\text{Time}(\text{CBF}, S), \text{Time}(\text{DBF}, S)) = \text{Time}(\text{DBF}, S)$. Thus we need to check whether $\text{Time}(\text{CBF}, \text{DBF}, S)/\text{Time}(\text{DBF}, S) \approx 1$, which means that [CBF, DBF] completes verification taking about the same time as DBF (even after trying CBF first).

To carry out our experiments we used the benchmark protocols included in the Murϕ distribution [1] that need at least (about) 100Kb of memory to be verified by standard Murϕ, and the kerb protocol from [7].

First, for each protocol $p$ in our benchmark we determined the minimum amount of memory $M(p)$ needed by Murϕ (version 3.1 from [1]) to complete the verification. Then we compared Serial–Murϕ performances with those of Disk–Murϕ for decreasing fractions of such a memory amount. Namely, we ran each protocol $p$ with memory limits $0.5M(p), 0.4M(p)$ and $0.3M(p)$.

In this way, we experimented our approach under conditions in which Serial–Murϕ at some point is forced to switch to Disk–Murϕ since there is not enough RAM for Cached–Murϕ to complete its verification task.

Our results are shown in Table 1, where columns correspond to the memory fraction used for the experiment (e.g. column 0.5 corresponds to half of the needed memory), and rows report the results obtained for a protocol in terms of fired rules, visited states and time to complete the verification. To highlight the usefulness of our approach, we report these results as ratios between the values obtained by Serial–Murϕ and the values obtained using Disk–Murϕ on the same protocols with the same memory restrictions. Thus row Time in Figure 1 gives us the serialization ratio.

The results in Table 1 show that using Serial–Murϕ two verification attempts (namely: Cached–Murϕ and then Disk–Murϕ) take about the same time of the fastest terminating one (namely Disk–Murϕ). Indeed, in Table 1 Time rows
range from 1.1 (i.e. a time overhead of 10%, worst case) to 0.8 (i.e. Serial–Murϕ is 20% faster than Disk–Murϕ), averaging to 0.99.

From Table 1 we see that sometimes Serial–Murϕ is faster than Disk–Murϕ. This is because Serial–Murϕ starts verification using a RAM based algorithm (CBF) which is faster than the disk based algorithm (DBF) to which Serial–Murϕ switches only after part of the verification work has been done (in RAM).

Summing up, the results in Table 1 show that Serial–Murϕ is typically as fast as (sometime faster than) the fastest terminating one among Cached–Murϕ and Disk–Murϕ. Thus, using Serial–Murϕ we can run two verification attempts in the time normally taken by one.

5 Conclusions

We presented a verification algorithm that can automatically switch from RAM based verification to disk based verification without discarding the work done during the RAM based verification phase.

Our experimental results show that typically our integrated algorithm is as fast as (sometime faster than) the fastest of the two base (i.e. RAM based and disk based) verification algorithms. This means that on a single machine we are able to run two verification attempts (RAM based and then disk based) within the time taken by the first terminating verification attempt.

References

1. url: \texttt{http://sprout.stanford.edu/dill/murphi.html}.
7. url: \texttt{http://verify.stanford.edu/ulii/research.html}.
Design and Verification of CoreConnect™ IP Using Esterel

Satnam Singh
Xilinx, 2100 Logic Drive, San Jose, California 95124
Satnam.Singh@xilinx.com

Abstract. This paper explores the practicality of describing and verifying both the hardware and software components of System-on-Chip (SOC) architectures using Esterel. We describe experiments to design and build working hardware based around IBM’s CoreConnect™ Intellectual Property (IP) bus. The flow we analyse has been used to produce working hardware realized on Xilinx’s FPGAs with soft 32-bit processors. Interesting properties about these systems have been proved by static analysis based on model checking.

1 Introduction

There has been a considerable interest in the ability to map a given function to either software or hardware in order to meet constraints for performance, area and time. Recently there has also been much interest in the use of assertions in hardware description languages to aid the verification process of complex systems. Conventional approaches for mapping a function to hardware or software typically involve performing totally separate software and hardware implementations (which is time consuming) or trying to derive one automatically from the other (which often produces poor results especially when one tries to infer efficient hardware from sequential software descriptions). The checking of assertions is often performed by a dynamic analysis i.e. simulation and this is known to have very poor coverage. This paper evaluates an approach which has some promising properties to help solve both of these problems.

We report experiments with the Esterel V7 programming language [7] which we have used for the synthesis of both hardware and software. We also report our experience of performing static analysis of hardware systems with properties expressed graphically as synchronous observers which are checked using an embedded model checker. It is our belief that such graphical descriptions of assertions may be more accessible to engineers than grammar based approaches.

2 Design and Property Specification in Esterel

Previous work in the area of using Esterel for generating efficient protocol code has already been reported as part of the HIPPCO project [4][5]. Much has been written about using Esterel for synthesizing software (specially C code) [3].
An Esterel description can be automatically synthesized either into reactive C software or into a variety of hardware descriptions languages (e.g. VHDL and Verilog) and then synthesized into hardware. The ability to synthesize either hardware or software from the same statically analysable description is one of the most appealing aspects of this technology. Although we do not present them here Esterel also contains many other language constructs for concurrent programming.

Our research investigation seeks to answer the following questions:

1. Can the intuitive graphical safe state machine notation be used effectively by engineers for specifying assertions which can be statically checked? Might this notation be more accessible than grammar based approaches?
2. Can Esterel descriptions be synthesized into efficient hardware and software (including a mixture of both) and work seamlessly with conventional vendor tool Assertions using Synchronous Observers?

Emerging techniques for specifying assertions typically involve using an extra language which has suitable operators for taking about time and logic relationships between signals. These languages are often concrete representations of formal logics and assertion languages are really temporal logics which can be statically analysed. Can the graphical safe state machine notation provide an alternative way of specifying properties about circuits which has the advantage of being cast in the same language as the specification notation? And can these circuit properties be statically analysed to formally prove properties about circuits?

To investigate these questions we performed an experiment which involved designing a peripheral for IBM’s OPB bus which forms part of IBM’s CoreConnect™ IP bus [6]. We chose the OPB bus because it is used by the MicroBlaze soft processor which is available on Xilinx’s FPGAs.

An example of a common transaction on the OPB-bus is shown in Fig. 1. The key feature of the protocol that we will verify with an example is that a read or write transaction should be acknowledged within 16 clock ticks. Unless a control signal is asserted to allow for more time if a peripheral does not respond within 16 ticks then an error occurs on the bus and this can cause the system to crash. Not shown is the OPB_RNW signal which determines whether a transaction performs a read or a write.

![Fig. 1. A sample OPB transaction](image)

We considered the case of a memory mapped OPB slave peripheral which has two device registers that a master can write into and a third device register that a master can read from. The function performed by the peripheral is to simply add the contents of the two ‘write’ registers and make sure that the sum is communicated by the ‘read’ register. A safe state machine for such a peripheral is shown in Figure 2.

This generated VHDL for this peripheral was incorporated into Xilinx’s Embedded Developer Kit and it was then used as a building block of a system which also included a soft MicroBlaze processor, an OPB system bus and various memory
resources and interfaces. We wrote test programs to check the operation of the peripheral with a 50MHz OPB system bus. The peripheral always produced the correct answer.

![Diagram of an OPB-slave peripheral](image)

**Fig. 2.** An OPB-slave peripheral

Having successfully implemented an OPB peripheral from the Esterel specification we then attempted to prove an interesting property about this circuit. We choose to try and verify the property that this circuit will always emit an OPB transfer acknowledge signal two clock ticks after it gets either a read or a write request. If we can statically prove this property we know that this peripheral can never be the cause of a transfer acknowledge timeout event.

We expressed this property as a regular Esterel safe state machine as shown in Figure 3. This synchronous observer tracks the signal emission behaviour in the implementation description and emits a signal if the system enters into a bad state i.e. a read or write request is not acknowledged in exactly two clock ticks.

One way to try and check this property is to try and use it in simulations to see if an error case can be found. Esterel Studio supports this by either simulation directly within the Esterel framework or by the automatic generation of VHDL implementation files and test benches which can check properties specified as synchronous observers.

However, the Esterel Studio system also incorporates a built-in model checker (Prover-SL from Prover Technology) which can be used to try and prove such properties. We use the latest V7 version of the Esterel language which allows us to reason about data as well as control which is an improvement from previous versions of the language. We configured the model check to see if the error signal corresponding to a bad state being entered is ever emitted i.e. might the circuit take
Fig. 3. An assertion expressed as a synchronous observer

longer than two clock ticks to acknowledge a transfer? It took Esterel Studio less than two seconds on a Sun Sparc Ultra-60 workstation to prove that this signal is never emitted.

```
esverify -v OPB.eid -checkis0 XFERACK_MISSING
--- esverify: Reading model from file "OPB.eid".
--- esverify: Checking if output "XFERACK_MISSING" is 0
--- esverify: Start model-checking properties
--- esverify: Verification complete for signal XFERACK_MISSING: --- esverify: --
--- esverify: Model-Checking results summary
```

We then produced a deliberately broken version of the peripheral which did not acknowledge read requests. Within two seconds the software was able to prove that there is a case when the acknowledge signal is not asserted after a transaction and provided a counter-model and VCD file.

A conventional approach to catching such approach bugs involves either simulation (which has poor coverage) or the use of bus monitors which snoop the bus at execution time looking for protocol violations. A failure to acknowledge a transaction is one of the types of bugs that such systems can be configured to catch. However, it is far more desirable to catch such problems with a static analysis. We are currently trying to convert a list of around 20 such bug checks used in a commercial OPB bus monitor into a collection of Esterel synchronous observers to allow us to check peripheral protocol conformance with static analyses.

### 3 Future Work

Here we described our experience with just one system and method for realizing assertions which can be statically analysed. We have started the process of repeating these static property checks using Sugar (with IBM’s FOCs system) and VERA with a view to writing a comparative study of the pros can cons of each approach. Sugar-based systems could be used indirectly in the Esterel flow by compiling synchronous observers into rules (which reside in a separate file from the VHDL design) and then
using the generated VHDL with vendor software for performing static and dynamic analyses of Sugar-based assertions. This would allow Esterel safe charts to act as graphical front end for some types of Sugar assertions.

For the verification of CoreConnect protocols we are developing Esterel models for the PLB (fast complex 64-bit system bus), OPB (a peripheral bus used as a 32-bit bus in Xilinx IP and used as the main bus for Xilinx’s soft processor) and DCR (a simpler device control register bus) components which can then be used to help write synchronous observers for IP blocks without replicating the functionality of the system arbiters in each observer. Previous work [8] using the model checker Rulebase [1][2] for proving properties about CoreConnect arbiters suggests that this approach should be feasible.

To investigate how feasible it is to make hardware/software trade-offs using this flow we are developing implementations for network-on-chip protocols which are implemented in a combination of hardware and software. Using this flow can experiment with what portions need to be in hardware for performance and we may also be able to perform interesting static analyses of protocol behaviour, performance and correctness.

4 Conclusions

The approach of using Esterel to produce hardware and software seems to show some promise. Initial experiments show that serviceable hardware and software can be produced and implemented on real hardware and embedded processors. The possibility to enter system specifications graphically makes this method much more accessible to regular engineers than competing formalisms which uses languages which are quite different to what engineers are used to. For any realistic system the developer still has to write some portions textually and become aware of the basic underlying principles of Esterel. It remains to be seen if the cost of learning this formalism is repaid by increased productivity, better static analysis and the ability to trade off hardware and software implementations.

An appealing aspect of this flow is the ability to write assertions in the same language as the system specification. This means that engineers do not need to learn yet another language and logic. Furthermore, the formal nature of Esterel’s semantics may help to make static analysis easier. Our initial experiments with using the integrated model checker are certainly encouraging. However, we need to design and verify more complex systems before we can come to a definitive conclusion about this promising technology for the design and verification of hardware and software from a single specification.

A very useful application of this technology would be to task-based dynamic reconfiguration. This method would avoid the need to duplicate implementation effort and it would also allow important properties of dynamic reconfiguration to be statically analysed to ensure that reconfiguration does not break working circuits.

There are some limitations to the technique we present here. There are some refinements that need to be made to the Esterel language to properly support hardware description. Most of these requirements are easily met without upsetting the core design of the language. Examples include a much more flexible way of converting between integers and bit-vectors and to allow arbitrary precision bit-vectors.
Currently performing an integer-based address decode for a 64-bit bus is possible in Esterel but one has to process the bus in chunks not larger than 31 bits.

“Virtex-II” is a trademark of Xilinx Inc. “CoreConnect” is a trademark of IBM. We would like to thank the staff at Esterel Technologies for their generous assistance during this project.

References

Inductive Assertions and Operational Semantics

J Strother Moore
Department of Computer Sciences
University of Texas at Austin
Austin, TX 78712-1188, USA
moore@cs.utexas.edu

Abstract. This paper shows how classic inductive assertions can be used in conjunction with an operational semantics to prove partial correctness properties of programs. The method imposes only the proof obligations that would be produced by a verification condition generator but does not require the definition of a verification condition generation. The paper focuses on iterative programs but recursive programs are briefly discussed. Assertions are attached to the program by defining a predicate on states. This predicate is then “completed” to an alleged invariant by the definition of a partial function defined in terms of the state transition function of the operational semantics. If this alleged invariant can be proved to be an invariant under the state transition function, it follows that the assertions are true every time they are encountered in execution and thus that the post-condition is true if reached from a state satisfying the pre-condition. But because of the manner in which the alleged invariant is defined, the verification conditions are sufficient to prove invariance. Indeed, the “natural” proof generates as subgoals the classical verification conditions. The invariant function may be thought of as a state-based verification condition generator for the annotated program. The method allows standard inductive assertion style proofs to be constructed directly in an operational semantics setting. The technique is demonstrated by proving the partial correctness of a simple bytecode program with respect to a pre-existing operational model of the Java Virtual Machine.

1 Summary

This paper connects two well-known approaches to program verification: operational semantics and inductive assertions. The paper shows how one can adopt the clarity and concreteness of a formal operational semantics while incurring just the proof obligations of the inductive assertion method, without writing a verification condition generator or other extra-logical tool. In particular, the formal definition of the state transition function can be used directly to generate verification conditions for annotated programs.

In this section the idea is presented in the abstract. Some details are skipped and a deliberate confusion of states with formulas is perpetrated to convey the basic idea. Subsequently, the method is applied to a particular formal operational
Consider a simple one loop program $\pi$ (Figure 1) that concludes with a $\text{HALT}$ instruction. Assume instructions are addressed sequentially, with $\alpha$ being the address or label of the first instruction and $\gamma$ being the address or label of the $\text{HALT}$. Let the pre- and post-conditions of the program be $P$ and $Q$ respectively. The arrows of Figure 1 indicate the control flow; functions $f$, $g$, and $h$ indicate the compound state transitions along the arcs and $t$ is the test for staying in the loop. $R$ is the loop invariant and “cuts” the only loop. The partial correctness challenge is to prove that if $P$ holds at $\alpha$ then $Q$ holds whenever (if) control reaches $\gamma$.

To give meaning to such programs with an operational semantics, one formalizes the abstract machine state and the effect of each instruction on the state. Typically the state, $s$, is a vector or $n$-tuple describing available computational resources such as environments, stacks, flags, etc. It is assumed here that the state includes a program counter, $pc (s)$, and the current program, $prog (s)$, which are used to determine the next instruction. Instructions are given meaning by defining a state transition function $\text{step}$. Typically, $\text{step} (s)$ is defined by considering the next instruction and transforming the state components accordingly. For example, a $\text{LOAD}$ instruction might advance the program counter and push onto some stack the contents of some specified variable. More complicated instructions, such as method invocation, may affect many parts of the state. The $\text{HALT}$ instruction is particularly simple; it is a no-op.

It is convenient to define an iterated step function:

$$run (k, s) = \begin{cases} s & \text{if } k = 0 \\ run (k - 1, \text{step} (s)) & \text{otherwise} \end{cases}$$

and to make the convention that $s_k = run (k, s)$. 

---

**Fig. 1. The One-Loop Program $\pi$ with Annotations**

<table>
<thead>
<tr>
<th>labels</th>
<th>program $\pi$</th>
<th>paths</th>
<th>assertions</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\alpha$</td>
<td></td>
<td>$f(s)$</td>
<td>$P(s)$ pre-condition</td>
</tr>
<tr>
<td>$\beta$</td>
<td></td>
<td>$g(s)$</td>
<td>$R(s)$ loop invariant</td>
</tr>
<tr>
<td>$\gamma$</td>
<td>$\text{HALT}$</td>
<td></td>
<td>$Q(s)$ post-condition</td>
</tr>
</tbody>
</table>
Given this operational semantics, the formalization of the partial correctness result is

**Theorem:** Correctness of Program \( \pi \).

\[ pc(s) = \alpha \land prog(s) = \pi \land P(s) \land pc(s_k) = \gamma \rightarrow Q(s_k). \]

**Proof.** In an operational semantics setting, theorems such as the Correctness of Program \( \pi \) are proved by establishing an invariance \( Inv(s) \) with the following three properties:

1. \( Inv(s) \rightarrow Inv(step(s)) \),
2. \( pc(s) = \alpha \land prog(s) = \pi \land P(s) \rightarrow Inv(s) \), and
3. \( pc(s) = \gamma \land prog(s) = \pi \land Inv(s) \rightarrow Q(s) \).

The main theorem is then proved as follows. The inductive application of property 1 produces

4. \( Inv(s) \rightarrow Inv(s_k) \).

Furthermore, instantiation of the \( s \) in property 3 with \( s_k \) produces

5. \( pc(s_k) = \gamma \land prog(s_k) = \pi \land Inv(s_k) \rightarrow Q(s_k) \).

We assume no instruction in \( \pi \) changes the program; hence \( prog(s) = prog(s_k) \). The Correctness of Program \( \pi \) then follows immediately from 2, 4, and 5. \( \square \)

Property 1, above, is problematic; it forces the user of the methodology to characterize all the states reachable from the chosen initial state. Contrast this situation with that enjoyed by the user of the inductive assertion method, where assertions are attached only to certain user-chosen cut-points, as in Figure 1. An extra-logical process, which encodes the language semantics as formula transformations, is then applied to the annotated program text to generate proof obligations or verification conditions

\[ VC1. P(s) \rightarrow R(f(s)), \]
\[ VC2. R(s) \land t \rightarrow R(g(s)), \] and
\[ VC3. R(s) \land \neg t \rightarrow Q(h(s)). \]

If these formulas are proved, the user is then assured that if \( P \) holds initially then \( Q \) holds when (if) the program terminates.

To render this assurance formal, i.e., write it as a formula, one must adopt some logic of programs, i.e., a logic that allows the combination of classical mathematical expressions about numbers, sequences, vectors, etc., with program text and terminology. The resulting programming language semantics is extra-logical in the sense that it is expressed as rules of inference in a metalanguage and is not directly subject to formal analysis within the logic.\(^1\) In contrast, in the operational approach, the semantics is expressed within the language (typically

\(^1\) See however the discussion of [3] the next section.
as defined functions or relations on states), programs are objects in the logical universe, and the properties of both — programs and the semantic functions and relations — are subject to proof within the logic.

The central question of this paper is whether it is possible to have the best of both worlds: the concreteness and clarity of an operational semantics in a classical logical setting but the elegance and simplicity of an inductive assertion-style proof. The central question may be put bluntly as “Is it possible to prove the formula named ‘Correctness of Program π,’ above, directly from VC1–VC3?” The answer is “yes.”

Recall that the proof of ‘Correctness of Program π’ required the definition of Inv(s) satisfying properties 1–3 above. The key to constructing an inductive assertion-style proof in an operational setting is the following definition of Inv(s).

\[
\text{Inv}(s) \equiv \begin{cases} 
\text{prog}(s) = \pi \land P(s) & \text{if } \text{pc}(s) = \alpha \\
\text{prog}(s) = \pi \land R(s) & \text{if } \text{pc}(s) = \beta \\
\text{prog}(s) = \pi \land Q(s) & \text{if } \text{pc}(s) = \gamma \\
\text{Inv}(\text{step}(s)) & \text{otherwise}
\end{cases}
\]

The logician will immediately ask whether there exists a predicate satisfying this equivalence. The affirmative answer is provided in [10]. The logical crux of the matter is that Inv(s) is defined with tail-recursion and there exists a satisfying and total witness for every tail-recursive equivalence. If some loop in the program is not cut, the equivalence may not uniquely define a predicate, but at least one witness exists.

Inv(s) clearly has properties 2 and 3. It therefore remains only to prove property 1. As will become apparent, the proof that Inv(s) has property 1 will generate the verification conditions as subgoals. To drive this home, we describe the process by which the proof is constructed rather than merely the formulas produced. Recall Figure 1. Successive steps from a state s with pc α eventually produce the state f(s) with pc β. Similarly, if t, then successive steps from a state s with pc β produce g(s) with pc β, and if ¬t, then successive steps from a state s with pc β produce h(s) with pc γ. Furthermore, repeated symbolic expansion and simplification of the step function produce the transformations described by f, g, and h.

**Theorem:** Property 1.

\[\text{Inv}(s) \rightarrow \text{Inv}(\text{step}(s))\]

**Proof.** Consider the cases on pc(s) as used in the definition of Inv.

Case: pc(s) = α. The hypothesis, Inv(s) may be simplified to prog(s) = π ∧ P(s). Consider the conclusion, Inv(step(s)). Symbolic simplification of step(s), given pc(s) = α and prog(s) = π, produces a symbolic state s’ with pc(s’) = α + 1. For program π either α + 1 is β or it is none of the cut points α, β or γ. In the latter case, Inv(step(s)) ≡ Inv(s’) ≡ Inv(step(s’)) and stepping
continues until $\beta$ is reached at state $f(s)$. Hence, $\text{Inv}(\text{step}(s)) \equiv R(f(s'))$ (since $\text{prog}(f(s)) = \pi$). Thus, this case simplifies to the goal

$$pc(s) = \alpha \land \text{prog}(s) = \pi \land P(s) \rightarrow R(f(s)).$$

This is just VC1 (with two now-irrelevant hypotheses, given traditional assertions $P$ and $R$).

Case: $pc(s) = \beta$. The hypothesis $\text{Inv}(s)$ simplifies to $\text{prog}(s) = \pi \land R(s)$. Then the symbolic simplification of $\text{step}(s)$ in the conclusion produces a bifurcated symbolic state whose program counter depends on test $t$. Repeated expansions of the definition of $\text{Inv}$ on both branches of the state eventually reach states $g(s)$ and $h(s)$ at which $\text{Inv}$ is defined. The results are VC2 and VC3, respectively.

Case: $pc(s) = \gamma$. The hypothesis $\text{Inv}(s)$ simplifies to $\text{prog}(s) = \pi \land Q(s)$. But the $\text{step}(s)$ in the conclusion simplifies to $s$ because the instruction at $\gamma$ in $\pi$ is the no-op $\text{HALT}$. Hence, $\text{Inv}(s) \equiv \text{Inv}(\text{step}(s))$ and this case is trivial (propositionally true independent of the assertions).

Case: otherwise. Since $pc(s)$ is not one of the cut-points, $\text{Inv}(s) \equiv \text{Inv}(\text{step}(s))$ by definition of $\text{Inv}$ and this case is also trivial.

$\square$

Hence, if the verification conditions VC1–VC3 have been proved, the proof of property 1, the step-wise invariance of $\text{Inv}$, involves no assertion-specific reasoning. More interestingly, given the definition of $\text{Inv}$, the proof generates the verification conditions by symbolic expansion of the operational semantics’ state transition function.

Practically speaking this means that with a mechanical theorem prover and a formal operational semantics one can enjoy the benefits of the inductive assertion method without writing a verification condition generator or other extra-logical tools to do formula transformations.

Another practical ramification of this paper is that it provides a simple means to define a step-wise invariant given only the assertions at the cut points. Step-wise invariants are frequently needed in operational semantics-based proofs of safety and liveness properties.

2 Related Work and Discussion

McCarthy [11] made explicit the notion of operational semantics, in which “the meaning of a program is defined by its effect on the state vector.”

The inductive assertion method for proving programs correct was implicitly used by von Neumann and Goldstine in [4] and made explicit in the classic papers by Floyd [2] and Hoare [5]. The first mechanized verification condition generator, which generates proof obligations from code and attached assertions, was written by King [8]. Hoare, of course, rendered the inductive assertion method formal by introducing a logic of programs. From the practical perspective most program logics are mechanized with two trusted tools, a formula generator, here called a
VCG, and a theorem prover. It is not uncommon for the VCG to include not just language semantics as formula transformers but also some logical simplification (i.e., theorem proving) to keep the generated proof obligations manageable.

A notable exception is the work of Gloess [3] where the Hoare semantics of a simple imperative programming language is formalized within the higher-order logic of PVS and mechanically checked proofs of several programs are carried out with PVS. As in the present work, Gloess’ proofs generate the verification conditions. The difference however is that the formal semantics is Hoare-style rather than operational and is thus designed to generate formulas.

This paper contains one apparently novel idea: a step-wise invariant can be defined from the inductive assertions using the state-transition function. One may think of this as a methodology for obtaining a state-based verification condition generator from an operational semantics. By doing it on a per program basis the method avoids the need to generate or trust extra-logical tools.

The use of inductive assertions in conjunction with a formal operational semantics to prove partial correctness results mechanically is not new. Robert S. Boyer and the author developed it for their Analysis of Programs course at the University of Texas at Austin as early as 1983. In that class, an operational semantics for a simple procedural language in Nqthm [1] was defined and the course explored program correctness proofs that combined operational semantics with inductive assertions. These proofs motivated the exploration of total versus partial correctness, Hoare logics, and verification condition generation. For an Nqthm proof script illustrating the use of inductive assertions in an operational semantics setting, see [12].

A recent example of the use of assertions to prove theorems about a program modeled operationally may be found in [15], where a safety property of a non-terminating multi-threaded Java system is proved with respect to an operational semantics for the Java Virtual Machine [14].

However, in the earlier work the invariant explicitly included an assertion for every value of the pc. (The invariant must recognize every reachable state and so must handle every pc; the issue is whether it does so explicitly or implicitly.)

An alternative way to combine inductive assertions at selected cut points with an operational semantics in a classical formal setting is to formalize and verify a VCG with respect to the operational semantics. In [6], for example, an HOL proof of the correctness of a VCG for a simple procedural language is described. The work includes support for mutually recursive procedures. Formal proofs of the verification conditions could, in principle, be used with the theorem stating the correctness of the VCG, to derive a property stated operationally. But the method described here does not require the definition of a VCG much less a proof of its correctness.

Logically speaking, a crucial aspect of the novel idea here is that the step-wise invariant is defined using tail recursion. The admission of a new function or predicate symbol via recursive definition is generally handled by a definitional principle that insures the existence (and often the uniqueness) of the defined concept. In many logics, this requires a termination proof. Admitting Inv under such
a definitional principle would require a measure of the distance to the next cut point and a proof that the distance decreases under step. That imposes a proof burden not generally incurred by the user of the inductive assertion method. (Every loop must be cut for the inductive assertion method to be effective; the question is whether that must be proved formally or merely demonstrated by the successful generation of the verification conditions.)

The technique used here exploits the observation that Inv is tail-recursive and hence admissible without proof obligation, given the work of Manolios and Moore [10] in which it was proved that every tail-recursive equation may be witnessed by a total function. The tail-recursive function may not be uniquely defined by the equation — this might occur if insufficient cut points are chosen. Such a failure is manifested by an infinite loop in the process of generating/proving the step invariance. This is the same behavior a VCG user would experience in the analogous situation.

The technique here is similar in spirit to one used by Pete Manolios [private communication] to attack the 2-Job version of the Apprentice problem [15]. There, he defined the reachable states of the Apprentice problem as all the states that could be reached from certain states by the execution of a fixed maximum number of steps.

See [13] for a long version of this paper, including all proof scripts.

3 A Demonstration of the Method

To illustrate the technique a mechanized formal logic and an operational semantics must be introduced. In this paper we use the ACL2 logic [7]. In this logic, function application is denoted as in Lisp, e.g., run \((k, s)\) is written \((\text{run } k \ s)\).

For the demonstration we choose a pre-existing operational semantics for a significant fragment of the JVM [9]. The model is called M5 [14] and it was chosen simply because it was available and it was realistic.

The M5 model is fairly complex, requiring about 250 ACL2 definitions consuming about 3000 lines of formalism on top of ACL2’s extensive support for discrete mathematics. In addition to many other JVM data types, M5 supports Java’s 32-bit two's complement integer arithmetic, here called “\text{int arithmetic},” in which overflow is not signaled; adding one to the most positive \text{int} produces the most negative \text{int}. M5 models 138 bytecode instructions including those for the creation and initialization of instance objects in the heap, manipulation of static and instance fields, the invocation of static, special, and virtual methods, Java’s inheritance rules for method resolution, the creation of multiple threads, and synchronization via monitors. The model is operational in the sense that it can be executed on the output of Sun’s \text{javac} compiler (after transformation of the class files into ACL2 constants).

The M5 model of the JVM is a good example of an abstract machine that is sufficiently complicated that writing a VCG for it a serious and error-prone undertaking.
M5 is formalized by defining *step* and *run* functions as above. The state includes a thread table containing stacks of method invocation frames, a heap, and a class table of loaded classes. Each frame contains a pc, bytecoded program, local variables, and operand stack. The M5 *step* function takes two arguments instead of just one: (*step* th s) is the state obtained by stepping thread th in state s. The *run* function, instead of taking the number of steps, takes a list of thread identifiers, called a schedule, and steps those threads sequentially.

Symbolic simplification of this semantics is central to the idea proposed here. Consider the following bytecode sequence (in the M5 parsed byte-stream format): (ILOAD_1) (ICONST_1)(IADD)(ISTORE_1). This sequence pushes the value of local variable 1 on the operand stack, pushes the constant 1, pops the first two items off the stack and pushes their int sum, and pops the stack into local variable 1. That is, the sequence corresponds to the Java assignment \( a = a+1; \) if a is allocated in local variable 1. Suppose M5 state s contains a thread, th, the active frame of thread th has pc 6 and that the bytecode sequence above is positioned starting at byte offset 6 in the current program. Suppose the locals of the frame are denoted by *locals* and the operand stack by *stack*. The symbolic simplification of (*step* th s) produces a symbolic state expression in which the active frame of thread th has pc 7 and operand stack (*push* (*nth* 1 *locals*) *stack*). If three more such steps are taken the result is a symbolic state expression in which the active frame of thread th has pc 10 and the following expression, *locals'*, for its locals (*update-nth* 1 (*int-fix* (+ (*nth* 1 *locals*) 1)) *locals*). Note that the symbolic expression for local 1 in this environment, (*nth* 1 *locals'*) simplifies to (*int-fix* (+ (*nth* 1 *locals*) 1)) using rewrite rules about *nth* and *update-nth*.

### 4 An Iterative Program

Below is an M5 program that decrements its first local, informally called n, by 2 and iterates until the result is 0. On each iteration it adds 1 to its second local variable, here called a, which is initialized to 0. Thus, the method computes \( n/2 \), henceforth written \((/ \ n \ 2)\), when \( n \) is even. It does not terminate when \( n \) is odd.

