Lecture Notes in Computer Science 1102
Edited by G. Goos, J. Hartmanis and J. van Leeuwen

Advisory Board: W. Brauer D. Gries J. Stoer

Rajeev Alur Thomas A. Henzinger (Eds.)

Computer Aided Verification

8th International Conference, CAV '96
New Brunswick, NJ, USA
July 31 - August 3, 1996
Proceedings

Springer

Volume Editors

Rajeev Alur
Bell Laboratories, Lucent Technologies
700 Mountain Avenue, Murray Hill, NJ 07974, USA

Thomas A. Henzinger
Department of Electrical Engineering and Computer Science
University of California at Berkeley
Berkeley, CA 94720, USA

Cataloging-in-Publication data applied for

Die Deutsche Bibliothek - CIP-Einheitsaufnahme

Computer aided verification : 8th international conference ; proceedings / CAV '96, New Brunswick, New Jersey, USA, July 31 - August 3, 1996. Rajeev Alur ; Thomas A. Henzinger (ed.). - Berlin ; Heidelberg ; New York ; Barcelona ; Budapest ; Hong Kong ; London ; Milan ; Paris ; Santa Clara ; Singapore ; Tokyo : Springer, 1996
(Lecture notes in computer science ; Vol. 1102)
ISBN 3-540-61474-5
NE: Alur, Rajeev [Hrsg.]; CAV <8, 1996, New Brunswick, NJ>; GT

CR Subject Classification (1991): F3, D.2.4, D.2.2, F.4.1, B.7.2, C.3, I.2.3

ISSN 0302-9743
ISBN 3-540-61474-5 Springer-Verlag Berlin Heidelberg New York

Typesetting: Camera-ready by author
SPIN 10513330 06/3142 – 5 4 3 2 1 0 Printed on acid-free paper

Preface

This volume contains the proceedings of the Eighth International Conference on Computer Aided Verification (CAV '96), organized July 31–August 3, 1996, in New Brunswick, New Jersey. The annual CAV series is dedicated to the advancement of the theory and practice of computer-assisted formal analysis methods for software and hardware systems. The conference covers the spectrum from theoretical results to concrete applications, with an emphasis on verification tools and the algorithms and techniques that are needed for their implementation. This year's call-for-papers invited submissions in two separate categories: regular research contributions, and short descriptions of tools and case studies. Of the 93 submissions in the first category, 32 were selected for presentation at the conference. From an enthusiastic response to the second category, 20 submissions were chosen.

The conference will include four invited lectures, and a morning session with invited talks by representatives from industry. Invited lectures will be given by Michael Rabin (Harvard University, USA, and Hebrew University, Israel) on *Randomization and Protocol Verification*, by John Rushby (SRI International, USA) on *Automated Deduction and Formal Methods*, and by Bill Roscoe (Oxford University, UK) on *Refinement-based Model Checking*. Amir Pnueli (Weizmann Institute, Israel) will give an after-banquet speech on *The Potential and Sensible Scopes of Formal Methods*. The industrial session will include invited talks by Justin Harlow and Peter Verhofstadt (Semiconductor Research Corporation, USA) on *Formal Methods in the IC Industry: Trends and Directions*, by Patrick Scaglia (Cadence Berkeley Labs, USA) on *From Wired Homes to Automated Highways: A Perspective on Verification in the 21st Century*, by Wolfram Büttner (Siemens Corporate Research and Development, Germany) on *Formal Verification at Siemens: Achievements, Problems, Trends*, by Gary De Palma (Lucent Technologies, USA) on *User Experiences with FormalCheck*, by Carl Pixley (Motorola, USA) on *Formal and Informal Functional Verification in a Commercial Environment*, and by Shoham Ben-David (IBM Haifa Research Lab, Israel) on *Model Checking at Work*.

The program of CAV '96 was selected by a program committee consisting of R. Alur (co-chair, Bell Labs, USA), R.K. Brayton (University of California at Berkeley, USA), K. Čerāns (University of Latvia, Latvia), D.L. Dill (Stanford University, USA), E.A. Emerson (The University of Texas at Austin, USA), O. Grumberg (The Technion, Israel), T.A. Henzinger (co-chair, University of California at Berkeley, USA), K.G. Larsen (Aalborg University, Denmark), D.E. Long (Bell Labs, USA), K.L. McMillan (Cadence Berkeley Labs, USA), A.K. Mok (The University of Texas at Austin, USA), D. Peled (Bell Labs, USA), A. Pnueli (Weizmann Institute, Israel), C.-J.H. Seger (Intel Development Labs, USA), J. Sifakis (VERIMAG, France), S.A. Smolka (SUNY at Stony Brook, USA), M.K. Srivas (SRI International, USA), W. Thomas (Universität Kiel, Germany), F. Vaandrager (Nijmegen University, The Netherlands), M.Y. Vardi (Rice University, USA), and P. Wolper (Université de Liège, Belgium).