The program is slightly simpler to deal with if it is assumed that \( n \) is a non-negative *int*. The program actually terminates for even negative *ints*, because Java’s *int* arithmetic wraps around; the most negative *int*, \(-2147483648\), is even and when it is decremented by 2 it becomes the most positive even, \(2147483646\). For simplicity, the program concludes with the fictitious *HALT* instruction, which stops the machine. The program constant below is named *flat-prog* because it does not return to a caller but stops the machine. Method invocation is discussed later in the paper.

```
(defconst *flat-prog*
'((ICONST_0) ; 0
 (ISTORE_1) ; 1 a := 0
 (ILOAD_0) ; 2 top of loop:
)
(IFEQ 14) ; 3 if n=0, goto 17
(ILOAD_1) ; 6
(ICONST_1) ; 7
(IADD) ; 8
(ISTORE_1) ; 9 a := a + 1
(ILOAD_0) ; 10
(ICONST_2) ; 11
(ISUB) ; 12
(ISTORE_0) ; 13 n := n - 2
(GOTO -12) ; 14 goto top of loop
(ILOAD_1) ; 17 push a
(HALT))) ; 18

Let the initial value of n be n0. The goal is to prove that if n0 is a non-negative int and control reaches pc 18, then n0 is even and (\(\lfloor n / 2 \rfloor\)) is on the stack. That is, if the program halts the initial input must have been even and the final answer is half that input.

Rather than deal with integer division during the code proof, the following function is introduced. The decision to use this function rather than algebraic expressions to express the properties of the code is independent of the decision to express the properties with inductive assertions.

(defun halfa (n a)
  (if (zp n)
    a
    (halfa (- n 2) (int-fix (+ a 1)))
))

Here, int-fix returns the integer represented by the low-order 32-bits of its argument and thus implements int wrap-around. The inductive assertion method will be used to establish that if the program terminates it will leave (halfa n0 0) on the stack. A second theorem, independent of the code, establishes that (halfa n0 0) is (\(\lfloor n / 2 \rfloor\)) under certain conditions. Such decomposition of code proofs into “algorithm” and “requirements” is standard in the ACL2 community and independent of whether inductive assertions are being used. It is possible, of course, to mix the two via inductive assertions about division or multiplication by two.

5 The Assertions at the Three Cut Points

The cut points, to which assertions will be attached, are at program counters 0 (\(\alpha\)), 2 (\(\beta\)), and 18 (\(\gamma\)). The assertions themselves, called P, R, and Q in the earlier treatment, are captured by the following function definitions. The names of the functions are, of course, irrelevant but indicate how they will be used. In the earlier treatment it was convenient to make these functions of state; here they are functions of the initial input n0 and the relevant state components, namely n and a.
(defun flat-pre-condition (n0 n)
  (and (equal n n0)
       (intp n0)
       (<= 0 n0)))

(defun flat-loop-invariant (n0 n a)
  (and (intp n0)
       (<= 0 n0)
       (intp n)
       (if (and (<= 0 n)
                 (evenp n))
           (equal (halfa n a)
                  (halfa n0 0))
           (not (evenp n)))
       (iff (evenp n0) (evenp n))))

(defun flat-post-condition (n0 value)
  (and (evenp n0)
       (equal value (halfa n0 0))))

The details of the assertions are not germane to this paper. The assertions are
typical inductive assertions for such a program. They are complicated primarily
because of Java’s int arithmetic. Halfa tracks the behavior of the program only
as long as n stays non-negative. Things would be simpler if the pre-condition
required that n0 be even or if the post-condition did not assert that n0 is even.
These assertions were chosen to illustrate that operational semantics could be
used to address partial correctness of non-terminating programs including the
characterization of when termination occurs.

6 Verification Conditions

Given *flat-prog*, the informal attachment of the three assertions to the cho-

en cut points, and a VCG for the JVM, the following verification conditions
would be produced.

(defthm VC1 ; entry to loop
  (implies (flat-pre-condition n0 n)
           (flat-loop-invariant n0 n 0)))

(defthm VC2 ; loop to loop
  (implies (and (flat-loop-invariant n0 n a)
                (not (equal n 0)))
           (flat-loop-invariant n0
                                (int-fix (- n 2))
                                (int-fix (+ 1 a)))))

(defthm VC3 ; loop to exit
  (implies (and (flat-loop-invariant n0 n a)
                (equal n 0))
           (flat-post-condition n0 a)))
These are easily proved. The challenge is: how can these three theorems be used to verify a partial correctness result for *flat-prog*?

7 Attaching the Assertions to the Code

In the earlier treatment of the method, the invariant conjoined each assertion with \( \text{prog}(s) = \pi \). Here we introduce an intermediate function to do this and also to name relevant components of the state.

\[
\text{(defun flat-assertion (n0 th s)}
\]

\[
\quad (\text{let ((n (nth 0 (locals (top-frame th s)))))}
\]

\[
\quad \quad (a (nth 1 (locals (top-frame th s)))))
\]

\[
\quad (\text{(and (equal (program (top-frame th s)) *flat-prog*)})
\]

\[
\quad (\text{(case (pc (top-frame th s))})
\]

\[
\quad \quad (0 (flat-pre-condition n0 n))
\]

\[
\quad \quad (2 (flat-loop-invariant n0 n a))
\]

\[
\quad \quad (18 (let (((value (top (stack (top-frame th s))))))
\]

\[
\quad \quad \quad (\text{(flat-post-condition n0 value)}))
\]

\[
\quad \quad (\text{(otherwise nil)})))
\]

The \text{let} identifies parts of the JVM state of interest: the 0\text{th} local of thread \text{th}, called \text{n}, and the 1\text{st} local of thread \text{th}, called \text{a}. It requires that the program being executed by the thread be *flat-prog* ("\(\pi\'\)). It then case splits on the \text{pc} of thread \text{th} and for program counters 0, 2, and 18 makes an assertion about \text{n}, \text{a}, and \text{n0}. The variable symbol \text{value} at the post-condition is bound to the value on top of the operand stack of the relevant thread at the conclusion of the program.

8 The Nugget: Defining the Invariant

The nugget in this paper is how the assertions, attached to selected cut points, are completed into a step-wise invariant on states.

The invariant is introduced with the \text{defpun} ("define partial function") utility of [10]. The assertions are tested at the three cut points and all other statements inherit the invariant of the next statement. This definition is analogous to that for \text{Inv} in the abstract treatment, except that the invariant also takes the initial input, \text{n0}, and the identifier of the relevant thread, \text{th}.

\[
\text{(defpun flat-inv (n0 th s)}
\]

\[
\quad (\text{(if (or (equal (pc (top-frame th s)) 0)}
\]

\[
\quad \quad (\text{(equal (pc (top-frame th s)) 2)}
\]

\[
\quad \quad \quad (\text{(equal (pc (top-frame th s)) 18)}))
\]

\[
\quad \quad (\text{(flat-assertion n0 th s)}
\]

\[
\quad \quad \quad (\text{(flat-inv n0 th (step th s)}))))
\]
9 Proofs

Here is the key theorem, called “property 1 of Inv” or the step-wise invariant theorem.

(defthm flat-inv-step
  (implies (flat-inv n0 th s)
    (flat-inv n0 th (step th s))))

As noted earlier, the proof attempt generates the verification conditions (with a few extra hypotheses about the program counter and current program). If ACL2’s data base already contains the theorems VC1–VC3, those theorems are used to complete the proof of flat-inv-step. If the verification conditions have not already been proved, the proof attempt here generates and proves them.

Central to the process is the symbolic simplification of state expressions under the state transition function step.

Having proved the invariance of flat-inv under step the next theorem in the mechanized “methodology” corresponds to property 4 of the earlier proof of the Correctness of Program π. is trivial. The theorem states that flat-inv is invariant under arbitrarily long runs of the thread in question.

(defthm flat-inv-run
  (implies (and (mono-threadedp th sched)
    (flat-inv n0 th s))
    (flat-inv n0 th (run sched s))))

where

(defun mono-threadedp (th sched)
  (if (endp sched)
      t
    (and (equal th (car sched))
         (mono-threadedp th (cdr sched))))).

Proof of flat-inv-run is trivial by induction and appeal to flat-inv-step.

Thus, if the initial state has pc 0 and satisfies the pre-condition, and, after some arbitrary mono-threaded run, a state with pc 18 is reached, then it satisfies the post-condition, namely, n0 is even and the answer is (halfa n0 0). Formally this can be written as follows.

(defthm flat-main
  (let ((s1 (run sched s0)))
    (implies (and (intp n0)
                  (<= 0 n0)
                  (equal (pc (top-frame th s0)) 0)
                  (equal (locals (top-frame th s0)) (list n0 any))
                  (equal (program (top-frame th s0)) *flat-prog*)
                  (mono-threadedp th sched)
                  (equal (pc (top-frame th s1)) 18)))
(and (evenp n0)
 (equal (top (stack (top-frame th s1)))
 (halfa n0 0))))

This is proved by using the instance of flat-inv-run obtained by letting s be s0.

Flat-main is essentially the goal, except it characterizes the answer as (halfa n0 0). If (/ n0 2) were preferred, either a separate proof relating (halfa n0 0) to (/ n0 2) could be performed, or the assertions could be stated in terms of division in the first place. In any case, this issue is independent of the use of inductive assertions.

It takes ACL2 approximately 8 seconds (on a 797MHz Pentium III) to prove flat-inv-step, in which the verification conditions are generated by repeated symbolic expansion of step on the bytecode in *flat-prog*. The subsequent proofs of flat-inv-run and flat-main take less than 1.5 seconds in all. The only proof-specific lemmas developed for this exercise were mathematical lemmas on the properties of evenp int arithmetic when subtracting 2.

Notice what has been accomplished. Flat-main is a partial correctness theorem about a JVM program, formalized with an operational semantics. The creative part of the proof consisted of the definition of the three assertions. Users familiar with inductive assertions would find these assertions straightforward (requiring only a few minutes to write down). The proof of the key lemma, flat-inv-step, generated (and requires the proof of) the classic verification conditions just as though a VCG for the JVM were available. But no VCG was defined. The proof does not establish termination of the code under the pre-conditions but does characterize necessary conditions to reach the HALT statement. Finally, neither the theorem nor the proof involved counting instructions or defining what is called a “clock function” in the Boyer-Moore community.

10 Method Invocation and Return

The HALT instruction in the previous program is fictitious but handy. Stepping the machine while on a HALT leaves the machine at the HALT. Thus, the invariance of the exit assertion is easy to prove once the exit is reached. In realistic code, the machine does not halt but returns control to the caller and non-trivial stepping continues. A useful inductive assertion methodology must deal with call and return. This paper does not discuss call and return in detail; see [13].

On the JVM, method invocation pushes a new stack frame on the invocation stack of the active thread. Abstractly, that frame may be thought of as containing the bytecode for the newly invoked method with initial pc 0. The new frame contains an initially empty “operand stack” for intermediate results. When certain return instructions are executed, the topmost item, v, on the operand stack is removed, the invocation stack is popped, and v is pushed onto the operand stack of the caller.2

2 Some forms of return implement void methods and return no v to the caller.
To deal with call and return via inductive assertions, two changes are made to the “methodology” described above. First, instead of using \texttt{run} to run the state a certain number of steps, the new function \texttt{run-to-return} is introduced, which runs a certain number of steps or until the state returns from the call depth, \texttt{d0}, at which the run was started. Second, the assertion function is changed so that the post-condition is asserted if the call depth is less than \texttt{d0}.

To deal with recursive methods, one must characterize the stack of frames created by previous recursive calls so that \texttt{returns} produce states in which continued symbolic evaluation is possible.

It should be possible to use this technique to express safety and liveness invariants for multi-threaded programs, significantly reducing the amount of definitional done in examples such as [15], but that experiment has not been done yet.

11 Conclusion

This paper has demonstrated that inductive assertion style proofs can be carried out in an operational semantics framework, without producing a verification condition generator or incurring proof obligations beyond those produced by such a tool. The key insight is that assertions attached to cut points in a program can be propagated by a tail-recursive function to create an alleged invariant. The proof that the alleged invariant is invariant under the state transition function produces the standard verification conditions. The invariance result can then be traded in for a partial correctness result stated in terms of the operational semantics, without requiring the construction of clocks or the counting of instructions.

No verification condition generator need be constructed. Given an operational semantics it is possible, more or less immediately, to perform inductive assertion style proofs of partial correctness theorems.

The process of proving the step-wise invariance of the completed assertions “naturally” produces the verification conditions.

This situation is attractive for three reasons. First, writing a verification condition generator for a realistic programming language like JVM bytecode is error-prone. For example, method invocation involves complicated non-syntactic issues like method resolution with respect to the object on which the method is invoked, as well as side-effects to many parts of the state including, possibly, the call frames of both the caller and the callee, the thread table (in the event that a thread is started), the heap (in the event of a synchronized method locking the object upon which it is invoked), and the class table (in the event of dynamic class loading). Coding this all in terms of formula transformation instead of state transformation is difficult. Second, when completed, the semantics of the language is encoded in the VCG process rather than as sentences in a logic. This encoding of the semantics makes it difficult to inspect. In our approach, the semantics is expressed explicitly in the logic so that it can be inspected. Indeed, it is possible to prove theorems about the semantics (not just theorems
about programs under the semantics). Finally, realistic VCGs contain simplifiers used to keep the generated proof obligations simple. These simplifiers are just theorems provers and must be trusted. In our approach, only one theorem prover is involved. It must be trusted but that trusted engine derives the verification conditions from the operational semantics and the user-supplied assertions.

References

A Compositional Theory of Refinement for Branching Time

Panagiotis Manolios

Georgia Institute of Technology, CERCS Lab
801 Atlantic Drive
Atlanta, Georgia, 30332, USA
manolios@cc.gatech.edu
http://www.cc.gatech.edu/~manolios

Abstract. I develop a compositional theory of refinement for the branching time framework based on stuttering simulation and prove that if one system refines another, then a refinement map always exists. The existence of refinement maps in the linear time framework was studied in an influential paper by Abadi and Lamport. My interest in proving analogous results for the branching time framework arises from the observation that in the context of mechanical verification, branching time has some important advantages. By setting up the refinement problem in a way that differs from the Abadi and Lamport approach, I obtain a proof of the existence of refinement maps (in the branching time framework) that does not depend on any of the conditions found in the work of Abadi and Lamport, e.g., machine closure, finite visible nondeterminism, internal continuity, the use of history and prophecy variables, etc. A direct consequence is that refinement maps always exist in the linear time framework, subject only to the use of prophecy-like variables.

1 Introduction

Computing systems are ubiquitous, consumed everything from cars and airplanes to financial markets and the distribution of information. Such systems tend to be very complicated and often contain costly errors. One approach to dealing with this complexity is to specify a sequence of related systems, starting with an abstract system, the specification, and ending with a concrete system, the implementation. One then proves that every pair of adjacent systems is related, via a suitable, compositional notion of correctness, thereby establishing that the specification is correctly implemented. For example, we can imagine verifying a netlist description of a pipelined microprocessor, the implementation, by relating it via a sequence of refinements to an instruction set level specification—the assembly programmer's view of the processor.

Two important concepts that notions of correctness must account for are:

- Stuttering. Since the specification is defined at a more abstract level than the implementation, notions of correctness should allow for stuttering: steps in the implementation may require several steps before matching a single step of the specification [14].
- **Refinement.** The implementation may contain more state components and may use different data representations than the specification. **Refinement maps** are used to show how to view an implementation state as a specification state [1].

The classic paper on the topic by Abadi and Lamport [1], which has motivated the work appearing in this paper, contains an in-depth discussion of these topics. The main idea is to use refinement maps to prove that systems have related infinite computations, by reasoning **locally**, about states and their successors, instead of **globally**, about infinite paths. Abadi and Lamport prove a theorem about when such refinement maps exist in the linear time framework, where the semantics of systems and properties correspond to sets of infinite sequences.

My approach differs in that I work in the branching time framework, where the semantics of systems are given by sets of infinite trees. Even so, the results can be applied to the linear time framework, as I explain later.

The theorem proved by Abadi and Lamport holds only under certain conditions. Briefly, they allow one to add history and prophecy variables to the implementation, they require that the implementation is machine closed, and they require that the specification has finite invisible nondeterminism and is internally continuous. My theorems do not depend on these conditions, but there are important differences between the two approaches that are explored in depth later.

There are two main reasons why I chose to work in the branching-time framework. The first is that in the simple case where one is dealing with finite-state systems, it makes sense to use algorithms that can check if one finite-state system refines another. For example, in [17] we use algorithms for deciding stuttering bisimulation to complete a proof of correctness for the alternating bit protocol (this is an infinite-state problem that was reduced to a finite-state problem using a theorem prover). The branching time notions of simulation and bisimulation, due to Milner and Park [18, 21], can be decided in polynomial time [20, 7]. In contrast, the corresponding linear time notions, trace equivalence and trace containment, are both PSPACE-complete problems [26].

Second, refinement maps allow one to show that one system **simulates** another. This is inherently a branching time notion which has the advantage of being structural and local. However, in order to use refinement maps in a linear time setting other mechanisms are needed to, in essence, hide the branching structure of systems. Thus, we expect the branching time case to be simpler than the linear time case. Obvious questions arise. How much simpler? What conditions in the Abadi and Lamport theorem are there for this purpose? It turns out that by using only prophecy-like variables, which have the effect of destroying the branching structure of systems, we can get a completeness theorem for the linear time.

Stuttering simulation is based on the notions of simulation and bisimulation, which have had a deep impact on how we think about specifications. The literature on this topic is vast and contains many fine surveys [23, 15, 6]. In ad-
dition, there have been various extensions of the Abadi and Lamport result [1],
including [5, 9, 2, 8]. In related previous work, Namjoshi [19] gives a sound
and complete proof rule for symmetric stuttering bisimulations which has heav-
ily influenced my work; however, Namjoshi does not consider simulations and
does not deal with refinement. Stuttering bisimulations and the related notion
of WEBs (Well-founded Equivalence Bisimulations) were used to link theorem
proving and model checking and to mechanically verify the alternating bit pro-
tocol in [17]. In [16], I proposed a notion of correctness for pipelined machines
based on WEBs and I showed that the variant of the Burch and Dill notion of
correctness [4] in [24, 25] can be satisfied by machines that deadlock. In addition,
I used the ACL2 theorem prover [12, 11, 10] to automate much of the verifica-
tion. I also verified variants of the pipelined machine including machines with
exceptions, interrupts (which lead to non-determinism), and netlist (gate-level)
descriptions and showed that my notion of correctness applies to these exten-
sions. Many of the variant machines were verified in stages, using the WEB
compositional proof rule. Unfortunately, stuttering bisimulation and WEBs are
often too strong a notion, just as trace equivalence is often too strong a notion in
the linear time case. I expect stuttering simulation to be much more applicable,
hence my interest in the topic.

The paper is organized as follows. In section 2, I describe my notational con-
tentions and review background material. In section 3, I develop a theory of
refinement based on stuttering simulation. In section 4, I discuss refinement in
the linear time framework and compare my work with that of Abadi and Lam-
port; some readers may want to start by skimming this section first. I conclude
in section 5.

2 Notation and Mathematical Preliminaries

$\mathbb{N}$ and $\omega$ both denote the natural numbers, i.e., \{0, 1, ...\}. The ordered pair
whose first component is $i$ and whose second component is $j$ is denoted $\langle i, j \rangle$. $[i..j]$
denotes the closed interval $\{k \in \mathbb{N} : i \leq k \leq j\}$; parentheses are used to denote
open and half-open intervals, e.g., $[i..j)$ denotes the set $\{k \in \mathbb{N} : i \leq k < j\}$. The
disjoint union operator is denoted by $\uplus$. Cardinality of a set $S$ is denoted by $|S|$. $\mathcal{P}(S)$
denotes the powerset of $S$. Function application is sometimes denoted by an
infix dot ".". For any binary relation $R$: I abbreviate $\langle s, w \rangle \in R$ by $sRw$, I write
$R(S)$ for the image of $S$ under $R$ (i.e., $R(S) = \{y : \exists x : x \in S : xRy\}$), and $R|_A$
denotes $R$ left-restricted to the set $A$ (i.e., $R|_A = \{\langle a, b \rangle : (aRb) \land (a \in A)\}$).
The composition of binary relations $R$ and $T$ is denoted $R;T$ or $T \circ R$, i.e.,
$R;T = T \circ R = \{\langle r, t \rangle : \exists x : xRx \land xTr\}$. The inverse of binary relation
$R$ is denoted $R^{-1}$ and is defined to be $\{\langle a, b \rangle : bRa\}$.

$\langle Qx : r : b \rangle$ denotes a quantified expression, where $Q$ is the quantifier, $x$ the
bound variable, $r$ the range of $x$ (true if omitted), and $b$ the body. I sometimes
write $\langle Qx \in X : r : b \rangle$ as an abbreviation for $\langle Qx : x \in X \land r : b \rangle$,
where $r$ is true if omitted, as before. From highest to lowest binding power, we
have: parentheses, function application, binary relations (e.g., $sBw$), equality
(=) and membership (∈), conjunction (∧) and disjunction (∨), implication (→),
and finally, binary equivalence (≡).

Spacing is used to reinforce binding: more space indicates lower binding.

A binary relation, \( B \subseteq X \times X \), is reflexive if \( \forall x \in X :: xBx \). \( B \) is symmetric
if \( \forall x,y \in X :: xBy \Rightarrow yBx \). \( B \) is antisymmetric if \( \forall x,y \in X :: xBy \land
yBx \Rightarrow x=y \). \( B \) is transitive if \( \forall x,y,z \in X :: xBy \land yBz \Rightarrow xBz \).
A binary relation is a preorder if it is reflexive and transitive. A preorder that
is also symmetric is an equivalence relation.

A finite sequence is a function from \([0..n]\) for some natural number \( n \). An
infinite sequence is a function from \( \mathbb{N} \). When I write \( x \in \sigma \), for a sequence \( \sigma \), I
mean that \( x \) is in the range of \( \sigma \). A well-founded structure is a pair \( \langle W,\prec \rangle \) where
\( W \) is a set and \( \prec \) is a binary relation on \( W \) such that there are no infinitely
decreasing sequences on \( W \), with respect to \( \prec \). I use \( < \) to compare natural
numbers and \( \prec \) to compare ordinal numbers.

A transition system (TS) is a structure \( \langle S,\rightarrow, L \rangle \), where \( S \) is a set of states,
\( \rightarrow \subseteq S \times S \) is the transition relation, \( L \) is the labeling function: its domain is \( S \)
and it tells us what is observable at a state. I also require that \( \rightarrow \) is left-total:
for every \( s \in S \), there is some \( u \in S \) such that \( s \rightarrow u \). Notice that a transition
system is a labeled graph where the nodes are states and are labeled by \( L \).

A path \( \sigma \) is a sequence of states such that for adjacent states \( s \) and \( u \), \( s \rightarrow u \).
A path, \( \sigma \), is a fullpath if it is infinite. \( fp,\sigma,s \) denotes that \( \sigma \) is a fullpath starting
at state \( s \) and \( \sigma' \) denotes the suffix fullpath \( \langle \sigma,i,\sigma(i+1),\ldots \rangle \). I use the symbol
";" for concatenation of paths where the left path is finite, e.g., \( \sigma;ab = aab \).

Temporal logic was proposed as a formalism for specifying the correctness of
computing systems in a landmark paper by Pnueli [22]. I assume that the reader
is familiar with temporal logic.

3 Stuttering Simulation Refinement

Stuttering simulation depends on the notion of matching I now define. I start
with an informal account. Given a relation \( B \) on a set \( S \), we say that an infinite
sequence \( \sigma \) (of elements from \( S \)) matches an infinite sequence \( \delta \) (of elements
from \( S \)) if the sequences can be partitioned into non-empty, finite segments such
that elements in related segments are related by \( B \). For example, if the first
segment of \( \sigma \) has three elements and the first segment of \( \delta \) has seven elements,
then each of the three elements is related by \( B \) to each of the seven elements. I
use matching, where the infinite sequences are fullpaths of a transition system,
to define stuttering simulation.

**Definition 1.** (match) Let \( i \) range over \( \mathbb{N} \). Let \( INC \) be the set of strictly in-
creasing sequences of natural numbers starting at 0; formally, \( INC = \{ \pi : \pi : \mathbb{N} \rightarrow \mathbb{N} \land \pi.0 = 0 \land \forall i \in \mathbb{N} :: \pi.i < \pi(i+1) \} \). The \( i^{th} \) segment of
an infinite sequence \( \sigma \) with respect to \( \pi \in INC \), \( \pi\sigma^i \), is given by the sequence
\( \langle \sigma(\pi.0),\ldots,\sigma(\pi(i+1)-1) \rangle \).

For \( B \subseteq S \times S \), \( \pi,\xi \in INC \), \( i,j \in \mathbb{N} \), and infinite sequences \( \sigma \) and \( \delta \), I
abbreviate \( \langle \forall s,w : s \in \pi\sigma^j \land w \in \xi\delta^j : sBw \rangle \) by \( (\pi\sigma^i)B(\xi\delta^i) \).
In addition: \( \text{corr}(B, \sigma, \pi, \delta, \xi) \equiv \forall i \in \mathbb{N} : (\pi^i \sigma^i) B(\xi^i \delta^i) \) and \( \text{match}(B, \sigma, \delta) \equiv \exists \pi, \xi \in \text{INC} : \text{corr}(B, \sigma, \pi, \delta, \xi) \).

**Lemma 1.** Given set \( S, B \subseteq S \times S \), and infinite sequences \( \sigma \) and \( \delta \),

\[ \exists \pi, \xi \in \text{INC} : \text{corr}(B, \sigma, \pi, \delta, \xi) \]

\[ \equiv \exists \pi', \xi' \in \text{INC} : \text{corr}(B, \sigma, \pi', \delta, \xi') \land \forall i \in \mathbb{N} : |\pi^i \sigma^i| = 1 \lor |\xi^i \delta^i| = 1) \]

The above lemma allows us to reason about segments using case analysis, where the three cases are both segments are of length 1, the right segment is of length 1 and the left of length greater than 1, and the left segment is of length 1 and the right of length greater than 1.

### 3.1 Stuttering Simulation

A relation on \( B \subseteq S \times S \) where \( \mathcal{M} = \langle S, \rightarrow, L \rangle \) is a stuttering simulation, if for every \( s, w \) such that \( sBw \), \( s \) and \( w \) are identically labeled and every fullpath starting at \( s \) can be matched by some fullpath starting at \( w \).

**Definition 2.** (Stuttering Simulation (STS)) \( B \subseteq S \times S \) is a stuttering simulation on \( TS \mathcal{M} = \langle S, \rightarrow, L \rangle \) iff for all \( s, w \) such that \( sBw \):

\[(\text{Sts1}) \quad L.s = L.w \]
\[(\text{Sts2}) \quad \langle \forall \sigma : fp.\sigma.s : \exists \delta : fp.\delta.w : \text{match}(B, \sigma, \delta) \rangle \]

**Lemma 2.** \( (B \subseteq C) \Rightarrow [\text{match}(B, \sigma, \delta) \Rightarrow \text{match}(C, \sigma, \delta)] \)

**Lemma 3.** Let \( C \) be a set of STS’s on \( TS \mathcal{M} \), then \( G = \langle \cup B : B \in C : B \rangle \) is an STS on \( \mathcal{M} \).

**Corollary 1** For every \( TS \mathcal{M} \), there is a greatest STS on \( \mathcal{M} \).

**Lemma 4.** If \( R \) and \( S \) are STS’s, so is \( T = R; S \).

**Lemma 5.** The reflexive, transitive, subsumption of an STS is an STS.

**Theorem 1.** Given \( TS \mathcal{M} \), there is a greatest STS on \( \mathcal{M} \), which is a preorder.

**Theorem 2.** Let \( B \) be a STS on \( \mathcal{M} \) and let \( sBw \). For every ACTL\(^*\) \( \backslash X \) formula \( f \), if \( \mathcal{M}, w \models f \) then \( \mathcal{M}, s \models f \).

### 3.2 Well-Founded Simulation

In order to check that a relation is an STS, we have to show that infinite sequences “match”. This can be problematic when using computer-aided veriﬁcation techniques. I present the notion of a well-founded simulation to remedy this situation. To show that a relation is a well-founded simulation, we need only check local properties; this is analogous to proving program termination by
exhibiting a function that maps states into a well-founded relation and showing that the function decreases during every step of the program. As mentioned previously, the intuition is that for every pair of states \( s, w \) that are related by an STS and \( u \) such that \( s \rightarrow u \), there are essentially three cases: either there is a \( v \) such that \( w \rightarrow v \) and \( u \) is related to \( v \), or \( u \) is related to \( w \), or there is a \( v \) such that \( w \rightarrow v \) and \( s \) is related to \( v \). In the last two cases, we must also ensure that we do not have an infinite sequence of states, each of which is related to a single state. This is where the well-founded relation comes in: we must show that in these cases there is an appropriate measure function into a well-founded relation that decreases. Formally, we have:

**Definition 3.** (Well-Founded Simulation (WFS)) \( B \subseteq S \times S \) is a well-founded simulation on TS \( \mathcal{M} = \langle S, \rightarrow, L \rangle \) iff:

(Wfs1) \( \forall s, w \in S : sBw : L.s = L.w \)

(Wfs2) There exists functions, \( \text{rankt} : S \times S \rightarrow W, \text{rankd} : S \times S \times S \rightarrow \mathbb{N} \), such that \( \langle W, \preceq \rangle \) is well-founded, and

\( \forall s, u, w \in S : sBw \land s \rightarrow u : \)

(a) \( \exists v : w \rightarrow v : uBv \lor \)

(b) \( uBw \land \text{rankt}(u, w) \preceq \text{rankt}(s, w) \lor \)

(c) \( \exists v : w \rightarrow v : sBv \land \text{rankd}(v, s, u) < \text{rankd}(w, s, u) \)

### 3.3 Equivalence

In this section, I show that well-founded simulation completely characterizes stuttering simulation. Thus, we can think of well-founded simulation as a sound and complete proof rule.

**Proposition 1** (Soundness) If \( B \) is a WFS, then it is an STS.

**Proof** Let \( aBb \); we need to show Sts1 and Sts2. \( L.a = L.b \) since \( B \) is a WFS (Wfs1), thus Sts1 holds. We show \( \forall \sigma : fp.\sigma.a : \exists \delta : fp.\delta.b : \text{match}(B, \sigma, \delta) \), namely that Sts2 holds. Suppose \( fp.\sigma.a \). We define fullpath \( \delta \) and increasing sequences \( \pi, \xi \) recursively as follows: \( \delta.0 = b, \pi.0 = 0, \xi.0 = 0 \). The idea is that from \( \pi.i, \xi.i, \delta(\xi.i) \) we can define \( \pi(i + 1), \xi(i + 1), \xi^i, \xi(\xi(i + 1)) \) with \( \pi^i, \xi \) matching. \( \square \)

We now prove that every STS is a WFS. For the proof, we have to exhibit the rank functions as per the definition of WFS. Here is a high-level overview.

The value of \( \text{rankt}(s, w) \) is important only if \( sBw \), as otherwise there are no restrictions required by the definition of WFS. If \( sBw \), then consider the largest subtree of the computation tree rooted at \( s \) such that no node in the subtree matches a successor of \( w \). The “rank” (a kind of height) of this subtree is the value of \( \text{rankt}(s, w) \). The “rank” of \( s \) is greater than the “rank” of any of its children in the tree, so case Wfs2b is satisfied.

The value of \( \text{rankd}(w, s, u) \) is important only if \( sBw \) and \( s \rightarrow u \), as otherwise there are no restrictions required by the definition of WFS. If \( sBw \) and \( s \rightarrow u \),
then \( \text{rank}(w,s,u) \) is the length of the shortest path from \( w \) that matches \( s,u \). In the case of \( \text{Wfs2c} \), we can choose the next successor of \( w \) in this path to satisfy the condition.

Given a TS \( \mathcal{M} = \langle S, \rightarrow, L \rangle \), the notion of the computation tree rooted at a state \( s \in S \) is standard. It is the tree obtained by unfolding \( \mathcal{M} \) starting from \( s \) and can be defined as follows. The nodes of the tree are finite sequences over \( S \). The tree is defined to be the smallest tree satisfying the following.

1. The root is \( \langle s \rangle \).
2. If \( \langle s, \ldots, w \rangle \) is a node and \( w \rightarrow v \), then \( \langle s, \ldots, w, v \rangle \) is a node whose parent is \( \langle s, \ldots, w \rangle \).

**Definition 4.** (tree) Given an STS \( B \), if \( \neg(sBw) \), then \( \text{tree}(s,w) \) is the empty tree, otherwise \( \text{tree}(s,w) \) is the largest subtree of the computation tree rooted at \( s \) such that for every non-root node of the tree, \( \langle s, \ldots, x \rangle \), we have that \( xBw \) and \( \langle \forall v : w \rightarrow v : \neg(xBv) \rangle \).

**Lemma 6.** Every path of \( \text{tree}(s,w) \) is finite.

Since the child relation on nodes in \( \text{tree} \) is well-founded, we can recursively define a labeling function, \( l \), that assigns an ordinal to nodes in the tree as follows: \( ln = \langle \cup c : c \text{ is a child of } n : (ln) + 1 \rangle \). This is the standard “rank” function encountered in set theory [13]. We use the convention that the label of a tree is the label of its root.

**Lemma 7.** If \( |S| \leq \kappa \), where \( \kappa \) is an infinite cardinal (i.e., \( \omega \leq \kappa \)) then for all \( s,w \in S \), \( \text{tree}(s,w) \) is labeled with an ordinal of cardinality \( \leq \kappa \).

**Lemma 8.** If \( sBw, s \rightarrow u, u \in \text{tree}(s,w) \) then \( l\text{tree}(u,w) < l\text{tree}(s,w) \).

**Definition 5.** (length) Given \( B \), an STS, \( \text{length}(w,s,u) = 0 \) if \( \neg(sBw) \) or \( \neg(s \rightarrow u) \), otherwise \( \text{length}(w,s,u) \) is the length of the shortest initial segment starting at \( w \) that matches \( \langle s,u \rangle \). Formally:

\[
\text{length}(w,s,u) = \langle \min \sigma, \delta, \pi, \xi : \text{fp} \sigma.s \land \sigma.1 = u \land \text{fp} \delta.w \land \pi, \xi \in \text{INC} \land \text{corr}(B,\sigma, \pi, \delta, \xi) : [\xi \delta^0] \rangle
\]

As \( sBw \) and \( s \rightarrow u \), the above range is non-empty and \( \text{length}(w,s,u) \in \mathbb{N} \).

**Lemma 9.** If \( sBw, s \rightarrow u \) and \( \neg(\exists \sigma, \delta, \pi, \xi : \text{fp} \sigma.s \land \sigma.1 = u \land \text{fp} \delta.w \land \pi, \xi \in \text{INC} : \text{corr}(B,\sigma, \pi, \delta, \xi) \land \xi \delta^0 = \langle w \rangle) \), then \( \langle \forall v : w \rightarrow v : \text{length}(v,s,u) < \text{length}(w,s,u) \land \text{sbv} \rangle \).

**Proposition 2** (Completeness) If \( B \) is an STS, then \( B \) is a WFS.

**Proof** Wfs1 follows from Sts1. Let \( W = (|S|+\omega)^+ \). Note that \( + \) denotes cardinal arithmetic; we add \( \omega \) to \( |S| \) to guarantee that we have an infinite cardinal; \( \kappa^+ \) is the successor cardinal to \( \kappa \).

Clearly, \( (W, \prec) \) is well-founded. Let \( \text{rankt} = l\text{tree} \) and let \( \text{rankl} = \text{length} \). Let \( sBw \) and \( s \rightarrow u \). There are three cases:
1. \( \exists v : w \rightarrow v : uBv \). By lemma 1, if (1) does not hold, then for every 
\( \sigma, \delta, \pi, \xi \) such that \( fp_{\sigma}, s \wedge \sigma_1 = u \wedge fp_{\delta}, w \wedge \pi, \xi \in INC \cap corr(B, \sigma, \pi, \delta, \xi) \),
either \( s \) marks the end of \( \sigma^0 \) or \( w \) marks the end of \( \xi^0 \), but not both.

2. \( \exists \sigma, \delta, \pi, \xi : fp_{\sigma}, s \wedge \sigma_1 = u \wedge fp_{\delta}, w \wedge \pi, \xi \in INC : corr(B, \sigma, \pi, \delta, \xi) \wedge \xi^0 = \langle w \rangle \). By lemma 9 and the definition of \( rankd, \exists \langle v : w \rightarrow v : rankd(v, s, u) < rankd(w, s, u) \wedge sBv \rangle. \)

**Theorem 3.** (Equivalence) \( B \) is an STS iff \( B \) is a WFS.

A consequence of the above theorem is that all of the properties proved for
STSs carry over to WFSs; I use this fact freely, without reference, in the sequel.

3.4 Refinement

Up to this point, I have developed a theory for relating states. I now show how
to apply the theory to transition systems. In this section, I define a notion of
refinement and show that STSs can be used in a compositional fashion. For
states \( s \) and \( w \), I write \( s \subseteq w \) to mean that there is an STS \( B \) such that \( sBw \).

By theorem 1, \( s \subseteq w \) if \( sGw \), where \( G \) is the greatest STS. I now lift this
idea to transition systems.

**Definition 6.** (Simulation Refinement) Let \( M = \langle S, \rightarrow, L \rangle, M' = \langle S', \rightarrow', L' \rangle \), and \( r : S \rightarrow S' \). We say that \( M \) is a simulation refinement of \( M' \) with
respect to refinement map \( r \), written \( M \equiv_r M' \), if there exists a relation, \( B \),
such that \( \langle \forall s \in S : sB(r.s) \rangle \) and \( B \) is an STS on the TS \( \langle S \cup S', \rightarrow \cup \rightarrow', L \rangle \),
where \( L.s = L'(s) \) for \( s \) an \( S' \) state and \( L.s = L'(r.s) \) otherwise.

In the above definition, it helps to think of \( M' \) as the specification and \( M \) as
the implementation. That \( M \) is a simulation refinement of \( M' \) with respect to
\( r \) implies that every visible behavior of \( M \) (where what is visible depends on \( r \))
is a behavior of \( M' \). There are often other considerations, e.g., it might be that
\( M \) and \( M' \) have certain states that are "initial". In this case one might wish to
show that initial states in \( M \) are mapped to initial states in \( M' \).

One has a great deal of flexibility in choosing refinement maps. The danger is
that by choosing a complicated refinement map, one can bypass the verification
problem all together. To make this point clear, let PRIME be the system whose
single behavior is the sequence of primes and let NAT be the system whose single
behavior is the sequence of natural numbers. We do not consider NAT to be an
implementation of PRIME, but using the refinement map from NAT to PRIME
that maps \( i \) to the \( i \)th prime, we can indeed prove the peculiar theorem that
NAT is a refinement of PRIME. The moral is that we must be careful to not
bypass the verification problem with the use of such refinement maps. Simple
refinement maps with a clear relationship between implementation states and
their image under the map are best. The reason we do not place restrictions
on refinement maps is that it is not a priori apparent what the "reasonable" relationships between implementation states and specification states might be, e.g., suppose that the specification system represents numbers in decimal but the implementation system represents numbers in binary, or that numbers in the specification are spread across several registers in the implementation, and so on. Often refinement maps are especially clear, which makes it easy to check that they are in fact appropriate. Suppose that associated with states is a set of variables, each of a particular type. Furthermore, suppose that the variables in the implementation are a superset of the variables in the specification and that the refinement map just hides the implementation variables that do not appear in the specification. Then, it is clear that the refinement map is a reasonable one. More precisely, given TS \( \mathcal{M} = \langle S, \rightarrow, L \rangle \), if \( L \) has the following structure, we say that \( \mathcal{M} \) is typed.

Let \( VARS \) be a set and let \( TYPE \) be a function whose domain is \( VARS \). Think of \( VARS \) as the variables of TS \( \mathcal{M} \), where \( TYPE \) gives the type of the variables. For all \( s \in S \), let \( L.s \) be a function from \( VARS \) such that \( L.s.v \in TYPE.v \). The lemma below shows why the appropriateness of refinement maps that hide some of the implementation variables is easy to ascertain.

**Lemma 10.** If \( \mathcal{M} = \langle S, \rightarrow, L \rangle \subseteq_{r} \mathcal{M}' = \langle S', \rightarrow', L' \rangle \), both \( \mathcal{M} \) and \( \mathcal{M}' \) are typed TSs, and \( L'(r,s) = L.s \mid V \), then for every pair of states \( s, r, s \) such that \( s \in S \), and every ACTL* \( \setminus X \) formula, \( f \), built out of expressions that only depend on variables in \( V \), we have \( \mathcal{M}', r.s \models f \Rightarrow \mathcal{M}, s \models f \).

**Lemma 11.** If \( B \) is an STS on TS \( \mathcal{M} = \langle S \supseteq S_1 \cup S_2, \rightarrow, L \rangle \), \( S_1 \cap S_2 = \emptyset \), states in \( S_1 \) can only reach states in \( S_1 \), and states in \( S_2 \) can only reach states in \( S_2 \), then \( \hat{B} = \{ (s_1, s_2) : s_1 \in S_1 \land s_2 \in S_2 \land s_1 B s_2 \} \) is an STS on \( \mathcal{M} \).

**Theorem 4.** (Composition) If \( \mathcal{M} \subseteq_{r} \mathcal{M}' \) and \( \mathcal{M}' \subseteq_{q} \mathcal{M}'' \) then \( \mathcal{M} \subseteq_{r \cdot q} \mathcal{M}'' \).

4 The Linear Time Case

The theorem on the existence of refinement maps in the previous section does not apply to the linear time framework because simulation is a stronger property than trace containment. However, note that if we destroy the branching structure of transition system \( \mathcal{M} \) to obtain transition system \( \mathcal{M}' \), then \( \mathcal{M}' \subseteq_{r} \mathcal{N} \) iff the set of infinite sequences of \( \mathcal{M} \), labeled by \( r \), is a subset of the set of sequences of \( \mathcal{N} \). We can destroy the branching structure of \( \mathcal{M} \) by using an oracle variable to record values for every non-deterministic choice made along an infinite path in the computation tree of \( \mathcal{M} \). We have thus sketched a proof of the existence of refinement maps in the linear time framework.

**Theorem 5.** If the set of traces of \( \mathcal{M} \) is a subset of the traces of \( \mathcal{N} \), then there exists \( \mathcal{M}' \), a transition system obtained from \( \mathcal{M} \) by adding an oracle variable, and a refinement map \( r \) such that \( \mathcal{M}' \subseteq_{r} \mathcal{N} \).
I now review the work of Abadi and Lamport on the existence of refinement maps. The review addresses the essential points, but is necessarily concise and readers are urged to read the full paper. I then present several examples, taken from Abadi and Lamport, that are used to justify the conditions appearing in their theorem. At the end of this section, I compare the two approaches.

4.1 Review of Abadi and Lamport Results

I begin by reviewing some initial definitions. A **behavior** is an infinite sequence and a **property** is a set of behaviors closed under finite stuttering. A **specification** is a (possibly infinite) state machine, consisting of externally visible components and internal components, and a **supplementary** property to represent fairness constraints. The **complete property** of a state machine is obtained by closing the set of behaviors allowed by the machine under (possibly infinite) stuttering. The **externally visible property** of a state machine is obtained by projecting the externally visible components of the complete property of the state machine. The **property** defined by a specification is obtained by intersecting the complete property of its state machine with the supplementary property. The **externally visible property** of a specification is obtained by projecting the externally visible components of the property of the specification.

We say that $I$, a “concrete” specification (the Implementation), **implements** $S$, an “abstract” specification (the Specification) if every externally visible behavior of $I$ is also a behavior of $S$. Proving that $I$ implements $S$ can require reasoning about arbitrary sequences because one has to show that if $I$ admits the behavior $\langle (e_0, z_0), (e_1, z_1), \ldots, (e_n, z_n), \ldots \rangle$, where the $e_i$ correspond to the externally visible components and the $z_i$ to the internal components, then $S$ admits the behavior $\langle (e_0, y_0), (e_1, y_1), \ldots, (e_n, y_n), \ldots \rangle$. Notice that $y_n$ can depend upon the entire sequence $\langle (e_0, z_0), (e_1, z_1), (e_2, z_2), \ldots \rangle$, which can make the proof difficult. We prefer to avoid such global reasoning and rather reason locally e.g., if there is a function $f$ such that $\langle e_i, y_i \rangle = f(e_i, z_i)$, it can be used to prove that $I$ preserves the safety property of $S$ by reasoning about pairs of states instead of arbitrary sequences of states. If such a function also preserves liveness, it is called a **refinement mapping** and Abadi and Lamport prove the following completeness theorem, showing under what conditions refinement mappings exist.

**Theorem 6.** If the machine-closed specification $I$ implements $S$, a specification that has finite invisible nondeterminism and is internally continuous, then there is a specification $I^h$ obtained from $I$ by adding a history variable and a specification $I^{hp}$ obtained from $I^h$ by adding a prophecy variable such that there exists a refinement mapping from $I^{hp}$ to $S$.

The above theorem depends on various conditions, which I now explain. We say that a specification $I$ is **machine-closed** if the supplementary property of $I$ does not specify any safety property not already specified by the state machine of $I$. A specification $S$ has **finite invisible nondeterminism** if for every finite
4.2 Examples Due to Abadi and Lamport

This section contains several examples that Abadi and Lamport use to explain the conditions found in their completeness theorem. After the examples are introduced, I show how they can be handled using in my framework.

In the first example, system $S$ is a three-bit clock, where only the low-order bit is externally visible and system $I$ is a one-bit clock. $I$ implements $S$ since they have the same traces (up to stuttering). However, no refinement mapping can be used to show this because there is no way to define the internal state of $S$: consider an arbitrary refinement mapping, $r$, and suppose that $r(\langle 0 \rangle) = \langle 0, y_0 \rangle$ and $r(\langle 1 \rangle) = \langle 1, y_1 \rangle$, then either $\langle 0, y_0 \rangle$ does not transit to $\langle 1, y_1 \rangle$ or $\langle 1, y_1 \rangle$ does not transit to $\langle 0, y_0 \rangle$. This is one reason for introducing history variables and they are used to resolve the dilemma as follows. A history variable is added to $I$ and the variable "remembers" what $I$ did in the past. The result is that the state space of $I$ is expanded so that there are enough states to define an appropriate refinement mapping.

Using the approach outlined in this paper, we find that history variables are not needed as we can define a refinement map that maps the state in $I$ whose counter is 0 to any state in $S$ whose low-order bit is 0 and similarly with the other state in $I$. The equivalence relation that relates states with the same low-order bit in the disjoint union of the two systems is a stuttering simulation.

The second example is used to motivate the need for prophecy variables. System $S$ chooses ten values non-deterministically and displays each in turn, whereas system $I$ chooses each value as it is displayed. $I$ implements $S$ since they have the same traces, but there is no refinement mapping that can be used to show this, as should be clear. This example highlights that proofs based on refinement mappings are based on simulation, a branching time notion. Thus, when $I$ is not a stuttering simulation of $S$, one cannot directly use refinement
mappings to prove that \( \mathcal{I} \) implements \( \mathcal{S} \) (in the linear time sense). This is one reason for introducing prophecy variables and they are used to resolve the dilemma as follows. A prophecy variable is added to \( \mathcal{I} \) and the variable "guesses" what \( \mathcal{I} \) will decide to do in the future. There is now a refinement map, based on this prophecy variable, that can be used to show that \( \mathcal{I} \) implements \( \mathcal{S} \). What is happening is that the prophecy variables allow one to push all of the branching in the computation tree of \( \mathcal{I} \) up to the root, thereby destroying the branching structure of \( \mathcal{I} \).

This example shows why oracle variables are used in theorem 5. Note that from the branching point of view \( \mathcal{I} \) does not implement \( \mathcal{S} \), e.g., from the initial state in \( \mathcal{I} \), there is a successor that has more than one possible future, a branching-time expressible property that does not hold in the initial state of \( \mathcal{S} \). It seems that any refinement-based approach will need a mechanism for dealing with this issue, whether it is by destroying the branching structure of implementations, by adding branching structure to specifications, or by some combination thereof.

The third example shows why a prophecy variable is needed to slow down an implementation that runs faster than a specification, even though the specification is just stuttering. Both \( \mathcal{I} \) and \( \mathcal{S} \) specify clocks in which the hours and minutes are externally visible, whereas the seconds are internal. Furthermore, \( \mathcal{I} \) increments the clock by one second, whereas \( \mathcal{S} \) increments the clock by ten seconds. Both \( \mathcal{I} \) and \( \mathcal{S} \) have the same externally visible behaviors and proving that \( \mathcal{S} \) implements \( \mathcal{I} \) using refinement mappings is easy. However, there is no way to show that \( \mathcal{I} \) implements \( \mathcal{S} \), because there is a behavior of \( \mathcal{S} \) such that the minute hand changes every six steps, but any behavior of \( \mathcal{I} \) requires at least sixty steps between minute hand changes.

In my formulation, the implementation is allowed to run faster than the specification, as we can both add and remove stuttering steps, thus it is easy to deal with the third example.

Abadi and Lampert present examples showing why the conditions of finite invisible nondeterminism and internal continuity are required. The examples are similar in that the implementation, \( \mathcal{I} \), has the same externally visible behaviors as the specification, \( \mathcal{S} \), but \( \mathcal{I} \) has a richer branching structure than \( \mathcal{S} \), i.e., \( \mathcal{S} \) is a simulation refinement of \( \mathcal{I} \), but not the other way. As we have seen in the second example, above, prophecy variables can be used to deal with this problem. However, in these examples there are states in \( \mathcal{I} \) that are related to an infinite number of states in \( \mathcal{S} \), and Abadi and Lampert’s prophecy variables cannot be used in this case (see their paper for the full details). To summarize, the conditions of internal continuity and finite invisible nondeterminism in the completeness theorem of Abadi and Lampert can be traced to the branching structure of the systems involved.

Oracle variables can be used in my approach to deal with these examples. The intuition is that oracle variables allow us to quantify over every possible nondeterministic choice and can be used to transform \( \mathcal{I} \) into a linear time equivalent system in which all nondeterministic choices have been made at the onset.
4.3 Comparison with the Approach of Abadi and Lamport

There are various differences between my approach and that of Abadi and Lamport. A major difference is that I deal with branching time notions because in the context of mechanical verification they provide certain advantages, as outlined above. However, in order to simplify the comparison, in this section I consider only the linear time aspects of my results.

There are differences in how stuttering is dealt with; namely, Abadi and Lamport allow infinite stuttering, whereas I do not. Consider the example of pipelined machine verification. Using the Abadi and Lamport approach, we would define the instruction set architecture using a state machine, say where every component is externally visible. By definition, the property generated by the state machine includes infinite stuttering, e.g., it includes the behavior where nothing happens. Thus, a supplementary property would be used to rule out such behaviors by requiring that non-stuttering steps are eventually taken, a liveness property. In contrast, in my approach, every step of the transition system modeling the instruction set architecture corresponds to the execution of an instruction, with the stuttering being handled by the definition of stuttering simulation. Notice that no supplementary property is required. In addition, the condition that a pipelined machine makes progress is now a safety property, because the number of steps required is bounded by the number of stages in the pipeline [16].

Lamport and Abadi require that systems have the same externally visible states. They make the point that one cannot say whether the value 11111100 corresponds to $-3$ without knowing how to interpret a sequence of bits as an integer. They go on to say that given such an interpretation, they can translate the externally visible states to the appropriate representation. In my case, instead of having a separate interpretation phase, I allow refinement maps to alter the labels of states directly. I have found that in practice this extra power is necessary. For example, when proving that a pipelined machine implements the instruction set architecture, I have used refinement maps that either modify the value of the program counter (when using my "commit" approach to correctness) or modify the register file and memory (using the Burch and Dill "flushing" approach to correctness) [16]. The point is that when using my commit approach to correctness, if we consider the program counter to be externally visible then we cannot use the Abadi and Lamport approach to prove that a pipelined machine implements the instruction set architecture. Similarly, when using the Burch and Dill approach, if we consider the register file or memory to be externally visible, then we cannot use the Abadi and Lamport approach to prove that a pipelined machine implements the instruction set architecture.

The refinement mappings of Abadi and Lamport are required to preserve the supplementary property of the specification. As they point out, this is not a local condition, but one can apply local methods such as well-founded induction for the proof. Unfortunately, they do not provide any guidance on constructing such arguments. In my case, the proof of proposition 2 (if $B$ is an STS, then $B$ is a WFS) shows how to construct the appropriate well-founded relations
and measure functions, \( \text{rank}_{\text{d}} \) and \( \text{rank}_{\text{f}} \). The proof also shows that two measure functions, one from pairs of states and one from triples of states to the naturals, are enough regardless of the transition systems involved.

Finally, my theorems are stronger than the ones given by Abadi and Lamport. For example, they show that even when \( S \) is not internally continuous a refinement map exists to show that \( I \) satisfies the safety property specified by \( S \). They continue “We do not know if anything can be said about proving arbitrary liveness properties.” Since my refinement theorems apply to any systems, a simple corollary is that, with my approach, refinement maps can always be used to prove both safety and liveness properties. This is something that we used in [17] where we used theorem proving to reduce an infinite-state system to a finite-state system in such a way that stuttering-insensitive properties, including liveness, were preserved. We then model checked the reduced system and were able to lift the results to the original system.

5 Conclusions

I have introduced compositional notions of refinement for stuttering simulation. I have shown that if one system refines another in the branching time framework, then a refinement map always exists, without relying on any of the conditions present in the approach taken by Abadi and Lamport, e.g., machine closure, finite invisible nondeterminism, internally continuity, the use of history and prophecy variables, etc. I also showed that refinement maps always exist in the linear time framework, subject only to the use of oracle variables.

My main motivation is the mechanical verification of systems. Notions of refinement based on stuttering bisimulation have proved useful for this purpose [17, 16]. However, stuttering bisimulation is applicable only in limited contexts, as usually specifications contain more nondeterminism than implementations. Thus, I expect that stuttering simulation will turn out to be more useful than stuttering bisimulation.

References


Linear and Nonlinear Arithmetic in ACL2

Warren A. Hunt, Jr., Robert Bellarmine Krug, and J Moore

Department of Computer Sciences
University of Texas at Austin
Austin, TX 78712-1188, USA
{hunt,rkrug,moore}@cs.utexas.edu

Abstract. As of version 2.7, the ACL2 theorem prover has been extended to automatically verify sets of polynomial inequalities that include nonlinear relationships. In this paper we describe our mechanization of linear and nonlinear arithmetic in ACL2. The nonlinear arithmetic procedure operates in cooperation with the pre-existing ACL2 linear arithmetic decision procedure. It extends what can be automatically verified with ACL2, thereby eliminating the need for certain types of rules in ACL2’s database while simultaneously increasing the performance of the ACL2 system when verifying arithmetic conjectures. The resulting system lessens the human effort required to construct a large arithmetic proof by reducing the number of intermediate lemmas that must be proven to verify a desired theorem.

1 Introduction

Mechanical theorem proving or proof checking systems offer a rigorous methodology with which to structure and check proofs. Each such system offers a different degree of automation—directly affecting its capability and ease of use. We have extended the ACL2 theorem proving system [7,8,9] with an automated verification procedure that enhances the linear arithmetic decision procedure. ACL2 can now more easily verify sets of inequalities containing nonlinear arithmetic relationships.

In this paper we describe our mechanization of linear and nonlinear arithmetic in ACL2. Before doing so, we briefly describe the theory behind the procedures and provide a couple of trivial examples of their use. The procedures operate on inequalities over the rationals. These inequalities can be combined by cross-multiplication and addition to permit the deduction of an additional inequality. For example, if \(0 < \text{poly1}\) and \(0 < \text{poly2}\), and \(c\) and \(d\) are positive rational constants, then \(0 < c \cdot \text{poly1} + d \cdot \text{poly2}\). Here, we are use two facts: multiplication by a positive rational constant does not change the sign of a polynomial and the sum of two positive polynomials is itself positive. This is linear arithmetic. We also have that \(0 < c \cdot \text{poly1} \cdot \text{poly2}\). In this nonlinear case, we are using the fact that the product of two positive polynomials is itself positive.

Now suppose we want to prove

\[3 \cdot x + 7 \cdot a < 4 \quad \land \quad 3 < 2 \cdot x \quad \Rightarrow \quad a < 0.\]
To do this, we assume the two hypotheses and the negation of the conclusion, and look for a contradiction. We therefore start with the three inequalities:

\begin{align*}
0 &< -3 \cdot x + -7 \cdot a + 4 \\
0 &< 2 \cdot x + -3 \\
0 &\leq a.
\end{align*}

We cross-multiply and add the first two – that is, multiply inequality (1) by two and inequality (2) by three, and then add the respective sides. This yields

\[ 0 < -14 \cdot a + -1. \]

Note that the new inequality does not mention \( x \). If we choose two inequalities with the same leading term and leading coefficients of opposite sign, we can generate an inequality in which that leading term is not present. This is the general strategy employed by the linear arithmetic decision procedure.

If we next cross-multiply and add inequality (4) with inequality (3), we get

\[ 0 < -1, \]

a false polynomial. We have, therefore, proved our theorem.

This process illustrated above of cross-multiplying and adding two inequalities will be referred to as “cancelling” the two inequalities. We shall refer to such obviously false inequalities as (5) as “contradictions,” and speak of any process that results in one of these as “generating a contradiction.”

Next, suppose that we have the three assumptions

\begin{align*}
3 \cdot x \cdot y + 7 \cdot a &< 4 \quad \text{or} \quad 0 < -3 \cdot x \cdot y + -7 \cdot a + 4 \\
3 &< 2 \cdot x \quad \text{or} \quad 0 < 2 \cdot x + -3 \\
1 &< y \quad \text{or} \quad 0 < y + -1,
\end{align*}

and we wish to prove that \( a < 0 \). We proceed by assuming the negation of our goal, \( 0 <= a \), and looking for a contradiction.

Note that in this case no two inequalities have a leading term in common. In this situation there are no cancellations to perform. However, (6) has a product as its leading term, \( x \cdot y \), and for each of the factors of that product, \( x \) and \( y \), there is an inequality which has such a factor as a leading term. When nonlinear arithmetic is enabled, ACL2 will multiply (7) and (8), obtaining

\[ 0 < 2 \cdot x \cdot y + -3 \cdot y + -2 \cdot x + 3. \]

The addition of this polynomial will allow cancellation to continue\(^1\) and, in this case, we will prove our goal. Thus, just as ACL2 adds two polynomials when they have the same largest unknown of opposite signs in order to create a new smaller polynomial, ACL2 can now multiply polynomials when the product of their largest unknowns is itself the largest unknown of another polynomial.

\(^1\) Inequality 9 can be canceled with 6. The result can be canceled with 8, and so on. The final cancellation will be with the negation of our goal, \( 0 <= a \).
1.1 Related Work and Plan of the Paper

It is often desirable to verify the correct operation of computer hardware or software. These operations may involve arithmetic, as in the floating-point hardware of a modern microprocessor or pointer arithmetic in a C program.

Several approaches to automating the verification of arithmetic lemmas have been tried. Great progress has been made as is illustrated by the many substantial proofs recently completed in PVS, HOL, and ACL2 [10,5,13]. The existing state-of-the-art is, however, not sufficient. The level of user expertise and effort required for the above-mentioned work is too high.

One of the primary difficulties encountered has been the fact that the formulae to be proved are rarely limited to just the four basic arithmetic operations, +, −, *, and /, but often involve more diverse semantic constructs or, at the least, user-defined functions. Theorem provers, therefore, cannot limit themselves to “pure” arithmetic but must work with combinations of theories.

Our approach has developed from an engineering, results-oriented perspective, and we have therefore concentrated on decreasing the user’s effort for the types of lemmas we see ACL2 users attempting to prove. Others have taken a more theoretical approach, whereby they can guarantee algorithmic completeness\(^2\) over an exactly specified domain.

Several groups have built such systems by combining small-domain specific provers. Nelson and Oppen [11] and Shostak [14] describe frameworks with which one can combine separate existing decision procedures into one larger procedure. This work has been extended by others; e.g., Kapur [6] and SRI [12]. While this approach has some nice properties, such as completeness and efficiency, it can be somewhat limiting. Some of these limitations arise because the procedures to be integrated are treated as fixed black-boxes. Armando and Ranise [1] describe a method for augmenting the black-boxes. Other limitations arise from concerns over efficiency. Harrison [4] explores the use of a full decision procedure over the reals and discusses its desirability.

We build on earlier work by Boyer and Moore [2] and share a common design philosophy with theirs. We regard ACL2’s various procedures as a mutually recursive nest of functions, and have tuned both the interfaces and internals of these procedures using feedback from users to guide the process. It is also similar to work by Cyrluk and Kapur [3]; they too were concerned with augmenting existing linear arithmetic decision procedures to handle nonlinear inequalities, and their design was also driven by engineering rather than theoretical concerns. We have the advantage of possessing much faster computers than were available at that time and believe that the time has come to reexamine the feasibility of more ambitious, but still incomplete algorithms, for handling nonlinear arith-

---

\(^2\) A decision procedure is said to be complete if it always returns a (correct) answer when asked to verify a true theorem. An incomplete procedure, by contrast, may return an “I don’t know” or even not return at all.
metric\textsuperscript{3}. In this paper we present our first attempt at fully integrating a nonlinear arithmetic semi-decision procedure into ACL2. We present merely an outline of our work, and do not discuss nor even mention many of the heuristics that we have employed to limit and guide our algorithms.

We provide the required background including the definitions of polys, pots, and labels, as well as a short discussion of type reasoning and linearization. Thereafter we describe the subprocedures that make up the linear and nonlinear arithmetic procedures. The linear arithmetic procedure consists of two nested loops. The innermost of these, the linear arithmetic decision procedure (described in Section 2) is responsible for adding inequalities to the pot-list. In Section 3 we present linear arithmetic’s outer loop, the linear lemmas procedure which attempts to gather additional inequalities in order to allow further cancellations. The nonlinear arithmetic procedure consists of three nested loops. The innermost is the same as for linear arithmetic. The next, described in Section 4, is an augmentation of that described in Section 3. Nonlinear arithmetic’s outer loop is presented in Section 5. We conclude with a few remarks about the labor that can now be saved.

1.2 Polys, Pots, Pot-lists, and Labels

The procedures we will describe here operate on polynomial inequalities over the rationals. A “polynomial” is a sum of terms, each of which is either a rational constant or the product of a rational constant and an “unknown.” An example polynomial is $3 \cdot x - 7/2 \cdot a + 2$; here $x$ and $a$ are the unknowns. The unknowns, however, need not be variable symbols; e.g., $|x|$, $x^n$, or $f(x, y)$ may be used as unknowns. Thus, $-3 \cdot |x| + a$ is also a polynomial.

A “polynomial inequality,” or a “poly” for short, is an inequality (either $<$ or $\leq$) between 0 and a polynomial; e.g., $0 < 3 \cdot x - 7/2 \cdot a + 2$ and $0 \leq -3 \cdot |x| + a$ are polys. We refer to obviously false polys such as $0 < 0$ as “contradictions.”

Polys are stored in groups called “pots.” All the polys with the same largest unknown\textsuperscript{4} are stored in a single pot, which is said to be “labeled by” or “about” that unknown. These pots are further divided into two compartments—one for “positive” polys (with a positive leading coefficient) and one for “negative” polys (with a negative leading coefficient). A pot represents the conjunction of the polys in it.

The pots are stored in a “pot-list,” which represents the conjunction of the pots in it. An example\textsuperscript{5} is:

\textsuperscript{3} One simple example of incompleteness is our inability to (automatically) prove $x \cdot x \neq 2$. If we were operating over the reals, $x = \sqrt{2}$ would be a solution, but recall that we are operating over the rationals. The authors have not done a study of how to demarcate the class of formulas on which our algorithms succeed or fail. They are, however, unaware of any examples which “should” be proveable using these techniques but which are not proveable for reasons other than limiting heuristics.

\textsuperscript{4} The order used here is basically lexicographic, considering number of variables first, number of function symbols second, and alphabetical order last.

\textsuperscript{5} Note that there are cancellations which can be performed.
We refer to the $b$ and $f(x, y)$ above as their pots’ “labels.”

The procedures that we will describe here all take, among other arguments, a pot-list and a list of polys to be added to the pot-list. They return either an augmented pot-list or a contradiction (a false poly) – the latter case indicating success as in Section 1.

1.3 Type Reasoning and Polys from Type-set

We shall treat type reasoning – carried out by calling the ACL2 function type-set – as something of a black-box. For present purposes only a few things need be known about it.

First, we use it here to quickly answer the question “To what arithmetic category does this expression belong?” where the possible answers are zero,\(^6\) positive integer, negative integer, positive ratio, negative ratio, or combinations thereof such as non-negative rational.

Second, we can sometimes form polys about an expression based on the answer given. For example, if $x$ is said to be a nonpositive rational, we can create the poly $0 \leq -1 \cdot x$ from that information. We refer to this mechanism as “creating polys from type-set.”

Third, although type-set’s reasoning abilities are fairly limited, they can be extended through the use of type-prescription rules. ACL2 comes with some of these already built in including rules about the basic arithmetic functions such as that $x \cdot y$ is a positive rational if both $x$ and $y$ are. This rule can, via the above-mentioned mechanism of creating polys from type-set, provide a small amount of nonlinear reasoning to the linear arithmetic procedures. We shall see this shortly.

1.4 Linearization

Linearization is the process of converting an ACL2 term into one or more polys. We note the following:

1. An equality can be expressed as a conjunction of two inequalities; $x = y$ is true if and only if both $x \leq y$ and $y \leq x$ are true.\(^7\).
2. We normalize polys so that their leading coefficient is $/+1$.
3. Consider the ACL2 term\(^8\) $(< x y)$. If we know that both $x$ and $y$ are integers, we can assume that we are linearizing $(\leq (\times x 1) y)$ instead, and

\[^6\] A category with only one member.
\[^7\] Note that the negation of an equality can similarly be expressed as a disjunction of two inequalities. We do not address this further in the present paper except to say that ACL2 does handle such situations.
\[^8\] ACL2 terms are a subset of Lisp expressions, and therefore use a Lisp-style prefix syntax.
so convert \((< \times y)\) to the poly \(0 \leq y + -1 \cdot x + -1\) rather than the weaker \(0 < y + -1 \cdot x\). We shall refer to this as “the 1+ trick.” This is the only place in which the procedures described here take advantage of the discreteness of the integers.

## 2 Linear Arithmetic

In this section, we describe the innermost loop of ACL2’s arithmetic procedures. The algorithm described in this section is a decision procedure for linear arithmetic over the rationals. We later refer to this as the linear arithmetic algorithm.

As in the examples of Section 1, our goal is to derive a contradiction. In order to do so, all of the unknowns of a poly must be eliminated by cancellation. We can choose to eliminate them in any order, but we eliminate the first. That is, two polys are canceled against each other only when their largest unknowns are the same and have coefficients of opposite signs. Note that this occurs precisely when two polys are (or will be) in opposite sides of the same pot.

### 2.1 The Linear Arithmetic Algorithm

We start with a (possibly empty) pot-list and a list of polys to be added to it. We repeat the following until we reach a fixed point.

1. For each poly to be added:
   - find its pot (the one whose label matches the poly’s largest unknown), if there is one, or make a new one. Add the poly to this pot and cancel the new poly with any polys of the opposite sign. If this generates a contradiction, quit and return the contradiction; otherwise set any new polys aside.
2. If there were any new polys set aside in step 1, go back to step 1 with the new polys. Otherwise, go on to step 3.
3. For each pot that is new or has been changed by having polys added to it in step 1, try to create a poly from \texttt{type-set} about the label of the changed pot. Collect any such newly created polys and return to step 1 with them.

### 2.2 An Example

Suppose that we want to prove

\[
\text{integer } a \land \text{ integer } b \land 0 \leq a \land a < b \implies a + 1 \leq a \cdot b + b
\]

As before, we assume the hypotheses and the negation of the conclusion, and look for a contradiction. Since \(a\) and \(b\) are assumed to be integers, the linearization of \(a < b\) is \(0 \leq b + -1 \cdot a + -1\). Similarly \(a \cdot b\) is known to be an integer since \(a\) and \(b\) are, so the linearization of the negation of \(a + 1 \leq a \cdot b + b\) is \(0 < -1 \cdot a \cdot b + -1 \cdot b + a\). Both of these linearizations used the 1+ trick. Finally, the linearization of \(0 \leq a\) is just \(0 \leq a\).
Since no cancellations can be performed between these three polys, executing steps 1 and 2 above results in the pot-list

<table>
<thead>
<tr>
<th>label</th>
<th>positives</th>
<th>negatives</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a$</td>
<td>$0 \leq a$</td>
<td></td>
</tr>
<tr>
<td>$b$</td>
<td>$0 \leq b + -1 \cdot a + -1$</td>
<td></td>
</tr>
<tr>
<td>$a \cdot b$</td>
<td>$0 &lt; -1 \cdot a \cdot b + -1 \cdot b + a.$</td>
<td></td>
</tr>
</tbody>
</table>

In step 3 we create three polys from \texttt{type-set}. There are three new (and therefore changed) pots, $a$, $b$, and $a \cdot b$. \texttt{Type-set} knows that $a$ is a nonnegative integer and that $b$ is a positive integer\footnote{The variable $a$ is nonnegative by hypothesis, and since $b$ is strictly greater than $a$, $b$ must be positive. This is about as complicated as type-reasoning gets.} and so we create the polys $0 \leq a$ and $0 < b$. As mentioned in 1.3, \texttt{type-set} therefore also knows that $a \cdot b$ is a nonnegative integer, and so we create the poly $0 \leq a \cdot b$. This small amount of nonlinear reasoning has long been built into ACL2. Note that we used the pot labels to guide our search for additional polys.

When we add these to the pot-list, after executing step 1 \emph{once}, we get

<table>
<thead>
<tr>
<th>label</th>
<th>positives</th>
<th>negatives</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a$</td>
<td>$0 \leq a$</td>
<td></td>
</tr>
<tr>
<td>$b$</td>
<td>$0 &lt; b$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$0 \leq b + -1 \cdot a + -1$</td>
<td></td>
</tr>
<tr>
<td>$a \cdot b$</td>
<td>$0 \leq a \cdot b$</td>
<td>$0 &lt; -1 \cdot a \cdot b + -1 \cdot b + a$</td>
</tr>
</tbody>
</table>

with the poly $0 < -1 \cdot b + a$ having been set aside. This poly is the result of cancelling the two polys in the $a \cdot b$ pot.\footnote{Note that cancellation does not remove any polys. We augment, but never diminish, the pot-list.} Upon adding it and canceling the polys in the $b$ pot (executing steps 2 and 1 again), we get the contradiction $0 < -1$ and our lemma is proved.

3 Linear Lemmas

Prior to version 2.7, ACL2’s arithmetic procedure encompassed little more than is described in this section. It is still the standard behavior of ACL2 when nonlinear arithmetic is disabled. Note that this procedure is not complete.

Suppose that the procedure described above does not produce a contradiction but instead yields a set of nontrivial polys. A contradiction might still be generated if we could add to the set some additional polys which allow further cancellation. That is where linear lemmas come in. Linear lemmas are more general and powerful than polys from type-set. (An example follows shortly.) When the set of polys has stabilized under the procedure described above and no contradiction has been produced, we form a list of the labels of any newly created pots and search the database of linear rules for ones that pattern match with a pot label. For each rule found, if we are able to relieve its hypotheses, we add its
conclusion to the pot-list (using the above linear arithmetic algorithm) in the hope that this will allow further cancellations to proceed. Just as for polys from \textit{type-set}, we are using pot labels to guide our search for additional polys. Such labels, recall, correspond to the unknowns that are candidates for cancellation.

3.1 The Linear Lemmas Algorithm

As before, we start with a list of polys and a (possibly empty) pot list. We repeat the following until we reach a fixed point or are interrupted by the user aborting the proof attempt.

1. Add the polys with the linear arithmetic algorithm as described in section 2.1; if no contradiction was generated, go on to step 2.
2. Make a list of the labels from any new pots created in step 1 (or 3). If there aren’t any, quit and return the pot-list; otherwise, go on to step 3.
3. For each item in this list and for each applicable linear-lemma:
   - If we can relieve the lemma’s hypotheses, add the concluding poly(s) to the pot-list as described in section 2.1.
4. If a contradiction was generated, quit and return it. Otherwise, go back to step 2.

3.2 An Example

Suppose that we are given the following linear lemma,\textsuperscript{11} \texttt{expt-lemma}, about $x^n$

$$1 < x \land \text{ integer } n \land 1 < n \implies x < x^n,$$

and that we wish to prove

$$2 < x \land \text{ integer } n \land 1 < n \land a \leq x + b \implies a < x^n + b.$$

After linearizing the inequalities among the hypotheses and the negation of the conclusion and adding them to the empty pot-list (step 1) we get

<table>
<thead>
<tr>
<th>label</th>
<th>positives</th>
<th>negatives</th>
</tr>
</thead>
<tbody>
<tr>
<td>$b$</td>
<td>$n &lt; 1$</td>
<td>$0 &lt; -1 \cdot b + a$</td>
</tr>
<tr>
<td>$x$</td>
<td>$0 \leq x + b - 1 \cdot a$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$0 &lt; x + 2$</td>
<td></td>
</tr>
<tr>
<td>$x^n$</td>
<td>$0 &lt; x^n$</td>
<td>$0 \leq -1 \cdot x^n + 1 \cdot b + a$</td>
</tr>
</tbody>
</table>

Note that the poly $0 < x^n$ was created from \texttt{type-set} about the pot-label $x^n$.

\textsuperscript{11} Note that the conclusion of this lemma does not encode a type such as positive integer, and so could not be expressed as a type-prescription rule. We also mention here that the hypotheses of a linear lemma may be relieved by general purpose rewriting and (recursively) linear and nonlinear arithmetic, while a type-prescription rule’s hypotheses must be relieved by type-reasoning only.
In step 2, we note that there were four new pots created in step 1, and in step 3 we will eventually find \texttt{expt-lemma}. We sketch here how we relieve the first hypothesis, \( 1 < x \). Rewriting cannot do anything with this, so we linearize the negation of the hypothesis \((\text{yielding } 0 \leq -1 \cdot x + 1)\) and recursively call the very procedure we are describing. In this situation we do not start with an empty pot-list. This poly will be added to the \( x \) pot. Upon cancellation with \( 0 < x + -2 \), we get the contradiction \( 0 < -1 \), and the hypothesis has been relieved.

We therefore add the linearization of the conclusion of \texttt{expt-lemma}, \( 0 < x^n + -1 \cdot x \), to the pot-list. After a couple of rounds of cancellation we derive the contradiction \( 0 < -2 \), and the theorem has been proved.

## 4 Linear Lemmas Revised

When nonlinear arithmetic is enabled, we do the above procedure a little differently. The gathering of polys from linear lemmas is intended to let the process of cancellation continue. In the procedure described in this section we still use linear lemmas, but we intertwine their use with other ways of gathering polys in preparation for what is to come — the nonlinear arithmetic procedure.

### 4.1 Exploded Pot Labels, Bounds Polys, and Inverse Polys

Previously, we examined pot labels to direct our gathering of additional polys from such sources as \texttt{type-set} and linear lemmas. That is, when there was a pot labeled with, say, \( x^n \), we looked to \texttt{type-set} or linear lemmas for additional information about \( x^n \). We shall soon examine “exploded” pot labels. These exploded pot labels consist of the original pot label and, if the pot label is a product, each of the label’s factors. A few examples will make this clearer:

\[
\begin{align*}
- x & \implies x \\
- \lfloor x \rfloor & \implies \lfloor x \rfloor \\
- x \cdot \lfloor x \rfloor & \implies x, \lfloor x \rfloor, \text{ and } x \cdot \lfloor x \rfloor \\
- x \cdot y \cdot z & \implies x, y, z, \text{ and } x \cdot y \cdot z
\end{align*}
\]

We are doing this so that we can seed the database with information about the factors of products. Note that in the last example we do not examine, for instance, \( x \cdot y \).

A “bounds poly” is a poly with exactly one unknown and can be considered to bound the unknown. For instance, \( 0 < x + 1 \) can be considered to give a lower bound of \(-1\) for \( x \). Similarly, \( 0 < -1 \cdot x + 3 \) bounds \( x \) from above at 3. A term is said to have “good bounds” if there are bounds polys for that term which bound the term away from zero. This will become important later when we multiply certain polys. For example, we may wish to multiply, \( 0 < x \cdot \frac{1}{y} \) and \( 0 < y \) to form the new poly \( 0 < x \). But, since this requires rewriting \( y \cdot \frac{1}{y} \) to 1, it can be done only if \( y \) is known to be non-zero.

Division thus introduces additional issues. We represent the ratio \( x/y \) as \( x \cdot \frac{1}{y} \). A term is said to “involve division” if it is of the form
1. \( \frac{1}{x} \) or \( \left( \frac{1}{x} \right)^n \), or
2. \( x^c \) or \( x^{c-n} \), where \( c \) is a constant negative integer.

As preparation for the nonlinear procedure, given such a term about division, ACL2 “adds its inverse polys.” We do not attempt to describe the method for generating these polys here other than to say that we gather our initial information from the bounds polys present in the pot-list. We give a few examples:

- If we can determine that \( 4 < x \), we know both \( 0 < \frac{1}{x} \) and \( \frac{1}{x} < 1/4 \).
- If we can determine that \( 0 < \frac{1}{x} \) and \( \frac{1}{x} < 3 \), we know \( 1/3 < x \).
- If we can only determine that \( -2 < x \), we do not know anything about \( \frac{1}{x} \).

### 4.2 The Revised Linear Lemmas Algorithm

As mentioned above, when nonlinear arithmetic is enabled we do things a bit differently. As before, we start with a list of polys and a (possibly empty) pot list. We repeat the following until we reach a fixed point or are interrupted by the user aborting the proof attempt.

1. Add the polys with the linear arithmetic algorithm as described in section 2.1; if no contradiction was generated, go on to step 2.
2. Make an exploded list of the labels of any new pots. If there are not any, quit and return the new pot-list. Otherwise, go on to step 3.
3. For each item in this list:
   a) Add any polys created from type-set.
   b) For each applicable linear lemma: If we can relieve its hypotheses, add the concluding poly(s) to the pot-list as in section 2.1.
   c) If the item involves division, add any inverse polys
4. If a contradiction was generated, quit and return it. Otherwise, go back to step 2.

### 4.3 An Example – Part I

Let us consider \( 0 < a \ \land \ a < b \ \implies 1 < b/a \). After adding the initial polys, the pot-list will look like

<table>
<thead>
<tr>
<th>label</th>
<th>positives</th>
<th>negatives</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a )</td>
<td>( 0 &lt; a )</td>
<td></td>
</tr>
<tr>
<td>( b )</td>
<td>( 0 &lt; b + -1 \cdot a )</td>
<td></td>
</tr>
<tr>
<td>( b \cdot \frac{1}{a} )</td>
<td>( 0 \leq -1 \cdot b \cdot \frac{1}{a} + 1 )</td>
<td></td>
</tr>
</tbody>
</table>

In step 2, we make the list \( a, b, \frac{1}{a}, b \cdot \frac{1}{a} \). Note the presence of \( \frac{1}{a} \), which would not have been there if we used regular pot labels. In step 3a we create the poly \( 0 < \frac{1}{a} \), among others, from type-set and add it to the pot.\(^{12}\) We will continue this example in Section 5.1.

\(^{12}\) We also create the same poly in step 3c.
5 Nonlinear Arithmetic

Before proceeding, let us pause a moment to recollect where we are. In Section 2.1, we presented the linear arithmetic algorithm which lies at the heart of ACL2’s arithmetic procedures – both linear and nonlinear. We next described the previously existing linear lemmas algorithm in Section 3.1. This algorithm uses the linear arithmetic algorithm and is still the default behaviour when nonlinear arithmetic is not enabled. Next, in Section 4.2 we described a variant of the linear lemmas algorithm which is used when nonlinear arithmetic is enabled. Whereas, when nonlinear arithmetic is disabled, the previously existing linear lemmas algorithm is the outermost loop for arithmetic reasoning; the new variant is only the middle loop when nonlinear arithmetic is enabled. We are now about to describe the outermost loop of the nonlinear arithmetic algorithm.

The nonlinear arithmetic procedure consists of three subprocedures: deal-with-product, deal-with-factor, and deal-with-division. Each of these subprocedures is guided by pot-labels and attempts to multiply polys. In order to multiply two polys, we unlinearize the polys (converting them back into ACL2 terms), create the term representing their product, use general-purpose rewriting to rewrite the product terms, and linearize the result. For example, the product of the two polys $0 < -1 \cdot x + 3$ and $0 \leq y + a$ is $0 \leq -1 \cdot y \cdot x + -1 \cdot x \cdot a + 3 \cdot y + 3 \cdot a$. In order to multiply two pots, form a list of the polys in each pot and multiply each poly in the first list with each poly in the second. We multiply more than two polys or pots by generalizing the above.

5.1 Deal-with-Product and Deal-with-Factor

When we have polys about a product and we have polys about the product’s factors, we can multiply those polys about the factors to form polys about the product and perhaps thereby allow cancellation to proceed.

For instance, if we have a new pot about the product $a \cdot b \cdot c$, we can form new polys about the product by finding pots with any of the following combinations of labels and then multiplying the pots.

- $a$, $b$, and $c$
- $a$, and $b \cdot c$
- $a \cdot c$, and $b$
- $a \cdot b$, and $c$

This is done by the subprocedure deal-with-product.

Similarly, if the new pot is about $a$, we look for pots of which $a$ is a factor, such as $a \cdot b \cdot c$, and then see if we can complete the product. This is done by the subprocedure deal-with-factor. We use these two procedures in tandem so that we are less sensitive to the order in which pots are created.

Let us revisit the example from 4.2. When we left it, having just added the poly from type-set, it looked like
<table>
<thead>
<tr>
<th>label</th>
<th>positives</th>
<th>negatives</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a$</td>
<td>$0 &lt; a$</td>
<td></td>
</tr>
<tr>
<td>$\frac{1}{a}$</td>
<td>$0 &lt; \frac{1}{a}$</td>
<td></td>
</tr>
<tr>
<td>$b$</td>
<td>$0 &lt; b$</td>
<td></td>
</tr>
<tr>
<td>$b \cdot \frac{1}{a}$</td>
<td>$0 &lt; b \cdot \frac{1}{a}$</td>
<td>$0 \leq -1 \cdot b \cdot \frac{1}{a} + 1$</td>
</tr>
</tbody>
</table>

Both deal-with-product and deal-with-factor take a pot-label to consider and a pot-list. Deal-with-product will be used only with products, such as $b \cdot \frac{1}{a}$ above, while deal-with-factor will be used only with individual factors, such as $a$, $b$, and $\frac{1}{a}$ above.

When deal-with-product is given $b \cdot \frac{1}{a}$, it will find the pots for $b$ and $\frac{1}{a}$ and multiply the two pots. In particular, it will multiply the polys $0 < \frac{1}{a}$ and $0 < b + -1 \cdot a$ getting $0 < b \cdot \frac{1}{a} + -1 \cdot a \cdot \frac{1}{a}$. This will be rewritten to

$$0 < b \cdot \frac{1}{a} + -1$$

since $a$ is known to be non-zero. Upon adding this to the pot-list we would get the contradiction $0 < 0$ and be done.

When deal-with-factor is given $a$, it will do nothing because $a$ is not a factor of any pot-labels. However, when it is given $b$, it will find the product $b \cdot \frac{1}{a}$. The pot-label $\frac{1}{a}$ will complete this product, and so deal-with-factor will multiply the pots for $b$ and $\frac{1}{a}$ with the same results as for deal-with-product. The pot label $\frac{1}{a}$ is dealt with similar.

### 5.2 Deal-with-Division

Let us next consider $0 < b \land b < a \implies 1 < a/b$. After executing the revised linear lemmas algorithm, the pot-list will look like

<table>
<thead>
<tr>
<th>label</th>
<th>positives</th>
<th>negatives</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a$</td>
<td>$0 &lt; a$</td>
<td></td>
</tr>
<tr>
<td>$b$</td>
<td>$0 &lt; b$</td>
<td></td>
</tr>
<tr>
<td>$\frac{1}{b}$</td>
<td>$0 &lt; \frac{1}{b}$</td>
<td>$0 \leq -1 \cdot b + a$</td>
</tr>
<tr>
<td>$a \cdot \frac{1}{b}$</td>
<td>$0 &lt; a \cdot \frac{1}{b}$</td>
<td>$0 \leq -1 \cdot a \cdot \frac{1}{b} + 1$</td>
</tr>
</tbody>
</table>

This time, deal-with-product and deal-with-factor are insufficient. If we multiply $0 < a$ and $0 < \frac{1}{b}$, we get $0 < a \cdot \frac{1}{b}$, which we already knew via polys from type-set. Rather, we want to multiply the polys $0 < b$ and $0 \leq -1 \cdot a \cdot \frac{1}{b} + 1$. After rewriting $a \cdot b \cdot \frac{1}{b}$ to $a$, we have $0 < b + -1 \cdot a$. Upon adding this latter poly to the pot-list, we get $0 < 0$ and the lemma is proved.

We now sketch the algorithm behind deal-with-division.

1. If the current pot label being considered is itself a product, quit. If we have good bounds for the label, go to step 2; if not, quit.
2. Make a list of all the pot labels that have the multiplicative inverse of the current label as a factor. To distinguish them from the current pot, we will refer to the pots these labels belong to as the “found” pots. For each entry in this list:
a) Multiply the bounds polys from the current pot and the polys in the found pot.
b) Multiply the bounds polys from the found pot and the bounds polys from the current pot.

Let us see how this lines up with what we said we wanted to do above. When deal-with-division is examining the pot-label $b$ (which has good-bounds) it finds the pot-labels $\frac{1}{b}$ and $a \cdot \frac{1}{b}$. For the second of these, since $0 < b$ and $0 \leq -1 \cdot a \cdot \frac{1}{b} + 1$ are both bounds polys, we multiply these polys in step 2. As above, upon adding this latter poly to the pot-list, we get $0 < 0$ and the lemma is proved.

5.3 The Nonlinear Arithmetic Algorithm

After adding polys as in Section 2.1, loop through the following at most three times. If at any point we generate a contradiction, quit and return it.

1. Execute the revised linear lemmas algorithm, described in Section 4.2.
2. Make a list (not an exploded list) of the labels from any new pots and for each item in that list:
   a) If we have good-bounds for the current item, carry out deal-with-division and add any polys generated.
   b) If the current item is a product, carry out deal-with-product and add any polys generated.
   c) If the current item is not a product, carry out deal-with-factor and add any polys generated.

This concludes our presentation of ACL2’s nonlinear arithmetic algorithm.

6 Conclusion

The nonlinear arithmetic procedure is tightly integrated with the rest of ACL2 and allows lemmas such as the following to be proven automatically.

- This lemma was needed for an industrial project to verify the correctness of a microprocessor. It inspired our original work on nonlinear arithmetic and was an early success.

\[
e < a \quad \land \quad a \leq d \quad \land \quad i < h \quad \land \quad h < g \quad \land \quad g \leq f
\]

\[
a \cdot f - a \cdot h \leq b + c \cdot g - c \cdot h
\]

\[
\implies e \cdot f - e \cdot i \leq b + c \cdot g - c \cdot h + d \cdot h - d \cdot i
\]

- Proving this equality helped us to refine deal-with-division.

\[
(bc i j) = \frac{i!}{j!(i-j)!}
\]
Where $bc$ is the binomial coefficient defined by Pascal’s Triangle:

$$(bc i \backslash; j) =$$

if $i < 0$ or $i < j$ then return 0
elseif $j <= 0$ then return 1
else return $(bc i--; j) + (bc i--; j--)$

- This was a long-standing challenge problem given to us by our sponsors. Previous versions of this proof required a dozen or more helper lemmas to be proven; we can now do the proof with only the one helper lemma given below.
Consider the following 6502 assembly program to multiply two 8-bit numbers:

```
LDX #8 ; Load X immediate with the integer 8
LDA #0 ; Load A immediate with the integer 0
LOOP ROR F1 ; Rotate F1 right circular through C
BCC ZCOEF ; Branch to ZCOEF if C = 0
CLC ; Set C to 0
ADC F2 ; Set A to A+F2+c and C to the carry
ZCOEF ROR A ; Rotate A right circular through C
ROR LOW ; Rotate LOW right circular through C
DEX ; Set X to X-1
BNE LOOP ; Branch to LOOP if Z = 0
```

The next lemma was the only one we needed to prove that the above code, generalized to an i-bit wide register, was correct.

- We can also prove this final example automatically. It states that rotating right an i-bit wide register through a carry flag fits back into the i-bit wide register.

$$x < 2^i \land \text{integer } i \land \text{integer } x \land (c = 0 \lor c = 1)$$

$$\implies \text{floor}(x/2) + c \cdot 2^{i-1} < 2^i$$

Our nonlinear arithmetic extension to ACL2 provides significant benefits at a small cost. Proofs that do not involve any nonlinear inequalities are not affected and run at the same speed. A typical “small” lemma with a couple of nonlinear inequalities, which ACL2 could prove automatically before, will generally be proven within a few percentage points of the time previously required – but we can now automatically prove more of these. For more complicated lemmas and theorems, little can be said about the computer time required.

However, within broad limits, the time a user takes to complete a proof is of greater importance than the time the computer takes. Examining failed proofs and writing helper lemmas can be time-consuming and psychologically draining. The fewer lemmas a user has to prove on the way to a desired result, the better.
References

Efficient Distributed SAT and SAT-Based Distributed Bounded Model Checking

Malay K Ganai, Aarti Gupta, Zijiang Yang, and Pranav Ashar

NEC Laboratories America, Princeton, NJ USA 08540
Fax: +1-609-951-2499
{malay,agupta,jyang,ashar}@nec-labs.com

Abstract. SAT-based Bounded Model Checking (BMC), though a robust and scalable verification approach, still is computationally intensive, requiring large memory and time. Interestingly, with the recent development of improved SAT solvers, it is frequently the memory limitation of a single server rather than time that becomes a bottleneck for doing deeper BMC search. Distributing computing requirements of BMC over a network of workstations can overcome the memory limitation of a single server, albeit at increased communication cost. In this paper, we present: a) a method for distributed-SAT over a network of workstations using a Master/Client model where each Client workstation has an exclusive partition of the SAT problem and uses knowledge of partition topology to communicate with other Clients, b) a method for distributing SAT-based BMC using the distributed-SAT. For the sake of scalability, at no point in the BMC computation does a single workstation have all the information. We experimented on a network of heterogeneous workstations interconnected with a standard Ethernet LAN. To illustrate, on an industrial design with ~13K FFs and ~0.5M gates, the non-distributed BMC on a single workstation (with 4 Gb memory) ran out of memory after reaching a depth of 120; on the other hand, our SAT-based distributed BMC over 5 similar workstations was able to go up to 323 steps with a communication overhead of only 30%.

1 Introduction

With increasing design complexity of digital hardware, functional verification has become the most expensive and time-consuming component of the product development cycle [1]. Verifying modern designs requires robust and scalable approaches in order to meet more-demanding time-to-market requirements. Formal verification techniques like symbolic model checking [2, 3], based on the use of Binary Decision Diagrams (BDDs) [4], offer the potential of exhaustive coverage and the ability to detect subtle bugs in comparison to traditional techniques like simulation. However, these techniques do not scale well in practice due to the state explosion problem. SAT solvers enjoy several properties that make them attractive as a complement to BDDs. Their performance is less sensitive to the problem sizes and they do not suffer from space explosion. As a result, various researchers have developed routines for performing Bounded Model Checking (BMC) using SAT [5-8]. Unlike symbolic model checking, BMC focuses on finding bugs of a bounded
length, and successively increases this bound to search for longer traces. Given a design and a correctness property, it generates a Boolean formula, such that the formula is true if and only if there exists a witness/counterexample of length $k$. This formula is then checked by a backend SAT solver. Due to the many recent advances in SAT solvers [9-13], SAT-based BMC can handle much larger designs and analyze them faster than before.

The main limitation of current applications of BMC is that it can do search up to a maximum depth allowed by the physical memory on a single server. This limitation comes from the fact that as the search bound $k$ becomes larger, the memory requirement due to unrolling of the design also increases. Especially for the memory-bound designs, a single server with a limited memory has now become the bottleneck to doing deeper search.

1.1 Motivation

Distributing computing requirements of BMC (memory and time) over a network of workstations can, however, overcome the memory limitation of a single server. In this paper, we explore this possibility, and discuss our approaches in a greater detail that made it feasible. Before we delve into that, we would like to give an intuition behind the feasible solution.

A BMC problem (described in Section 2) originating from an unrolling of the sequential circuit in different time frames provides a natural disjoint partitioning of the problem and thereby, allows the computing resources to be configured in a linear topology. The topology using one Master and several Clients is shown in Figure 1.

![Fig. 1. Partitioning of Unrolled Circuit](image)

Each Client $C_i$ hosts a part of the unroll circuit i.e., from $n_{i+1}$ to $n_{i+1}$ where $n_i$ represents the partition depth. Each $C_i$ (except for the terminals) is connected to $C_{i+1}$ and $C_{i-1}$. The Master is connected to each of the Clients. Using the linear topology, we can distribute parts of the unroll circuit dynamically over additional Clients as and when memory resources on current Clients get close to exhaustion.

To check the satisfiability of a Boolean problem originating from BMC wherein the unrolled circuit is distributed over several servers, we must identify the part of the SAT algorithm that may be delegated to each processor without requiring any processor to have the entire problem data. Since Boolean Constraint Propagation (BCP) on clauses can be done independently on an exclusive partition, it can be delegated to each processor. Moreover, since about 80% of SAT time involves BCP,
one could achieve some level of parallelism by doing distributed-BCP. Note that any approach similar to SAT-based BMC can use similar concept to exploit parallelism.

With this motivation we now briefly describe the organization of the rest of the paper. With a brief discussion on prior related work in Section 1.2, we give a short background in Section 2, our contributions in Section 3-7, experiments in Section 8, and conclusions in Section 9.

1.2 Related Work

Parallelizing SAT solvers have been proposed by many researchers [14-19]. Most of them target performance improvement of the SAT solver. These algorithms are based on partitioning the search space on different processors using partial assignments on the variables. Each processor works on the assigned space and communicates with other processors only after it is done searching its portion of the search space. Such algorithms are not scalable memory-wise due to high data redundancy as each processor keeps the entire problem data (all clauses and variables).

In a closely related work on parallelizing SAT [16], the authors partition the problem by distributing the clauses evenly on many application specific processors. They use fine grain parallelism in the SAT algorithm to get better load balancing and reduce communication costs. Though they have targeted the scalability issue by partitioning the clauses disjointedly, the variables appearing in the clauses are not disjoint. Therefore, whenever a Client finishes BCP on its set of clauses, it must broadcast the newly implied variables to all the other processors. The authors observed that over 90% of messages are broadcast messages. Broadcasting implications can become a serious communication bottleneck when the problem contains millions of variables.

Reducing the space requirement in model checking has been suggested in several works [20-22]. These studies suggest partitioning the problem in several ways. The work in [20] shows how to parallelize the model checker based on explicit state enumeration. They achieve it by partitioning the state table for reached states into several processing nodes. The work in [21] discusses techniques to parallelize the BDD-based reachability analysis. The state space on which reachability is performed is partitioned into disjoint slices, where each slice is owned by one process. The process performs a reachability algorithm on its own slice. In [22], a single computer is used to handle one task at a time, while the other tasks are kept in external memory. In another paper [23], the author suggested a possibility of distributing SAT-based BMC but has not explored the feasibility of such an approach.

2 Background

State-of-the-Art SAT Solver

The Boolean Satisfiability (SAT) problem consists of determining a satisfying assignment for a Boolean formula on the constituent Boolean variables or proving that no such assignment exists. The problem is known to be NP-complete. Most SAT solvers [9-13] employ DPLL style [24] algorithm as shown in Figure 2 with three main engines: decision, deduction, and diagnosis. A Boolean problem can be
expressed either in CNF form or logical gate form or both. A hybrid SAT solver as in [12], where the problem is represented as both logical gates and a CNF expression, is well suited for BMC.

```cpp
SAT_Solve(P=1) { // Check if constraint P=1 satisfiable? 
    while(Decide()==SUCCESS) //Selects a new variable 
        while(Deduce()==CONFLICT)//BCP till conflict/no-conflict 
            if (Diagnose()==FAILURE) //Add conflict learnt clause(s) 
                return UNSAT; //Conflict found at decision level 0 
    return SAT; } //No more decision to make
```

**Fig. 2.** DPLL style SAT Solver

**Bounded Model Checking**

In BMC, the specification is expressed in LTL (Linear Temporal Logic). Given a Kripke structure $M$, an LTL formula $f$, and a bound $k$, the translation task in BMC is to construct a propositional formula $[M, f]_k$, such that the formula is satisfiable if and only if there exists a witness of length $k$ [25]. The satisfiability check is performed by a backend SAT solver. Verification typically proceeds by looking for witnesses or counter-examples (CE) of increasing length until completeness threshold [25, 26]. The overall algorithm of a SAT-based BMC for checking (or falsifying) a simple safety property is shown in the Figure 3. The SAT problems generated by the BMC translation procedure grow bigger as $k$ increases. Therefore, the practical efficiency of the backend SAT solver becomes critical in enabling deeper searches to be performed.