The following researchers helped in the evaluation of the submissions, and we are grateful for their efforts: L. Aceto, J. Andersen, E. Asarin, P. Attie, F. Balarin, C. Barrett, T. Basten, I. Beer, H. Ben-Abdallah, S. Ben-David, S. Berezin, K. Bernstein, B. Bloom, B. Boigelot, D. Bosscher, A. Bouajjani, R. Bryant, N. Buhrke, S.-T. Cheng, C.-T. Chou, D. Clarke, D. Cyrluk, D. Dams, S. Dawson, Z. Dayar, S. Edwards, C. Eisner, K. Engelhardt, J. Esparza, A. Felty, J.-C. Fernandez, T. Fernando, M. Fisher, W. Fokkink, N. Francez, H. Garavel, T. Gelsema, R. Gerth, D. Giest, P. Godefroid, S. Graf, D. Griffioen, E. Gukovski, P. Habermehl, N. Halbwachs, W. Hesselink, R. Ho, R. Hojati, G. Holzmann, A. Hu, H. Huttel, A. Ingolfsdottir, C.N. Ip, A. Isles, S.P. Iyer, B. Jacobs, H.E. Jensen, M. Kaltenbach, S. Katz, A. Kerbrat, J. Kleist, S. Krishnan, K. Kristoffersen, K. Kuehnle, Y. Kukimoto, Y. Lakhnech, A. Landver, K. Laster, D. Lee, H. Lescow, A. Levin, X. Liu, S. Ma, O. Maler, F. Maraninchi, E. Mikk, H. Miller, F. Moller, L. Mounier, M. Mukund, K. Namjoshi, V. Natarajan, D. Niwinski, S. Park, C. Petersohn, A. Philippou, I. Polak, A. Ponse, C. Puchol, S. Rajamani, Y.S. Ramakrishna, R. Ramanujam, R. Ranjan, P. Raymond, A. Rensink, J. Romijn, H. Rueß, J. Sanghavi, I. Schiering, R. Segala, S. Seibert, N. Shankar, T. Shiple, V. Singhal, A. Skou, O. Sokolsky, J. Springintveld, R. Staerk, M. Staskauskas, F. Stomp, K. Stroetmann, R. Sumners, K. Sunesen, G. Swamy, S. Tasiran, P.S. Thiagarajan, R. Trefler, J. Tretmans, S. Tripakis, S. Ur, A. van Deursen, M. van Hulst, B. Victor, T. Villa, J. Voege, T. Vos, I. Walukiewicz, P. Weidmann, H. Wupper, C.H. Yang, M. Yannakakis, and S. Yovine.

The CAV steering committee consists of the conference founders Ed Clarke (Carnegie Mellon University, USA), Bob Kurshan (Bell Labs, USA), Amir Pnueli (Weizmann Institute, Israel), and Joseph Sifakis (VERIMAG, France). We thank them and Pierre Wolper (Université de Liège, Belgium), the conference chair of CAV '95, for valuable advice on the organization of the conference.

This year, CAV will be part of the Federated Logic Conference (FLoC '96), and organized jointly with the 13th International Conference on Automated Deduction (CADE), the 11th Annual IEEE Symposium on Logic in Computer Science (LICS), and the 7th International Conference on Rewriting Techniques and Applications (RTA). FLoC '96 will be hosted by the Center for Discrete Mathematics and Computer Science (DIMACS), an NSF Science and Technology Center located at Rutgers University, as part of a Special Year on Logic and Algorithms. The FLoC steering committee consists of Stephen Mahaney (Rutgers University, USA) and Moshe Vardi (chair, Rice University, USA). The FLoC organizing committee consists of Rajeev Alur (Bell Labs, USA), Leo Bachmair (SUNY at Stony Brook, USA), Amy Felty (Bell Labs, USA), Doug Howe (Bell Labs, USA), and Jon Riecke (chair, Bell Labs, USA). We gratefully acknowledge the help of Priscilla Rasmussen from ARCS, who is responsible for registration and site arrangements.