```cpp
BMC(k,P){//Falsify safety property P within bound k 
    for (int i=0; i<=k ; i++) { 
        P_i=Unroll(P,i);//Get property node at i\textsuperscript{th} unrolled frame 
        if (SAT_Solve(P_i=0)=SAT) return CE;//Try to falsify 
    } 
    return NO_CE; } //No counter-example found
```

**Fig. 3.** SAT-based BMC for Safety Property P

3 Our Contributions

**Overview of Distributed-SAT**

Given an exclusive partitioning of the SAT problem, we give an overview of the fine grain parallelization of the three engines of the SAT algorithm (as described in Section 2) on a Master/Client distributed memory environment. The Master controls the execution of distributed-SAT. The decision engine is distributed in such a way that each Client selects a good local variable and the Master then chooses the globally best variable to branch on. During the deduction phase, each Client does BCP on its exclusive local partitions, and the Master does BCP on the global learned conflict clauses. Diagnosis is performed by the Master, and each Client performs a local
backtrack when request by the Master. The Master does not keep all problem clauses and variables; however, the Master maintains the global assignment stack and the global state for diagnosis. This requires much less memory than the entire problem data. To ensure proper execution of the parallel algorithm, each Client is required to be synchronized. We give details of the parallelization and different communication messages in Section 5-9.

Novelties of Our Approach

In this paper, we present a method for distributing SAT over a network of workstations using a Master/Client model where each Client workstation has an exclusive partition of the SAT problem. Though this work is closely related to [16], there are some important differences: a) In [16], though each Client has disjoint set of clauses, variables are not disjoint. So, Clients after completing BCP, broadcast their new implications to all other Clients. After decoding the message, each receiving Client either reads the message or ignores it. In a communication network where BCP messages dominate, broadcasting implications can be an overkill when the number of variables runs into millions. In our improved distributed BCP, however, each Client has the knowledge of the SAT-problem partition topology and uses that to communicate with other Clients. This ensures that the receiving Client has to never read a message that is not meant for it. b) The algorithm in [16] is developed primarily for application specific processors, while our algorithm uses easily available existing networks of workstations. We have described several innovative optimization schemes to reduce the effect of communication overhead on performance in general-purpose networks by identifying and executing tasks in parallel while messages are in transit.

In this paper, we also extend the SAT-based BMC (as a part of our formal verification platform called DiVer) using topology-cognizant distributed-SAT to obtain a SAT-based distributed BMC over a distributed-memory environment. For the sake of scalability, our method makes sure that at no point in the BMC computation does a single workstation have all the information. We developed our distributed algorithms for a network of processors based on standard Ethernet and using the TCP/IP protocol. We can also potentially use dedicated communication infrastructures that may yield better performance, but for this work, we wanted to use an environment that is easily available, and whose performance can be considered a lower bound. We used a socket interface message passing library to provide standard bidirectional communications primitives.

4 Topology-Cognizant Distributed-BCP

BCP is an integral part of any SAT solver. We distribute BCP on multiple processes that are cognizant of topology of the SAT-problem partition running on a network of workstations. In [16], during the distributed-SAT solve each Client broadcasts its implications to all other processors. After decoding the message, each receiving process either reads the message or ignores it. We improve this approach in the following way. Each process is made cognizant of the disjoint partitioning. The process then sends out implications to only those processes that share the partitioning
interface variables with it. Each receiving process simply decodes and reads the message. This helps in two ways: a) the receiving buffer of the process is not filled with useless information; b) receiving process does not spend time in decoding useless information. This ensures that the receiving process has to never read a message that is not meant for it.

We use a distributed model with one Master and several Client processors. The Master’s task is to distribute BCP on each Client that owns an exclusive partition of the problem. A bi-directional FIFO (First-in First-out) communication channel exists only between the process and its known neighbor, i.e., each process is cognizant of its neighbors. The process uses the partition topology knowledge for communication so as to reduce the traffic of the receiving buffer. A FIFO communication channel ensures that the channel is in-order, i.e., the messages sent from one process to another will be received in the order sent. Besides distributing BCP, the Master also records implications from the Clients as each Client completes its task.

The main challenging task for the Master is to maintain causal-effect (“happens before”) ordering of implications in distributed-BCP since we cannot assume channel speeds and relative times of message arrivals during parallel BCP. Maintaining such ordering is important because it is required for correct diagnosis during conflict analysis phase of SAT. In the following we discuss the problem in detail and techniques to overcome it.

Consider the Master/Client model as shown in Figure 1. Client Ci can communicate with Ci-1 and Ci+1 besides the Master M. The Master and Clients can generate implication requests to other Clients; however, Clients can send replies to the Master only for the request made to it. Along with the reply message, Client also sends the message ids of the requests, if any, it made to the other Clients. This is an optimization step to reduce the number of redundant messages. To minimize reply wait time, the Master is allowed to send requests to the Clients even when there are implications pending from the Client provided that the global state (maintained by the Master) is not in conflict.

Let p->q denote an implication request from p to q and p<-q denote implication replies from q to p. Note that though the channel between Ci and the Master is in-order, what happens at the Event E3 cannot be guaranteed in the following.

\[
\begin{align*}
E1: & \text{ M->C1} \\
E2: & \text{ C1->C2} \\
E3: & \text{ M<-C2 or M<-C1}
\end{align*}
\]

If M<-C2 “happens before” M<-C1, then we consider it an out-of-order reply since the implications due to M<-C2 depend on C1->C2, which in turn depend on M->C1. Moreover, any out-of-order reply from a Client makes subsequent replies from that Client out-of-order until the out-of-order reply gets processed.

We propose a simple solution to handle out-of-order replies to the Master. For each Client, the Master maintains a FIFO queue where the out-of-order replies are queued. Since the channel between a Client and Master is in-order, this model ensures that messages in the FIFO will not be processed until the front of the FIFO is processed. We illustrate this with a short event sequence. For simplicity we show the contents for FIFO for the Client C2.

\[
\begin{align*}
E1: & \text{ M->C1} \\
E2: & \text{ C1->C2} \\
\text{FIFO(C2):} & \text{ -}
\end{align*}
\]
E3: M->C2
E4: M<->C2 (in response to E2)
E5: M<->C2 (in response to E3)
E6: M<->C1 (in response to E1)

FIFO(C2): -
FIFO(C2): E4
FIFO(C2): E4,E5
FIFO(C2): - (E4 is processed before E5)

Note that in the reply event E6, the Client C1 also notifies the Master of the event E2. Master queues E4 reply as an out-of-order reply as it is not aware of the responsible event E2 until E6 happens. E5 reply is also queued as out-of-order as earlier out-of-order reply E4 has not been processed yet. When E6 occurs, the Master processes the messages from the events E6, E4 and E5 (in the order). This maintains the ordering of the implications in the global assignment stack.

5 Distributed-SAT

We use fine grain parallelism in our distributed-SAT algorithm similar to the one proposed in [16]. However, we use the topology-cognizant distributed-BCP (as described in the previous section) to carry out distributed-SAT over network of workstations. First, we describe the task partitioning between the Master and Clients as shown in the Figure 4.

Fig. 4. Distributed-SAT and SAT-based Distributed-BMC
**Tasks of the Master**
- Maintains list of constraints, global assignment stack, learnt clauses, antecedents
- Selects a new decision variable from the best local decision sent by each Client
- Global conflict analysis using the assignments and antecedents
- Local BCP on clauses; manages distributed-BCP
- Receives from Ci: New implications with antecedents and best local decision
- Sends to Ci: Implication on variables local to Ci variables, backtrack request, learnt local clauses, update score request

**Tasks of a Client Ci**
- Maintains the ordered list of variables, scores, local assignment stack, local learnt clauses
- Keeps the exclusive partition of the problem and topological information
- Executes on request: Backtrack, decay score, update variable score, local BCP
- Receives from Master: Implications, backtrack request, update score, clause
- Receives from neighbor Cj : Implications on interface
- Sends to Master: New Implications with antecedents and best local decision, best local decision when requested, conflict node when local conflict occurs during BCP, request id when implication request comes from other Clients
- Sends to neighbor Cj: New implication requests on interface

### 6 SAT-Based Distributed-BMC

A SAT-based BMC problem originating from an unrolling of the sequential circuit over different time frames has a natural linear partition and thereby allows configuring the computing resources in a linear topology. The topology using one Master and several Clients is shown in Figure 1. Each Client Ci is connected to C_{i+1} and C_{i-1}. The Master controls the execution of the SAT-based distributed BMC algorithm. The BMC algorithm in Figure 3 remains the same except for the following changes. The Unroll procedure is now replaced by a distributed unrolling in which the procedure Unroll is actually invoked on the Client that hosts the partition for the depth i. Note that depending on the memory availability, the host Client is decided dynamically. After the unrolling, the distributed-SAT algorithm is invoked (in place of $SAT\_Solve$) to check the satisfiability of the problem on the unrolled circuit that has been partitioned over several workstations. Following are the tasks distribution of the Master and Clients.

**Tasks of the Master**
- Allocates an exclusive problem partition to each host Client (box 300 in Figure 4)
- Requests an unrolling to the terminal Client (box 301 in Figure 4)
- Controls distributed-SAT as described in Section 5

**Tasks of a Client**
- Handle current unroll request and also advance by one (box 302 in Figure 4)
- Initiate a new Client as defined by the topology when new unroll size is too large
- Participate in distributed-SAT
7 Optimizations

Memory Optimizations in Distributed-SAT
The bookkeeping information kept by the Master grows with the unroll depth. The scalability of our distributed-BMC is determined by how low is the ratio of the memory utilized by the Master to the total memory used by the Clients. Following steps are taken to lower the scalability ratio:

- By delegating the task of choosing the local decision and maintaining the ordered list of variables to the Client, we save the memory otherwise used by the Master.
- Master does not keep the entire circuit information anytime. It relies on the Clients to send the reasons of implications that will be used during diagnosis.

In our experiments, we observed that the scalability ratio for large designs is close to 0.1, which implies that we can do a 10 times deeper search using a distributed-BMC as compared to a non-distributed (monolithic) BMC over network of similar machines (In our observation, the global learnt clauses maintained by Master is not exponentially large).

Tight Estimation of Communication Overhead
Inter-workstation communication time can be significant and adversely affects the performance. We can mitigate this overhead by hiding execution of certain tasks behind the communication latency. To have some idea of communication overhead, we first need some strategy to measure the communication overhead and actual processing time. This is non-trivial due to asynchronous clock domain of the workstations. In the following, we first discuss a novel strategy to make tight estimation of the wait time incurred by the Master due to inter-workstations communication in Parallel BMC.

Consider a request-reply communication. Time stamps are local to the Master and Client. At time $T_s$, the Master sends its request to the Client. The Client receives the message at its time $t_r$. The Client processes the message and sends the reply to the Master at time $t_s$. The Master, in the meantime, does some other tasks and then starts waiting for the message at time $T_w$. The Master receives the message at time $T_r$.

Without accounting for the Client processing time, wait time would be simply,

$$\text{Wait Time} = T_r - T_w$$  if  $T_r > T_w$  (= 0 otherwise)

This calculated wait time would be an over-estimation of the actual wait time. To account for the Client processing time, we propose the following steps:

- Master sends the request with $T_s$ embedded in the message.
- Client replies back to the Master with the time stamp $(T_s + (t_s - t_r))$.
- The Master, depending on the time $T_w$, calculates the actual wait time as follows:
  - Case Tw1: $T_w < (T_s + (t_s - t_r))$  \[ \text{Wait Time} = T_r - (T_s + (t_s - t_r)) \]
  - Case Tw2: $(T_s + (t_s - t_r)) < T_w < T_s$  \[ \text{Wait Time} = T_r - T_w \]
  - Case Tw3: $T_w < T_s$  \[ \text{Wait Time} = 0 \]

Performance Optimizations in Distributed-SAT
Now we discuss several performance optimizations in the distributed-SAT algorithm.

- A large number of communication messages tend to degrade the overall performance. We took several means to reduce the overhead:
The Master waits for all Clients to stabilize before sending a new implication request. This reduces the number of implication messages sent.

Clients send their best local decision along with every implication and backtrack replies. At the time of decision, the Master, then, only selects from the best local decisions. It is not required to make explicit requests for a decision variable to each Client separately.

For all implication requests, Clients send replies to only the Master. This reduces the number of redundant messages on the network.

Client sends active variables to the Master before doing the initialization. While the Master waits and/or processes the message, the Client does its initialization in parallel.

When Master requests each Client to backtrack, it has to wait for the Clients to respond with a new decision variable. The following overlapping tasks are done to mitigate the wait time:

- Local backtrack (box 207b in Figure 4) by the Master is done after the remote request is sent (box 207b in Figure 4). While the Master waits for the decision variable from the Client, the Master also sends the learnt local conflict clauses to the respective Client.
- The function for adjusting variable score (box 217 in Figure 4) is invoked in the Client after it sends the next decision variable (during backtrack request from the Master) (box 216 in Figure 4). Since message-send is non-blocking, potentially the function is executed in parallel with send. On the downside, the decision variable that is chosen may be a stale decision variable. However, note that the local decision variable that is sent is very unlikely be chosen as decision variable. The reason is that in the next step after backtrack there will be an implication. Since the Client sends the decision variable after every implication request, the staleness of the decision variable will be eventually eliminated.

Performance Optimization in SAT-Based Distributed-BMC

- The design is read and initialization is done in all the Clients to begin with. This reduces the processing time when the unrolling is initiated onto a new Client.
- Advance unrolling is done in the Client while the Client is waiting for implication request from the Master. This includes invoking a new partition in a new Client.

8 Experiments

We conducted our evaluation of distributed -SAT and SAT-based distributed BMC on a network of workstations, each composed of dual Intel 2.8GHz Xeon Processor with 4Gb physical memory running Red Hat Linux 7.2, interconnected with a standard 10Mbps/100Mbps/1Gbps Ethernet LAN. We compare the performance and scalability of our distributed algorithm with a non-distributed (monolithic) approach. We also measure the communication overhead using the accurate strategy as described in Section 7.

We performed our first set of experiments to measure the performance penalty and communication overhead for the distributed algorithms. We employed our SAT-based distributed algorithm on 15 large industrial examples, each with a safety property. For these designs, the number of flip-flops ranges from ~1K to ~13K and number of 2-
input gates ranges from ~20K to ~0.5M. Out of 15 examples, 6 have counterexamples and the rest do not have counterexample within the bound chosen. We used a Master (referred to as M) and 2 Clients (referred as C1 and C2) model where C1 and C2 can communicate with each other. We used a controlled environment for the experiment under which, at each SAT check in the distributed-BMC, the SAT algorithm executes the tasks in a distributed manner as described earlier except at the time of decision variable selection and backtracking, when it is forced to follow the sequence that is consistent with the sequential SAT. We also used 3 different settings of the Ethernet switch to show how the network bandwidth affects the communication overheads. We present the results of the controlled experiments in Table 1[a-b].

In Table 1a, the 1st Column shows the set of designs (D1-D6 have a counterexample), the 2nd Column shows the number of Flip Flops and 2-input Gates in the fanicone of the safety property in the corresponding design, the 3rd Column shows the bound depth limit for analysis, the 4th Column shows the total memory used by the non-distributed BMC, the 5th Column shows the partition depth when Client C2 took an exclusive charge of the further unrolling, Columns 6-8 show the memory distribution among the Master and the Clients. In the Column 9, we calculate the scalability ratio, i.e., the ratio of memory used by the Master to that of the total memory used by Clients. We observe that for larger designs, the scalability factor is close to 0.1 though for comparatively smaller designs, this ratio was as high as 0.8. This can be attributed to the minimum bookkeeping overhead of the Master. Note that even though some of the designs have same number of flip-flops and gates, they have different safety properties. The partition depth chosen was used to balance the memory utilization; however, the distributed-BMC algorithm chooses the partition depth dynamically to reduce the peak requirement on any one Client processor.

| Table 1 [a-b]. Memory & Performance evaluation of the distributed SAT-based BMC |
|---|---|---|---|---|---|---|---|
| EX | FF (K)/Gate (K) | M M Mem (Mb) | Part D | P Mem (Mb) | S ratio | MT (sec) | PT (sec) | MWT (sec) | Perf Pntly | Com Ovr |
| D1 | 4.2/30 | 16 | 20 | 5 | 8 | 5 | 16 | 0.4 | 8.9 | 12.8 | 11.4 | 34.5 | 991.2 | 1.4 | 0.9 |
| D2 | 4.2/30 | 14 | 18 | 5 | 8 | 6 | 13 | 0.4 | 4.2 | 6.7 | 10.5 | 24.2 | 698.6 | 1.6 | 1.6 |
| D3 | 4.2/30 | 17 | 21 | 5 | 9 | 5 | 17 | 0.4 | 9.7 | 15.6 | 11.2 | 33.2 | 767.9 | 1.6 | 0.7 |
| D4 | 4.2/30 | 9 | 10 | 5 | 3 | 4 | 6 | 0.3 | 0.8 | 1.9 | 1.8 | 3.8 | 107.7 | 2.4 | 0.9 |
| D5 | 4.2/30 | 15 | 18 | 5 | 8 | 5 | 15 | 0.4 | 5.2 | 8.2 | 10 | 31.4 | 680.5 | 1.6 | 1.2 |
| D6 | 4.2/30 | 7 | 8 | 5 | 2 | 4 | 4 | 0.3 | 0.3 | 1.1 | 0.6 | 1.6 | 45.1 | 3.7 | 0.5 |
| D7 | 4.2/30 | 21 | 24 | 5 | 7 | 4 | 20 | 0.3 | 9.5 | 14.7 | 9 | 40 | 855.3 | 1.5 | 0.6 |
| D8 | 1.0/18 | 55 | 68 | 30 | 20 | 35 | 31 | 0.3 | 37.9 | 52.1 | 22.1 | 109 | 1895.3 | 1.4 | 0.4 |
| D9 | 0.9/18 | 67 | 124 | 30 | 65 | 33 | 49 | 0.8 | 314.6 | 454.5 | 130 | 702.4 | 12922.9 | 1.4 | 0.3 |
| D10 | 5.2/37 | 21 | 29 | 5 | 10 | 4 | 24 | 0.4 | 23.4 | 38.4 | 17.8 | 71.8 | 764.1 | 1.6 | 0.5 |
| D11 | 12.7/448 | 61 | 1538 | 45 | 172 | 1071 | 480 | 0.1 | 919 | 1261.4 | 1135.7 | 2403 | 5893.2 | 1.4 | 0.9 |
| D12 | 3.7/158 | 81 | 507 | 40 | 47 | 246 | 267 | 0.1 | 130.5 | 89.1 | 0.1 | 65.1 | 63.2 | 0.7 | 0.0 |
| D13 | 3.7/158 | 41 | 254 | 20 | 24 | 119 | 141 | 0.1 | 33.7 | 23.2 | 0.4 | 6.3 | 16.1 | 0.7 | 0.0 |
| D14 | 3.7/158 | 81 | 901 | 40 | 149 | 457 | 447 | 0.2 | 452.8 | 360.6 | 87.4 | 653.5 | 1288.6 | 0.8 | 0.2 |
| D15 | 3.7/158 | 81 | 901 | 40 | 135 | 457 | 443 | 0.2 | 442.2 | 344.6 | 97.2 | 679.9 | 1138.5 | 0.8 | 0.3 |
Table 2. Comparison of monolithic and distributed BMC on Industrial designs

<table>
<thead>
<tr>
<th>Ex</th>
<th>Mono Depth</th>
<th>Mono Time (sec)</th>
<th>Para Depth</th>
<th>Para Time (sec)</th>
<th>Para Memory (in Mb)</th>
<th>MWT (sec)</th>
<th>Comm Ovrhd</th>
<th>S Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>D11</td>
<td>120</td>
<td>1642.3</td>
<td>323</td>
<td>6778.5</td>
<td>634</td>
<td>1505</td>
<td>1740</td>
<td>1740</td>
</tr>
<tr>
<td>D12</td>
<td>553</td>
<td>4928.3</td>
<td>1603</td>
<td>13063.4</td>
<td>654</td>
<td>1846</td>
<td>1863</td>
<td>1863</td>
</tr>
<tr>
<td>D13</td>
<td>553</td>
<td>4899.5</td>
<td>1603</td>
<td>12964.5</td>
<td>654</td>
<td>1846</td>
<td>1864</td>
<td>1864</td>
</tr>
<tr>
<td>D14</td>
<td>567</td>
<td>642.8</td>
<td>1603</td>
<td>2506.2</td>
<td>654</td>
<td>1833</td>
<td>1851</td>
<td>1851</td>
</tr>
<tr>
<td>D15</td>
<td>567</td>
<td>641.9</td>
<td>1603</td>
<td>1971.5</td>
<td>654</td>
<td>1833</td>
<td>1851</td>
<td>1851</td>
</tr>
</tbody>
</table>

In Table 1b, the 1st Column shows the cumulative time taken (over all steps) by non-distributed BMC, the 2nd Column shows the cumulative time taken (start to finish of Master over all steps) by our distributed-BMC excluding the message wait time, Columns 3-5 show the total message wait time for the Master in a 10/100/1000Mbps Ethernet Switch setting. In the Column 6, we calculate the performance penalty by taking the ratio of the time taken by distributed to that of non-distributed BMC (=Para Time/ Mono Time). In the Column 7, we calculate the communication overhead for the 1Gbps switch setting by taking the ratio of the message waiting time to distributed BMC time (=wait time for 1 Gbps/ Para Time). On average we find that the performance penalty is 50% and communication overhead is 70% with overall degradation by a factor of 2.55 (=1.5 * 1.7).

In some cases, D12-D15, however, we find an improvement in performance over non-distributed BMC. This is due to the exploitation of parallelism during the Client initialization step as described in Section 7. Note that the message wait time adversely gets affected with lowering the switch setting from 1Gbps to 10Mbps. This is attributed to the fact that Ethernet LAN is inherently a broadcast non-preemptive communication channel.

In our second set of experiments, we used the 5 largest (of 15) designs D11-D15 that did not have a witness. For distributed-BMC, we configured 5 workstations into one Master and 4 Clients C1-C4; each connected with the 1Gbps Ethernet LAN. In this setting, Clients are connected in a linear topology and the Master is connected in a star with others. In this experiment, we show the ability of the distributed-BMC to do deeper search using distributed memory. For the design D11, we used a partition of 81 unroll depths on each Client and for designs D12-15, we used partition of 401 unroll depths on each Client. The results are shown in the Table 2.

In Table 2, the 1st Column shows the set of large designs that were hard to verify, the 2nd Column shows the farthest depth to which non-distributed BMC could search before it runs out of memory, the 3rd Column shows the time taken to reach the depth in the 2nd Column, the 4th Column shows the unroll depth reached by distributed-BMC using the allocated partition, the 5th Column shows the time taken to reach the depth in the 4th Column excluding the message wait time, Columns 6-10 show the memory distribution for the Master and Clients, the 11th Column shows the total message wait time. In the Column 12, we calculate the communication overhead by taking the ratio of message wait time to the distributed-BMC time (=MWT time/ Para Time). In the Column 13, we calculate the scalability ratio by taking the ratio of memory used by the Master to that of the total memory used by the Clients.

We use the design D11 with ~13K flip-flops and ~0.5Million gates to show the performance comparison. For the design D11 we could analyze up to a depth of 323 with only 30% communication overhead, while using a non-distributed version we...
could analyze only up to 120 time frames under the per-workstation memory limit. Low scalability factor, i.e., 0.1 for large designs indicates that for these designs our distributed-BMC algorithm could have gone 10 times deeper compared to the non-distributed version for similar set of machines. We also observe that the communication overhead for these designs was about 45% on average, a small penalty to pay for deeper search.

9 Conclusions

For verifying designs with high complexity, we need a scalable and robust solution. SAT-based BMC is quite popular because of its robustness and better debugging capability. Although, SAT-based BMC is able to handle increasingly larger designs than before as a result of advancement of SAT solvers, the memory of a single server has become a serious limitation to carrying out deeper search. Existing parallel algorithms either focus on improving the SAT performance or are used in either explicit state-based model checkers or in unbounded implicit state-based model checkers. To the best of our knowledge ours is the first detailed study on providing a feasible solution for SAT-based distributed-BMC using an improved distributed SAT algorithm.

Our distributed algorithm uses the normally available large pool of workstations that are inter-connected by standard Ethernet LAN. For the sake of scalability, our distributed algorithm makes sure that no single processor has the entire data. Also, each process is cognizant of the partition topology and uses the knowledge to communicate with the other process; thereby, reducing the process’s receiving buffer with unwanted information. We have also proposed several memory and performance optimization schemes to achieve scalability and decrease the communication overhead.

In the future, we would like to evaluate our distributed-SAT and SAT-based distributed-BMC on a clustered system for high performance computing that has low latency and high bandwidth communication [27].

Acknowledgements. We thank Guoqiang Pan for implementing the socket-based message-passing library.

References


Convergence Testing in Term-Level Bounded Model Checking*

Randal E. Bryant\textsuperscript{1,2}, Shuvendu K. Lahiri\textsuperscript{2}, and Sanjit A. Seshia\textsuperscript{1}

School of Computer Science & Electrical and Computer Engineering Department
Carnegie Mellon University, Pittsburgh, PA 15213
\{Randy.Bryant, Sanjit.Seshia\}@cs.cmu.edu
shuvendu@ece.cmu.edu

Abstract. We consider the problem of bounded model checking of systems expressed in a decidable fragment of first-order logic. While model checking is not guaranteed to terminate for an arbitrary system, it converges for many practical examples, including pipelined processors. We give a new formal definition of convergence that generalizes previously stated criteria. We also give a sound semi-decision procedure to check this criterion based on a translation to quantified separation logic. Preliminary results on simple pipeline processor models are presented.

1 Introduction

Systems with parameters of finite but arbitrary or large size are often modeled as infinite-state systems. Such systems include superscalar processors, communication protocols with unbounded channels, and networks of an arbitrary number of identical processes. While state elements can still be of Boolean type, richer data types such as unbounded integers or unbounded arrays of integers are also used. Employing this richer expressive power is one approach to tackling the state explosion problem.

In the area of hardware verification, the logic of Equality with Uninterpreted Functions and Memories (EUFM) has been successfully used for the automated verification of pipelined processor designs [8,3]. The more general logic of Counter Arithmetic with Lambda Expressions and Uninterpreted Functions [4] (CLU) has been used for bounded model checking and inductive invariant checking of out-of-order microprocessors with unbounded resources [14]. Bounded model checking proceeds by symbolically simulating the system for a finite number of steps starting from an initial state, checking on each step that a state property holds. As the state elements can be terms in a first-order logic, we will refer to this technique as term-level bounded model checking. Since term-level models can express Turing machines [12], the symbolic simulation might never reach a fixpoint in general. However, in many practical cases, the simulation does

\* This research was supported in part by the Semiconductor Research Corporation, Contract RID 1029 and by ARO grant DAAD 19-01-1-0485.
converge. It is therefore necessary to check, after each simulation step, whether the simulation has converged.

In this paper, we make two main contributions. First, we give a formal definition of convergence for term-level bounded model checking, where CLU logic is used as the modeling formalism. The convergence criterion is formulated as a quantified second-order formula with one quantifier alternation and is undecidable in general. Second, we give a semi-decision procedure for this class of second-order formulas. Our procedure is based on a sound translation to a decidable fragment of first-order logic called quantified separation logic (QSL). QSL formulas are quantified Boolean combinations of Boolean variables and predicates of the form \( x_i < x_j + c \) or \( x_i = x_j + c \), where \( x_i \) and \( x_j \) are real or integer variables, and \( c \) is a constant. The QSL formulas are then decided by a translation to quantified Boolean logic \([15]\). Although we use the semi-decision procedure for convergence checking, our results are also more generally applicable to automated theorem proving of second-order formulas.

Previous term-level model checkers vary in expressiveness of the underlying logic, and either use syntactic convergence criteria or approximation techniques that guarantee convergence at the cost of completeness. Hojati et al. \([12]\) presented a modeling formalism called ICS which is similar in expressiveness to EUFM. They showed that ICS models do not converge in general, except under highly restrictive assumptions that are not of practical interest. Isles et al. \([13]\) built on this work, giving a conservative, syntactic definition of convergence of ICS models, and using it to verify versions of the DLX pipeline. Our logic is more expressive than ICS. Also, as we show in Section 5.2, their convergence criterion is a special case of the one we present in this paper. Corella et al. \([9]\) have used Multiway Decision Graphs (MDGs) for term-level model checking. MDGs are BDD-like data structures used for representing formulas in quantifier-free logics such as EUFM and CLU; the exact logic represented depends on the set of interpreted function symbols used in the model. Thus, Corella et al. use MDGs to represent the characteristic function of the set of states of a term-level model. Unlike our work, their models cannot have variables of function type, and hence cannot verify systems with embedded memories. However, they address a more general class of properties expressible in a first order temporal logic. With respect to convergence checking, Corella et al. use syntactic rewriting techniques similar to those employed for ICS \([13]\). Bultan et al. \([6]\) have used Presburger arithmetic for verifying concurrent algorithms. Checking convergence for systems expressed in Presburger arithmetic is decidable; however, since the model checking might not converge in general, they conservatively approximate the fixpoint, allowing the possibility of spurious counterexamples. In comparison, our use of CLU logic allows us to use uninterpreted functions and also lets us model richer systems with memories. This expressive power, however, results in convergence checking becoming undecidable.

The rest of the paper is organized as follows. Section 2 presents CLU logic and our system modeling formalism. Section 3 defines the term-level bounded model checking problem. In Section 4, we formally define the convergence cri-
terion. Section 5 describes how we check this criterion. Finally, we conclude in Section 6 with some preliminary results with pipelined processor models. For brevity, we have omitted proofs of theorems and an alternate complete semi-decision procedure; these can be found in an accompanying technical report [5].

2 Preliminaries

2.1 CLU Logic

Syntax. The syntax includes four classes of expressions, representing computations of truth values or integers, as well as functions over integers yielding truth values or integers. We use symbols to represent abstract values and functions.

\[
\begin{align*}
\text{bool-expr} & ::= \text{true} | \text{false} | \text{bool-symbol} | \neg \text{bool-expr} | (\text{bool-expr} \land \text{bool-expr}) \\
& \quad | (\text{int-expr} = \text{int-expr}) | (\text{int-expr} < \text{int-expr}) \\
& \quad | \text{predicate-expr}(\text{int-expr}, \ldots, \text{int-expr}) \\
\text{int-expr} & ::= \text{lambda-var} | \text{int-symbol} | \text{ITE}(\text{bool-expr}, \text{int-expr}, \text{int-expr}) \\
& \quad | \text{int-expr} + \text{int-constant} | \text{function-expr}(\text{int-expr}, \ldots, \text{int-expr}) \\
\text{predicate-expr} & ::= \text{predicate-symbol} | \lambda \text{lambda-var}, \ldots, \text{lambda-var}. \text{bool-expr} \\
\text{function-expr} & ::= \text{function-symbol} | \lambda \text{lambda-var}, \ldots, \text{lambda-var}. \text{int-expr}
\end{align*}
\]

Fig. 1. Expression Syntax. Expressions can denote computations of Boolean values, integers, or functions yielding Boolean values or integers.

Symbols are written with a typewriter font, such as \(a\) or \(f\). Associated with each symbol is a type indicating what kind of value it represents (truth, integer, function, or predicate). For function and predicate symbols, the type includes its arity indicating the number of arguments it takes. For function symbol \(f\), we write its arity as \(arity(f)\). For a set of symbols \(A\), we let \(E(A)\) denote the set of all expressions that can be formed using these symbols, obeying the usual rules on type matching.

The syntax includes integer lambda variables. These only serve to represent the arguments to lambda expressions. Note also that the lambda expression syntax is constrained so that they cannot have functions as arguments, and they cannot express any form of looping or recursion.

Sets of Expressions. We use two ways to refer to sets of expressions in which we must identify the different elements. The first is a vector notation, in which we index the elements with integer subscripts. We use the notation \(e_{\overline{n}}\) to denote a vector with elements \(e_1, \ldots, e_n\). The second is a named-element notation, in which we have a set of symbolic names \(A\) and write a set of expressions \(e\) as having an element \(e_a\) for each \(a \in A\).

With both notations, we can indicate the syntactic substitution of elements for symbols or variables in an expression. That is, the expression \(s [e_{\overline{n}}/\overline{x_n}]\) denotes the expression where each instance of \(x_i\) in \(s\) is replaced by the expression
\(e_i\) for \(1 \leq i \leq n\). These substitutions are performed in parallel, so there is no ambiguity of some expression \(e_i\) contains the symbol \(x_j\). Similarly, \(s[\overline{e}/A]\) indicates the result of replacing each instance of a symbol \(a \in A\) with the expression \(e_a\).

**Semantics.** For a set of symbols \(A\), we let \(\sigma_A\) indicate an *interpretation* of each of these symbols. That is, \(\sigma_A\) maps each symbol to an integer, a truth value, or a function according to the symbol type. For any expression \(e \in E(A)\), we define its *evaluation under interpretation* \(\langle e \rangle_{\sigma_A}\), denoted \(\langle e \rangle_{\sigma_A}\) as the value obtained by evaluating \(e\) when each symbol \(a\) is replaced by its interpretation \(\sigma_A(a)\). We omit the detailed definition.

A truth expression \(e \in E(A)\) is said to be *universally valid* when it evaluates to \(true\) for all interpretations of its symbols, i.e., when \(\langle e \rangle_{\sigma_A} = true\) for all \(\sigma_A\).

As a final notation, for disjoint symbol sets \(A\) and \(B\), each having interpretations \(\sigma_A\) and \(\sigma_B\), we let \(\sigma_A \cdot \sigma_B\) denote the interpretation over the symbols in \(A \cup B\) obtained by applying the respective interpretations to the symbols in \(A\) and \(B\).

As noted earlier, our syntax for function applications requires all arguments to be integer expressions. We can therefore transform any integer or truth expression containing lambda expressions into an equivalent lambda-free one by performing *Beta reduction*, in which the actual parameter expressions are syntactically substituted in parallel with the actual parameter expressions.

### 2.2 System Model

We model the system as having a number of *state elements*, where each state element may be a truth or integer value, or a function or predicate. This latter class of state elements allows us to describe various forms of memories. For example, a conventional random-access memory can be modeled as a function that yields an integer data value given an integer address as argument. We use symbolic names to represent the different state elements giving the set of *state symbols* \(S\). We also introduce a set of *input symbols* \(T\), representing a set of input signals that can be set to different values on each step of operation. That is, on each step \(i\), we introduce a symbol \(a_i\) for each input symbol \(a\). We refer to such signals as the *indexed input symbols*. We introduce two more sets of symbols \(K\) and \(I\) to allow one run by the verifier to compute the behavior of systems with different functionality operating with different initial state and input values. The symbols in \(K\) parameterize system functionality. This could include, for example, function symbols for the ALU, and the contents of the instruction memory. The symbols in \(I\) parameterize the initial state and system input sequence. These could include a function symbol to encode the initial state of a memory. They also include the indexed input symbols.

The overall system operation is characterized by an *initial state* \(s^0\) and a *transition behavior* \(\delta\). The initial state contains an expression for each state element. The initial value of state element \(a\) is given by an expression \(s^0_a \in E(I)\). The transition behavior consists of an expression for each state element. The behavior for state element \(a\) is given by an expression \(\delta_a \in E(K \cup S \cup T)\). In this expression, we use the state element symbols to represent the current system
state, and the input symbols to represent the current values of the inputs. The expression then gives the new state for that state element.

From these expressions, we define the state sequence for the system 
\( s^0, \ldots, s^i, \ldots \), where the state at step \( i \) consists of an expression for each state element \( s^i_a \in E(K \cup I) \). This expression is given by performing the double substitution

\[
s^i_a = \delta_a \left[ s^{i-1} / \mathcal{S}, t^i / \mathcal{T} \right],
\]

where the input expression \( t^i \) has \( t^i_a = a_i \) for each \( a \in \mathcal{T} \). As mentioned earlier, we always perform Beta reduction following a substitution such as this. We use the shorthand \( s^i = \delta(s^{i-1}, t^i) \) to indicate this process of generating the expressions for the state at step \( i \).

3 Property Checking

A system property \( P \) is represented as a Boolean expression over the state elements \( P \in E(\mathcal{S}) \). Typically we want to determine whether \( P \) holds at some particular step \( k \), or whether \( P \) holds at every step. We can determine whether \( P \) holds at some particular step \( k \) by applying a decision procedure for CLU logic. However, our interest here is to prove that \( P \) holds for every step \( i \geq 0 \). In general, this task is undecidable. The problem remains undecidable even if we restrict the class of systems to ones with only integer state elements, and where the system behavior is described using a logic of equality with uninterpreted functions [12].

Instead, we focus on a more restricted class of systems that satisfy a property we call \( k \)-convergence. With these systems, every reachable state can be reached within \( k \) steps for some combination of initial state and inputs, for some fixed bound \( k \). If we can prove that a system is \( k \)-convergent, then we can guarantee property \( P \) holds on every step by verifying that it holds on every step up through \( s^k \).

Formally, we say that a system with initial state \( s^0 \) and transition behavior \( \delta \) converges in \( k \) steps, when for every interpretation \( \sigma_I \) of the initial state and inputs and for every interpretation \( \sigma_K \) of the system parameters, there exists a step \( i \leq k \) and an alternate interpretation \( \theta_I \) of the initial state and inputs, such that for every state symbol \( a \in \mathcal{S} \)

\[
\langle s^i_a \rangle_{\theta_I \cdot \sigma_K} = \langle s^{k+1}_a \rangle_{\sigma_I \cdot \sigma_K}.
\]

We use the shorthand \( \langle s^i \rangle_{\theta_I \cdot \sigma_K} = \langle s^{k+1} \rangle_{\sigma_I \cdot \sigma_K} \) to indicate this equality for every state element. Property (2) states that by step \( k+1 \), the system will not reach any new states. That is, for every possible interpretation of the system parameters \( \theta_K \), and for every possible operation of the system for \( k+1 \) steps, as determined by the interpretation \( \sigma_I \) of the initial state and indexed input symbols \( I \), there is some alternate initial state and input sequence, given by interpretation \( \theta_I \) that would have led to the exact state in \( i \) steps for some \( 0 \leq i \leq k \).
We show that this property guarantees that the system will not reach new states beyond step \( k \).

**Theorem 1.** If a system converges in \( k \) steps, then for any \( j \geq 0 \) and any interpretation \( \sigma_K \) of the system parameters, there exists a step \( i \leq k \) and an alternate interpretation \( \theta_I \) of the initial state and inputs, such that

\[
\langle s^i \rangle_{\theta_I \cdot \sigma_K} = \langle s^j \rangle_{\sigma_I \cdot \sigma_K}.
\] (3)

### 4 Formulation of the Convergence Criterion

We now reach the main topic of this paper: determining whether a system is \( k \)-convergent for some value of \( k \). We can express this as a problem in second-order logic as follows. Introduce a symbol set \( J \) consisting of a symbol \( a' \) for each initial state symbol \( a \in I \), and a symbol \( a'_i \in I \) for each indexed input signal \( a_i \), for \( 1 \leq i \leq k \). Rewrite each state expression \( s^i \), for \( 0 \leq i \leq k \) to an expression \( r^i \), by replacing each symbol in \( I \) with its counterpart in \( J \).

Using the notation of predicate calculus, we consider the symbols in \( I, J, \) and \( K \) to be quantified variables, either first-order (for integer or Boolean symbols) or second-order (for function or predicate symbols). We can then write the convergence criterion as:

\[
\forall K \forall I \exists J \left[ \bigvee_{0 \leq i \leq k} \bigwedge_{a \in S} \{ r^i_a = s^k + 1 \} \right] \tag{4}
\]

With these quantifiers, we are really quantifying over the possible interpretations of the symbols. Note that this formula cannot be expressed in first-order logic, because we have existentially quantified function symbols.

**Example 1.** Consider a system with the integer state variables \( x, y \) and Boolean state variable \( b \). The operations are defined by:

\[
\begin{align*}
\text{init}[x] &= c_0 & \text{init}[y] &= c_0 & \text{init}[b] &= \text{true} \\
\text{next}[x] &= f(x) & \text{next}[y] &= f(y) & \text{next}[b] &= (x = y)
\end{align*}
\]

where \( c_0 \) is an integer symbol and \( f \) is an uninterpreted function symbol. Using our notation, the sets of symbols are defined as follows — \( S = \{x, y, b\}, K = \{f\}, I = \{c_0\} \) and \( J = \{c'_0\} \).

After simulating the system for one step, the convergence condition (given by equation 4, where \( k = 0 \)) becomes:

\[
\forall f \forall c_0 \exists c'_0 [ c'_0 = f(c_0) \land c'_0 = f(c_0) \land \text{true} = (f(c_0) = f(c_0)) ]
\]

which simplifies to \( \forall f \forall c_0 \exists c'_0 [ c'_0 = f(c_0) ] \), which is clearly valid, with \( c'_0 \) taking the value \( f(c_0) \).

Therefore the system converges after one step of simulation. As expected, the state variable \( b \) is always \text{true} in the reachable set of states.
For a function or predicate state element $F$, the expression $r_F^k = s_F^{k+1}$ is a second-order equation—it states that two functions or predicates are identical for all possible arguments.

For systems without function or predicate state elements, our convergence criterion yields a formula with the quantification structure shown in (4), with only first-order equations. Even for the simple case of a system with one integer symbol in $T$, one function symbol of arity 2 in $K$, deciding the truth of a formula with this structure is undecidable [2].

Again we find ourselves facing an undecidable property. We deal with this by 1) using syntactic transformations to eliminate the second-order equations for function and predicate state elements, and 2) using a sound, but incomplete decision procedure for second-order formulas of the form shown in (4). Our procedure is quite simple, but it seems to work well for the formulas arising in our convergence testing.

5 Checking Convergence

5.1 Function and Predicate State Elements

We can convert our convergence formula (4) to one containing only first-order equations by introducing a set of argument symbols $Z = z_1, \ldots, z_n$, where $n$ is the maximum arity of any predicate or function state element. Suppose state element $F$ has arity $\text{arity}(F) = m$. Then define $r_F^k \equiv r_F^k(z_1, \ldots, z_m)$, and similarly define $s_F^k \equiv s_F^k(z_1, \ldots, z_m)$. Then we can rewrite the convergence criterion as:

$$\forall K \forall I \exists J \forall Z \left[ \bigvee_{0 \leq i \leq k} \bigwedge_{a \in S} r_a^i = s_a^k \right]$$

(5)

Unfortunately, we have no general approach to handle formulas with this quantifier structure. Instead, we use rewriting techniques to handle limited forms of function and predicate state elements. Our technique is sufficient to handle random-access memories, including the data memory and register file of a microprocessor.

A random-access memory is modeled as a function state element $\text{Mem}$ where the argument is an address, and the function returns the value stored at that address. Consider a memory with address input $\text{Adr}$, data input $\text{Dat}$ and write-enable signal $\text{Wrt}$. We describe the memory operation in our term-level modeling language as:

$$\begin{align*}
\text{init}[\text{Mem}] &= m_0 \\
\text{next}[\text{Mem}] &= \lambda x. \text{ITE}(\text{Wrt} \land x = \text{Adr}, \text{Dat}, \text{Mem}(x))
\end{align*}$$

where $m_0$ is an uninterpreted function giving the initial memory contents. Note the restricted class of expressions that will result when modeling the operation of this memory over time to generate the expression $r_{\text{Mem}}^i$. At the base is an
uninterpreted function, which can be assigned an interpretation that matches any desired functionality. There will then be a bounded number of updates due to write operations, but these will each be to a single (symbolic) address.

Suppose we wish to determine whether the system has converged for some fixed time point \( i \), so that Equation 5 reduces to

\[
\forall K \forall I \exists J \forall Z \left[ \bigwedge_{a \in S} \bar{t}_a^i = s_a^k \right]
\] (6)

Then the convergence criterion for state element \( \text{Mem} \) will have the general form:

\[
\forall A \exists B \forall z F'(z) = F(z)
\] (7)

where expression \( F \) has only symbols in \( A \), while expression \( F' \) has symbols from both \( B \) and \( A \).

We apply a set of rewrites to the symbols in \( B \) and generate a set of verification conditions that guarantees (7) holds, based on the structure of expression \( F' \). In general, our rules apply to equations of the form \( P(z) \Rightarrow F'(z) = F(z) \), where \( P \) is a predicate expression with symbols from both \( B \) and \( A \). At the top level, we start with \( P \) being an expression that always yields \text{true}.

1. For equations of the form \( P(z) \Rightarrow f'(z) = F(z) \), where \( f' \) is a function symbol in \( B \), rewrite all occurrences of \( f' \) in \( \bar{t}^i \) to be \( \lambda x. \text{ITE}(P(x), F(x), f'(x)) \).

2. For equations of the form \( P(z) \land z = E \Rightarrow F'(z) = F(z) \), where \( E \) is an expression with symbols from both \( B \) and \( A \), reduce the equation to \( P(E) \Rightarrow F'(E) = F(E) \). This eliminates any reference to \( z \) in the equation.

3. For equations of the form \( P(z) \Rightarrow [\lambda x. \text{ITE}(Q(x), G'(x), H'(x))](z) = F(z) \), where \( Q \), \( G' \), and \( H' \) are predicate and function expressions containing symbols in both \( A \) and \( B \), we generate two verification conditions: \( P(z) \land Q(z) \Rightarrow G'(z) = F(z) \), and \( P(z) \land \neg Q(z) \Rightarrow H'(z) = F(z) \), and solve these recursively.

4. For equations of the form \( P(z) \Rightarrow f(z) = F(z) \), where \( f \) is a function symbol in \( A \), we recursively analyze the structure of \( F \).
   - If \( F \) is of the form \( \text{ITE}(Q(x), G(x), H(x)) \), where \( Q \), \( G \), and \( H \) are predicate and function expressions containing symbols in \( A \), we generate two verification conditions: \( P(z) \land Q(z) \Rightarrow f(z) = G(z) \), and \( P(z) \land \neg Q(z) \Rightarrow f(z) = H(z) \), and solve these recursively.
   - If \( F \) is of the form \( g(z) \), then the symbols \( f \) and \( g \) need to be the same. If the two symbols are different, we return \text{false} which implies that no rewrite exists.

5. For equations of the form \( P(z) \Rightarrow F'(z + c) = F(z) \) with integer constant \( c \), transform the equation to be \( P(z - c) \Rightarrow F'(z) = F(z - c) \), and solve it recursively.
Similar rules hold for equations of the form $P \implies F'(z) = F(z)$, i.e., $P$ is a Boolean expression independent of $z$.

Given the special form of the expressions describing the updating of a random-access memory, we can see that by repeated application of these rules, we can eliminate all occurrences of symbol $z$ in (6). The first rule handles the uninterpreted function representing the initial memory state. The second rule handles updates to individual memory addresses. The third rule lets us split based on the case structure of the expression. The last two rules would be required for more complex memory structures.

Note that CLU logic can be used to model memories in which multiple entries can be updated in parallel [14]. The rewriting techniques proposed in this section do not work for such memories.

5.2 Convergence with First-Order Equations

Assume we have applied transformation rules to eliminate all second-order equations, and hence the convergence criterion is expressed by an equation of the form shown in (4) with only first-order equations. We would therefore like to decide the validity of a formula $\psi$ of the form

$$\psi \equiv \forall A \exists B \phi$$

where $\phi$ does not contain any quantifiers. In fact, $\phi$ is a CLU formula, and we can assume that transformations have been applied to eliminate all ITE operations\(^1\) and lambda applications.

Our system model is sufficiently general that we can generate any second-order formula having the structure shown in (8) as part of a convergence test. To see this, let the variables in $\phi$ be $A = \overline{a}$ and $B = \overline{b}$. Introduce a set of $m+1$ state elements, consisting of an element $q_i$ for each existentially quantified variable $b_i \in B$, and a final truth-valued state element $q_{m+1}$. For each universally quantified variable $a_i \in A$, introduce a system parameter $a_i$. Let the system have transition behavior $\delta$ such that $\delta_{q_{m+1}} = \phi \overline{a}$. Let $\delta_{q_i} = q_i$ for $1 \leq i \leq m$. Finally, let the initial state $s_0$ of each state element $q_i$ for $1 \leq i \leq m$ be $a_i$, and the initial state of $q_{m+1}$ be true. Then the system is 0-convergent if and only if the formula $\forall A \exists B \phi$ is valid.

This construction shows that we cannot assume any particular restrictions on the formulas we must decide to prove convergence, other than the quantifier structure shown in (8).

\(^1\) These can be eliminated by the “push to the leaves” transformation [16].
Proposition 1. Let $b$ denote a set containing an expression $b_a \in E(\mathcal{A})$ for each $a \in \mathcal{B}$. If $\forall \mathcal{A} \phi [b/\mathcal{B}]$ is valid, then so is $\forall \mathcal{A} \exists \mathcal{B} \phi$.

The proof of this proposition follows by instantiating any symbol $a \in \mathcal{B}$ with the value $\langle b_a \rangle_{\mathcal{A}}$.

With this approach, we can prove convergence by using a decision procedure for CLU logic to prove the universal validity of $\phi [b/\mathcal{B}]$. The challenge, of course, is to find an appropriate set of substitutions to the symbols in $\mathcal{B}$.

Semantic Approach. We describe a way to transform formulas of the structure $\psi \models \forall \mathcal{A} \exists \mathcal{B} \phi$ into a formula in the logic we call Quantified Separation Logic (QSL). QSL consists of quantified Boolean and integer variables, Boolean connectives, and predicates of the form $x = y + c$ and $x < y + c$, where $x$ and $y$ are integer variables, and $c$ is an integer constant. Our translation $T_s(\psi)$ (for “sound”) yields a formula that is valid only if $\psi$ is valid. By deciding the validity of the translation we can test for definite convergence.

We can rewrite any Boolean or integer expression in CLU into a normal form, in which all ITE operations have been eliminated, and the additions of integer constants are grouped together. Define an atomic expression as either an integer or Boolean symbol, or an application of a function or predicate symbol.

Without loss of generality, let us assume $\phi$ is in normal form. We start by enumerating all of the atomic expressions occurring in $\phi$ as a sequence $g_1, \ldots, g_n$. Let $\text{top}(g_i)$ denote the top-level symbol in subexpression $g_i$. We can see that each atomic expression $g_i$ must be of one of the following forms:

1. Boolean symbol. $g_i \doteq b$, giving $\text{top}(g_i) = b$.
2. Predicate application. $g_i \doteq p(g_{i_1} + c_{i_1}, \ldots, g_{i_k} + c_{i_k})$, giving $\text{top}(g_i) = p$.
3. Integer symbol. $g_i \doteq x$, giving $\text{top}(g_i) = x$.
4. Function application. $g_i \doteq f(g_{i_1} + c_{i_1}, \ldots, g_{i_k} + c_{i_k})$, giving $\text{top}(g_i) = f$.

We require the sequence to be ordered according to subexpression containment. That is, for the function and predicate application forms listed above, we require $i_l < i$ for $1 \leq l \leq k$. The soundness property of translation $T_s$ holds for any such ordering, but we get a tighter bound by listing the subexpressions having top-level symbols in $\mathcal{A}$ as early as possible. That is, if $\text{top}(g_i) \in \mathcal{A}$ and $\text{top}(g_j) \in \mathcal{B}$, then $i < j$, unless $g_j$ is a subexpression of $g_i$.

Now introduce a sequence of symbols $\overline{v} = v_1, \ldots, v_n$, where $v_i$ is an integer (respectively, Boolean) symbol when $\text{top}(g_i)$ is an integer or function symbol (respectively, Boolean or predicate symbol). We generate two formulas $C_\mathcal{A}$ and $C_\mathcal{B}$, each of which is a conjunction of consistency constraints by considering each pair of subexpressions $g_i$ and $g_j$, with $i < j$ and $\text{top}(g_i) = \text{top}(g_j)$. These are the same constraints used by Ackermann for removing function applications from a formula [1]. For subexpression $g_i$ of the form $f(g_{i_1} + c_{i_1}, \ldots, g_{i_k} + c_{i_k})$, and $g_j$ of the form $f(g_{j_1} + c_{j_1}, \ldots, g_{j_k} + c_{j_k})$, we include the constraint

$$v_{i_1} = v_{j_1} + (c_{j_1} - c_{i_1}) \land \cdots \land v_{i_k} = v_{j_k} + (c_{j_k} - c_{i_k}) \implies v_i = v_j \quad (9)$$
This constraint is included in either $C_A$ or $C_B$ according to whether $f \in A$ or $f \in B$. Similar constraints are generated when the top-level symbol in $g_i$ and $g_j$ is a predicate symbol $p$.

Let $\hat{\phi}$ be the formula generated by replacing each atomic expression $g_i$ in $\phi$ with the symbol $v_i$. We always replace maximal subexpressions, so that the resulting formula no longer contains any symbols from $\phi$.

Let quantifier $Q_i$ be $\forall$ when $\text{top}(g_i) \in A$, and $\exists$ when $\text{top}(g_i) \in B$.

The soundness-preserving translation of $\psi$ is given by

$$T_s(\psi) \doteq Q_1v_1 Q_2v_2 \cdots Q_nv_n \left[ C_A \implies (C_B \land \hat{\phi}) \right]$$

(10)

**Theorem 2.** For any formula $\psi$ having the structure $\psi \doteq \forall A \exists B \phi$, if $T_s(\psi)$, as given by (10), is valid, then so is $\psi$.

We also provide a completeness preserving translation in [5]. We can test for possible convergence by deciding the validity of this translation.

We now give some examples to demonstrate the capabilities and limitations of our translation method.

**Example 2.** Our first example is a case where we successfully prove soundness.

$$\forall f, y \left[ \forall x \; x = f(x) \right] \implies y = f(f(y))$$

(11)

To get this into the required form, we rewrite it as

$$\forall f, y \exists x \; \neg(x = f(x)) \lor y = f(f(y))$$

We write the subexpressions as follows. To make the resulting formulas more readable, we introduce symbols with names based on the subexpressions, rather than the more generic $v_1, v_2, \ldots, v_n$:

<table>
<thead>
<tr>
<th>Subexpression</th>
<th>$g_1$</th>
<th>$g_2$</th>
<th>$g_3$</th>
<th>$g_4$</th>
<th>$g_5$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Symbol</td>
<td>$y$</td>
<td>$f(y)$</td>
<td>$f(f(y))$</td>
<td>$x$</td>
<td>$f(x)$</td>
</tr>
</tbody>
</table>

For $C_A$ we then get

$$(x = y \implies fx = fy) \land (x = fy \implies fx = fffy) \land (y = fy \implies fy = fffy)$$

For formula $C_B$ we get true, while for $\hat{\phi}$ we get $\neg(x = fx) \lor y = fffy$, and the overall quantifier structure is:

$$\forall y \forall fy \forall fffy \exists x \forall fx$$

It can be easily shown that the QSL formula is valid. We omit the details.
**Example 3.** Our second example illustrates a case where the formula is valid, but the soundness-preserving transformation fails to show this.

\[ \forall f \left[ \forall x f(x) < f(x + 1) \right] \implies \left[ \forall y f(y) < f(y + 2) \right] \]  

(12)

To get this into the required form, we rewrite it as

\[ \forall f \forall y \exists x \neg (f(x) < f(x + 1)) \lor f(y) < f(y + 2) \]

We write the subexpressions as follows.

<table>
<thead>
<tr>
<th>Subexpression</th>
<th>$g_1$</th>
<th>$g_2$</th>
<th>$g_3$</th>
<th>$g_4$</th>
<th>$g_5$</th>
<th>$g_6$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Symbol</td>
<td>$y$</td>
<td>$f(y)$</td>
<td>$f(y + 2)$</td>
<td>$x$</td>
<td>$f(x)$</td>
<td>$f(x + 1)$</td>
</tr>
</tbody>
</table>

For $C_A$ we then get

\[
(x = y \implies f(x) = f(y)) \land (x = y - 1 \implies f(x) = f(y)) \\
\land (x = y + 2 \implies f(x) = f(y + 2)) \land (x = y + 1 \implies f(x) = f(y + 2))
\]

For formula $C_B$ we get \textit{true}, while for $\hat{\phi}$ we get

\[ \neg (f(x) < f(x + 1)) \lor f(y) < f(y + 2) \]

and the overall quantifier structure is:

\[ \forall y \forall f(y) \forall f(y + 2) \exists x \forall f(x) \forall f(x + 1) \]

This formula is not valid.

This example shows the limited capability of our translation $T_s$. It does not do the multiple instantiations of $x$ required to replace the quantified antecedent in (12) with $f(y) < f(y + 1) \land f(y + 1) < f(y + 2)$.

### 6 Results and Discussion

We have implemented a prototype of the convergence testing framework within the UCLID [4] verification tool. Currently, we have only implemented the soundness-preserving translation to QSL. The QSL solvers use different techniques to transform a QSL formula to a quantified Boolean formula (QBF) [15]. All the experiments are performed on a 2GHz Pentium-4 running Linux, with 1 GB of memory.

In this section, we describe our experience with the convergence testing framework for a three-stage arithmetic pipeline given in figure 2. This example originated with the first work on symbolic model checking [7], and has subsequently become a standard for verification research [10,13]. In our version, we make use of both stalling and forwarding to resolve read-after-write hazards in the pipeline. Previous versions used only forwarding, with the result that a new result is written to the register file on each step of operation.
The state elements of the pipeline include a function state variable, an unbounded register file $pRF$. The integer state elements include the different register identifiers, namely $eSRC2$, $eDEST$ and $wDEST$, the data values $eARG1$, $eARG2$ and $wVAL$, and the program counter $pPC$. The Boolean state elements consist of the write enable registers $eWRT$ and $wWRT$. The system functionality is parameterized by uninterpreted function symbols for decoding an instruction, updating the program counter and the ALU. The Boolean state elements are initialized to false and the rest of the state elements take on arbitrary initial values.

The pipeline was symbolically simulated starting from the initial state. The QSL formula produced by the soundness preserving translation was false after $k = 1$ and $k = 2$ steps of simulation. A look at the Boolean state elements indicated that the system indeed does not converge within two steps. However, after $k = 3$ steps of simulation, the QSL formula produced was too large to be solved with the current QSL solver implementation we use [15]. The formula had 53 quantified integer variables, with 6 levels of quantifier alternations, 836 nodes in a Directed Acyclic Graph (DAG) representation of the formula, and the BDD representing the QBF formula exceeds 1 GB of memory. However, we have been able to prove the convergence of two simplified versions of the pipeline processor.

1. For the first case, we removed the data-path components of the processor including the register file, operand values and the write-back value. The remaining pipeline still contains the entire control complexity of the original pipeline including the stalling and the forwarding mechanisms. This model converges after $k = 3$ steps of simulation and our decision procedure detects so within 2 seconds with less than 11 MB of memory. The QSL formula contains 27 quantified integer variables, with 4 levels of quantifier alternations and 249 nodes in the DAG form. Notice that this example contains uninterpreted function symbols but does not contain any function state elements.
2. For the second case, we combined the execute and the write-back stages of
the pipeline into a single stage (making the pipeline 2-stage), but retained
the register file pRF and the data-path. The pipeline was modified to ac-
commodate both stalling and forwarding of data. This example converges
after $k = 2$ steps of simulation and our decision procedure takes 8 seconds
to prove it valid. The memory consumption was about 80 MB. The QSL
formula contains 29 quantified integer variables, with 4 levels of quantifier
alternations and 203 nodes in the DAG form.

We are currently working on alternate translations of QSL formulas to QBF
formulas and hope to test the convergence of the pipeline with a few optimizations.
We are also experimenting with enumeration based QBF solvers including
Quaffle [17].

Discussion. The notion of $k$-convergence is not useful for systems with un-
bounded buffers, since many such systems do not converge. Moreover, our pre-
liminary results indicate that the convergence criterion we present is precise, but
computationally difficult to check. Abstraction techniques, such as predicate ab-
straction [11], allow for greater efficiency at the expense of using an approximate
notion of convergence, and are a promising area for future work.

References

1. W. Ackermann. Solvable Cases of the Decision Problem. North-Holland, Amster-
dam, 1954.
3. R. E. Bryant, S. German, and M. N. Velev. Processor verification using efficient
reductions of the logic of uninterpreted functions to propositional logic. ACM
a logic of counter arithmetic with lambda expressions and uninterpreted functions.
In Computer-Aided Verification (CAV’02), LNCS 2404, pages 78–92, July 2002.
5. Randal E. Bryant, Shuvendu K. Lahiri, and Sanjit A. Seshia. Convergence test-
ing in term-level bounded model checking. Technical Report CMU-CS-03-156,
6. T. Bultan, R. Gerber, and W. Pugh. Symbolic model checking of infinite state
systems using Presburger arithmetic. In Computer-Aided Verification (CAV ’97),
LNCS 1254. Springer-Verlag, June 1997.
verification using symbolic model checking. In Design Automation Conference,
8. J. R. Burch and D. L. Dill. Automated verification of pipelined microprocessor
1994.
Multiway decision graphs for automated hardware verification. Formal Methods in


The ROBDD Size of Simple CNF Formulas

Michael Langberg, Amir Pnueli, and Yoav Rodeh
Weizmann Institute of Science, Rehovot, Israel
\{mikel,amir,yrodeh\}@wisdom.weizmann.ac.il

Abstract. Reduced Ordered Binary Decision diagrams (ROBDDs) are nowadays one of the most common dynamic data structures for Boolean functions. Among the many areas of application are verification, model checking, and computer aided design. In the last few years, SAT checkers, based on the CNF representation of Boolean functions are getting more and more attention as an alternative to the ROBDD based methods. We show the difference between the CNF representation and the ROBDD representation in one of the most degenerate cases – random monotone 2CNF formulas. We examine this model and give almost matching lower and upper bounds for the ROBDD size in different cases, and show that as soon as the formulas are non-trivial the ROBDD size becomes exponential, thus showing perhaps one of the most fundamental advantages of SAT solvers over ROBDDs.

1 Introduction

Automatic manipulation of formulas in propositional logic is of major importance in both theoretical and practical computer science. In the VLSI and process analysis communities Reduced Ordered Binary Decision Diagrams (ROBDDs) are popular. Their usage, initiated by Bryant [B86], has caused a considerable increase of the scale of systems that can be verified. In the last few years SAT checkers have appeared as a very competitive alternative to the ROBDD based techniques, Clarke et al. [BCCF99] probably being the initiator of this trend.

It is a common place saying that ROBDDs and SAT complement each other, i.e., there are cases where the ROBDD technique will work better, and those where SAT will. Indeed, Groote and Zantema [GZ01] show that the ROBDD proof of the pigeon hole principal takes exponential size ROBDDs while the unit resolution proof is polynomial. In the other direction, they also give a family of formulas, where an ROBDD based proof is polynomial, while already the CNF representation is exponential. Ideally, for understanding the different faults and merits of both techniques, we would like to have a characterization of the size relation between the two representations of boolean formulas – in CNF form, and in ROBDD form. Hopefully, such an understanding will help in the construction of a new data structure which will combine the good qualities of both ROBDDs and SAT solvers.

There has been some previous work on the size of ROBDDs, Gropl et al. [GPS01] for example, investigates the largest possible size of an ROBDD over...
all functions over $n$ variables. Bollig and Wegener [BW00] examine the worst case ROBDD size of a function with a given number of 1-inputs (among other questions). Woelfel [W01] gives very tight bounds on the ROBDD size of the integer multiplication function, which was one of the first examples of a function with a polynomially sized circuit but an exponential size ROBDD, proved originally by Bryant [B86].

In this paper we examine a very degenerate type of CNF formulas, monotone 2CNF formulas, consisting only of clauses with 2 variables, and no negation. We consider random monotone 2CNF formulas with $n$ variables where each of the $\binom{n}{2}$ possible clauses is chosen with probability $p$. These formulas are clearly always satisfiable, and the (expected) number of satisfying assignments depends on $p$ (this number decreases as $p$ increases). Moreover, the simple syntactic structure of these formulas may lead one to believe that their ROBDD structure is succinct. We show that this is far from being true.

In this work, we present a full characterization of the ROBDD size of random monotone 2CNF formulas. Namely, for practically every value of $p$, we study the ROBDD size of such random formulas and present matching (up to low order terms) lower and upper bounds on this size. Our results show that except for very small $p$, where the formula is degenerate, or very large $p$, where the formula has only a polynomial number of satisfying assignments, the most probable ROBDD size (under any ordering of the variables in the formula) is highly exponential, very closely related to the number of satisfying assignments to the formula. Thus we show that the ROBDD reductions are of little use when handling these simple CNF formulas.

Let $\varphi_p$ be a random monotone 2CNF formula with $n$ variables, in which each of the $\binom{n}{2}$ possible clauses is chosen with probability $p$. Our results can be (roughly) summarized as follows:

1. Let $p < (1 - \epsilon) \frac{1}{n}$, where $\epsilon > 0$ is constant. Notice that in this case a random formula $\varphi_p$ is expected to have less than $n/2$ clauses (implying that each variable is expected to appear at most once in $\varphi_p$). Then w.h.p. the ROBDD size of $\varphi_p$ is polynomial.

2. Let $p$ satisfy (a) $(1 + \epsilon) \frac{1}{n} < p$ for some constant $\epsilon > 0$, and (b) For every constant $\alpha > 0$, $p < 1 - \frac{1}{n^\alpha}$ (i.e. $p$ is not very small or large). Then w.h.p. the ROBDD size of $\varphi_p$ is super polynomial. Specifically, we show that for small values of $p$ in the range defined above, the ROBDD size of $\varphi_p$ is in the range $[2^{\frac{1}{n} polylog^\infty}, 2^{\log^2 n}]$; and for large values of $p$, the ROBDD size of $\varphi_p$ is equal to $2^{\Theta\left(\frac{\log^2 n}{\log \frac{1}{(1 - p)}}\right)}$ (w.h.p.). For example for $p = 1/\sqrt{n}$ the ROBDD size of $\varphi_p$ is roughly $2^{\sqrt{n} polylog n}$, and for $p = 1/2$ this size is roughly $2^{\log^2 n} = n^{\log n}$. Notice the sharp jump in the ROBDD size, with respect to case 1 above, with a very small increase of $p$.

3. If there exists some constant $\alpha > 0$ such that $p > 1 - \frac{1}{n^\alpha}$, then w.h.p. the ROBDD size of $\varphi_p$ is again polynomial.
An important point in these bounds, is that the upper bounds in items 2 and 3 above are derived by showing an upper bound to the number of satisfying assignments to the formula. The fact that these bounds practically match the lower bounds means that the ROBDD reductions are of very little use for these kinds of formulas – we might as well have written a list of all satisfying assignments as a description of the formula.

Along the way, we show that for small $p$, it is the pathwidth of the formula which determines the optimal ROBDD size. This parameter captures in a simple manner the concept of information flow that is caused by the variable ordering in the ROBDD method. In our restricted setting, this result can be seen as a matching lower bound to Berman’s [B89] classic upper bound on ROBDD size, relating circuit structure and ROBDD size using a notion similar to our pathwidth. Also, this result formalizes the common sense intuition of ROBDD ordering, and thus shows one of the fundamental drawbacks of ROBDDs, if an ordering does not put related variables close to one another – the ROBDD size will be large.

The remainder of this paper is organized as follows. In Section 2 we present the main definitions and notation that will be used throughout this work. Specifically we show a natural characterization of random monotone 2CNF formulas $\varphi_p$ on $n$ variables by the distribution $G_{n,p}$ on graphs with $n$ vertices. In Section 3 we show a connection between the ROBDD size of monotone 2CNF formulas and certain combinatorial graph properties. We then define the pathwidth of a formula, a notion which plays a major role in our analysis. Finally, in Section 4 we state the upper and lower bounds sketched above rigorously and proceed in their proof. Due to space limitations, some of our results appear without detailed proof. A full version can be found at,

http://www.wisdom.weizmann.ac.il/~verify/publications/2003/LPR03.html

2 Preliminaries and Notation

2.1 Graphs

For a graph $G$, denote its set of vertices by $V$, and its set of edges by $E$. Let $n$ be the size of $V$, and $m$ be the size of $E$. We denote by $d(G)$ the maximum degree of a vertex in $G$. For a set of vertices $U \subseteq V$ define its set of neighbors as $\Gamma_G(U) = \{ v \in V \mid v \notin U, \exists u \in U, (u,v) \in E \}$. Denote the subgraph induced by a subset $U$ of vertices as $G|_U$, i.e., $G|_U = \langle U, E \cap (U \times U) \rangle$. We say $U \subseteq V$ is an independent set if the edge set of $G|_U$ is empty. Let $ID(G)$ denote the set of independent sets of the graph $G$. Denote the size of the largest independent set in $G$ by maxID($G$). The definitions above imply that,

**Proposition 1.** $|ID(G)| \leq n^{\text{maxID}(G)}$

Let $G_V$ be the set of graphs on vertex set $V$. For short, we mark $G_n = G_{[1,n]}$. 

2.2 Boolean Formulas

Let $\Delta_V$ denote the set of Boolean assignments to the variable set $V$, $\Delta_V = \{\alpha \mid \alpha : V \to \{0,1\}\}$. Let $\Phi_V = \{\varphi \mid \varphi \subseteq \Delta_V\}$ denote the set of all Boolean formulas on the variable set $V$ (a function is characterized by its set of satisfying assignments). For $\alpha \in \Delta_V$, $U \subseteq V$, denote by $\alpha|_U \in \Delta_U$ the restriction of assignment $\alpha$ to the set $U$. We would also like to consider the restriction of the formula $\varphi$ to a partial assignment. For $\varphi \in \Phi_V$, $U \subseteq V$, and some $\alpha \in \Delta_U$, let

$$\varphi|_\alpha = \{\beta \in \Delta_{V\setminus U} \mid \exists \gamma \in \varphi, \gamma|_U = \alpha \text{ and } \gamma|_{V\setminus U} = \beta\}$$

Again we will mark $\Phi_n = \Phi_{[1,n]}$, and $\Delta_n = \Delta_{[1,n]}$.

2.3 Random Monotone 2CNF Formulas

In 2.2 we considered only the semantics of boolean formulas by characterizing them using their satisfying set of assignments. We now proceed to consider the representation of a formula, its syntax. We consider a restricted class of CNF formulas, monotone 2CNF formulas. A monotone 2CNF formula over variable set $V$ is the conjunction of a set of clauses of the form $(a \vee b)$ where $a, b$ are in $V$.

We can equivalently model such a formula by a graph $G \in \mathcal{G}_V$, where each edge $(a, b)$ in the graph stands for the clause $(a \vee b)$. We then get that the formula corresponding to the graph $G$ is

$$\varphi_G = \{\alpha \in \Delta_V \mid \forall (i, j) \in E(G), \alpha(i) = 1 \text{ or } \alpha(j) = 1\}$$

We will consider such random formulas, using the random model $\mathcal{G}_{n,p}$, where $G \in \mathcal{G}_{n,p}$ is a graph on vertices $[1,n]$, where each possible edge is in the graph with probability $p$, uniformly and independently. We will say an event in $\mathcal{G}_{n,p}$ happens with high probability if it happens with probability tending to 1 as $n$ approaches infinity.

2.4 ROBDDs – Reduced Ordered Binary Decision Diagrams

**Definition 1.** An OBDD on $[1,n]$ is an edge labeled directed graph, whose sinks are labeled by Boolean constants FALSE and TRUE, and whose non sink (or inner) nodes are labeled by elements of $[1,n]$. Each inner node has two outgoing edges, one labeled by 0 and the other by 1. An edge leading from an $i$-node must end in a sink or a $j$-node, where $j > i$. Each inner node $v$ with label $k$, represents a Boolean formula $\varphi_v \in \Phi_{[1,n]}$ defined in the following way. In order to check if $\alpha \in \varphi_v$, $\alpha \in \Delta_{[1,n]}$, start at $v$. After reaching an $i$-node, choose the outgoing edge with label $\alpha(i)$, until a sink is reached. If the label of the sink is TRUE then $\alpha \in \varphi_v$, if it is FALSE then $\alpha \notin \varphi_v$. The size of the OBDD is defined to be its number of nodes.
Lemma 2. \( \alpha \) for simplicity.

For Proposition 2, we have a slightly different version of ROBDDs, called Quasi-reduced OBDDs (QOBDDs). In this paper we will actually consider this latter type, because of the following two lemmas (see [BW00] for example):

**Lemma 1.** The number of i-nodes, \( 1 < i \leq n \), of the QOBDD of \( \varphi \in \Phi_n \) is \( |\{ \varphi_{|\alpha} \mid \alpha \in \Delta_{i-1} \}| \).

**Lemma 2.** If \( s_R \) is the size of the ROBDD of \( \varphi \in \Phi_n \), and \( s_Q \) is the size of its QOBDD, then \( \frac{1}{n} s_Q \leq s_R \leq s_Q \).

The first Lemma allows us to deal with the size of QOBDD in a simple manner, and the second Lemma shows that the size of QOBDDs is practically the same as that of ROBDDs, especially since all size lower bounds we show will have an exponential nature. Therefore, for the remainder of the paper, we will examine only QOBDDs. For \( \varphi \in \Phi_n \), we denote by BDD(\( \varphi \)), the size of \( \varphi \)'s QOBDD. For simplicity, we will not count the root node and the two leaf nodes of the QOBDD when calculating BDD(\( \varphi \)), this changes the QOBDD size by at most 3, and so is immaterial. We get the following proposition,

**Proposition 2.** For \( \varphi \in \Phi_n \), BDD(\( \varphi \)) = \( \sum_{k=1}^{n-1} |\{ \varphi_{|\alpha} \mid \alpha \in \Delta_k \}| \)

We note the following useful upper bound on QOBDD size.

**Proposition 3.** For \( \varphi \in \Phi_n \), BDD(\( \varphi \)) < \( n(|\varphi| + 1) \).

**Proof.** By Proposition 2,

\[
\text{BDD}(\varphi) = \sum_{k=1}^{n-1} |\{ \varphi_{|\alpha} \mid \alpha \in \Delta_k \}| \leq \sum_{k=1}^{n-1} \left( |\{ \alpha \in \Delta_k \mid \varphi_{|\alpha} \neq \emptyset \}| + 1 \right)
\]

For every \( \alpha \in \Delta_k \), such that \( \varphi_{|\alpha} \neq \emptyset \), there is at least one \( \beta \in \varphi \) s.t. \( \beta_{|[1,k]} = \alpha \). Choose one of these \( \beta \) and mark it by \( \beta_{\alpha} \). Clearly if \( \alpha_1 \neq \alpha_2 \) then \( \beta_{\alpha_1} \neq \beta_{\alpha_2} \), and so \( |\{ \alpha \in \Delta_k \mid \varphi_{|\alpha} \neq \emptyset \}| \leq |\varphi| \) and we conclude, BDD(\( \varphi \)) \( \leq (n-1)(|\varphi| + 1) < n(|\varphi| + 1) \). \( \square \)

As is well known, the QOBDD of a formula \( \varphi \) depends on the specific ordering of variables in \( \varphi \). Denote by \( S_n \) the set of permutations on the set \([1,n]\). For a formula \( \varphi \in \Phi_n \), and a permutation \( \sigma \in S_n \), denote

\( \varphi^\sigma = \{ \alpha \mid \exists \beta \in \varphi, \forall v \in V, \alpha(\sigma(v)) = \beta(v) \} \)

\( \varphi^\sigma \) is the result of changing the names of the variables of \( \varphi \). This change may result in a change of BDD(\( \varphi \)), and in fact there are known examples (see for example [CGP]), where BDD(\( \varphi \)) is polynomial, while for some \( \sigma \), BDD(\( \varphi^\sigma \)) is exponential. We therefore denote

\[
m\text{BDD}(\varphi) = \min_{\sigma \in S_n} \text{BDD}(\varphi^\sigma)
\]

Clearly, Proposition 3 applies also to mBDD(\( \varphi \)).
3 QOBDD Size vs. Combinatorial Graph Properties

Let $G$ be a graph in $\mathcal{G}_n$. Let $\varphi = \varphi_G \in \Phi_n$ be the 2CNF formula corresponding to $G$. In this section we show various connections between combinatorial properties of $G$ and the size of the QOBDD of $\varphi$. We will need the following definition. For $\alpha \in \Delta_n$ denote $Z_\alpha = \{ v \in V \mid \alpha(v) = 0 \}$.

Lemma 3. ID$(G) = \{ Z_\alpha \mid \alpha \in \varphi \}$

Proof. Let $Z$ be an independent set in $G$. Consider the assignment $\alpha$ which assigns a value of 0 to every vertex in $Z$ and a value of 1 to the remaining vertices in $V \setminus Z$. Clearly $Z = Z_\alpha$, furthermore as $Z$ is independent we conclude that $\alpha \in \varphi$ implying that $Z \in \{ Z_\alpha \mid \alpha \in \varphi_G \}$. For the other direction, consider an assignment $\alpha \in \varphi$. By the definitions above, $Z_\alpha$ must be an independent set in $G$. \hfill \Box

Corollary 1. For $\varphi \in \Phi_n$, BDD$(\varphi) < n(|\text{ID}(G)| + 1)$.

Theorem 1. For $G \in \mathcal{G}_n$, Setting,

$$A_G = \left\{ \Gamma \mid \Gamma = \Gamma_G(I) \cap [k+1,n], \ I \in \text{ID} \left( G_{[1,k]} \right) \right\}$$

The size of the $k+1$ level in $\varphi$’s QOBDD (under natural ordering) is either $|A_G|$ or $|A_G| + 1$

Proof. Consider the set

$$A_\varphi = \left\{ \varphi|_{\alpha} \mid \alpha \in \Delta_{[1,k]}, \ \varphi|_{\alpha} \neq \emptyset \right\}.$$

The size of the $k+1$ level in $\varphi$’s QOBDD (under natural ordering) is exactly the size of $A_\varphi$, possibly plus 1, if there is some $\alpha$ s.t. $\varphi|_{\alpha} = \emptyset$. Hence, it suffices to present a one to one function from $A_\varphi$ to $A_G$ and vice versa. For the first direction consider the function which associates with every $\varphi|_{\alpha}$ the set $\Gamma_G(Z_\alpha) \cap [k+1,n]$ (where $Z_\alpha$ is as defined above). As $\varphi|_{\alpha} \neq \emptyset$ we have that $Z_\alpha$ in as independent set in $G_{[1,k]}$. Now assume two formulas $\varphi|_{\alpha_1}$ and $\varphi|_{\alpha_2}$ that are not equal. Namely (w.l.o.g.) there exists some assignment $\beta \in \Delta_{[k+1,n]}$ such that $\beta \in \varphi|_{\alpha_1}$ but $\beta \notin \varphi|_{\alpha_2}$. For $i = 1, 2$ let $\gamma_i \in \Delta_{[1,n]}$ be the assignment obtained by concatenating $\alpha_i$ and $\beta$. By these definitions $\gamma_1 \in \varphi$ and $\gamma_2 \notin \varphi$. Hence, it must be the case that $\gamma_2$ violates some clause, say the clause including the $i$’th and $j$’th variables, where $i < j$ (that is $\gamma_2(i) = \gamma_2(j) = 0$).

Now (by contradiction) assume that $\Gamma_1 = \Gamma_G(Z_{\alpha_1}) \cap [k+1,n]$ is equal to $\Gamma_2 = \Gamma_G(Z_{\alpha_2}) \cap [k+1,n]$. Recall that $\varphi$ is a monotone 2CNF formula, it is satisfied by $\gamma_1 = \alpha_1 \beta$, and it is not satisfied by $\gamma_2 = \alpha_2 \beta$. Moreover, $\varphi|_{\alpha_2}$ is not equal to $\emptyset$. By the fact that $\varphi$ is satisfied by $\gamma_1$ we conclude that all variables in $\Gamma_1$ have value 1 under the assignment $\beta$ implying that they have value 1 both in the assignment $\gamma_1$ and $\gamma_2$. Hence, it cannot be the case that $i$ or $j$ belong
to \( \Gamma_2 \). By the fact that \([1,k] \setminus Z_{\alpha_2}\) is set to 1 in \( \gamma_2 \) it cannot be the case that \( i \) or \( j \) are in \([1,k] \setminus Z_{\alpha_2}\). By the fact that \( \varphi|_{\alpha_2} \neq \emptyset \) it cannot be the case that both \( i \) and \( j \) are in \( Z_{\alpha_2} \). We conclude that it must be the case that both \( i \) and \( j \) are in \([k+1,n] \setminus \Gamma_2 \). But the value of such \( i \) and \( j \) are determined by \( \beta \), and by the fact that \( \gamma_1 = \alpha_1 \beta \in \varphi \) we conclude that either the value of \( i \) or \( j \) is 1 in \( \gamma_2 \).

For the other direction, consider the function which associates with each \( \Gamma \in A_G \) the assignment \( \alpha \in \Delta_{[1,k]} \) which is defined as follows. Let \( Z \) be some independent set in \( G_{[1,k]} \) such that \( \Gamma_G(Z) \cap [k+1,n] = \Gamma \), define \( \alpha(i) \) to be zero iff \( i \in Z \). As \( Z \) in an independent set in \( G_{[1,k]} \) it is the case that \( \varphi|_{\alpha} \neq \emptyset \) and thus in \( A_G \). Let \( \Gamma_1 = \Gamma_G(Z_1) \cap [k+1,n] \) and \( \Gamma_2 = \Gamma(Z_2) \cap [k+1,n] \) be two different subsets in \( A_G \). We will show that for corresponding \( \alpha_1 \) and \( \alpha_2 \) as defined above the functions \( \varphi|_{\alpha_1} \) and \( \varphi|_{\alpha_2} \) differ. Let (w.l.o.g.) \( i \) be a vertex in \( \Gamma_1 \setminus \Gamma_2 \) (note that \( i \in [k+1,n] \)). Let \( \beta \in \Delta_{[k+1,n]} \) be defined such that \( \beta(i) = 0 \) and \( \beta(j) = 1 \) for all \( j \neq i \). The vertex \( i \) is connected by an edge to \( Z_1 \) implying that the assignment \( \gamma_1 \) which is the concatenation of \( \alpha_1 \) and \( \beta \) does not satisfy \( \varphi \). We conclude that \( \beta \notin \varphi|_{\alpha_1} \). On the other hand, the vertex \( i \) is not connected to any vertices in \( Z_2 \), implying (in a similar manner) that \( \beta \in \varphi|_{\alpha_2} \).

In the following, we define the notion of the pathwidth of a graph (as introduced in [RS83]). Given an ordering of the vertices of a given graph \( G \) the pathwidth of \( G \) is defined as follows:

**Definition 2.** For \( G \in \mathcal{G}_n \), denote \( \text{PW}(G) = \max_{k \in [1,n]} |\Gamma_G([1,k])| \).

Next we present upper and lower bounds on the QOBDD size of \( \varphi \) using the pathwidth notion. Afterwards we show that the pathwidth of a graph is monotone with respect to edge contractions and vertex and edge deletions. We will use this property later on in Section 4.

### 3.1 Upper Bound

**Lemma 4.** \( \text{BDD}(\varphi) \leq n(2^{\text{PW}(G)} + 1) \)

**Proof.** Using Theorem 1 we need to show that for every \( k \) the size of the set

\[
\left\{ \Gamma_G(I) \cap [k+1,n] \mid I \in \text{ID}(G_{[1,k]}) \right\}
\]

is of size at most \( 2^{\text{PW}(G)} \). However, since \( I \subseteq [1,k] \), then \( |\Gamma_G(I) \cap [k+1,n]| \leq |\Gamma_G([1,k])| \leq \text{PW}(G) \), and therefore the number of possible sets of the form \( \Gamma_G(I) \cap [k+1,n] \) is at most \( 2^{\text{PW}(G)} \). \( \square \)

### 3.2 Lower Bound

We first state without proof the following lemma, which is proved using a simple greedy strategy.

**Lemma 5.** For \( G \in \mathcal{G}_n \), \( \max\text{ID}(G) \geq \frac{n}{d(G)+1} \)
Lemma 6. \( BDD(\varphi) \geq 2^{\frac{PW(G)}{(d(G))}} \)

Proof. Mark \( h = PW(G) \) and \( d = d(G)+1 \). Set \( k \) to be such that \( |\Gamma_G([1,k])| = h \). Using Theorem 1 we want to show that

\[
\left| \left\{ I \mid \Gamma = \Gamma_G(I) \cap [k+1,n], \ I \in \text{ID} \left( G_{[1,k]} \right) \right\} \right| \geq 2^{\frac{h}{d^2}} \tag{1}
\]

For every vertex \( v \in [1,k] \) denote \( A_v = \Gamma_G(\{v\}) \cap [k+1,n] \). We will find a specific independent set \( \mathcal{I} \) of \( G_{[1,k]} \) such that

1. For every \( u \in \mathcal{I}, A_u \neq \emptyset \).
2. For every \( u,v \in \mathcal{I}, A_u \cap A_v = \emptyset \).
3. \( |\mathcal{I}| \geq \frac{h}{d^2} \)

Finding such an \( \mathcal{I} \) will prove Equation (1), by letting \( I \) run over all subsets of \( \mathcal{I} \).

Since \( |\Gamma_G([1,k])| = h \), then \( | \cup A_v | \geq h \). Therefore there are at least \( \frac{h}{d} \) such sets \( A_v \neq \emptyset \). Noticing that each vertex \( u \in [k+1,n] \) can appear in at most \( d \) sets \( A_v \), and since \( |A_v| < d \), we have that each \( A_v \) intersects at most \( d^2 \) other such sets. By Lemma 5, there are at least \( \frac{h}{d^2} \cdot \frac{1}{d^2} = \frac{h}{d^2} \) such sets that do not intersect each other. Denote by \( H \subseteq [1,k] \) the set of \( v \)'s corresponding to these \( A_v \)'s. Again, using Lemma 5, and by the fact that \( |H| \geq \frac{h}{d^2} \), we can find a subset \( \mathcal{I} \) of \( H \) that is an independent set in \( G \). This \( \mathcal{I} \) satisfies all three properties above. \( \square \)

### 3.3 Optimal Ordering

The previous results we have shown all consider the natural ordering of variables in \( \varphi \). In the following we extend these results naturally to obtain the connections needed between the properties of \( G \) and the QOBDD size of an arbitrary ordering of \( \varphi \). Let \( \sigma \in S_n \) and \( G \in G_n \). The graph \( G \) obtained after a renaming of \( V \) according to \( \sigma \) is defined as

\[
G^\sigma = (V, \{(\sigma(i), \sigma(j)) \mid (i,j) \in E(G)\}).
\]

It is not hard to verify that \( (\varphi^\sigma_G)^\sigma = \varphi_{(G^\sigma)} \), implying that \( \text{mBDD}(\varphi_G) = \min_{\sigma} \text{BDD}(\varphi_{(G^\sigma)}) \). We now define the minimal pathwidth of a graph.

**Definition 3.** The minimal pathwidth of \( G \) is \( \text{mPW}(G) = \min_{\sigma} \text{PW}(G^\sigma) \).

It is straightforward to verify that Lemma 6 and Lemma 4 now imply:

**Theorem 2.** \( 2^{\frac{\text{mPW}(G)}{(d(G))}} \leq \text{mBDD}(\varphi_G) \leq n(2^{\text{mPW}(G)} + 1) \).

We believe this result to be of independent interest, since it shows the close connection between the pathwidth of the graph and the QOBDD size of the formula. If all orderings of the vertices result in many clauses being separated – the QOBDD size will be large, exponential in the pathwidth.
3.4 Minors

For a graph \( G \in \mathcal{G}_n \), and an edge \((i, j) \in E(G)\), the result of contracting the edge \((i, j)\) in \( G \) is the graph \( G[\{1, n\} \setminus \{i\}] \) with the addition of the edges \( \{(j, x) \mid (i, x) \in E(G)\} \). We say \( H \) is a minor of \( G \) if it is the result of consecutive edge contractions of \( G \), vertex deletions and edge deletions of \( G \). In our application, \( H \) does not have any multiple edges (i.e. \( H \) is not a multi graph).

**Lemma 7.** If \( H \) is a minor of \( G \) then \( \text{mPW}(H) \leq \text{mPW}(G) \).

**Proof.** For one vertex or edge deletion the result is trivial. We therefore prove it for one edge contraction and the Lemma follows by induction. Let \( G \in \mathcal{G}_n \), and assume w.l.o.g. that \( \text{PW}(G) = \text{mPW}(G) \). Assume an edge \((i, j)\) is contracted in \( G \) to give \( H \), where \( i < j \). We claim that the following ordering of \( H \)'s vertices gives a pathwidth of \( H \) which is at most \( \text{PW}(G) \): \( 1, 2, \ldots, i - 1, i + 1, \ldots, n \).

1. For all \( k \leq i - 1 \), \( \Gamma_H([1, k]) = \Gamma_G([1, k]) \setminus \{i\} \).
2. For all \( k \geq j \), \( \Gamma_H([1, k] \setminus \{i\}) = \Gamma_G([1, k]) \).
3. For all \( i < k < j \), \( \Gamma_H([1, k] \setminus \{i\}) \subseteq \Gamma_G([1, k] \setminus \{i\}) \setminus \{i\} \cup \{j\} \subseteq \Gamma_G([1, k]) \)

And so, for all \( k \): \(|\Gamma_H([1, k] \setminus \{i\})| \leq |\Gamma_G([1, k])|\), to conclude. \(\square\)

4 QOBDD Size of Random 2CNF

We now proceed to examine the most probable QOBDD size of a random formula in \( \mathcal{G}_{n,p} \) for different values of \( p \). Our analysis is divided into several cases, each examining a different range of values for \( pn \). The value \( pn \) is (approximately) twice the expected ratio between the number of clauses and the number of variables in the formula, and is therefore a good indicator for the expected structure and complexity of the formula. We prove the following results (with high probability over the random formula \( \varphi \)).

1. For \( pn < 1 - \epsilon \), where \( \epsilon > 0 \) is constant, \( \text{mBDD}(\varphi) = O(n \log n) \). We will see that the probable formulas in this case are very degenerate, since the graph will most probably contain only very small connected components.
2. For \( 1 + \epsilon < pn < o(n) \), where \( \epsilon > 0 \) is constant,

\[
2^{O(\frac{1}{p} \log^{-6} n)} < \text{mBDD}(\varphi) < 2^{O(\frac{1}{p} \log^2 n)}
\]

This implies that the QOBDD size is highly exponential\(^1\) for small values of \( p \), and slowly decreases as \( p \) approaches 1. For example, when \( pn = \sqrt{n} \), the QOBDD size is \( 2^{\sqrt{n} \cdot \text{polylog} n} \) (which is still highly exponential). Notice the sharp jump in the QOBDD size, with respect to the previous case, with a very small increase of \( pn \).

\(^1\) For \( pn \geq 12 \) we show an improved lower bound of \( 2^{O(\frac{1}{p} \log^{-4} n)} \).
3. We improve the bounds above for large values of $p$. Let $p$ satisfy (a) For every constant $\epsilon > 0$, $pn \geq n^{1-\epsilon}$, and (b) For every constant $\alpha < 1$, $pn \leq n - n^\alpha$. (I.e. $pn$ is large but not too large). Then

$$\text{mBDD}(\varphi) = 2^{\Theta\left(\frac{\log^2 n}{\log (1/1-p)}\right)}$$

In this case we get matching lower and upper bounds (up to constant factors in the exponent). Since $pn < n - n^\alpha$ for all $\alpha < 1$, this means that $\text{mBDD}(\varphi)$ is super polynomial. For example, when $p = \frac{1}{2}$, $\text{mBDD}(\varphi) = 2^{\Theta(\log^2 n)} = n^{\Theta(\log n)}$

4. If there exists a constant $0 < \alpha < 1$ s.t. $pn > n - n^\alpha$, then $\text{mBDD}(\varphi) = n^{O(1)}$, i.e., is polynomial.

An important point in these bounds, is that all upper bounds (except the one for $pn < 1 - \epsilon$) are derived using Corollary 1, by showing an upper bound to the number of satisfying assignments to the formula. The fact that these bounds practically match the lower bounds means that the QOBDD reductions are of very little use for these kinds of formulas – we might as well have written a list of all satisfying assignments as a description of the formula.

4.1 Case 1: $pn < 1 - \epsilon$

We start by stating the following theorem appearing in [JLR] which states that w.h.p. $G$’s connected components are all of size at most $O(\log n)$ and are all almost trees

**Theorem 3.** ([JLR]): If $G \in \mathcal{G}_{n,p}$, where $pn < 1 - \epsilon$ for some constant $\epsilon > 0$, then w.h.p. $G$’s connected components are of size $O(\log n)$, and are either trees, or trees with one extra edge.

We now show that the QOBDD size of a graph that is a tree is small. This is done by showing that the pathwidth of a tree is small. Combining these two facts we will conclude that w.h.p. $\text{mBDD}(\varphi) < O(n \log n)$.

**Lemma 8.** For $T \in \mathcal{G}_n$, where $T$ is a tree, $\text{mpW}(T) \leq \log_2 n$.

**Proof.** If $n = 1$ then clearly $\text{mpW}(T) = 0 = \log_2(1)$. We order the vertices of the tree recursively. Number the $s$ subtrees rooted at the children of the root vertex $r$ according to their size, i.e., $T_1$ is the largest, $T_2$ the second, and so on until $T_s$, the smallest subtree. Order each of the subtrees recursively, the vertices of $T_1$ are ordered $t^1_1, t^1_2, \ldots, t^1_{k_1}$, the vertices of $T_2$ are ordered $t^2_1, t^2_2, \ldots, t^2_{k_2}$ and so on. Now order all the vertices in the following way:

$$t^1_1, t^1_2, \ldots, t^1_{k_1}, t^2_1, t^2_2, \ldots, t^2_{k_2}, \ldots, t^s_1, t^s_2, \ldots, t^s_{k_s}, r$$

We claim that this ordering gives a pathwidth of at most $\log_2 n$.

1. For $k \in [1, k_1 - 1], \Gamma_T(\{t^1_1, \ldots, t^1_k\}) = \Gamma_{T_1}(\{t^1_1, \ldots, t^1_k\})$. By the induction hypothesis this set is of size at most $\log_2 |T_1| \leq \log_2 n$. 
2. For \( k = k_1 \), \( \Gamma_T(\{t^1_1, \ldots, t^1_k\}) = |\{r\}| = 1 \leq \log_2 n \), since \( n \) is at least 2.
3. For \( 1 < i \leq s \), for \( k \in [1, k_i] \), \( \Gamma_T(\{t^1_1, \ldots, t^1_{k_1}, \ldots, t^i_1, \ldots, t^i_k\}) = \Gamma_T(T_i) \cup \cdots \cup \Gamma_T(\{t^i_1, t^i_k\}) = \{r\} \cup \Gamma_T(\{t^i_1, t^i_k\}) \). By the induction hypothesis we get that this set is of size at most \( \log_2 |T_i| + 1 \). However, since \( i > 1 \), then \( T_i \) is not the largest subtree child of \( r \), and therefore must satisfy \( |T_i| < \frac{1}{2}|T| \). Which gives \( \log_2 |T_i| + 1 \leq \log_2 n \).

\[ \square \]

**Theorem 4.** If \( G \in \mathcal{G}_{n,p} \) where \( pn < 1 - \epsilon \) for some constant \( \epsilon > 0 \), then w.h.p. \( \text{mBDD}(\varphi_G) = O(n \log n) \).

**Proof.** According to Theorem 3, w.h.p. \( G \)'s connected components \( C_1, \ldots, C_k \) are all of size at most \( O(\log n) \) and are each a tree with maybe an addition of one edge. Since an extra edge can increase the pathwidth of a graph by at most 1, then by Lemma 8 we have that for all \( i \), \( \text{mPW}(G_{|C_i}) \leq \log_2 |C_i| + 1 \). Therefore, by Theorem 2 we have \( \text{mBDD}(G_{|C_i}) \leq |C_i| \cdot (2^{\log_2 |C_i| + 1} + 1) < 3|C_i|^2 \). It is not hard to verify that this implies

\[
\text{mBDD}(\varphi_G) \leq n + \sum_{i=1}^{k} \text{mBDD}(G_{|C_i}) \leq n + 3 \sum_{i} |C_i|^2
\]

Denoting \( M = \max_i |C_i| \), we have that \( \text{mBDD}(\varphi_G) \leq n + 3 \frac{n}{M} M^2 \), and since for all \( i \), \( |C_i| = O(\log n) \), \( \text{mBDD}(\varphi_G) = O(n \log n) \).

\[ \square \]

**4.2 Lower Bound of Case 2: \( 1 + \epsilon < pn = o(n) \)**

We start by showing that for \( pn > 12 \) w.h.p. \( \text{mPW}(G) > \frac{1}{4} n \). We also show that for \( pn = O(1) \), w.h.p. \( d(G) = O(\log n) \), and now using Theorem 2 we get an exponential lower bound for \( \text{mBDD}(\varphi) \) in the case \( 12 < pn = O(1) \). From this we easily derive a lower bound for larger \( pn \), while \( pn = o(n) \).

The result for \( 1 + \epsilon < pn \leq 12 \) now follows by finding a minor \( H \) of \( G \) that has a large pathwidth. We show that \( G \) contains a minor \( H \) which is actually an element of \( \mathcal{G}_{l,p'} \), where \( l, p' > 12 \), and since \( \text{mPW}(G) \geq \text{mPW}(H) \), we get an exponential (in \( l \)) lower bound for \( BDD(\varphi) \). Details follow.

**Lemma 9.** For \( G \in \mathcal{G}_{n,p} \), where \( pn > 12 \), w.h.p., \( \text{mPW}(G) > \frac{1}{4} n \).

**Proof.** We show that if \( pn > 12 \), then w.h.p., for \( G \in \mathcal{G}_{n,p} \), every set \( V \subseteq V(G) \), where \( |V| = \frac{1}{2} n \), satisfies \( |\Gamma_G(V)| > \frac{1}{4} n \). This will prove the lemma.

For fixed \( A, B \subseteq V \), where \( |A| = \frac{1}{2} n \) and \( |B| = \frac{1}{4} n \),

\[
\Pr[\Gamma_G(A) \subseteq B] = (1 - p)^{|A|(n-|A|+|B|))} = (1 - p)^{\frac{1}{2} n \frac{1}{2} n} < e^{-\frac{pn^2}{8}}
\]

If we have that for all relevant \( A \) and \( B \), \( \Gamma_G(A) \not\subseteq B \) then the graph is as we want it. We bound the probability of this not happening using a simple union bound:

\[
2^n \cdot 2^n \cdot e^{-\frac{pn^2}{8}} = e^n(2\log 2 - \frac{1}{8} pn)
\]

This tends to zero if \( pn > 12 \).

\[ \square \]
It is not hard to verify that w.h.p. the maximal degree \( d(G) \) of a graph \( G \in \mathcal{G}_{n,p} \) with \( pn = O(1) \) is of size \( O(\log n) \). We thus conclude, by Theorem 2 that

**Corollary 2.** For \( G \in \mathcal{G}_{n,p} \) where \( 12 < pn = O(1) \), w.h.p., \( \text{mBDD}(\varphi_G) > 2^{\Omega\left(\frac{n}{\log^2 n}\right)} \).

We now turn to study values of \( p \) that satisfy \( 12 < pn = o(n) \).

**Theorem 5.** For \( G \in \mathcal{G}_{n,p} \), where \( 12 < pn = o(n) \), w.h.p., \( \text{mBDD}(\varphi_G) > 2^{\Omega\left(\frac{1}{p\log^4 n}\right)} \).

**Proof.** Set \( k = \frac{13}{p} \), and examine the random behavior of \( G_{[1,k]} \), which is actually an element of \( \mathcal{G}_{k,p} \). Since \( pn = o(n) \), \( p = o(1) \) and therefore \( k \) is unbounded, so by Corollary 2, w.h.p. \( \text{mBDD}(\varphi_{G_{[1,k]}}) = 2^{\Omega\left(\frac{k}{\log^2 n}\right)} = 2^{\Omega\left(\frac{1}{p\log^4 n}\right)} \). Since \( \frac{1}{p} < n \), we get \( \frac{1}{p}\log^{-4} \frac{1}{p} > \frac{1}{p}\log^{-4} n \).

A simple observation is that if \( H = G_{[1]} \), then \( \text{mBDD}(\varphi_G) \geq \text{mBDD}(\varphi_H) \), and this gives us the desired result. \( \square \)

It is left to show our bounds for \( 1 + \epsilon < pn \leq 12 \). To do so we show that for \( G \in \mathcal{G}_{n,p} \), \( pn > 1 + \epsilon \), \( G \) contains a minor \( H \) that behaves as a random graph in \( \mathcal{G}_{k,p'} \), where \( p'k > 12 \). This, combined with the analysis above will prove that \( H \) has large pathwidth.

**Theorem 6.** ([JLR]): If \( G \in \mathcal{G}_{n,p} \) and \( pn > 1 + \epsilon \), for some constant \( \epsilon > 0 \), then there is some constant \( \theta \) s.t. w.h.p. the biggest connected component of \( G \) is of size at least \( \theta n \).

**Theorem 7.** For \( G \in \mathcal{G}_{n,p} \), where \( 1 + \epsilon < pn \leq 12 \) and \( \epsilon > 0 \) is constant, w.h.p. \( \text{mBDD}(\varphi_G) > 2^{\Omega\left(\frac{n}{\log^6 n}\right)} \).

**Proof.** For two reals \( 0 \leq p_1, p_2 \leq 1 \), s.t., \( p_1 + (1 - p_1)p_2 = p \), we can view \( G \) as the union of two graphs, \( G_1 \) and \( G_2 \), where \( G_1 \in \mathcal{G}_{n,p_1} \), and \( G_2 \in \mathcal{G}_{n,p_2} \). Setting \( p_1 = \frac{1}{n}(1 + \frac{\epsilon}{2}) \), we get that \( \frac{\epsilon}{2} < np_2 \leq 12 \).

In the following, we find a minor \( H_1 \) of \( G_1 \) which will contain no edges at all, and then consider how the edges of \( G_2 \) appear in \( H_1 \). This gives us a minor \( H \) of \( G \) which will have a large pathwidth.

By Theorem 6, we have that \( G_1 \) contains a tree of size \( \theta n \). As before we may assume that the maximum degree in this tree is \( d = O(\log n) \) (this will happen w.h.p.). It is not hard to verify that this implies that for any \( k \), \( G_1 \) contains \( l = \frac{\theta n}{kd} \) disjoint connected sets \( V_1, \ldots, V_l \), each of size \( k \) (such a partition can be obtained by traversing the tree mentioned above). Now set \( k = \frac{24\epsilon}{\theta d} d = O(\log n) \), notice that \( l \) is unbounded. In the following we assume that both \( k \) and \( l \) are integers, otherwise we must use the \( \lfloor \cdot \rfloor \) notation.

Define a minor \( H_1 \) of \( G_1 \), by contracting all of the edges internal to each \( V_i \), and removing all vertices outside of \( \cup_i V_i \), and all edges not internal to the \( V_i \)'s
in other words, $H_1$ contains $l$ vertices, and no edges. Define a minor $H$ of $G$, by considering the edges of $G_2$ as they appear in $H_1$. An edge of $H$ corresponds to $k^2$ (possible) edges of $G_2$, and so will appear with probability $p_3$:

$$p_3 = p_2(1 + (1 - p_2) + \ldots + (1 - p_2))k^2 - 1 \geq p_2k^2(1 - p_2)^{k^2 - 1} \geq p_2k^2\frac{1}{e}$$

Now,

$$lp_3 \geq \frac{\theta n}{kd}p_2k^2\frac{1}{e} \geq \frac{\theta k}{2ed} = 12.$$  

According to Lemma 9, w.h.p. $\text{mPW}(H) > \frac{1}{4}l = \Omega\left(\frac{n}{\log^2 n}\right)$, and by Lemma 7, $\text{mPW}(G) \geq \text{mPW}(H) > \Omega\left(\frac{n}{\log^2 n}\right)$. Lastly, w.h.p. $d(G) = O(\log n)$, and then by Theorem 2 we have that w.h.p. $\text{mBDD}(\varphi_G) > 2^{\Omega\left(\frac{n}{\log c n}\right)}$, to conclude. $\square$

### 4.3 Lower Bound of Case 3: $n^{1-\epsilon} < pn < n - n^\alpha$

Notice that the lower bound presented in the previous Section 4.2 is not super polynomial if $p$ is taken to be very large (namely for values of $p$ greater than $1/\log^6 n$). In the following section, we study large values of $p$ and obtain super polynomial lower bounds. To show a lower bound in these cases, we will work directly with Theorem 1 and not with the pathwidth of the graph. To get a lower bound using this theorem we need to first estimate the number of independent sets in a random graph of $G_{n,p}$.

For the reminder of this section, we will assume (a) For every constant $\epsilon > 0$, $pn > n^{1-\epsilon}$, and (b) For every constant $\alpha < 1$, $pn < n - n^\alpha$.

**Independent Sets in $G_{n,p}$**. Recall that Theorem 1 shows a connection between certain combinatorial properties of $G$ and the QOBDD size of $\varphi_G$. In particular, a necessary condition for a large $\text{mBDD}(\varphi_G)$ is the existence of many (super polynomial) number of independent sets in $G$. We start by showing this condition holds w.h.p. on random graphs in $G_{n,p}$, and then use it for proving the lower bound of case 3.

Denote $q = 1 - p$. We will consider the number of independent sets of size $k = k_c$ in $G_{n,p}$, where $k = c\log n / \log 1/q$, and therefore $q^k = n^{-c}$. Since $q > n^{\alpha-1}$ for every constant $\alpha < 1$, we get that $k$ is unbounded, and we can therefore assume $k$ is a natural number. We take $c$ to be a small constant. Since $pn > n^{1-\epsilon}$ for every constant $\epsilon > 0$, we have $k = O(n^\epsilon \log n)$ for every constant $\epsilon > 0$. Let $\gamma > 0$ be an arbitrarily small constant, in the following we will use the fact that $k \leq n^\gamma$.

Denote the expected number of independent sets of size $k_c$ by $E = E_c$. Clearly, $E = \binom{n}{k}q^{\binom{k}{2}}$. It is not hard to verify that $E = n^{\Omega(k)}$ given $c$ is small enough. Furthermore, it can be seen (using standard techniques) that the variance $V$ of the number of independent sets of size $k$ is at most $\frac{1}{4}E^2$. Thus, by Chebyshev’s inequality,
Corollary 3. For small enough $c$, the number of independent sets of size $k$ in $G \in \mathcal{G}_{n,p}$ is $n^{\Omega(k)}$ with probability greater than $\frac{1}{2}$.

The constant $\frac{1}{2}$ bound on the probability obtained in Corollary 3 will not suffice for our purpose, and we will therefore amplify the probability of this result. Roughly speaking, this is done by applying Corollary 3 on a large class of almost disjoint subsets of vertices in $G$ (namely subsets that share at most a single vertex) where each subset is of polynomial size. If one of these subsets has many independent sets, so does $G$. Due to space limitations, full proof is omitted.

Lemma 10. For small enough $c$, the number of independent sets of size $k$ in $G \in \mathcal{G}_{n,p}$ is $n^{\Omega(k)}$ with probability greater than $1 - 2^{-n^{3/5}}$.

QOBDD Size Lower Bound. We will now use Theorem 1 to prove the lower bound of case 3 on the QOBDD size of $G \in \mathcal{G}_{n,p}$. It is not hard to verify that it suffices to prove

Lemma 11. Let $G \in \mathcal{G}_{n,p}$. Let $k = k_c$ be as defined in Section 4.3. For small enough $c$, w.h.p. $\text{mBDD}(\varphi_G) = n^{\Omega(k)}$.

Proof. By Theorem 1 it is enough to show that w.h.p., for every set $U \subseteq [1, n]$, $|U| = \sqrt{n}$,

\[|\{\Gamma_G(I) \cap ([1, n] \setminus U) \mid I \in \text{ID}(G|_U)\}| \geq n^{\Omega(k)}.\]

Since this will show, that for every ordering of the vertices of $G$, the size of the $\sqrt{n} + 1$ row in $\varphi_G$’s QOBDD is at least $n^{\Omega(k)}$. We will therefore show that for every such $U$ this happens with probability greater than $1 - \frac{1}{n} \left(\frac{n}{\sqrt{n}}\right)^{-1}$, and so using the union bound, we get that it is true for all $U$ w.h.p.

Let $U_1$ and $U_2$ be two independent sets of size $k$ in $G|_U$. For $i = 1, 2$, let $\Gamma_i = \Gamma_G(U_i) \cap ([1, n] \setminus U)$. The probability that a specific vertex is in $\Gamma_1$ but not $\Gamma_2$ is greater than $pq^k$, and therefore the probability that there is no such vertex in $[1, n] \setminus U$, i.e., $\Gamma_1 = \Gamma_2$, is at most,

\[(1 - pq^k)n - \sqrt{n} < (1 - \frac{D}{n^c})^\frac{1}{2} < e^{-\frac{1}{2} n^{\alpha_c}} < e^{-\frac{1}{2} n^{1-\gamma-c}} < e^{-n^{3/4}},\]

where $\gamma > 0$ is an arbitrarily small constant. Since the number of independent sets $U_i$ in $U$ is at most $|U|^k < e^{k \log n} < e^{n^\gamma \log n}$, then the probability that all the sets $\Gamma_G(U_i) \cap ([1, n] \setminus U)$ differ is at least

\[1 - e^{2n^{\gamma} \log n} e^{-n^{3/4}} > 1 - e^{-n^{2/3}}\]

For a specific $U$, by Lemma 10, with probability at least $1 - 2^{-n^{3/4}}$, the number of independent sets of size $k$ in $U$, is $\sqrt{n}^{\Omega(k)} = n^{\Omega(k)}$. To conclude,

\[1 - (e^{-n^{2/3}} + 2^{-n^{3/4}}) > 1 - e^{-\sqrt{n} \log n} > 1 - \frac{1}{n} \left(\frac{n}{\sqrt{n}}\right)^{-1} \]

\[\square\]
4.4 Upper Bounds of Cases 2, 3, and 4

We now prove the upper bound of case 4. The upper bounds of cases 2 and 3 are proven similarly (their proof involves setting the parameter $k$ in the proof below to $4 \log n / \log 1/q$).

**Theorem 8.** Let $G \in G_{n,p}$, where $pn > n - n^\alpha$ for some constant $0 < \alpha < 1$. Then, w.h.p. $m_{BDD}(\varphi_G) = n^{O(1)}$.

**Proof.** The expectation of the number of independent sets of size $k = \lceil \frac{3}{1-\alpha} \rceil + 1$ is at most,

$$\left( \begin{array}{c} n \\ k \end{array} \right) (1 - p)^{\binom{k}{2}} = \left( \begin{array}{c} n \\ k \end{array} \right) \left( \frac{n^\alpha - 1}{2} \right)^{\binom{k}{2}} \leq n^k n^{(\alpha - 1) \frac{k(k - 1)}{2}} = n^k \frac{1}{2} (2 + (\alpha - 1)(k - 1)).$$

Since $(\alpha - 1)(k - 1) = (\alpha - 1) \lceil \frac{3}{1-\alpha} \rceil \leq -3$, the expectation is at most $n^{-\frac{1}{2}k} = o(1)$, and so by Markov’s inequality w.h.p. $\maxID(G) \leq k$. By Proposition 1 and Corollary 1, $m_{BDD}(\varphi_G) \leq n \cdot n^k = n^{O(1)}$. $\square$

**Acknowledgments.** We would like to thank Uriel Feige and Alon Rosen for initiating our interest in the problem at hand, and for helpful discussions.

**References**


[BW00] B. Bollig and I. Wegener, “Asymptotically optimal bounds for OBDDs and the solution of some basic OBDD problems”, In *International Colloquium on Automata, Languages and Programming ICALP ’00*.


[GZ01] J.F. Groote and H. Zantema, “Resolution and Binary decision diagrams cannot simulate each other polynomially”, *Ershov Memorial Conference ’01*.


Efficient Hybrid Reachability Analysis for Asynchronous Concurrent Systems*

Enric Pastor and Marco A. Peña

Department of Computer Architecture
Technical University of Catalonia
08860 Castelldefels (Barcelona), Spain
{enric,marcoa}@ac.upc.es

Abstract. Symbolic reachability analysis based on Binary Decision Diagrams (BDDs) is a technique that allows the implementation of efficient state space exploration algorithms. However, in practice it is well known that the BDD blowup problem limits the size of the systems that can be analyzed. Conversely, simulation is a low-cost state generation technique, although its effectiveness is limited due to its inherent sequentiality. We present a hybrid methodology that combines simulation and symbolic traversal in order to improve the state space exploration of large systems. The methodology concentrates on asynchronous concurrent systems, whose peculiarities are not fully exploited by other existing techniques for hybrid verification. Our approach exploits the information obtained from simulations to improve the knowledge of the state space, effectively guiding symbolic traversal. We demonstrate the applicability of this methodology in the verification of complex control-dominated asynchronous circuits.

1 Introduction

State space computation is the main bottleneck for most formal verification techniques. As an example, for invariant verification all reachable states of the system are calculated and the desired invariants are checked to hold in all of them. If the system fails to satisfy the invariants, it is necessary to identify a counter-example that reproduces the sequence of actions that the system performs before failing. The computational complexity of invariant verification is revealed when systems that exhibit high degrees of concurrency with irregular state spaces are analyzed (the well-known state explosion problem). In those cases, even the utilization of BDD-based symbolic techniques [1,2] does not allow the complete analysis of the state space.

In recent years mixed approaches combining simulation and formal verification have been introduced, coining the term hybrid verification [3]. Instead of ensuring the complete exploration of the state space, hybrid verification intends to provide efficient mechanisms to identify significantly large portions of

* Funded by the Ministry of Science and Technology of Spain TIC 2001-2476-C03-02 and DURSI of Generalitat de Catalunya 2001SGR-00226.
the space space with a reduced computational complexity. Hybrid verification has been traditionally useful when the size of the system under analysis is too large to be fully verified by conventional means. In these cases, hybrid verification provides the designer with positive feedback to improve the reliability of the system in terms of failures discovered in a first step of the verification flow. These techniques may also help in the early stages of the design, when failures are not real design errors but holes in the specifications.

This paper presents a hybrid reachability strategy tailored for asynchronous concurrent systems, i.e. to consider the interleaved execution of concurrent events. We propose a two-step mechanism based on a combination of simulation and reachability analysis (see Figure 1). In a first step, simulation provides an initial depth-first view of the states in the system. In order to guarantee a good coverage of the state space, simulation detects those states where the system chooses between alternative execution sequences (i.e. branching sequences). Then, each one of the possible alternative sequences will be further explored. The analysis introduced in this work guarantees that interleaving branches due to concurrency will not be exhaustively explored during simulation. Conversely, only one of the sequences is explored, resembling those techniques used in partial order reduction methods [4,5].

In a second step symbolic traversal is applied to improve the state coverage. The information about the ordering in which events are fired, obtained by simulation, is used to guide the way in which traversal is applied. Reachability analysis is performed for each one of the sequences generated by the simulation phase, accumulating the obtained states.

The remainder of the paper is organized as follows. Section 2 introduces existing previous research related to our methodology. Section 3 provides background on the model used for asynchronous concurrent systems, and on the peculiarities of their reachability analysis. The proposed simulation scheme is described in Section 4. The analysis of the dynamic behavior of the system and its application to guided traversal is described in Section 5. Experimental results on the application to invariant checking on control-dominated asynchronous circuits are analyzed in Section 6. Section 7 concludes the paper.

Fig. 1. Two-step scheme: simulation followed by guided-traversal.
2 Previous Work

State space exploration using guided techniques has become subject of wide interest. These techniques tackle the guidance of reachability analysis toward failure detection rather than to complete state space computation. Guided search typically uses “score-boarding” to find sequences from the initial states to failure states. Various metrics have been proposed to prioritize the state exploration based on the Hamming distance [6], tracking [6], reachability probability [7], lighthouses or guide-posts [8,6,9], and rarity search [10].

Several techniques have been introduced to guide the search toward uncovered regions of the state space. Ganai et al. [8] introduced a combination of adaptive simulation with retrograde analysis. Adaptive simulation is based on random simulation with a backtracking mechanism to avoid getting stuck during the search. Retrograde analysis involves a combination of forward analysis with pre-images from the failure states. Bloem et al. [11] use hints to guide the symbolic search and to alleviate the BDD explosion problem. Each hint indicates which portion of the transition relation should be used at each step to avoid a BDD blowup. Ganai et al. [8] and Yang et al. [6] suggest the manual insertion of guide-posts. User defined guide-posts are variables inserted in the system, which if activated during the traversal indicate that we are in the right way to find a failure. In [9] an automatic guide-post insertion mechanism is proposed. Kuehlmann et al. [7] suggest using the state reachability probability as a guide for state prioritizing. Again, Ganai et al. [10] propose a rarity-based guide that tracks latch toggle activity to improve state coverage.

Some authors suggest the combination of symbolic reachability analysis with BDD-subsetting. In [12] when the BDD representing the state space grows beyond a certain limit, a subset is taken such that the BDD size is reduced but a large fraction of the state space is kept. [3] attempts to improve the subsetting mechanism by differentiating control and data-path, and keeping subsets that preserve all possible control behaviors.

The work presented in this paper resembles some of the strategies using by partial order reduction techniques [4,5]. However, some key aspects differentiate our approach from these techniques. First of all, the goal of the approach is to generate the largest possible portion of the state space. This goal is radically opposite to partial order reduction. Second, the state successors to be explored are selected taking into account exclusively the causality relations between events in the system. No assumption is made on the type of temporal property being verified. Additionally, the reduced state space is never rebuild, only finite sequences of states are generated.
3 Background

3.1 Transition Systems

A TS is a formalism oriented to modeling asynchronous concurrent systems that emphasizes the execution of abstract events rather than how they are encoded. Events may have different semantics depending on the level of detail of the model (signal changes, protocol operations, etc). The concurrent execution of events is described by means of *interleaving*, *i.e.* weaving the execution into sequences.

Formally, a *transition system* (TS) [13] is composed of a non-empty set of states $\mathcal{S}$, a non-empty alphabet of events $\Sigma$, a transition relation $T \subseteq \mathcal{S} \times \Sigma \times \mathcal{S}$, and a set of initial states $\mathcal{S}_{in}$. Transitions are denoted by $s \xrightarrow{e} s'$. The *firing region* of an event $e$ is defined as $Fr : \Sigma \rightarrow 2^\mathcal{S}$ such that $Fr(e) = \{ s \in \mathcal{S} | \exists s \xrightarrow{e} s' \in T \}$. Thus, event $e$ is *firable* at state $s$ if $\exists s \xrightarrow{e} s' \in T$, *i.e.* $s \in Fr(e)$. The set of events *firable* at state $s$ is denoted by $\mathcal{E}(s)$. A *run* of a TS is a *firing sequence* $\sigma = s_1 \xrightarrow{e_1} s_2 \xrightarrow{e_2} \cdots$, such that $s_1 \in \mathcal{S}_{in}$ and $\forall i \geq 1 : s_i \xrightarrow{e_i} s_{i+1} \in T$. Given the significance of individual events, the transition relation (TR) of a TS can be naturally partitioned into a disjoint set of relations, one for each event $e \in \Sigma$; $T_e = \{ s \xrightarrow{e} s' \in T | \exists s, s' \in \mathcal{S} \}$.

Figure 2 shows a TS that will be used as a running example. The system contains 22 states and a set of events $\Sigma = \{ a, b, c, d, e, f, g \}$. State $s_1$ is its initial state. Note the existence of multiple interleaving sequences due to concurrency, e.g. $a \xrightarrow{} b \xrightarrow{} c$ and $a \xrightarrow{} c \xrightarrow{} b$.

3.2 Reachability Analysis

The set of states that is reachable in any number of steps from a set of states $C$ ($\text{Reach}(T, C)$) is defined as the least fix-point of the following recurrence:
where $Img(T, S_i)$ is the one-step image computation applying the TR $T$ on a set of states $S_i$. When, $C$ equals $S_{in}$ for a given TS, this algorithm generates the state space of a system in a Breath First Search (BFS) style. The number of iterations performed by such traversal is determined by the maximum number of steps from the initial state to the first occurrence of each of the reachable states (called the sequential depth of the TS). In the example of Figure 2, the application of BFS from the initial state gives state $s_2$ in a first step, states $s_3, s_5, s_{13}$ in a second step, states $s_4, s_8, s_6, s_{10}, s_{14}, s_{17}, s_{15}$ in a third, etc.

The classical BFS algorithm can be improved based on two key observations. First, at each iteration of the BFS traversal, most transitions described in the monolithic TR are not applied (e.g. at the second BFS step, only events $b, c, g$ are significant). And second, the TR of a TS can be naturally partitioned into disjunctive TRs, one for each event, that can be applied individually.

These observations have suggested alternative traversal algorithms, named chaining [14,15]. Chaining applies the individual TRs of events in a predetermined order such that the number of new states generated at each step is maximized. After the application of the transition relation of an event, the newly generated states are immediately used as domain for the next event in order, hence coining the term chaining.

Figure 3 shows the general concept for two TRs $A$ and $B$. If $A$ and $B$ are applied to the same set FROM in a BFS style, a certain number of states is reached (see Figure 3(a) and (b)). However, chaining would apply $A$ to FROM and generate a new set of states ($FROM + TO(A)$ in Figure 3(c)), and afterward
apply \( B \) to this set (in Figure 3(d)). The number of reached states increases with almost the same computational effort.

In practice, chaining can significantly reduce the number of iterations of the BFS algorithm [15,16]. The method is specially effective if the appropriate firing order of the events is selected. Chaining outperforms BFS techniques in the verification of asynchronous concurrent systems because states are computed at a much faster ratio and with less effort, thus reducing the number TR applications. Moreover, partitioning the TR provides important memory savings and CPU speed-ups when implemented over BDD structures.

4 Simulating Transition Systems

This section presents a simulation approach for asynchronous concurrent systems that automatically provides a good state space coverage. At each explored state the causality between firable events is analyzed to identify firing conflicts between them. Conflict detection allows to identify execution sequences that exclude each other in a variety of ways, including mutual exclusion for example. This simulation scheme resembles those state exploration techniques used by partial order reduction [4,5].

Simulation chooses a particular firing order among all possible interleaved executions of concurrent events. Thus, simulation alone has limited coverage effectiveness for concurrent systems. We will show in Section 5 that the interleaving of events due to concurrency can be explored more efficiently by symbolic traversal once the information from a particular simulation sequence is available.

4.1 Conflict Detection to Improve Coverage

Conflict detection is the key mechanism that allows to distinguish between sequences of events representing alternative behaviors of a system or interleaving sequences of concurrent events. The first type of sequences are relevant and must be explored in order to guarantee a good coverage of all possible behaviors of the system. Exploring interleaved sequences must be avoided and postponed to the symbolic traversal phase.

An event \( e_1 \) disables another event \( e_2 \) if a pair of states exists \( s_1, s_2 \) such that \( s_1 \xrightarrow{e_1} s_2 \in T \) and \( e_2 \) is firable in \( s_1 \) (\( e_2 \in \mathcal{E}(s_1) \)) but \( e_2 \) is not firable in \( s_2 \) (\( e_2 \not\in \mathcal{E}(s_2) \)). Two events \( e_1, e_2 \) are in conflict if \( e_1 \) disables \( e_2 \) or \( e_2 \) disables \( e_1 \). A conflict is called symmetric if \( e_1 \) disables \( e_2 \) and \( e_2 \) disables \( e_1 \). The conflict is called asymmetric if \( e_1 \) disables \( e_2 \) but \( e_2 \) does not disable \( e_1 \), or vice versa.

Figure 4 depicts a portion of the state space of a concurrent system. The figure illustrates the conflict situations previously described. From the initial state (a) shows three events \( e_1, e_2, e_3 \) that are mutually concurrent; (b) shows a symmetric conflict between \( e_1 \) and \( e_2 \); and (c) shows an asymmetric conflict in which \( e_2 \) disables \( e_1 \) but not the contrary (event \( e_3 \) remains concurrent to \( e_1 \) and \( e_2 \)).
A state in which two or more events are in conflict is called a branching state in which alternative execution sequences exist (see Figure 1). Each separate sequence can be followed, resulting into different behaviors of the system.

Symmetric conflicts are associated to states in which the system takes a decision. The behavior in each branch may involve completely different sets of events and thus produce distinct/disjoint sets of states. A different simulation sequence is generated for each branch in order to achieve a better coverage of the state space. On the contrary, asymmetric conflicts can be associated to disablings, in which the firing of one event (the disabler) prevents the firing of a second event (the disabled). In this type of conflict two different firing sequences exists. In one of the sequences, both events can fire concurrently and no disabling occurs. In the other sequence, the disabler event fires thus disabling the second event and, in consequence, disabling also some part of the system behavior. As an example, disablings can be associated to races in digital circuits (e.g. producing either glitches or dead-locks at the output of some gates).

4.2 Simulation Algorithm

This section presents an improved simulation mechanism based on the analysis of the conflicts found along the simulation sequences. Every time a pair of conflicting events is identified, a new sequence is generated and stored in a list of pending sequences. The sequence duplication scheme is detailed in Figure 5. In the example there is a firing sequence ($\sigma_1$) in which three events $e_1, e_2, e_3$ are firable in state $s_1$. Let us assume that events $e_1$ and $e_2$ are in symmetric conflict. A copy of the branch state $s_1$ is generated ($s'_1$), together with a copy ($\sigma'_1$) of the sequence (up to state $s_1$) being explored. The exploration continues by removing the disabled event ($e_2$) from the list of firable events at $s_1$. Then, event $e_1$ is fired in the active sequence $\sigma_1$ generating a state where only $e_3$ remains firable. On the other hand, $\sigma'_1$ is stored for later exploration. The disabled event ($e_1$) is removed from the list of firable events at state $s'_1$. Note that the order in which events have been selected it is not necessarily the order in which our algorithm may proceed. In that case, concurrent events like $e_3$ may be given priority.

Simulation sequences are processed following a configurable priority scheme. States can be analyzed following a DFS or BFS style, or a mixture of both.

Fig. 4. Concurrent and conflict situations.
Fig. 5. Branching sequences due to conflicts.

However, other parameters can be taken into account, e.g. the number of choices already taken. The firing order of the events can be also decided according to some priority scheme. In our simulator, we keep track of the number of times that each event is fired. To avoid locking the state exploration in some local region of the state space, we give additional priority to those events which have been fired less often.

The algorithm in Figure 6 describes the suggested simulation scheme. The simulation engine stores the set of sequences, together with the events that are ready to fire, in the active list. Sequences are stored as linked lists of BDD cubes, each cube representing a state of the system. Terminated sequences due to state repetition, deadlocks or simulation limits are stored in seq. All states visited along the simulation are stored in visit. Each sequence analyzed along the simulation is stored as a tuple $\tau = (s, \sigma, E, D, B)$ that consists of: $s$ the last state in the sequence; $\sigma$ the firing sequence required to reach $s$ from $S_in$; a set $E \subset \Sigma$ that indicates the events that remain firable at $s$; and two integers to indicate the firing depth $D$ with respect to $S_in$, and the number of taken choices $B$ required to reach that depth.

Without loss of generality we will assume that the simulation starts from a single initial state $s_in$. A tuple is created for this state by using the empty sequence $\{s_in\}$ and all possible firable events (retrieved by function firable($s_in$)). This initial sequence is placed into the list of active sequences pending of being processed.

The simulator takes one sequence from the list of pending sequences. The last state of the sequence is checked to determine if the simulation should proceed from it. If the state has been already visited, or simply the depth/branch limit have been surpassed, the sequence is stored in seq. If the last state can be processed, a firable event $e \in \tau.E$ is selected. Events can be selected giving priority to either: events that are not in conflict, events that are in symmetrical conflict and events that are in asymmetric conflict. If event $e$ is in conflict**, then sequence $\tau$ is duplicated into an exact copy $\tau'$. Event $e$ is marked as non-firable in $\tau'$ to avoid exploring the same sequence multiple times (see Figure 5). Finally, $\tau'$ is inserted back into the list of active sequences for a later exploration of alternative branches.
1. visit := seq := active := ∅;
2. \( \tau := \text{alloc\_sequence} (s_i, \{s_i\}, E(s_i), 0, 0) \);
3. active := active \cup \tau
4. while (active \neq \emptyset) do
5. \( \tau := \text{get\_priorized\_sequence} (\text{active}) \)
6. visit := visit \cup \tau.s
7. if (termination\_condition (\tau)) then
8. seq := seq \cup \tau.\sigma
9. free\_sequence (\tau)
10. continue;
11. e := \text{select\_firable\_event}(\tau.E);
12. if (event\_disables (\tau, e)) then
13. \( \tau' := \text{duplicate\_sequence} (\tau) \)
14. \( \tau'.E := \tau'.E \setminus e \)
15. \( \tau'.B = \tau'.B + 1 \)
16. active := active \cup \tau'
17. \( \tau.s := \text{Img}(T_e, \tau.s) \)
18. \( \tau.D := \tau.D + 1 \)
19. \( \tau.E := \text{firable}(\tau.s) \)
20. \( \tau.\sigma := \tau.\sigma \stackrel{e}{\rightarrow} s \)
21. active := active \cup \tau

**Fig. 6.** Pseudocode of the simulation algorithm.

The selected event \( e \) is fired from state \( \tau.s \), generating its successor \( \text{Img}(T_e, \tau.s) \) that is updated in the sequence. The remaining information is also updated, including the extension of the firing sequence by \( \sigma \stackrel{e}{\rightarrow} s \). Finally, \( \tau \) is updated and placed back into the list of active sequences.

Given the example of Figure 2, the simulator generates two firing sequences (assuming an alphabetical firing order of the events) shown in Figures 7(a) and 8(a), respectively. Two sequences are generated because a conflict is detected at state \( s_2 \) between events \( b \) and \( g \), when \( b \) is selected to fire. A third sequence is also generated, although not shown due to lack of space, because events \( c \) and \( g \) are also in conflict at \( s_2 \). Note that sequences are annotated with the events firable at each state.

This simulation scheme allows a fast in-depth analysis of the system, providing a good state coverage since all conflict branches are identified. A number of heuristic termination conditions are included to avoid repeating equivalent execution sequences: stop exploring a sequence whenever an already visited state is reached, and bound the depth of the sequences to a factor of the total number of events in the TS.
5 Guided Traversal

This section shows how the sequences generated by the simulation phase, together with the information about which events are firable at each state, allows analyzing the causality relations between events. Such causality is later exploited to improve the symbolic traversal by using chaining. An efficient traversal algorithm is applied for selected sequences, thus improving the state coverage. Following the ideas in [16], the TR of the system is partitioned, and the application of each part is scheduled by analyzing the causality relations between the events in the sequence, thus maximizing the state generation ratio.

5.1 Extracting Causality Relations

Causal event structures (CES) describe all possible sequential and concurrent executions of a set of events. A CES [17] is a tuple \( \langle \Sigma, \prec \rangle \) where \( \Sigma = \{e_1, \ldots, e_n\} \) is a finite set of events and \( \prec \subseteq \Sigma \times \Sigma \) is a strict partial order (irreflexive and transitive) over \( \Sigma \) called the causality relation.

Given a CES the following relations can be defined where \( \text{co} \) is called the concurrency relation:

\[
\text{id} \defeq \{(e, e) \mid e \in \Sigma\}
\]

\[
\succ \defeq \{(e_1, e_2) \mid (e_2, e_1) \in \prec\}
\]

\[
\text{co} \defeq \Sigma \times \Sigma - \{\prec \cup \succ \cup \text{id}\}
\]

Provided a firing sequence of events, we can recover a partial order showing the causality relationships among those events, i.e. a CES. A partial order \( \prec \) over a set of events \( \Sigma \) and the associated relations \( \succ \), \( \text{id} \) and \( \text{co} \) completely partition \( \Sigma \times \Sigma \). We can use this fact to derive a CES from a sequence \( \sigma \), such that: \( \text{id} \) is obviously defined; \( e_1 \text{ co } e_2 \) if \( e_1 \) and \( e_2 \) are firable simultaneously in some state visited by \( \sigma \); and \( e_1 \prec e_2 \) if \( e_1 \) precedes \( e_2 \) and are not firable simultaneously in \( \sigma \) (similarly for \( \succ \)).

Figures 7(b) and 8(b) show the CESs derived from the sequences in Figures 7(a) and 8(a), respectively. Arcs denote causality relations between events. For sake of clarity, we also indicate with dotted arcs the conflict relations, although they are not part of the actual CES. Thus, if event \( b \) disables \( g \), a dotted arc from \( b \) to \( g \) is drawn. In Fig. 7, note that event \( g \) appears two times. The first time is disabled by \( b \), while the second one is fired after \( g \).

5.2 Reachability Analysis

A topological order of the events of a CES is a sequence \( e_1 \cdots e_n \in \Sigma^* \ (n = |\Sigma|) \), such that all \( e_i \) are distinct and \( \forall 1 \leq i, j \leq n : e_i < e_j \Rightarrow i < j \). Firing the events following such topological order often guarantees that when an event is fired all its causal predecessors have been already fired. Given an event \( e_i \) ready to fire, if all the events concurrent to \( e_i \) are fired before \( e_i \), most states in \( \text{Fr}(e_i) \) will
be already reached. A traversal algorithm in which events are fired following the topological order guarantees a good effectiveness. Unfortunately, the causality relations of a complex system cannot be described with a single CES. A pair of events may be causally ordered in some part of the state space, whereas they may be concurrent in another. On the contrary, causality relations derived from a single firing sequence provides a quite precise approximation of the behavior at localized areas of the state space. Hence, traversal can successfully exploit that information in those cases.

Given a set of sequences generated from the described simulation process, we propose the following three-step traversal strategy (see Figure 9): Generate the CES for each firing sequence (2). Find a topological order of the events in the
Fig. 9. Guided traversal algorithm.

CES(3). Execute a symbolic traversal algorithm from the initial state by applying
the TR of each event in sequence (4–5). Events will be applied following the
topological order extracted from the CES. The states generated after the image
computation of one event will be immediately applied as domain for the image
computation of the successor event in order, thus chaining the effect.

Note that, in practice, not all firing sequences need to be considered for
traversal. Some sequences will be almost equivalent to other, with only a small
suffix of the simulation being different. In those cases, causality analysis and
traversal should be only applied to the suffix. Otherwise large amounts of states
will be repeatedly generated from different sequences.

Figures 7(b) and 8(b) show the CES annotated with an index that indicates
the position in the topological order selected for traversal. Observe that the
disabled events are not annotated since they do not actually belong to the CES.
Figures 7(c) and 8(c) show the portions of the state space in the original TS
(see Figure 2) generated by the guided traversal for each firing sequence. After
both traversals only state $s_{10}$ remains unreached. Note that in Figure 8, event
d appears two times along the sequence. In that case, duplicated events are
renamed to satisfy the topological order requirements.

5.3 Methodology Implementation

Figure 10 sketches an implementation of the proposed strategy for hybrid explo-
reration of the state space. The process is divided in two parts: a simulation phase
followed by a traversal phase.

Simulation phase: From the initial state, the simulation engine generates
multiple branching sequences. The number of sequences will be determined by
the set of conflicts found during the state exploration, or limited by a user-defined
limiting parameter. Causality is extracted and attached to each sequence to be
used later in the traversal phase.

Traversal phase: Sequences are iteratively taken to apply symbolic traversal
on them. Heuristically, we choose the sequence that contains more states not
covered by previous sequences. Note that if all states in a sequence are already
contained in the set of reached states so far, the sequence will be discarded for
traversal. The events in the selected sequence are fired following a topological
order. The order is either extracted from the causality information attached to
the sequence, or directly taken from the order in which events are fired along
the sequence (also a valid topological order). Events are iterated once if applied from the causality information, or until a fix-point is reached if applied from the simulation order.

6 Experimental Results

In the following tables several asynchronous concurrent systems are analyzed using the hybrid reachability scheme described in this paper. A brief description of each systems follows:

PCC  Pausible clock controller for heterogeneous systems in [18].

GALS  Globally-Asynchronous Locally-Synchronous design in [19].

RGD-arbiter asP*, RGD arbiter in [20] described at transistor level.

IPCMOS  A pulse-based controller for asynchronous pipelines in [21].

STARI  A self-timed pipeline in [22].

All these results are from executions on a 2Ghz Pentium IV Linux computer with 512Mb of memory. Note that the behavior of all these systems is delay-dependent. In our experiments we only concentrate on the untimed state space.

Table 1 compares the results of full reachability analysis when using different traversal strategies on the selected benchmarks. Suffix C is used for circuits and A for abstractions. The number in parenthesis indicates the number of stages in case of pipelines. We provide results for BFS traversal (BFS), chained traversal using a greedy ordering strategy (C Greedy) [14], and a token-traverse chained strategy (C Token) [16]. Our goal when presenting these experiments is to demonstrate the significant impact that the chaining methodology has on the efficiency of traversal. Moreover, we will use these results as a reference to evaluate the proposed hybrid methodology.
Table 1. Experimental results: various forms of traversal.

<table>
<thead>
<tr>
<th>Name</th>
<th>States</th>
<th>Iter</th>
<th>BDD</th>
<th>CPU</th>
<th>Iter</th>
<th>BDD</th>
<th>CPU</th>
<th>Iter</th>
<th>BDD</th>
<th>CPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>GALS-C</td>
<td>1.232e+3</td>
<td>68</td>
<td>10498</td>
<td>42.7</td>
<td>17</td>
<td>10914</td>
<td>6.2</td>
<td>10</td>
<td>10767</td>
<td>6.2</td>
</tr>
<tr>
<td>PCC-C</td>
<td>9.89184e-5</td>
<td>64</td>
<td>80979</td>
<td>42.4</td>
<td>15</td>
<td>18104</td>
<td>2.6</td>
<td>5</td>
<td>12573</td>
<td>2.7</td>
</tr>
<tr>
<td>RGD-arbiter-A</td>
<td>3.33813e-9</td>
<td>79</td>
<td>218088</td>
<td>695.7</td>
<td>20</td>
<td>113757</td>
<td>22.6</td>
<td>5</td>
<td>13938</td>
<td>6.1</td>
</tr>
<tr>
<td>RGD-arbiter-C</td>
<td>5.46918e+13</td>
<td>Tott</td>
<td>27</td>
<td>823820</td>
<td>1469.5</td>
<td>16</td>
<td>44238</td>
<td>26.3</td>
<td>5</td>
<td>13938</td>
</tr>
<tr>
<td>IPCMOS-C (4 c)</td>
<td>8.15635e+9</td>
<td>129</td>
<td>201380</td>
<td>96.9</td>
<td>12</td>
<td>153160</td>
<td>28.0</td>
<td>10</td>
<td>121994</td>
<td>44.1</td>
</tr>
<tr>
<td>IPCMOS-C (6 c)</td>
<td>1.78657e+14</td>
<td>Tott</td>
<td>41</td>
<td>126707</td>
<td>41.3</td>
<td>13</td>
<td>207124</td>
<td>19.1</td>
<td>10</td>
<td>121994</td>
</tr>
<tr>
<td>IPCMOS-A (4 c)</td>
<td>1.16785e+7</td>
<td>237</td>
<td>209198</td>
<td>1055.1</td>
<td>16</td>
<td>54978</td>
<td>22.1</td>
<td>8</td>
<td>88061</td>
<td>27.3</td>
</tr>
<tr>
<td>IPCMOS-A (6 c)</td>
<td>9.15592e-9</td>
<td>237</td>
<td>209198</td>
<td>1055.1</td>
<td>16</td>
<td>54978</td>
<td>22.1</td>
<td>8</td>
<td>88061</td>
<td>27.3</td>
</tr>
<tr>
<td>STARI-C (8 c)</td>
<td>1.07225e+12</td>
<td>Tott</td>
<td>56</td>
<td>170544</td>
<td>105.5</td>
<td>11</td>
<td>219575</td>
<td>73.0</td>
<td>8</td>
<td>88061</td>
</tr>
</tbody>
</table>

Table 2. Experimental results: simulation followed by guided-traversal.

<table>
<thead>
<tr>
<th>Name</th>
<th>Seq</th>
<th>BDD</th>
<th>States</th>
<th>CPU</th>
<th>Seq</th>
<th>BDD</th>
<th>States</th>
<th>CPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>GALS-C</td>
<td>27</td>
<td>13485</td>
<td>381</td>
<td>0.5</td>
<td>1</td>
<td>16208</td>
<td>1.232e+3</td>
<td>0.8</td>
</tr>
<tr>
<td>PCC-C</td>
<td>1</td>
<td>9120</td>
<td>306</td>
<td>0.5</td>
<td>1</td>
<td>21185</td>
<td>9.89184e+5</td>
<td>3.7</td>
</tr>
<tr>
<td>RGD-arbiter-A</td>
<td>17</td>
<td>10493</td>
<td>142</td>
<td>0.5</td>
<td>1</td>
<td>33355</td>
<td>1.05433e+9</td>
<td>2.7</td>
</tr>
<tr>
<td>RGD-arbiter-C</td>
<td>30</td>
<td>17480</td>
<td>221</td>
<td>1.2</td>
<td>1</td>
<td>148711</td>
<td>9.18829e+12</td>
<td>17.4</td>
</tr>
<tr>
<td>IPCMOS-C (4 c)</td>
<td>1</td>
<td>8088</td>
<td>179</td>
<td>0.3</td>
<td>1</td>
<td>99799</td>
<td>8.05928e+9</td>
<td>21.6</td>
</tr>
<tr>
<td>IPCMOS-C (6 c)</td>
<td>1</td>
<td>15191</td>
<td>263</td>
<td>0.6</td>
<td>1</td>
<td>278575</td>
<td>1.75992e+14</td>
<td>14.9</td>
</tr>
<tr>
<td>IPCMOS-A (4 c)</td>
<td>1</td>
<td>13727</td>
<td>133</td>
<td>0.3</td>
<td>1</td>
<td>151493</td>
<td>1.16785e+7</td>
<td>25.6</td>
</tr>
<tr>
<td>IPCMOS-A (6 c)</td>
<td>1</td>
<td>28481</td>
<td>241</td>
<td>0.9</td>
<td>1</td>
<td>179577</td>
<td>9.15592e+9</td>
<td>32.9</td>
</tr>
<tr>
<td>STARI-C (8 c)</td>
<td>8</td>
<td>141299</td>
<td>5646</td>
<td>16.9</td>
<td>2</td>
<td>283725</td>
<td>9.73548e+11</td>
<td>126.9</td>
</tr>
</tbody>
</table>

The first column in Table 1 shows the total number of states (States). The second set of columns shows the number of iterations (Iter) of BFS traversal, the peak BDD size (BDD) and the computation time (CPU) (in seconds). The third set of columns shows the same parameters but for the chained traversal with greedy ordering. The last set of columns shows the same parameters but for the token traverse chained strategy. Note that in both modified traversals Iter refers to the iterations of the algorithm, not to the sequential depth of the experiment.

Table 2 shows the results of the hybrid traversal strategy. The first set of columns provides data to evaluate the simulation phase. Column Seq indicates the total number of sequences that have been explored; column BDD shows the peak BDD size; column States shows the total number of states visited along the simulation; finally CPU indicates the computation time in seconds. It is important to note that the ratio of visited states versus CPU time is small compared to standard simulations. The reason is that, at each state, conflict relations should be analyzed penalizing the simulation efficiency. However, this initial effort should pay off later in the traversal phase.

The second set of columns provides data to evaluate the traversal phase. Column Seq indicates the subset of sequences that have been traversed; column BDD shows the BDD peak size during traversal, Column States shows the states reached after guided-traversal; and column CPU indicates the computation time in seconds for this phase. Note that some BFS results not shown are due to a CPU time-out set to 1 hour.

The initial set of experiments, BFS traversal versus chained traversal highlights the importance of a good chained strategy. The impact in both number of
iterations and peak BDD size allows reducing the computation times for traversal.

These preliminary experiments show that significant portions of the state space can be reached by our hybrid approach in reduced CPU times. BDD sizes remain reasonable for all examples, as expected, due to the spatial locality obtained by the guided-traversal step. In addition, the portion of the state space generated by the approach is a good starting point to execute symbolic traversal until the full state space is reached.

In the future we intend to improve the strategies used to select which events must be fired first during simulation. These strategies should influence to a great extent the coverage of the state space achieved during simulation, and later on during guided-traversal. We also want to explore in more detail how the firing order for events influences in the BDD sizes and the state coverage.

7 Conclusions

We believe that the incremental analysis of the state space of a system by techniques that exploit state locality is the key for the success of traversal algorithms. Instead, most existing approaches try to exploit the locality available in the transition relations to minimize them, rather than considering the impact in the representation of the state space. Following this line of reasoning, we have proposed a two-step hybrid reachability analysis strategy that combines fast simulation and guided-traversal. Simulation provides information to identify subsets of the state space in which the causality between events can be properly identified. This information can be exploited in a second phase. Causality provides enough information to efficiently generate large portions of the state space. Additionally, information about good chaining order is also extracted, which is used to guide the later traversal. The combination of both strategies should allow the reduction of BDD sizes as well as the execution times.

References

Finite Horizon Analysis of Markov Chains with the Murϕ Verifier*

Giuseppe Della Penna¹, Benedetto Intrigila¹, Igor Melatti¹, Enrico Tronci², and Marisa Venturini Zilli²

¹ Dip. di Informatica, Università di L’Aquila, Coppito 67100, L’Aquila, Italy
{dellapenna,intrigila,melatti}@di.univaq.it
² Dip. di Informatica Università di Roma “La Sapienza”,
Via Salaria 113, 00198 Roma, Italy
{tronci,zilli}@dsi.uniroma1.it

Abstract. In this paper we present an explicit disk based verification algorithm for Probabilistic Systems defining discrete time/finite state Markov Chains. Given a Markov Chain and an integer k (horizon), our algorithm checks whether the probability of reaching an error state in at most k steps is below a given threshold. We present an implementation of our algorithm within a suitable extension of the Murϕ verifier. We call the resulting probabilistic model checker FHP-Murϕ (Finite Horizon Probabilistic Murϕ). We present experimental results comparing FHP-Murϕ with (a finite horizon subset of) PRISM, a state-of-the-art symbolic model checker for Markov Chains. Our experimental results show that FHP-Murϕ can handle systems that are out of reach for PRISM, namely those involving arithmetic operations on the state variables (e.g. hybrid systems).

1 Introduction

Model checking techniques [5,11,16,15,21,28] are widely used to verify correctness of digital hardware, embedded software and protocols by modeling such systems as Nondeterministic Finite State Systems (NFSSs).

However, there are many reactive systems that exhibit uncertainty in their behaviour, i.e. which are stochastic systems. Examples of such systems are: fault tolerant systems, randomized distributed protocols and communication protocols. Typically stochastic systems cannot be conveniently modeled using NFSSs. However, they can often be modeled by Markov Chains [2,12]. Roughly speaking, a Markov Chain can be seen as an automaton labelled with (outgoing) probabilities on its transitions.