FLoC '96 receives financial support from DIMACS, and also from AT&T Research, IBM Almaden Research, the IEEE Computer Society, Lucent Technologies Bell Labs, and the Max-Planck Institute. Student registration at CAV '96 is

subsidized due to financial support from Cadence Berkeley Labs, Lucent Technologies, and Siemens Corporate Research and Development. We thank all sponsors for their generosity.

Murray Hill, New Jersey
Berkeley, California

May 1996

Rajeev Alur
Tom Henzinger

Table of Contents

Tools and Case Studies

Symbolic Verification of Communication Protocols with Infinite State Spaces Using QDDs (Extended Abstract)

Bernard Boigelot*
Université de Liège
Institut Montefiore, B28
4000 Liège Sart-Tilman, Belgium
Email: boigelot@montefiore.ulg.ac.be

Patrice Godefroid
Lucent Technologies - Bell Laboratories
1000 E. Warrenville Road
Naperville, IL 60566, U.S.A.
Email: god@bell-labs.com

Abstract

We study the verification of properties of communication protocols modeled by a finite set of finite-state machines that communicate by exchanging messages via *unbounded* FIFO queues. It is well-known that most interesting verification problems, such as deadlock detection, are undecidable for this class of systems. However, in practice, these verification problems may very well turn out to be decidable for a subclass containing most "real" protocols.

Motivated by this optimistic (and, we claim, realistic) observation, we present an algorithm that may construct a *finite* and *exact* representation of the state space of a communication protocol, even if this state space is *infinite*. Our algorithm performs a *loop-first search* in the state space of the protocol being analyzed. A loop-first search is a search technique that attempts to explore first the results of successive executions of loops in the protocol description (code). A new data structure named *Queue-content Decision Diagram* (QDD) is introduced for representing (possibly infinite) sets of queue-contents. Operations for manipulating QDDs during a loop-first search are presented.

A loop-first search using QDDs has been implemented, and experiments on several communication protocols with infinite state spaces have been performed. For these examples, our tool completed its search, and produced a finite symbolic representation for these infinite state spaces.

1 Introduction

State-space exploration is one of the most successful strategies for analyzing and verifying properties of finite-state concurrent reactive systems. It proceeds by exploring a global state graph representing the combined behavior of all concurrent components in the system. This is done by recursively exploring all successor states of all states encountered during the exploration, starting from a given initial state, by executing all enabled transitions in each state. The state graph that is explored is called the *state space* of the system. Many different types of properties of a system can be checked by exploring its state space: deadlocks, dead code, violations of user-specified assertions, etc. Moreover, the range of properties that state-space exploration techniques can verify has been substantially broadened during the last decade thanks to the development of model-checking methods for various temporal logics (e.g., [CES86, LP85, QS81, VW86]).

*"Aspirant" (Research Assistant) for the National Fund for Scientific Research (Belgium). The work of this author was done in part while visiting Bell Laboratories.

Verification by state-space exploration has been studied by many researchers (cf. [Liu89, Rud87]). The simplicity of the strategy lends itself to easy, and thus efficient, implementations. Moreover, verification by state-space exploration is fully automatic: no intervention of the designer is required. The main limit of state-space exploration verification techniques is the often excessive size of the state space. Obviously, this *state-explosion problem* is even more critical when the state space being explored is infinite.

In contrast with the last observation, we show in this paper that verification by state-space exploration is also possible for systems with *infinite* state spaces. Specifically, we consider communication protocols modeled by a finite set of finite-state machines that communicate by exchanging messages via *unbounded* FIFO queues. We present a state-space exploration algorithm that may construct a *finite* and *exact* representation of the state space of such a communication protocol, even if this state space is *infinite*. From this symbolic representation, it is then straightforward to verify many properties of the protocol, such as the absence of deadlocks, whether or not the number of messages stored in a queue is bounded, and the reachability of local and global states.

Of course, given an arbitrary protocol, our algorithm may not terminate its search. Indeed, it is well-known that unbounded queues can be used to simulate the tape of a Turing machine, and hence that most interesting verification problems are undecidable for this class of systems [BZ83]. However, in practice, these verification problems may very well turn out to be decidable for a subclass containing most "real" protocols. To support this claim, properties of several communication protocols with infinite state spaces have been verified successfully with the algorithm introduced in this paper.