For stochastic systems correctness can only be stated using a probabilistic approach, e.g. using a Probabilistic Logic (e.g. [32,8,13]). This motivates the development of Probabilistic Model Checkers [9,1,17], i.e. of model checking algorithms and tools whose goal is to automatically verify (probabilistic) properties

* This research has been partially supported by MURST projects: MEFISTO and SAHARA.
of stochastic systems (typically Markov Chains). For example, a probabilistic model checker may automatically verify a system property like “the probability that a message is not delivered after 0.1 seconds is less than 0.80”.

Many methods have been proposed for probabilistic model checking, e.g. [10, 3,8,13,14,19,24,27,32].

To the best of our knowledge, currently, the state-of-the-art probabilistic model checker is PRISM [25,1,18]. PRISM overcomes the limitations due to the use of linear algebra packages in Markov Chain analysis by using Multi Terminal Binary Decision Diagrams (MTBDDs) [6], a generalization of Ordered Binary Decision Diagrams (OBDDs) [4] allowing real numbers in the interval [0, 1] on terminal nodes. More precisely, PRISM can carry out the required Markov Chain analysis using a matrix based approach (based on linear algebra packages), a symbolic approach (based on the CUDD package [7]) as well as a hybrid approach. The user can choose the best approach for the problem at hand.

Here we are mainly interested in automatic analysis of discrete time/finite state Markov Chains modeling Discrete Time Hybrid Systems. Such Markov Chains can in principle be analyzed using PRISM. However, our experience is that, using PRISM on our systems, quite soon we run into a state explosion problem, i.e. we run out of memory because of the huge OBDDs built during the model checking process. This is due to the fact that hybrid systems dynamics typically entails many arithmetical operations on the state variables. This makes life very hard for OBDDs, thus making usage of a symbolic probabilistic model checker (e.g. like PRISM) on such systems rather problematic.

Indeed our experience shows that Explicit Model Checking can outperform Symbolic Model Checking in automatic analysis of Hybrid Control Systems [22]. This suggested us to explore the possibility of devising an explicit disk based algorithm for automatic Finite Horizon safety analysis of Markov Chains. In this paper we present our algorithm as well as experimental results showing its effectiveness. Our results can be summarized as follows.

- We present (Sections 3, 4) an explicit algorithm for automatic verification of discrete time/finite state Markov Chains. Given a Markov Chain $\mathcal{M}$, our algorithm checks whether the probability of reaching a given state $s$ within $k$ steps is less than a given bound $p$. Our algorithm is disk based, thus, because of the large size of modern hard disks, state explosion is hardly a problem for us. Computation time instead is our bottleneck. Our algorithm can trade RAM memory with computation time, i.e. the more RAM available the faster our computation. To the best of our knowledge, this is the first time that such a disk based algorithm for probabilistic model checking is proposed.
- We present (Sections 5) an implementation of our algorithm within the Mur$\varphi$ [21] verifier. We call the resulting probabilistic model checker FHP-Mur$\varphi$ (Finite Horizon Probabilistic Mur$\varphi$).
- We present (Section 6.1) experimental results comparing FHP-Mur$\varphi$ with PRISM on two suitably modified versions of the dining philosophers protocol included in the PRISM distribution. Our experimental results show that FHP-Mur$\varphi$ can handle systems that are out of reach for PRISM. However,
as long as PRISM does not hit state explosion, PRISM is faster than FHP-Murφ (as to be expected).

Note however that PRISM can handle more general models than FHP-Murφ, and can verify more general properties (namely all PCTL [13] properties) than FHP-Murφ. In fact, FHP-Murφ can only verify finite horizon safety properties for Markov Chains, a subclass (although an important one) of the verification tasks that PRISM can handle.

– We present (Section 6.2) experimental results on using FHP-Murφ for a probabilistic analysis of a “real world” hybrid system, namely the Turbogas Control System of the Co-generative power plant described in [22]. Because of the arithmetic operations involved in the definition of system dynamics, this hybrid system is out of reach for OBDDs (and thus for PRISM), whereas FHP-Murφ can complete (finite horizon) verification within reasonable time.

2 Basic Notation

Let \( S \) be a finite set. We regard functions from \( S \) to the real interval \([0, 1]\) and functions from \( S \times S \) to \([0, 1]\) as row vectors and as matrices, respectively. If \( \mathbf{x} \) is a vector and \( s \in S \) we also write \( \mathbf{x}_s \) or \((\mathbf{x})_s\) for \( \mathbf{x}(s) \). If \( \mathbf{P} \) is a matrix and \( s, t \in S \) we also write \( \mathbf{P}_{s,t} \) or \((\mathbf{P})_{s,t}\) for \( \mathbf{P}(s,t) \). On vectors and matrices we use the standard matrix operations. Namely: \( \mathbf{xP} \) is the row vector \( \mathbf{y} \) s.t. \( \mathbf{y}_s = \sum_{j \in S} \mathbf{x}_j \mathbf{P}_{j,s} \) and \( \mathbf{AB} \) is the matrix \( \mathbf{C} \) s.t. \( \mathbf{C}_{s,t} = \sum_{j \in S} \mathbf{A}_{s,j} \mathbf{B}_{j,t} \). We define \( \mathbf{A}^n \) in the usual way, i.e.: \( \mathbf{A}^0 = \mathbf{I} \), \( \mathbf{A}^{n+1} = \mathbf{A^nA} \), where \( \mathbf{I} \) (the identity matrix) is the matrix defined as follows: \( \mathbf{I}(s,j) = \begin{cases} 1 & \text{if } (s = j) \\ 0 & \text{else} \end{cases} \). We denote with \( \mathcal{B} \) the set \([0,1]\) of boolean values. As usual 0 stands for false and 1 stands for true.

We give some basic definitions on Markov Chains. For further details see, e.g. [2]. A distribution on \( S \) is a function \( \mathbf{x} : S \rightarrow [0,1] \) s.t. \( \sum_{i \in S} \mathbf{x}(i) = 1 \). Thus a distribution on \( S \) can be regarded as a \(|S|\)-dimensional row vector \( \mathbf{x} \). A distribution \( \mathbf{x} \) represents state \( j \in S \) iff \( \mathbf{x}(j) = 1 \) (thus \( \mathbf{x}(i) = 0 \) when \( i \neq j \)). If distribution \( \mathbf{x} \) represents \( s \in S \), by abuse of language we also write \( \mathbf{x} \in S \) to mean that distribution \( \mathbf{x} \) represents a state and we use \( \mathbf{x} \) in place of the element of \( S \) represented by \( \mathbf{x} \). In the following we often represent states using distributions. This allows us to use matrix notation to define our computations.

**Definition 1.** 1. A Discrete Time Markov Chain (just Markov Chain in the following) is a triple \( \mathcal{M} = (S, \mathbf{P}, q) \) where: \( S \) is a finite set (of states), \( q \in S \) and \( \mathbf{P} : S \times S \rightarrow [0,1] \) is a transition matrix, i.e. for all \( s \in S \), \( \sum_{i \in S} \mathbf{P}(s,t) = 1 \). (We included the initial state \( q \) in the Markov Chain definition since in our context this will often shorten our notation.)

2. An execution sequence (or path) in the Markov Chain \( \mathcal{M} = (S, \mathbf{P}, q) \) is a nonempty (finite or infinite) sequence \( \pi = s_0s_1s_2\ldots \) where \( s_i \) are states and \( \mathbf{P}(s_i, s_{i+1}) > 0 \), \( i = 0,1,\ldots \). If \( \pi = s_0s_1s_2\ldots \) we write \( \pi(k) \) for \( s_k \). The length of a finite path \( \pi = s_0s_1s_2\ldots s_k \) is \( k \) (number of transitions), whereas the length of an infinite path is \( \omega \). We denote with \( |\pi| \) the length of \( \pi \).
denote with \( \text{Path}(\mathcal{M}, s) \) the set of infinite paths \( \pi \) in \( \mathcal{M} \) s.t. \( \pi(0) = s \). If \( \mathcal{M} = (S, P, q) \) we write also \( \text{Path}(\mathcal{M}) \) for \( \text{Path}(\mathcal{M}, q) \).

3. For \( s \in S \) we denote with \( \sum(s) \) the smallest \( \sigma \)-algebra on \( \text{Path}(\mathcal{M}, s) \) which, for any finite path \( \rho \) starting at \( s \), contains the basic cylinders \( \{ \pi \in \text{Path}(\mathcal{M}, s) \mid \rho \text{ is a prefix of } \pi \} \). The probability measure \( Pr \) on \( \sum(s) \) is the unique measure with \( Pr(\{ \pi \in \text{Path}(\mathcal{M}, s) \mid \rho \text{ is a prefix of } \pi \}) = Pr(\rho) = \prod_{i=0}^{k-1} P(\rho(i), \rho(i+1)) = P(\rho(0), \rho(1))P(\rho(1), \rho(2)) \cdots P(\rho(k-1), \rho(k)) \), where \( k = |\rho| \).

E.g. given distribution \( x \), the distribution \( y \) obtained by one execution step of Markov Chain \( \mathcal{M} = (S, P, q) \) is computed as: \( y = xP \). In particular if \( y = xP \) and \( x(s) = 1 \) we have that \( \forall t[y(t) = (P)_{s,t}] \).

3 Finite Horizon Safety Verification of Markov Chains

Given a Markov Chain, we want to compute the probability that a path of length \( k \) starting from a given initial state \( q \) reaches a state \( s \) satisfying a given boolean formula \( \phi \) (i.e. \( \phi(s) = 1 \)). If \( \phi \) models an error condition the above computation allows us to compute the probability of reaching an error condition in at most \( k \) transitions.

Problem 1. Let \( \mathcal{M} = (S, P, q) \) be a Markov Chain, \( k \in \mathbb{N} \), and \( \phi \) be a boolean function on \( S \). We want to compute: \( P(\mathcal{M}, k, \phi) = Pr((\exists i \leq k \phi(\pi(i))) \mid \pi \in \text{Path}(\mathcal{M})) \) That is, we want to compute the probability of reaching a state satisfying \( \phi \) in at most \( k \) steps in Markov Chain \( \mathcal{M} \) (starting from \( \mathcal{M} \) initial state \( q \)).

Definition 2. Let \( \mathcal{M} = (S, P, q) \) be a Markov Chain and let \( \phi \) be a boolean function on \( S \), i.e. \( \phi : S \to B \). We define Markov Chain \( \mathcal{M}_\phi \) as follows.

\[
\mathcal{M}_\phi = (S, P_\phi, q), \text{ where for all } s, t \in S, P_\phi(s, t) = \begin{cases} P(s, t) & \text{if } \neg\phi(s) \\ 1 & \text{if } \phi(s) \land (s = t) \\ 0 & \text{if } \phi(s) \land (s \neq t) \end{cases}
\]

In other words, Markov Chain \( (S, P_\phi, q) \) is obtained from \( (S, P, q) \) by removing all outgoing edges from any state \( s \) satisfying \( \phi \) (error state) and replacing such outgoing edges with just one edge leading back to \( s \). Thus, once an error state is entered there is no way to leave it. This, in turn, means that for \( (S, P_\phi, q) \) the probability of reaching in exactly \( k \) steps a state satisfying \( \phi \) is exactly the same as the probability of reaching in at most \( k \) steps a state satisfying \( \phi \). Note that according to item 1 of Definition 1 \( (S, P_\phi, q) \) is indeed a Markov Chain.

From the above considerations follow that \( P(\mathcal{M}, k, \phi) \) can be computed from \( P_\phi \) as shown in Proposition 1. Essentially Proposition 1 is a specialization to our finite horizon case of known results on PCTL Model Checking of Markov Chains (e.g. [13,1]).

Proposition 1. Let \( \mathcal{M} = (S, P, q) \), and let \( \phi \) be a boolean function on \( S \). Then
\[
P(\mathcal{M}, k, \phi) = Pr((\exists i \leq k \phi(\pi(i))) \mid \pi \in \text{Path}(\mathcal{M})) = \sum_{s : \phi(s)} (qP_\phi^k)_s
\]
Let \( \phi \) be defined as follows: \( \phi(s) = (s = 2) \), i.e. only state 2 satisfies \( \phi \).

Then \( \mathbf{P}_\phi = \begin{bmatrix} 0.8 & 0.2 \\ 0.0 & 1.0 \end{bmatrix} \).

From Theor. 1 we have: \( P(M, 1, \phi) = 0.2 \); \( P(M, 2, \phi) = 0.36 \); \( P(M, 3, \phi) = 0.488 \).

Example 1. Consider Markov Chain \( M = (S, \mathbf{P}, \mathbf{q}) \) with \( S = \{1, 2\} \), \( \mathbf{P} = \begin{bmatrix} 0.8 & 0.2 \\ 0.7 & 0.3 \end{bmatrix} \) and \( \mathbf{q} = [1 \, 0] \) (i.e. distribution \( \mathbf{q} \) denotes state 1). The usual automata-like representation for \( M \) is given in Fig. 1.

4 Probabilistic Finite State Systems

The Markov Chain Definition in Definition 1 is appropriate to study mathematical properties of Markov Chains. However Markov Chains arising from probabilistic concurrent systems are usually defined using a suitable programming language rather than a stochastic matrix. As a matter of fact the (huge) size of the stochastic matrix of concurrent systems is one of the main obstructions to overcome in probabilistic model checking.

Thus a Markov Chain is presented to a model checker by defining (using a suitable programming language) a next state function that returns the needed information about the immediate successors of a given state. The following definition formalizes this notion.

Definition 3. A Probabilistic Finite State System (PFSS) \( S \) is a 3-tuple \( (S, q, \text{next}) \), where: \( S \) is a finite set (of states), \( q \in S \) and \( \text{next} \) is a function taking a state \( s \) as argument and returning a set \( \text{next}(s) \) of pairs \( (t, p) \) s.t. \( \sum_{(t,p) \in \text{next}(s)} p = 1 \).

To a PFSS we can associate a Markov Chain in a unique way.

Definition 4. 1. Let \( S = (S, q, \text{next}) \) be a PFSS. The Markov Chain \( S^{mc} = (S, \mathbf{P}, q) \) associated to \( S \) is defined as follows: \( \mathbf{P}(s, t) = \begin{cases} p & \text{if} \ (t, p) \in \text{next}(s) \\ 0 & \text{otherwise} \end{cases} \)

2. Given \( k \in \mathbb{N} \) and a boolean function \( \phi \) on \( S \) we write \( P(S, k, \phi) \) for \( P(S^{mc}, k, \phi) \) as defined in Problem 1. Thus Problem 1 for PFSSs becomes: given a PFSS \( S \) compute \( P(S, k, \phi) \).

Given a PFSS \( S \) we want to compute \( P(S, k, \phi) \) without generating the transition matrix for Markov Chain \( S^{mc} \). Using Proposition 1 this can be done as shown in Proposition 2.

Proposition 2. Let \( S = (S, q, \text{next}) \) be a PFSS, \( k \in \mathbb{N} \) and \( \phi \) be a boolean function \( \phi \) on \( S \). Then \( P(S, k, \phi) \) can be computed as shown in Fig. 2.
This modal safety property can be handled by formula, namely $p_i < 1$ for all $i$.

Given a PFSS $S = (S, q, \text{next})$, $k \in \mathbb{N}$, a boolean function $\phi$ on $S$ and a probability threshold $p$, in Section 5, exploiting Proposition 2, we will present an efficient disk based algorithm to check if it holds that $P(S, k, \phi) < p$. In other words, our algorithm checks validity of a Finite Horizon Probabilistic (FHP) Safety Property. FHP safety properties are a very important class of properties. This motivates our disk based algorithm.

Of course a FHP safety property can be easily defined with a PCTL [13] formula, namely $P_{<p} [\text{true} \ U \leq k \phi]$. Thus also the probabilistic model checker PRISM [25] can be used to verify FHP safety properties.

Note however that PRISM can handle all PCTL formulas, whereas our algorithm can only handle FHP safety properties. In particular PRISM can verify properties like $P_{<p} [\text{true} \ U \phi]$ (the probability of reaching a state satisfying $\phi$ is less than $p$). Such unbounded horizon properties cannot be handled with our algorithm.

5 Analysing Probabilistic Systems with the Mur$\varphi$ Verifier

Building on the computation scheme in Fig. 2, in the following we describe an efficient disk based algorithm to verify FHP-safety properties, as well as an implementation of such an algorithm within the Mur$\varphi$ verifier. We call the resulting tool FHP-Mur$\varphi$ (Finite Horizon Probabilistic Mur$\varphi$).

5.1 Functions and Data Structures

FHP-Mur$\varphi$ input defines a PFSS $S = (S, q, \text{next})$ to which we will refer in the sequel. The FHP-Mur$\varphi$ keyword $\text{startstate}$ defines $S$ initial state $q$. Indeed, Mur$\varphi$ can have a set of initial states, however, w.l.o.g. in the following we assume...
we have just one initial state. FHP-Muφ keyword invariant defines the boolean function φ on S as well as the probability threshold β s.t. \( P(S, k, \phi) < \beta \) must hold (Remark 1).

The meaning of the declarations in Fig. 3 is as follows. Constant k (implementing \( k \)) is our verification horizon and is given to FHP-Muφ as a command line parameter. Functions \( \Phi() \) implements \( \phi \). Function \( \text{next}() \) is the nextstate function of the PFSS \( S \) defined by FHP-Muφ input. Thus function \( \text{next}() \) takes a state \( s \) as argument and returns the set \( \text{next}(s) \) of pairs \( (t, p) \) s.t. \( s \) goes to \( t \) with probability \( p \). Queues \( Q_{\text{old}}, Q_{\text{new}} \) are used to store distributions. Thus queue elements are pairs \( (s, p) \) where \( s \) is a state and \( p \) is the probability of reaching \( s \) from the initial state of \( S \). Such queues play, respectively, the same role as queues \( Q(i) \) and \( Q(i+1) \) in the while loop in Fig. 2. Queues \( Q_{\text{old}} \) and \( Q_{\text{new}} \) are the only place in which state explosion may occur in our algorithm. For this reason we implement them on disk analogously to [31]. This allows us to handle fairly large state spaces. The hash table \( M \) is a cache whose entries are pairs \( (s, p) \) as for queues \( Q_{\text{old}}, Q_{\text{new}} \). Constant \( \text{max} \_\text{prob} \_\text{Phi} \) (implementing \( \beta \)) defines our probability threshold, i.e. the max allowed value for the probability \( \text{prob} \_\text{Phi} \) of reaching (within the given horizon \( k \)) an error state (i.e. a state \( s \) s.t. \( \Phi(s) = \text{true} \)).

Note that from the above discussion follows that Muφ hash compaction \(-c\) [21] has no effect in FHP-Muφ since no FHP-Muφ data structure uses state signatures [29,30].

### 5.2 Functions Search() and Insert()

Our main function \( \text{Search}() \) is shown in Fig. 4. This function efficiently implements the computation described in Fig. 2.

Function \( \text{Insert}() \) is shown in Fig. 4. This function uses a cache table \( M \) in RAM to save queue space and thus computation time. \( M[h] \) returns the pair \( (s, p) \) stored in entry \( h \) of \( M \). \( M[h].\text{state} \) denotes \( s \) and \( M[h].\text{prob} \) denotes \( p \).

Every time it is necessary to enqueue a new pair \( (\text{state} \ s, \ \text{probability} \ p) \), \( \text{Insert}(s, p) \) is called. If state \( s \) is already stored in cache \( M \), we simply update the stored probability in \( M \), adding \( p \) to it. If state \( s \) is not stored in \( M \), we check if the slot in \( M \) in which we have to put \( s \) is free. If it is free then we insert pair \( (s, p) \) in \( M \). If it is not free, we call function \( \text{Checktable}() \) to empty \( M \) and then we insert pair \( (s, p) \) in \( M \).
int Search() {
    prob_Phi = 0;
    enqueue(Q_old, (q, 1)); /* enqueue initial state q */
    for (level = 1; level <= k; level++) {
        clear cache table M;
        while (Q_old is not empty) {
            (s, p) = dequeue(Q_old);
            for all (s', a) in next(s) {
                if (Phi(s')) {
                    prob_Phi = prob_Phi + p*a;
                    if (prob_Phi >= max_prob_Phi)
                        return(0); /* property does not hold */
                } else Insert(s', p*a);
            } /* for all */
        } /* while, level terminated, Q_old is empty */
        Checktable();
        swap Q_new with Q_old; /* now, Q_new is empty */
    } /* for */
    return(1); /* property holds */
} /* Search() */

Insert(state s, double p) {
    if (s is in M) {
        h = hash(s);
        prob = M[h].prob + p;
        M[h] = (s, prob); /* new probability of s is prob */
    } else {
        collision = Insert_in_table(s, p);
        if (collision) {
            Checktable(); /* there is space to insert now */
            Insert_in_table(s, p);
        }
    }
} /* Insert() */

boolean Insert_in_table(state s, double p) {
    h = hash(s);
    if (M[h] is free) {
        M[h] = (s, p);
        return true;
    }
    else return (M[h].state == s);
} /* Insert_in_table() */

Checktable() {
    move M in Q_new and clear M; /* M is empty now */
} /* Checktable() */

Fig. 4. Functions: Search(), Insert(), Insert_in_table(), Checktable()
If we were not using $M$, for each state $s$ at level $i$ we would have $w$ copies of $s$ in the queue, where $w$ is the number of paths of length $i$ leading to state $s$ from initial state $q$. Using $M$ rather than $w$ copies of $s$ we have just one or slightly more than one (depending on how large is $M$). This saves queue space as well as computation time. Hence, the more RAM available for $M$, the less our duplicated states, queue sizes, number of states to be explored and, finally, our computation time. For this reason $M$ should be as large as possible.

5.3 Functions Insert_in_table() and Checktable()

Function Insert_in_table() is shown in Fig. 4. Function Insert_in_table() calculates the hash value $h$ of $s$. If $M[h]$ is a free slot, Insert_in_table() inserts $s$ and $p$ in $M[h]$ and returns true. If $M[h]$ is not free, Insert_in_table() returns false without inserting $s$ and $p$ in $M$.

Function Checktable() is shown in Fig. 4. It is the only function that enqueues values in $Q_{\text{new}}$; it simply flushes $M$ into $Q_{\text{new}}$.

Function Checktable() is used by function Insert() to free $M$ when a collision occurs. It is also called at the end of the while in function Search() (Fig. 4) to enqueue in $Q_{\text{new}}$ the states visited after the last call to function Insert(), so that all states reached in the current level will be expanded in the next one.

6 Experimental Results

To show effectiveness of our approach we run two kind of experiments.

First, in Section 6.1, we compare FHP-Mur$\varphi$ with the probabilistic model checker PRISM [25].

Second, in Section 6.2, we run FHP-Mur$\varphi$ on a quite large probabilistic hybrid systems. Since our main goal is to use FHP-Mur$\varphi$ on hybrid systems, this second kind of evaluation is very interesting for us.

6.1 Probabilistic Dining Philosophers

In this Section we give our experimental results on using FHP-Mur$\varphi$ on the probabilistic protocols included in PRISM distribution [25]. We do not consider the protocols that lead to Markov Decision Processes or to Continuous Time Markov Chains, since FHP-Mur$\varphi$ cannot deal with them. Hence we only consider Pnueli-Zuck [23] and Lehmann-Rabin [20, 26] probabilistic dining philosophers protocols. Moreover, we modify PRISM definitions for such protocols in order to have a finite horizon property to verify with FHP-Mur$\varphi$. In fact, FHP-Mur$\varphi$ is unable to verify the PCTL properties for these protocols included in the PRISM distribution, since they are not of the required (finite horizon probabilistic safety) form $P_{\leq p}[\text{true } U^{\leq k} \phi]$.

Finally, FHP-Mur$\varphi$ definitions for such protocols have been obtained by translating into FHP-Mur$\varphi$ their PRISM (modified) definitions so that for each
protocol, FHP-Murφ and PRISM definitions specify exactly the same Markov Chain.

Our modifications to PRISM protocols consist in adding variables to count the number of times that a philosopher fails in getting both forks. We then verify that these counters are always less than a given maximum threshold (\texttt{MAX\_CONT} in the following) with a given probability. This corresponds to verify quality of service properties, which are very frequent in practice. E.g., in the Pnueli-Zuck protocol, we changed the code fragment in Fig. 5 with the one in Fig. 6.

We want to know the probability \( P(\texttt{MAX\_CONT}, k) \) of a counter reaching \texttt{MAX\_CONT} in at most \( k \) (horizon) steps. We set \( k = 20 \) as our finite horizon (this value occurs in a property of the Lehmann-Rabin protocol in PRISM distribution [25]).

Fig. 7 shows the PCTL property to be verified stating that the probability that a counter reaches \texttt{MAX\_CONT} has to be at most \( p \). We set \( p = 1 \) since for computing \( P(\texttt{MAX\_CONT}, k) \) the value of \( p \) does not matter.

In Fig. 8 we have the FHP-Murφ code corresponding to the PRISM code fragment of Fig. 6. Of course FHP-Murφ input language is the same as Murφ one [21], only FHP-Murφ has probabilities rather than booleans on rule guards. FHP-Murφ invariant \texttt{invariant} \( p \) \( γ \) requires that with probability at least \( p \) “all states reachable in at most \( k \) steps from the initial state satisfy \( γ \)” (\( k \) is FHP-Murφ horizon). Thus, using the notation in Section 5 we have that: \( \phi = \neg γ \) and the probability threshold (\texttt{max\_prob\_Phi} in Fig. 3) is \( (1 − p) \).

Note that in Fig. 8 the probability threshold for FHP-Murφ invariant is 0, so that FHP-Murφ will not stop verification before completing all levels of the BF computation. This forces FHP-Murφ to compute \( P(\texttt{MAX\_CONT}, k) \).

To assess FHP-Murφ effectiveness in Figs. 9, 10 we compare the results obtained with FHP-Murφ and with PRISM on, respectively, Pnueli-Zuck and Lehmann-Rabin protocols (modified as described above).

From Fig. 9 we can see that, for Pnueli-Zuck algorithm, when \texttt{NPHIL} = 5 (5 philosophers) and \texttt{MAX\_CONT} is 4, PRISM is unable to complete any verification within 2GB of RAM, independently on which of the 3 PRISM verification algorithms (totally MTBDD based, algebraic and hybrid) is chosen. Similarly, for the Lehmann-Rabin algorithm, in Fig. 10 we see that when \texttt{NPHIL} is 4, and \texttt{MAX\_WAIT} is 3, then PRISM is unable to complete the verification task in the same environment as above.

FHP-Murφ was always able to complete all given verifications tasks. Note however that, as it can be seen from Figs. 9 and 10, for the verifications tasks in which PRISM terminates, PRISM is always faster than FHP-Murφ.

Our experimental results show that for probabilistic protocols involving arithmetical computations FHP-Murφ is to be considered among the available (and valuable) tools for automatic finite horizon analysis of safety properties.

As for the numerical quality of FHP-Murφ we have that when both PRISM and FHP-Murφ terminate both give the same value for \( P(\texttt{MAX\_CONT}, k) \) (column \textit{Probability} in Figs. 9, 10).
module phil1
    p1: [0..10] init 0;
    cont1: [0..3] init 0;
    . . . .
    [] p1=6 & cont1!=MAX_CONT -> (p1'=1) & (cont1'=cont1+1);
    [] p1=6 & cont1=MAX_CONT -> (p1'=1);
    [] p1=7 & cont1!=MAX_CONT -> (p1'=1) & (cont1'=cont1+1);
    [] p1=7 & cont1=MAX_CONT -> (p1'=1);
    . . . .
    [] p1=10 -> (p1'=0) & (cont1'=0);
endmodule

Fig. 5. Pnueli-Zuck algorithm fragment to be modified in PRISM.

module phil1
    p1: [0..10] init 0;
    cont1: [0..3] init 0;
    . . . .
    [] p1=6 & cont1!=MAX_CONT -> (p1'=1) & (cont1'=cont1+1);
    [] p1=6 & cont1=MAX_CONT -> (p1'=1);
    [] p1=7 & cont1!=MAX_CONT -> (p1'=1) & (cont1'=cont1+1);
    [] p1=7 & cont1=MAX_CONT -> (p1'=1);
    . . . .
    [] p1=10 -> (p1'=0) & (cont1'=0);
endmodule

Fig. 6. Pnueli-Zuck algorithm modified fragment in PRISM.

P>=1.0 [true U<=20 ((cont1 = MAX_CONT) | (cont2 = MAX_CONT) |
( cont3 = MAX_CONT))]

Fig. 7. PCTL formula in PRISM.

function calc_prob(i : 1..NPHIL; c : 0..10) : prob;
-- probability that p[i] becomes c, NPHIL is the number of philosophers
begin
        . . . .
        case 6: if (c = 1) then return 1.0 / NPHIL; else return 0.0; endif
        case 7: if (c = 1) then return 1.0 / NPHIL; else return 0.0; endif
        . . . .
    endswitch; end;

ruleset philosophers : 1..NPHIL do ruleset next : 0..10 do rule "next"
calc_prob(philosophers, next) ==> begin
    p[i] := c;
    -- cont[1] corresponds to PRISM cont1, cont[2] to PRISM cont2 etc
    if (c = 1 & (p[i] = 6 | p[i] = 7) & (cont[i] != MAX_CONT))
        then cont[i] := cont[i] + 1; endif;
    if (p[i] = 10 & c = 0) then cont[i] := 0; endif; end; end; end;

invariant "starvation" 0.0
forall i : 1..NPHIL do (cont[i] != MAX_CONT) endforall;

Fig. 8. Pnueli-Zuck algorithm in FHP-Murϕ.
Finite Horizon Analysis of Markov Chains with the Mur$\varphi$ Verifier

<table>
<thead>
<tr>
<th>SPHIL</th>
<th>MAX_WAIT</th>
<th>Probability</th>
<th>Mur$\varphi$ memory</th>
<th>PRISM memory</th>
<th>Mur$\varphi$ time</th>
<th>PRISM time</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>3</td>
<td>7.355194164e-05</td>
<td>200</td>
<td>0.9057</td>
<td>51.970</td>
<td>1.487</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>6.883132778e-05</td>
<td>200</td>
<td>1.6844</td>
<td>52.610</td>
<td>2.507</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>1.8895076e-06</td>
<td>200</td>
<td>28.1066</td>
<td>242.940</td>
<td>28.72</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>2.910383046e-12</td>
<td>200</td>
<td>66.2659</td>
<td>244.170</td>
<td>71.112</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>9.16495139e-08</td>
<td>200</td>
<td>916.8246</td>
<td>1408.290</td>
<td>1023.468</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>4.194304e-14</td>
<td>200</td>
<td>N/A</td>
<td>1412.210</td>
<td>N/A</td>
</tr>
<tr>
<td>8</td>
<td>3</td>
<td>1.210429649e-10</td>
<td>1000</td>
<td>N/A</td>
<td>213790.740</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Fig. 9. Results on a machine with 2 processors (both INTEL Pentium III 500Mhz) and 2GB of RAM. Mur$\varphi$ options: $-b$ (bit compression), $-m200$ (use exactly 200MB of RAM), $-maxl20$ (the finite horizon is 20). The last verification had $-m1000$ (use exactly 1GB of RAM). PRISM options: default options. N/A means that PRISM was unable to complete the verification; in this case, also the $-m$ and $-a$ (totally MTBDD and algebraic verification algorithm respectively) have been used, with the same result. Memory occupations are in MB, time is in seconds.

<table>
<thead>
<tr>
<th>SPHIL</th>
<th>MAX_WAIT</th>
<th>Probability</th>
<th>Mur$\varphi$ memory</th>
<th>PRISM memory</th>
<th>Mur$\varphi$ time</th>
<th>PRISM time</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>3</td>
<td>4.8039366e-06</td>
<td>800</td>
<td>39.0625</td>
<td>1040.330</td>
<td>84.556</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>0.0</td>
<td>800</td>
<td>70.1483</td>
<td>1041.700</td>
<td>121.147</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>5.609882064e-08</td>
<td>800</td>
<td>N/A</td>
<td>33407.740</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Fig. 10. W.r.t. Fig. 9, the only change is in the Mur$\varphi$ option $-m800$ (use exactly 800MB of RAM).

6.2 Analysis of a Probabilistic Hybrid Systems with FHP-Mur$\varphi$

In this section we show our experimental results on using FHP-Mur$\varphi$ for the analysis of a real world hybrid system. Namely, the Control System for the Gas Turbine of a 2MW Electric Co-generative Power Plant (ICARO) in operation at the ENEA Research Center of Casaccia (Italy).

Our control system (Turboagas Control System, TCS, in the following) is the heart of ICARO and is indeed the most critical subsystem in ICARO. Unfortunately TCS is also the largest ICARO subsystem, thus making the use of model checking for such hybrid system a challenge.

In [22] it is shown that by adding finite precision real numbers to Mur$\varphi$, we can use Mur$\varphi$ to automatically verify TCS. In particular in [22] it has been shown the following. If the the speed of variation of the user demand for electric power (MAX_D$\varphi$ in the following) is greater than or equal to 25 (kW/sec), TCS fails in maintaining ICARO parameters within the required safety ranges.

A TCS state in which one of ICARO parameters is outside its given safety range is of course considered an error state.

In [22] the user demand has been modeled rather roughly, using nondeterministic automata. Here we show that using FHP-Mur$\varphi$ we can define and, more importantly, automatically analyse, a more accurate model for the user demand by modeling it using a Markov Chain.

To do this we define a function $p(u, i)$ as follows:

$$p(u, i) = \begin{cases} 
0.4 + \beta \frac{(u-M)|u-M|}{M^2} & \text{if } i = 1 \\
0.2 & \text{if } i = 0 \\
0.4 + \beta \frac{(M-u)|u-M|}{M^2} & \text{if } i = -1 
\end{cases}$$

(1)
ruleset d_u : -1..1 do /* disturbance: takes values -1, 0 and 1 */
  rule "time step" user_demand(u, d_u) =>> main(u, d_u);
end; -- user demand disturbance

**Fig. 11.** Rulesets with probabilistic user demand

<table>
<thead>
<tr>
<th>MAX_D_U</th>
<th>Reachable States</th>
<th>Rules Fired</th>
<th>Finite Horizon</th>
<th>CPU Time</th>
<th>Probability</th>
</tr>
</thead>
<tbody>
<tr>
<td>25</td>
<td>3018970</td>
<td>8971839</td>
<td>1600</td>
<td>68562.570</td>
<td>7.373291768e-05</td>
</tr>
<tr>
<td>35</td>
<td>2226036</td>
<td>6602763</td>
<td>1400</td>
<td>50263.020</td>
<td>1.076644427e-04</td>
</tr>
<tr>
<td>45</td>
<td>1834684</td>
<td>5439327</td>
<td>1300</td>
<td>41403.150</td>
<td>9.957147381e-05</td>
</tr>
<tr>
<td>50</td>
<td>83189</td>
<td>246285</td>
<td>900</td>
<td>2212.360</td>
<td>3.984375e-03</td>
</tr>
</tbody>
</table>

Fig. 12. Results on a machine with 2 processors (both INTEL Pentium III 500Mhz) and 2GB of RAM. Murϕ options used: -b (bit compression), -m500 (use 500 MB of RAM). Time is given in seconds.

where \( M = \text{MAX}_D\_U \) (maximum user demand value) and \( \alpha = \text{MAX}_D\_U \).

Denoting with \( u(t) \) the user demand value at time \( t \) we can define the (stochastic) dynamics for the user demand as follows:

\[
  u(t + 1) = \begin{cases} 
    \min(u(t) + \alpha, M) & \text{with probability } p(u(t), 1) \\
    u(t) & \text{with probability } p(u(t), 0) \\
    \max(u(t) - \alpha, 0) & \text{with probability } p(u(t), -1) 
  \end{cases} \tag{2}
\]

In this way, we have that the further \( u(t) \) from \( u_0 \), the higher the probability to return towards \( u_0 \), i.e. to decrement \( u(t) \) if \( u(t) > u_0 \) and to increment it otherwise.

To see that (2) is indeed a Markov Chain, it is sufficient to observe that, for any \( \beta \), the sum of the outgoing transitions is obviously 1. Moreover, since \( \frac{(u(t) - M)}{M^2} \leq 1 \), as long as \(-0.4 \leq \beta \leq 0.4 \) holds, all probability values are between 0 and 1.

With FHP-Murϕ the definition of Markov Chain (2), starting from the TCS model, is quite simple. This is done in Fig. 11, where user_demand(u, d_u) computes \( p(u, d_u) \) (1) and function main updates the system state, in particular updates \( u \) as described in (2).

In Fig. 12 we report the results of some verification runs done by FHP-Murϕ with \( \beta = 0.4 \).

We are interested in cases where the error probability is greater than 0 (zero). From the results in [22] we know that this is the case if we choose \( \text{MAX}_D\_U \) greater than or equal to 25 and the horizon value no smaller than the transition graph diameter. In our experiments here we choose our horizon as follows. Let \( \text{Diam}(n) \) be the diameter of TCS transition graph when \( \text{MAX}_D\_U = n \). We set our horizon \( k \) to be equal to \( \left\lfloor \frac{\text{Diam}(n)}{100} \right\rfloor \). In this way we check the error probability in the error neighborhood.

Fig. 12 allows us to evaluate the probability of reaching an error state when \( \text{MAX}_D\_U \) is greater than or equal to 25. Note that such a probability is rather
small, suggesting that in many cases setting $\text{MAX}_D(U)$ to 25 may be acceptable. This kind of evaluations are not possible with the nondeterministic verification of TCS carried out in [22].

7 Conclusions

We presented (Sections 3, 4) an explicit disk based verification algorithm for Probabilistic Systems defining discrete time/finite state Markov Chains. Given a Markov Chain and an integer $k$ (horizon) our algorithm checks that the probability of reaching a given error state in at most $k$ steps is below a given probability threshold.

We presented (Section 5) an implementation of our algorithm within a suitable extension of the Mur$\phi$ verifier that we call FHP-Mur$\phi$ (Finite Horizon Probabilistic-Mur$\phi$).

We presented (Section 6) experimental results comparing FHP-Mur$\phi$ with (a finite horizon subset of) PRISM, a state-of-the-art symbolic model checker for Markov Chains. Our experimental results show that FHP-Mur$\phi$ can handle systems that are out of reach for PRISM, namely those involving arithmetic operations on the state variables (e.g. hybrid systems).

Future work includes extending our approach to other models (e.g. Continuous Time Markov Chains) as well as to other kinds of PCTL formulas, e.g. formulas with unbounded until.

References


Improved Symbolic Verification Using Partitioning Techniques

Subramanian Iyer¹,², Debashis Sahoo¹,³, Christian Stangier¹, Amit Narayan¹, and Jawahar Jain¹

¹ Fujitsu Laboratories of America, Sunnyvale, CA 94085, USA
FAX: (408)530-4515
{suyer,dsahoo,cstangier,amit,jawahar}@fla.fujitsu.com
² Dept. of Computer Sciences, University of Texas at Austin, TX 78712, USA
³ Dept. of Electrical Engineering, Stanford University, CA 94305, USA

Abstract. This paper presents an efficient method to avoid memory explosion in symbolic model checking through the use of partitioning techniques. Dynamic repartitioning of Partitioned OBDDs (POBDDs) is investigated to enhance the efficiency of symbolic verification techniques. New and improved algorithms are presented for reachability based invariant checking and for model checking a fraction of CTL that is found to be most important in practice. These algorithms hinge on dynamically repartitioning the state space and exploit the partitioned nature of the data structure. The effectiveness of the partitioning approach is demonstrated on both proprietary industrial designs as well as public benchmark circuits. Notably, the approach is able to verify, and in some cases falsify, properties of interest in industry on large designs which were otherwise intractable for verification by other state-of-the-art tools.

1 Introduction

Computation Tree Logic (CTL) [6] has proved to be a popular specification language for expressing properties for formal verification of designs, especially hardware. Model checking [6,7] is the prominent automatic formal verification methodology. Reduced Ordered Binary Decision Diagrams (ROBDDs) [4] currently serve as the data structure of choice during symbolic model checking [13], because they have the desirable property of being canonical as well as manipulable. ROBDDs have efficient representations for many functions of practical interest. Unfortunately, some applications require representation of functions that only have exponential ROBDD size. This limits the complexity of problems that can be attacked by ROBDDs.

A more efficient representation was proposed through the use of Partitioned-ROBDDs (POBDDs) [12] especially for large designs. In this approach, different partitions of the Boolean space are allowed to have different variable orderings and only one partition needs to be in memory at any given time. In this paper, we extend and improve this approach to address the following issues.

© Springer-Verlag Berlin Heidelberg 2003
Firstly, we propose the use of dynamically Partitioned-OBDDs. This partitioning technique dynamically varies the number of partitions that are created and is thereby able to avoid memory explosion. Theoretical evidence [2] suggests that representations using this approach can be exponentially more compact than an approach using a fixed constant number of partitions. We incorporate this dynamic repartitioning in reachability based invariant checking as well as model checking for a portion of CTL.

Secondly, we also propose a new algorithm for model checking a significant portion of CTL. This portion is defined as those formulae, which can be represented without the use of the greatest fixpoint in existential normal form. More precisely, we efficiently handle the temporal modalities $EX$, $EF$ and their duals as well as $EU$. Such formulae are found to be a significant fraction of the properties that are of practical interest to hardware designers. In particular, this includes invariants as well as FSM deadlock avoidance properties.

It has been previously shown [16,15] that POBDDs can be used analogously to OBDDs for most applications. However, a straightforward implementation using the conventional algorithm leads to excessive overhead in the form of disk accesses, BDD variable reorderings, etc. The proposed algorithm leverages the partitioned nature of the data structure in order to significantly reduce these overheads. This is, to our knowledge, the first algorithm to take full advantage of the ideas of partitioning at an algorithmic level in the model checking procedure.

Thirdly, though it may not be obvious, use of partitioning based representation is not practical at all if one can not devise a practical and competitive strategy to discover, when appropriate, a path leading to an erroneous state. We provide a novel method to determine the same. In many cases, this method may be able to provide an error trace more efficiently than using classical OBDD based methods.

To our knowledge, this is one of the few papers demonstrating the use of partitioning based data structures in an industrial setting. On many public benchmark circuits also it shows non-linear gains in space and time, often an order of magnitude or more, over the best known state of the art tool (VIS). Thus, we demonstrate that BDD-based verification can be expanded over the limits of classical ROBDD approaches.

1.1 Comparison with Related Work

The use of partitioned transition relations [5] was proposed to control the size of symbolic representation of transition relations. The set of latches is divided into different groups which control the ROBDD-size of the transition relation and allow early quantification as well. In POBDDs, the entire Boolean space is partitioned. Thus, in order to distinguish the sense in which partitioning is performed, it would be more appropriate to call the former as clustered-transition relations. Indeed, the two approaches are orthogonal and these “clustered”-transition relations are used in the image computation of our approach as well.

Recently, a method for distributed model checking was studied by [10,9]. It parallelizes the classical symbolic model checking algorithm using the partition-
ing approach suggested in [15]. This approach uses slicing, which is similar to partitioning, with the objective of doing model checking in a distributed fashion. This approach does not address issues related to costs of communication and variable ordering in different partitions. In particular, this approach partitions the computation into a fixed number of fragments equal to the number of processors available in the distributed environment. However as noted in the literature [2], a partitioning scheme with \( k \) partitions can be exponentially more succinct than one with just \( k - 1 \) partitions. Thus, the apriori selection of the number of fragments greatly limits the efficiency of the partitioned data structure. Indeed the gain from such a static method would be obtained substantially from parallelization rather than from the inherent algorithmic advantages offered by the POBDD data structure.

In contrast, our algorithms effectively capitalize on the partitioned nature of the data structure. We require only one partition to be in memory for any image computation, and each partition can be independently ordered. Significantly, this approach incorporates a dynamic re-partitioning scheme which allows for an unbounded number of partitions to be automatically created when necessary. At the same time, we show how to drastically cut down the number of instances of inter-partition communications as compared to the classical approach. This reduces the number of transfers and reorderings of large BDDs between partitions and is found to be a significant gain in practice. We also address the issue of efficient determination of error trace in the presence of partitioning.

In the rest of this paper, we first give an overview of POBDDs and the appropriate verification techniques. Then, we describe the proposed algorithms followed by the experimental results and finally conclusions.

2 Preliminaries

The idea of partitioning was used to discuss a function representation scheme called partitioned-ROBDDs in [12,11] which was extensively developed in [16].

**Definition.** [16] Given a Boolean function \( f : B^n \rightarrow B \), defined over \( n \) inputs \( X_n = \{x_1, \ldots, x_n\} \), the partitioned-ROBDD (henceforth, POBDD) representation \( \chi_f \) of \( f \) is a set of \( k \) function pairs, \( \chi_f = \{(w_1, f_1), \ldots, (w_k, f_k)\} \) where, \( w_i : B^n \rightarrow B \) and \( f_i : B^n \rightarrow B \), are also defined over \( X_n \) and satisfy the following conditions:

1. \( w_i \) and \( f_i \) are ROBDDs respecting the variable ordering \( \pi_i \), for \( 1 \leq i \leq k \).
2. \( w_1 \lor w_2 \lor \ldots \lor w_k = 1 \)
3. \( w_i \land w_j = 0 \), for \( i \neq j \)
4. \( f_i = w_i \land f \), for \( 1 \leq i \leq k \) The set \( \{w_1, \ldots, w_k\} \) is denoted by \( W \). Each \( w_i \) is called a window function and represents a partition of the Boolean space over which \( f \) is defined. Each partition is represented separately as an ROBDD and can have a different variable order. Most ROBDD based algorithms can be adapted easily for POBDDs.

Partitioned-ROBDDs are canonical and various Boolean operations can be efficiently performed on them just like ROBDDs. In addition, they can be ex-
2.1 Reachability and Model Checking

We omit the syntax of CTL as it is widely known and readily available in the literature. We shall only note that it is possible to express any CTL formula in terms of the Boolean connectives of propositional logic and the existential temporal operators $EX$, $EU$ and $EG$. Such a representation is called the existential normal form.

Model Checking is usually performed in two stages: In the first stage, the finite state machine is reduced with respect to the formula being model checked and then the reachable states are computed. The second stage involves computing the set of states falsifying the given formula. The reachable states computed earlier are used as a care set in this step.

Since there exist computational procedures for efficiently performing Boolean operations on symbolic BDD data structures, including POBDDs, model checking of CTL formulas primarily is concerned with the symbolic application of the temporal operators. $EXq$ is a backward image and uses the same machinery as image computation during reachability, with the adjustment for the direction. $EpUq$ (resp. $EGp$) has been traditionally represented as the least (resp. greatest) fixpoint of the operator $\tau(Z) = q \lor (p \land EXZ)$ (resp. $\tau(Z) = p \land EXZ$).

*Invariants* are CTL formulas of the form $AGp$, where $p$ is a proposition, and can therefore be checked during the initial reachability computation itself.

The standard reachability algorithm is based on a breadth-first traversal of finite-state machines [8,13,19]. The algorithm takes as inputs the set of initial states, $I(s)$, expressed in terms of the present state variables, $s$, and a transition relation, $T(s,s',i)$, relating the set of next states, $N(s')$, that a system can reach from a state $s$ on an input $i$. The transition relation, $T(s,s',i)$, is obtained by taking a conjunction of the transition relations, $s'_k = f_k(s,i)$, of the individual state elements, i.e., $T(s,s',i) = \prod(s'_k = f_k(s,i))$. Given a set of states, $R(s)$, that the system can reach, the set of next states, $N(s')$, is calculated using the equation $N(s') = \exists_{s,i}[T(s,s',i) \land R(s)]$. This calculation is also known as image computation. The set of reached states is computed by adding $N(s)$ (obtained by replacing variables $s'$ with $s$) to $R(s)$ and iteratively performing the above image computation step until a fixed point is reached.

2.2 Reachability Using POBDDs

In the context of Partitioned-OBDDs, we can derive a transition relation, $T_{jk}$, from partition $j$ into partition $k$ by conjoining $T$ with the respective window functions as $T_{jk}(s,s',i) = w_j(s)w_k(s')T(s,s',i)$.
The Partitioned-ROBDD based traversal algorithm uses the ROBDD based algorithm in its inner loop to perform fixed point on individual partitions. Let us assume that we are given a partitioned-ROBDD representation $\chi_R = \{(w_j(s), R_j)|1 \leq j \leq k\}$. If we take the image of $R_j$ under $T_{jj}$, we obtain $N_j(s') = \exists_{s,i}[w_j(s)w_j(s')T(s, s', i)R_j(s)]$. Since $w_j(s')$ is independent of the variables that are to be quantified, it can be taken out of existential quantification, giving us $N_j(s') = w_j(s')[\exists_{s,i}[w_j(s)T(s, s', i)R_j(s)]]$

The image of $R_j$ under $T_{jj}$ lies completely within partition $j$. Similarly, the image, $N_l$ of $R_j$ under $T_{jl}$ will lie completely within partition $l$. This observation motivates us to define the image computation in terms of the image computed within the same partition and the image communicated to another partition. The former will be called $\text{ImgPart}$ and the latter will be called as $\text{ImgComm}$. Analogously, we define the pre-image computations $\text{preImgPart}$ and $\text{preImgComm}$. They are illustrated in the pseudo-code shown in Fig 1.

The pre-image, i.e. $\text{computeEX}$, is then obtained by their union, as $\text{preImage}(p) := \bigvee_i \text{preImgPart}(p_i, i) \lor \text{preImgComm}(p)$.

The pseudo-code for $\text{computeEX}$, as applied to POBDD, is in Fig 2a.

Notice that two approaches are possible for the computation of the communicated image: In the first, an image is computed from partition $j$ into each partition $k \neq j$ separately, using the transition relation $T_{jk}$. Alternately, one can compute the image from partition $j$ into the boolean space that is the complement of partition $j$, denoted by $\overline{j}$. The former has the advantage that the BDD representations of the transition relations $T_{jk}$ are much smaller, but in return it has to perform $O(n^2)$ image computations. We use the second method in defining $\text{imgComm}$. This method requires only $O(n)$ image computations, but each of these is followed by $O(n)$ restrict operations.

---