In the next section, we formally define communication protocols. Our algorithm performs a *loop-first search* in the state space of the protocol being analyzed. A loop-first search is a search technique that attempts to explore first the results of successive executions of loops in the protocol description (code). This search technique is presented in Section 3. A new data structure, the *Queue-content Decision Diagram* (QDD), is introduced in Section 4 for representing (possibly infinite) sets of queue-contents. Operations for manipulating QDDs during a loop-first search are presented in Section 5. A loop-first search using QDDs has been implemented, and experiments on several communication protocols with infinite state spaces are reported in Section 6. This paper ends with a comparison between our contributions and related work.

2 Communicating Finite-State Machines

Consider a protocol modeled by a finite set $\mathcal{M}$ of finite-state machines that communicate with each other by sending and receiving messages via a finite set Q of unbounded FIFO queues, modeling communication channels. Let M_i denote the set of messages that can be stored in queue q_i, $1 \leq i \leq |Q|$. For notational convenience, let us assume that the sets M_i are pairwise disjoint. Let C_i denote the finite set of states of machine $\mathcal{M}_i$, $1 \leq i \leq |\mathcal{M}|$.

Formally, a protocol P is a tuple (C, c_0, A, Q, M, T) where $C = C_1 \times \cdots \times C_{|\mathcal{M}|}$ is a finite set of *control states*, $c_0 \in C$ is an *initial control state*, A is a finite set of *actions*, Q is a finite set of unbounded FIFO queues, $M = \cup_{i=1}^{|Q|} M_i$ is a finite set of *messages*, and T is a finite set of *transitions*, each of which is a triple of the form (c_1, op, c_2) where c_1 and c_2 are control states, and op is a label of one of the forms $q_i!w$, where $q_i \in Q$ and $w \in M_i^*$, $q_i?w$, where $q_i \in Q$ and $w \in M_i^*$, or a, where $a \in A$.

A transition of the form $(c_1, q_i!w, c_2)$ represents a change of the control state from c_1 to c_2 while appending the messages composing w to the end of queue q_i. A transition of the form $(c_1, q_i?w, c_2)$ represents a change of the control state from c_1 to c_2 while removing the messages composing w from the head of queue q_i.

A global state of a protocol is composed of a control state and a *queue-content*. A queue-content

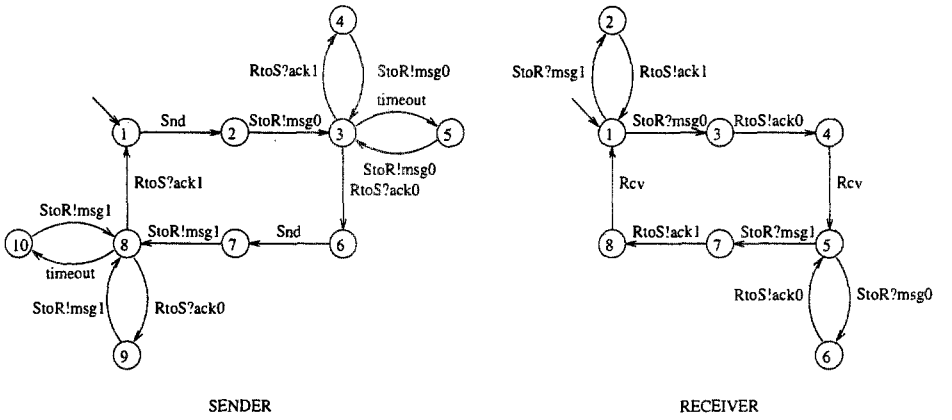


Figure 1: Alternating-Bit Protocol

associates with each queue q_i a sequence of messages from M_i. Formally, a *global state* γ, or simply a *state*, of a protocol is an element of the set $C_1 \times \cdots \times C_{|\mathcal{M}|} \times M_1^* \times \cdots \times M_{|Q|}^*$. A global state $\gamma = (c(1), c(2), \ldots, c(|\mathcal{M}|), w(1), w(2), \ldots, w(|Q|))$ assigns to each finite-state machine $\mathcal{M}_i$ a "local" (control) state $c(i) \in C_i$, and associates with each queue q_j a sequence of messages $w(j) \in M_j^*$ which represents the content of q_j in the global state γ. The *initial global state* of the system is $\gamma_0 = (c_0(1), c_0(2), \ldots, c_0(|\mathcal{M}|), \varepsilon, \ldots, \varepsilon)$, i.e., we assume that all queues are initially empty.

A *global transition relation* $\rightarrow$ is a set of triples (γ, a, γ'), where γ and γ' are global states, and $a \in A \cup \{\tau\}$. Let $\gamma \xrightarrow{a} \gamma'$ denote $(\gamma, a, \gamma') \in \rightarrow$. Relation $\rightarrow$ is defined as follows:

- if $(c_1, q_i!w, c_2) \in T$, then $(c_1(1), c_1(2), \ldots, c_1(|\mathcal{M}|), w'(1), w'(2), \ldots, w'(|Q|)) \xrightarrow{\tau}$ $(c_2(1), c_2(2), \ldots, c_2(|\mathcal{M}|), w''(1), w''(2), \ldots, w''(|Q|))$ where $w''(i) = w'(i)w$ and $w''(j) = w'(j)$, $j \neq i$ (the control state changes from c_1 to c_2 and w is appended to the end of queue q_i);
- if $(c_1, q_i?w, c_2) \in T$, then $(c_1(1), c_1(2), \ldots, c_1(|\mathcal{M}|), w'(1), w'(2), \ldots, w'(|Q|)) \xrightarrow{\tau}$ $(c_2(1), c_2(2), \ldots, c_2(|\mathcal{M}|), w''(1), w''(2), \ldots, w''(|Q|))$ where $w'(i) = ww''(i)$ and $w''(j) = w'(j)$, $j \neq i$ (the control state changes from c_1 to c_2 and w is removed from the head of queue q_i);
- if $(c_1, a, c_2) \in T$, then $(c_1(1), c_1(2), \ldots, c_1(|\mathcal{M}|), w'(1), w'(2), \ldots, w'(|Q|)) \xrightarrow{a}$ $(c_2(1), c_2(2), \ldots, c_2(|\mathcal{M}|), w''(1), w''(2), \ldots, w''(|Q|))$ with $w''(i) = w'(i)$, for all $1 \leq i \leq |Q|$ (the control state changes from c_1 to c_2 while the action a is performed).

A global state γ' is said to be *reachable* from another global state γ if there exists a sequence of global transitions $(\gamma_{i-1}, a_i, \gamma_i)$, $1 \leq i \leq n$, such that $\gamma = \gamma_0 \xrightarrow{a_1} \gamma_1 \cdots \gamma_{n-1} \xrightarrow{a_n} \gamma_n = \gamma'$. The *global state space* of a system is the (possibly infinite) set of all states that are reachable from the initial global state γ_0.

Example 1 As an example of communication protocol, consider the well-known Alternating-Bit Protocol [BSW69]. This protocol can be modeled by two finite-state machines *Sender* and *Receiver* that communicate via two unbounded FIFO queues *StoR* (used to transmit messages from the Sender to the Receiver) and *RtoS* (used to transmit acknowledgments from the Receiver to the Sender).

Precisely, the Alternating-Bit Protocol is modeled by the protocol (C, c_0, A, Q, M, T) where $C = C_{Sender} \times C_{Receiver}$, where $C_{Sender} = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$ and $C_{Receiver} = \{1, 2, 3, 4, 5, 6, 7, 8\}$; $c_0 = (1, 1)$; $A = \{Snd, Rcv, timeout\}$; $Q = \{StoR, RtoS\}$; $M = M_{StoR} \cup M_{RtoS}$, where $M_{StoR} = \{msg0, msg1\}$ and $M_{RtoS} = \{ack0, ack1\}$; and T contains the transitions $((s_1, r_1), op, (s_2, r_2))$ where

either $r_1 = r_2$ and (s_1, op, s_2) is a transition in the *Sender* machine of Figure 1, or $s_1 = s_2$ and (r_1, op, r_2) is a transition in the *Receiver* machine of Figure 1. The action *Snd* models a request to the *Sender*, coming from a higher-level application, to transmit data to the *Receiver* side. The actual data that are transmitted are not modeled, only message numbers $msg0$ and $msg1$ are transmitted over the queues. Similarly, the action *Rcv* models the transmission of data received by the *Receiver* to a higher-level application. The actions labeled by *timeout* model the expiration of timeouts. ■

3 Loop-First Search

All state-space exploration techniques are based on a common principle: they spread the reachability information along the transitions of the system to be analyzed. The exploration process starts with the initial global state of the system, and tries at every step to enlarge its current set of reachable states by propagating these states through transitions. The process terminates when a stable set is reached.