```
preImgPart(Bdd, j) {
    return preImage(Bdd, T_{jj})
}

preImgComm(S){
    result = \emptyset
    foreach (partition j)
        temp = preImage(S_j, T_{jj})
        foreach (partition k \neq j)
            temp_k = temp restricted to w_k
            reorder BDD temp_k from partition order j to order k
            result_k = result_k \lor temp_k
        end for
    end for
    return result
}
```

**Fig. 1.** Image Computation Algorithm
3 Improved State Space Traversal

In this section, we will describe the use of a dynamic partitioning scheme where the number of partitions can be increased or decreased as the computation progresses. This can be shown to be exponentially more succinct than the use of a fixed constant number of partitions. We also present a novel algorithm for computing a path from a state with an error to the initial state.

3.1 Dynamic Repartitioning

Dynamic repartitioning of the state space is triggered whenever the size of any partition under observation crosses a certain threshold. The partitioning variables are selected using the history of previously computed windows. Repartitioning is performed by splitting the given partition by cofactoring the entire state space based on one or more splitting variables until the blow-up has been ameliorated for each partition, which was created so far. Initially, the partitioning is done using one splitting variable. The choice of this variable is as explained before. At this point, each new partition is checked to see whether the blow-up has subsided. If not, repartitioning is called again on that partition until the blow-up has subsided in all partition.

Sometimes it is found that the blow-up in the BDD-sizes during an intermediate step of image computation is a temporary phenomenon which eventually subsides by the time the image computation is completed. In such a case the invocation of dynamic global repartitioning of the state space could create a large number of partitions, whose BDD-sizes become eventually very small. These partitions create an unnecessary amount of computational overhead. Hence, it is advantageous to create these partitions locally only for that particular image computation and then recombine them before the end of the image computation. To create these local partitions, we can cofactor the state space using the ordered list of splitting variables that was generated earlier.

Our algorithm for checking invariants performs successive steps of image computation on each $R_j$ under $T_{jj}$. Since these steps, $imgPart$, of image computation add states only within the same partition, and since different partitions are disjoint, we are guaranteed that the same state is not being visited multiple times within different partitions. Once a fixpoint is reached within a partition $j$, the procedure $imgComm$ is used to communicate the new set of states to the partition $l$ for for $1 \leq l \leq k$ and $l \neq j$. At any stage, where new states are added into the reached states set, we check for the violation of the invariant presented. If failure is detected, we stop and call the error trace mechanism to retrieve a path from the initial states to an error state. Otherwise, we proceed with traversing more states until the entire state space is exhausted, at which point, the formula has passed.

3.2 Tracing Erroneous Paths

In order to obtain a path from an error state $e$ back to an initial state $i$, the naive idea would be to compute successive preImages beginning with $e$, until
can be, in part.

For each state $s$ in the set of reachable states $S$, this tree contains the image computation when the state $s$ was first added to the reachable set $S$. The structure stores the information required to trace a backward path as follows: For each partition of the boolean space, its frontier is defined as the states added to this partition by the most recent invocation of imgComm and the subsequent imgPart operations. Each such frontier is actually a collection of sets, each represented as a BDD, whose set union represents the set of all states that have been reached in this partitions for the first time, but have not yet been used for communication to other partitions. Thus, the number of BDDs in this frontier can be, in the worst case $O(M + d_i)$ where $M$ is the number of partitions, and $d_i$ is the depth of the fixpoint in partition $i$. For the entire graph this can, in the worst case be, $O(M \times (M + d_{max}))$.

To retrieve a path from an initial state to a state $s$, we do the following:

1. Obtain the location in the computation tree that contains $s$.
2. Take the predecessor frontier of this location in the tree, and compute a backward image into this frontier to find one or more predecessor states.
3. Pick one such predecessor state.
4. Repeat steps 2 and 3 on successive states until an initial state is reached.

This gives us the backward path from state $s$ with an error to an initial state. **Advantages of partitioned error trace:** Notice that in the case of ROBDDs, the onion rings can get large in size. An effect of having these large sized representations is that image computations get more expensive. As noted before, ignoring the frontier states and performing a backward reachability is even more expensive, and in that case the backward path can be longer in length too.

Observe that partitions can often be assymmetric with respect to the space and time required for performing image computations on them. Therefore, in the presence of multiple paths from an error state to the initial states, it would be advantageous to compute the shortest path in terms of computational effort rather than the length of the path. In order to do this, we annotate the nodes of the tree with information about the amount of time the corresponding image computation required. These annotations can be used as an indicator of how much time the backward image would take, and thus, in step 3 above, they can
assist in reducing the time spent in finding a more practical path back to the initial states.

4 Model Checking Fixpoint Formulas

As mentioned in section 2.2, the modalities EX, EU and EG suffice to represent any CTL formula in existential normal form.

In particular, we note that the deadlock property $AG(p \rightarrow EFq)$ can be represented in the “greatest fixpoint free” fragment of CTL Since invariant checking and deadlocks form a large fraction of formulas that are of practical interest to designers, we will first look at the least fixpoint operator $E(pUq)$. Note that, $p$ and $q$ are not restricted to propositions and can be any CTL formulae.

4.1 Why Communication Is Expensive

It is important to notice that there are fundamental differences between the two image operations - $imgPart$ and $imgComm$. Observe that $imgPart(R_j)$ is in the same partition $j$ as the original BDD $R_j$ and therefore only one partition needs to be in memory for its computation. On the other hand, $imgComm(R_j)$ computes an image into $\overline{j}$, i.e., every partition other than $j$, therefore it needs to finally access and modify every partition. This gives rise to two important issues with respect to communication.

Firstly, the reached state set of every partition needs to be accessed. In the case of large designs, where the BDDs of even a single partition can run into millions of nodes, this usually means accessing stored partitions from the disk.

Secondly, the BDD variable order of the computed imageset must be changed from the order of the $j^{th}$ partition to that of each of its target partitions, before the new states can be added to the reached set in the target. Again, for large designs, reordering a large BDD can be an extremely expensive operation.

In this context, image computation within a partition, $ImgPart$, is a relatively inexpensive operation as compared to communication between partitions, $ImgComm$. Therefore, in the interest of minimising transfer of BDDs from one partition to another, we need a new algorithm that would decrease the number of invocations of $ImgComm$ whenever possible.

An associated advantage of performing image computation repeatedly within a partition before communicating, is that it allows some errors to be caught much earlier. When a formula fails in any partition, it becomes unnecessary to explore the other partitions any further. In this manner, it may be possible to locate the error by exploring a smaller fraction of the state space than otherwise necessary.

In the rest of this section, we will present, in the context of POBDDs, the improved model checking algorithm designed to take advantage of partitioning.

4.2 Evaluating the Least Fixpoint $E(pUq)$

The classical algorithm for the least fixpoint operator is presented in Figure 2a in terms of the POBDD data structure.
computeEX($p$) \{ 
  $R \leftarrow p$
  forall (partitions $j$)
  $S_j \leftarrow \text{preImgPart}(R_j, j)$
  end for
  $S \leftarrow S \lor \text{preImgComm}(R)$
  output $S$
}\}

computeEU($p, q$) \{ 
  $S \leftarrow q$ and $S.\text{old} \leftarrow \phi$
  repeat
    $S.\text{old} \leftarrow S$
    $S \leftarrow q \lor (p \land \text{computeEX}(S))$
  until($S = S.\text{old}$)
  output $S$
}\}

a) Classical Algorithm

Fig. 2. Algorithms for $E(pUq)$ using Partitioned-OBDDs

Notice that in the computation of $E(pUq)$, the preImage computation forms the bulk of the work performed by the algorithm. As noted in section 4.1, the cost of performing communication during every preImage is quite large. This penalty is due to resources required to transfer BDDs between partitions, to reorder the BDDs before such transfer can occur and to fetch the partitions from storage in order that the new states can be conjuncted with $p$ and disjuncted with $q$. Therefore, it is important to postpone the invocation of preImgComm, i.e., to perform as many image computations as possible locally within each partition before communication is performed across partitions.

A New Algorithm for $E(pUq)$

In this section we describe a new algorithm for model checking least fixpoint CTL formulas and sketch a proof of its correctness. Algorithm 2b for computing the set $E(pUq)$ is designed to take advantage of the partitioned nature of the data structure. Notice that we explore each partition independently of the others until they reach a fixpoint individually. Then, we perform the communication across partitions.

This allows us to keep just one partition in memory at any given time. It also greatly reduces the number of communication induced BDD transfers, disk accesses and variable reordering calls.

Before proving the correctness of the new algorithm, we define some notation. Let the set of states $S$ at the end of the $k$th iteration of the outermost repeat-until loop in algorithm 2b be represented by $S^k$.

For every state $s \models E(pUq)$, either $s \models q$ or there exists a sequence of states $s_0, s_1, \ldots, s_k$ that has the smallest length $k \neq 0$ such that $s_0 = s$, $s_k \models q$, ...
\[ \forall i < k : s_i \models p \quad \text{and} \quad \forall i < k : s_i \in \text{preImage}(s_{i+1}). \] Such a sequence of states is called a witness for the inclusion of \( s \) in \( E(pUq) \), and \( k \) is its length.

For the sake of convenience, we will use the symbol for a formula to also represent the set of states it represents. We first show that algorithm 2b terminates.

**Lemma 1.** (Termination) For any integer \( i \), \( S^{i+1} \supseteq S^i \). The inequality is strict unless a fixpoint is reached.

The proof is evident from the construction of sets \( S^k \). Since any step of the procedure must add at least one new state to the set \( S \), we have termination at the end of at most as many iterations as there are states in the space under consideration.

**Theorem 1.** The procedure computeEU of algorithm 2b, given the set of states corresponding to formulas \( p \) and \( q \) as inputs, terminates with the output \( S \) being precisely the set of states that model the formula \( E(pUq) \).

**Proof:** Soundness: We prove by induction on the sets \( S^k \) that the procedure is sound, i.e., at all times \( S \models E(pUq) \). This clearly holds for any state in the initial set \( S^0 = q \), since any state satisfying \( q \) also satisfies \( E(pUq) \).

Assume, it holds for \( S^i \), i.e., that \( S^i \models E(pUq) \). Consider a state \( s \in S^{i+1} - S^i \). Then, by construction of \( S^{i+1} \) from \( S^i \), we have \( s \models p \). Either \( s \) is added during some step of the inner fixpoint loop or it is added in a step of communication, i.e., \( s \in \text{preImgComm}(S^i) \).

Suppose \( s \) is added in the inner fixpoint loop of some partition \( j \). Since \( S^i \) is a POBDD, let us call the projection of \( S^i \) in partition \( j \) as \( S^i_j \). From before, we know \( \text{preImgPart}(S^i_j, j) \subseteq \text{preImgPart}(S^i) \subseteq \text{preImage}(S^i) \). Also notice that the variable for the inner fixpoint is initialized to \( S^i_j \). Therefore, every state added in the first step of the inner fixpoint models \( p \wedge EX(E(pUq)) \) and therefore models \( E(pUq) \). Consequently, we can show by induction that any state added in the inner fixpoint loop for partition \( j \) must model \( E(pUq) \).

In the second case, \( s \) was added in some step of the communication. Considering that \( \text{preImgComm}(S^i) \subseteq \text{preImage}(S^i) \), any state added in the communication step models \( p \wedge EX(E(pUq)) \), and therefore \( E(pUq) \). In particular, \( s \models E(pUq) \).

Consequently, \( S^{i+1} - S^i \models E(pUq) \) and the soundness of the procedure follows by induction.

Completeness: We next show the completeness, i.e., that every state of \( E(pUq) \) is indeed in set \( S \). Let \( T^k \) be the set of states, whose inclusion in \( E(pUq) \) is witnessed by a path of length at most \( k \). We prove by induction on \( k \) that \( T^k \subseteq S \). In the base case, this trivially holds because \( T^0 = q = S^0 \subseteq S \).

Now, let us assume that \( T^i \subseteq S \). For any state \( s \in T^{i+1} \) consider the sequence of states \( s_0 = s, s_1, \ldots, s_{i+1} \) that witnesses its inclusion in \( E(pUq) \). We will show that \( s \in S \).

Now, the sequence \( s_1, \ldots, s_{i+1} \) is a witness for \( s_1 \), therefore \( s_1 \in T^i \subseteq S \). In particular, there exists a smallest \( j \) so that \( s_1 \in S^j \). We know that \( s \models p \) and
s ∈ preImage(s₁) ⊆ preImage(Sᵢ). From the definition of Sᵢ and Algorithm 2b, we have that

\[ S^{i+1} = S^i \cup (p \land preImgPart(S^i)) \cup (p \land preImgComm(S^i)) \]

Therefore, \( s \in S^{i+1} \subseteq S \), whereby \( T^{i+1} \subseteq S \). By induction, this gives us \( E(pUq) \subseteq S \).

Together with lemma 1, this proves that algorithm 2b terminates with the set \( S = E(pUq) \).

4.3 Evaluating the Greatest Fixpoint \( EGp \)

The model checking of \( EGp \) is done by computation of the greatest fixpoint of the operator \( \tau(Z) = p \land EXZ \). As in the case of least fixpoint, one would like to postpone the communication until after each partition has reached its individual fixpoint independent of the other partitions. However, the description of this is considerably more complex and thus far we have only implemented a simple, classical, version of the greatest fixpoint algorithm for \( EGp \) in terms of POBDDs.

Even so, most specifications of interest in practice are expressible in the fragment of CTL free of greatest fixpoints. For e.g., deadlock avoidance properties of the form \( AG(p \rightarrow EFq) \) and invariants can both be expressed in existential normal form using only least fixpoints. Therefore, we find that the inability to postpone communications for the greatest fixpoint does not impose a great disadvantage in most practical applications.

5 Experiments

We implemented dynamic partitioning-based model checking using the CUDD-package [18] (version 2.3.0) for OBDD representation. We use the routines from VIS [3] (version 1.4) for reading in the design and to build the initial transition relation using the IWLS95 method [17]. Our implementation can be thought of as building on top of VIS and therefore a comparison with VIS is natural.

We found empirically that for our benchmarks VIS-2.0 using the MLP [14] method performs worse than VIS-1.4 using the IWLS95 method, probably due to known problems in preimage computation. Thus, we compared our methods to VIS by using the IWLS95 method for both.

Benchmarks and Experimental Setup

For our experiments, we used the designs from the Vis Verilog benchmark suite [1]. This suite also contains properties given in CTL formulas for verification. We pick the properties which when expressed existentially are “greatest
fixpoint free”. On the entire benchmark suite this is found to cover about 80% of all properties, which is believed to be typical. Finally, we also used proprietary designs that were made available by Fujitsu designers.

The parameters of VIS and CUDD are left unchanged at their default values. Experiments on the public benchmarks were performed on dual-processor Xeon 2.2Ghz workstations with 2 GB of RAM running Linux. The invariant checking as well as model checking experiments used dynamic partitioning. Both were run with a timeout limit of 24 hours.

The peak number of live nodes is given by Peak Node. The CPU time is measured in seconds and given as Time. The column denoted with Time Gain (resp. Space Gain) describes the gain in time (space) of POBDDs over VIS.

Results on Invariant Checking. We compare our POBDD method to the standard VIS approach on invariant checking in Table 1. Note that this table is restricted to the largest entries (BDD-nodes > 300K) in the benchmark suite. Our partitioned approach clearly outperforms the state-of-the-art VIS in time as well as in space. Especially for the larger circuits the improvement is drastic, since we complete the verification of four circuits that timed out using VIS.

Comparison with Static Partitioning It is natural to analyse what benefit dynamic partitioning offers over static partitioning. In Fig. 3, we compare the performance of the proposed dynamic partitioning based invariant checking approach with invariant checking based on the static partitioning method of [15]. In particular, note that in the last case, vrc32_8, the previous approach timed out after 86,400 seconds whereas we are able to complete in about 12,000 seconds.

Results on Model Checking. The results on runtime and space requirements in model checking are presented in Table 2. POBDDs may not sometimes show their full potential on the smaller circuits due to the overhead of creating and maintaining partitions. Nevertheless, the results show that POBDD-based model checking can out-perform VIS even on such cases in time as well as in space.

But, more important are the last few entries in the table, showing the harder benchmarks. Here, the POBDD-based model checking clearly outperforms the
Fig. 3. Comparison of Times taken (Normalized) by Different Partitioning Approaches for Invariant Checking on some Large designs

Table 2. Model Checking on Large Designs

<table>
<thead>
<tr>
<th>Circuit</th>
<th>Number of Partitions</th>
<th>Peak Nodes</th>
<th>Time (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>VIS</td>
<td>POBDD</td>
</tr>
<tr>
<td>product</td>
<td>4</td>
<td>919 K</td>
<td>108 K</td>
</tr>
<tr>
<td>s1269b</td>
<td>4</td>
<td>2.3 M</td>
<td>317 K</td>
</tr>
<tr>
<td>am2910</td>
<td>4</td>
<td>&gt;4.9 M</td>
<td>127 K</td>
</tr>
<tr>
<td>twoQ</td>
<td>6</td>
<td>&gt;5.5 M</td>
<td>1.8 M</td>
</tr>
<tr>
<td>palu</td>
<td>12</td>
<td>&gt;10.5 M</td>
<td>3 M</td>
</tr>
<tr>
<td>am2901</td>
<td>5</td>
<td>&gt;5.7 M</td>
<td>1.94 M</td>
</tr>
</tbody>
</table>

classical approach and is able to even finish four of the designs that cannot be finished within the given 24 hour timeout when using VIS.

It is also noteworthy, that the maximum peak BDD-size of one partition is often an order of magnitude smaller than the maximum peak node size for ROBDDs. We have observed that this reduction is in many cases more than the number of partitions created.

**Industrial Circuits** The properties for industrial circuits were taken from actual Fujitsu designs with sizes ranging from 2000 to 10000 flip-flops. Table 3 shows the summarized results for the comparison of POBDD-based model checking with VIS for three different types of properties. For the first two properties, *Index range* and *full-case*, the POBDD method is able to finish 11 (resp. 5) more properties than the OBDD method.
Table 3. Model Checking of Industrial Circuits (2,000 to 10,000 flip-flops)

<table>
<thead>
<tr>
<th>Property Type</th>
<th>Method</th>
<th>Pass</th>
<th>Fail</th>
<th>Timeout</th>
</tr>
</thead>
<tbody>
<tr>
<td>Index out of range</td>
<td>POBDD</td>
<td>678</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>VIS</td>
<td>667</td>
<td>0</td>
<td>11</td>
</tr>
<tr>
<td>Full Case</td>
<td>POBDD</td>
<td>16</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>VIS</td>
<td>11</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>Synchronizer data stability</td>
<td>POBDD</td>
<td>2</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>VIS</td>
<td>0</td>
<td>2</td>
<td>4</td>
</tr>
</tbody>
</table>

For the third property, *data stability*, the POBDD method is actually able to detect 2 failures more in addition to the passing properties than the OBDD approach.

6 Conclusions

In this paper we addressed the memory explosion problem associated with model checking through the use of dynamically Partitioned-OBDDs. We have shown that it can be significantly better for problems, where the state of the art can require impractically large computational resources. The significant advantage of the proposed verification technique is its ability to control the memory required. Usually, this has the added advantage of improvement in run-time, which is primarily governed by the BDD-sizes. On large circuits we find that the computational savings offered by the proposed partitioning based model checking can be significant. We have shown cases, where our proposed method could finish in just a few thousand seconds, whereas other approaches timed out after a day. Importantly, a new algorithm for invariant checking and for model checking the fragment of CTL free of greatest fixpoint in the existential normal form are presented. This can handle many more properties of practical interest and truly exploit the theoretical and practical benefits of dynamically partitioned-OBDDs.

Acknowledgment. The authors would like to thank Prof. E. Allen Emerson and Prof. David Dill for their advice and encouragement.

References


Author Index

Aagaard, Mark D. 66
Abu-Haimed, Husam 158
Al Sammane, Ghiath 150
Ashar, Pranav 334

Barner, Sharon 35
Beer, Ilan 141
Berger, Eli 141
Beringer, Lennart 270
Beyer, Sven 51
Borrione, Dominique 150
Bryant, Randal E. 348

Casas, Jeremy 170
Chaki, Sagar 19
Chockler, Hana 111
Clarke, Edmund 19

Della Penna, Giuseppe 277, 394
Dill, David L. 158

Emerson, E. Allen 216, 247
Encrenaz, Emmanuelle 164

Fisler, Kathi 185

Ganai, Malay K 334
Geist, Daniel 3
Gopalakrishnan, Ganesh 81
Gordon, Mike 200
Groce, Alex 19
Gupta, Aarti 334
Gurumurthy, Sankar 96

Hooman, Jozef 231
Hu, Alan J. 170
Hunt, Warren A. 319
Hurd, Joe 200
Hymans, Charles 263

Intrigila, Benedetto 277, 394
Iyer, Subramanian 410

Jacobi, Chris 51
Jain, Jawahar 410

Kahlon, Vineet 247
Kröning, Daniel 51
Krug, Robert Bellarmine 319
Kupferman, Orna 96, 111

Lahiri, Shuvendu K. 348
Langberg, Michael 363
Layouni, Mohamed 231
Leinenbach, Dirk 51
Lindstrom, Gary 81

Manolios, Panagiotis 304
Matusevich, Mark 141
Melatti, Igor 277, 394
Moore, J Strother 289, 319

Narayan, Amit 410

Ostier, Pierre 150

Pastor, Enric 378
Paul, Wolfgang J. 51
Peña, Marco A. 378
Pnueli, Amir 363

Rabinovitz, Ishai 35
Rodeh, Yoav 363
Roesner, Wolfgang 1
Roux, Cédric 164

Sahoo, Debasish 410
Schmaltz, Julien 150
Sebastiani, Roberto 126
Seshia, Sanjit A. 348
Sheeran, Mary 4
Singh, Satnam 283
Slind, Konrad 81, 200
Somenzi, Fabio 2, 96
Stangier, Christian 410
Strichman, Ofer 19
Tahar, Sofiène 231
Toma, Diana 150
Tonetta, Stefano 126
Tronci, Enrico 277, 394
Tzoref, Rachel 141
Vardi, Moshe Y. 96, 111

Venturini Zilli, Marisa 277, 394
Wahl, Thomas 216
Yang, Jin 170
Yang, Yue 81
Yang, Zijiang 334