In order to use the above state-space exploration paradigm for verifying properties of systems with infinite state spaces, two basic problems need to be solved: one needs a representation for infinite sets of states, as well as a search technique that can explore an infinite number of states in a finite amount of time.

In the context of the verification of communication protocols as defined in the previous section, our solution to the first problem is to represent the control part explicitly and the queue-contents "symbolically". Specifically, we will use special data structures for representing (possibly infinite) sets of queue-contents associated with reachable control states.

To solve the second problem, we will use these data structures for simultaneously exploring (possibly infinite) sets of global states rather than individual global states. This may make it possible to reach a stable representation of the set of reachable global states, even if this set is infinite. In order to simultaneously generate sets of reachable states from a single reachable state, *meta-transitions* [BW94] can be used. Given a loop that appears in the protocol description and a control state c in that loop, a meta-transition is a transition that generates all global states that can be reached after repeated executions of the body of the loop. By definition, all these global states have the same control state c.

The classical enumerative state-space exploration algorithm can then be rewritten in such a way that it works with sets of global states, i.e., pairs of the form (control state, data structure), rather than with individual states. Initially, the search starts from an initial global state. At each step during the search, whenever meta-transitions are executable, they are explored first, which is a heuristic aimed at generating many reachable states as quickly as possible. This is why we call such a search a *loop-first search*. The search terminates if the representation of the set of reachable states stabilizes. This happens when, for every control state, every new deducible queue-content is *included* in the current set of queue-contents associated with that control state. At this moment, the final set of pairs (control state, data structure) represents *exactly* the state space of the protocol being analyzed.

In order to apply the verification method described above, we need to define a data structure for representing (possibly infinite) sets of queue-contents, and algorithms for manipulating these data structures. Specifically, whenever a transition or a meta-transition is executed from a pair (control state, data structure) during a loop-first search, the new pair (control state, data structure) obtained after the execution of this (meta-)transition has to be determined. Therefore, from any given such data structure, one needs to be able to compute a new data structure representing the effect of sending messages to a queue $(q_i!w)$ and receiving messages from a queue $(q_i?w)$, as well as the result of executing frequent types of meta-transitions, such as repeatedly sending messages on a queue $((q_i!w)^*)$, repeatedly receiving messages from a queue $((q_i?w)^*)$, and repeatedly receiving the

sequence of messages w_1 from a queue q_i followed by sending another sequence of messages w_2 on another queue q_j, $i \neq j$, $((q_i?w_1; q_j!w_2)^*)$. Finally, basic operations on sets are also needed, such as checking if a set of queue-contents is included in another set, and computing the union of two sets of queue-contents.

4 Queue-content Decision Diagrams

Queue-content Decision Diagrams (QDDs) are data structures that satisfy all the constraints listed in the previous section. A QDD is a special type of finite-state automaton on finite words. A finite-state automaton on finite words is a tuple $A = (\Sigma, S, \Delta, s_0, F)$, where Σ is an alphabet (finite set of symbols), S is a finite set of states, $\Delta \subseteq S \times (\Sigma \cup \{\varepsilon\}) \times S$ is a transition relation (ε denotes the empty word), $s_0 \in S$ is the initial state, and $F \subseteq S$ is a set of accepting states. A transition (s, a, s') is said to be *labeled* by a. A finite sequence (word) $w = a_1 a_2 \ldots a_n$ of symbols in Σ is *accepted* by the automaton A if there exists a sequence of states $\sigma = s_0 \ldots s_n$ such that $\forall 1 \leq i \leq n : (s_{i-1}, a_i, s_i) \in \Delta$, and $s_n \in F$. The set of words accepted by A is called the *language accepted by* A, and is denoted by $L(A)$. Let us define the *projection* $w|_{M_i}$ of a word w on a set M_i as the subsequence of w obtained by removing all symbols in w that are not in M_i. An automaton is said to be *deterministic* if it does not contain any transition labeled by the empty word, and if for each state, all the outgoing transitions are labeled by different symbols.

Precisely, QDDs are defined as follows.

Definition 2 A QDD A for a protocol P is a deterministic finite-state automaton (M, S, Δ, s_0, F) on finite words such that

$$\forall w \in L(A) : w = w|_{M_1} w|_{M_2} \ldots w|_{M_n}.$$

■

A QDD is associated with each control state reached during a loop-first search, and represents a set of possible queue-contents for this control state. Each word w accepted by a QDD defines one queue-content $w|_{M_i}$ for each queue q_i in the protocol.

By Definition 2, a total order $<$ is implicitly defined on the set Q of all queues q_i in the protocol such that, for all QDDs for this protocol, transitions labeled by messages in M_i always appear before transitions labeled by messages in M_j if $i < j$. Therefore, for all QDDs for a protocol, a given queue-content can only be represented by one unique word. In other words, Definition 2 implicitly defines a "canonical" representation for each possible queue-content. Note that this does not imply that QDDs are canonical representations for sets of queue-contents.

5 Operations on QDDs

Standard algorithms on finite-state automata on finite words can be used for checking if the language accepted by a QDD is included in the language accepted by another QDD, for computing the union of QDDs, etc. (e.g., see [LP81]). In what follows, $A_1 \cup A_2$ will denote an automaton that accepts the language $L(A_1) \cup L(A_2)$, while DETERMINIZE(A) will denote a deterministic automaton that accepts the language $L(A)$. We will write "Add (s, w, s') to Δ" to mean that transitions (s_{i-1}, a_i, s_i), $1 \leq i \leq n$, such that $w = a_1 a_2 \ldots a_n$, $s_0 = s$, $s_n = s'$, and $s_i, 1 \leq i < n$, are new (fresh) states, are added to Δ.

We now describe how to perform the other basic operations on QDDs listed in Section 3.

Let A be the QDD associated with a given control state c. Let $L(A)$ denote the language accepted by A, and let $L_{op}(A)$ denote the language that has to be associated with the control state c' reached

SEND(queue-id i, word w, QDD (M, S, Δ, s_0, F)) {

For all states $s \in S$ such that

$\exists w' \in (\cup_{j=1}^{i} M_j)^* : s_0 \stackrel{w'}{\Rightarrow} s$,

do the following operations:

- Add a new state s' to S;
- For all transitions $t = (s, m, s'') \in \Delta$ such that $m \in M_j, j > i$:
  Replace t by (s', m, s'');
- For all transitions $t = (s'', m, s) \in \Delta$ such that $m \in M_j, j > i$:
  Replace t by (s'', m, s');
- Add (s, w, s') to Δ;
- If $s \in F$, add s' to F, and remove s from F;

Return DETERMINIZE((M, S, Δ, s_0, F)).

}

RECEIVE(queue-id i, word w, QDD (M, S, Δ, s_0, F)) {

For all states $s \in S$ such that

$\exists w' \in (\cup_{j=1}^{i-1} M_j)^* : s_0 \stackrel{w'}{\Rightarrow} s$,

do the following operations:

- Add a new state s' to S;
- For all transitions $t = (s, m, s'') \in \Delta$ such that $m \in M_j, j \geq i$:
  Replace t by (s', m, s'');
- For all transitions $t = (s'', m, s) \in \Delta$ such that $m \in M_j, j \geq i$:
  Replace t by (s'', m, s');
- For all states $s'' \in S$ such that $s' \stackrel{w}{\Rightarrow} s''$:
  Add a transition (s, ϵ, s'') to Δ;
- If $s \in F$, add s' to F, and remove s from F;

Return DETERMINIZE((M, S, Δ, s_0, F)).

}

Figure 2: $q_i!w$ and $q_i?w$

after the execution of a transition (c, op, c') from the control state c, with $op \in \{q_i!w, q_i?w\}$. We have the following:

- $L_{q_i!w}(A) = \{w'' | \exists w' \in L(A) : w''|_{M_i} = w'|_{M_i} w \wedge \forall j \neq i : w''|_{M_j} = w'|_{M_j}\}$,
- $L_{q_i?w}(A) = \{w'' | \exists w' \in L(A) : w'|_{M_i} = ww''|_{M_i} \wedge \forall j \neq i : w''|_{M_j} = w'|_{M_j}\}$.

Algorithms for computing a QDD A' that accepts all possible queue-contents obtained after the execution of a transition of the form $q_i!w$ or $q_i?w$ on a QDD $A = (M, S, \Delta, s_0, F)$ are given in Figure 2. The correctness of these algorithms is established by the following two theorems.

Theorem 3 *Let A be a QDD, let A' denote the automaton returned by SEND(i, w, A), and let $L(A')$ denote the language accepted by A'. Then A' is a QDD such that $L(A') = L_{q_i!w}(A)$.*

Proof Proofs are omitted here due to space limitations. See the full paper. ■

Theorem 4 *Let A be a QDD, let A' denote the automaton returned by RECEIVE(i, w, A), and let $L(A')$ denote the language accepted by A'. Then A' is a QDD such that $L(A') = L_{q_i?w}(A)$.*

SEND-STAR(queue_id i, word w, QDD (M, S, Δ, s_0, F)) {

For all states $s \in S$ such that

$\exists w' \in (\cup_{j=1}^{i} M_j)^* : s_0 \stackrel{w'}{\Rightarrow} s$,

do the following operations:

- Add two new states s' and s'' to S;
- For all transitions $t = (s, m, s''') \in \Delta$ such that $m \in M_j, j > i$:
  Replace t by (s'', m, s''');
- For all transitions $t = (s''', m, s) \in \Delta$ such that $m \in M_j, j > i$:
  Replace t by (s''', m, s'');
- Add (s, ε, s'), (s', ε, s'') and (s', w, s') to Δ;
- If $s \in F$, add s'' to F;

Return DETERMINIZE((M, S, Δ, s_0, F)).

}

RECEIVE-STAR(queue_id i, word w, QDD (M, S, Δ, s_0, F)) {

For all states $s \in S$ such that

$\exists w' \in (\cup_{j=1}^{i-1} M_j)^* : s_0 \stackrel{w'}{\Rightarrow} s$,

do the following operations:

- Add a new state s' to S;
- For all transitions $t = (s, m, s'') \in \Delta$ such that $m \in M_j, j \geq i$:
  Replace t by (s', m, s'');
- For all transitions $t = (s'', m, s) \in \Delta$ such that $m \in M_j, j \geq i$:
  Replace t by (s'', m, s');
- For all states $s'' \in S$ such that $\exists w' \in \{w\}^* : s' \stackrel{w'}{\Rightarrow} s''$:
  Add a transition (s, ε, s'') to Δ;
- If $s \in F$, add s' to F;

Return DETERMINIZE((M, S, Δ, s_0, F)).

}

Figure 3: $(q_i!w)^*$ and $(q_i?w)^*$

Proof See the full paper. ■

We now consider the meta-transitions discussed in Section 3. The operation $(q_i!w)^*$ denotes the union of all possible queue-contents obtained after sending k sequences of messages $w \in M_i^*$ to the queue q_i of the system, for all $k \geq 0$. The operation $(q_i?w)^*$ denotes the union of all possible queue-contents obtained after receiving k sequences of messages $w \in M_i^*$ from the queue q_i of the system, for all $k \geq 0$. The operation $(q_i?w_1; q_j!w_2)^*$ denotes the union of all possible queue-contents obtained after receiving k sequences of messages $w_1 \in M_i^*$ from the queue q_i and sending k sequences of messages $w_2 \in M_j^*$ to the queue q_j, for all $k \geq 0$, and for $i \neq j$.

Let A be the QDD associated with a given control state c. Let $L(A)$ denote the language accepted by A, and let $L_{op}(A)$ denote the language that has to be associated with the control state c reached after the execution of a meta-transition (c, op, c) with $op \in \{(q_i!w)^*, (q_i?w)^*, (q_i?w_1; q_j!w_2)^*\}$. We have the following:

- $L_{(q_i!w)^*}(A) = \{w'' | \exists w' \in L(A), k \geq 0 : w''|_{M_i} = w'|_{M_i} w^k \wedge \forall j \neq i : w''|_{M_j} = w'|_{M_j}\}$,
- $L_{(q_i?w)^*}(A) = \{w'' | \exists w' \in L(A), k \geq 0 : w'|_{M_i} = w^k w''|_{M_i} \wedge \forall j \neq i : w''|_{M_j} = w'|_{M_j}\}$,

RECEIVE-SEND-STAR(queue_id i, word w_1, queue_id j, word w_2, QDD (M, S, Δ, s_0, F)) {

Let n be the greatest integer such that

$$\exists s_1, \ldots s_{n+1} \in S : s_1 \stackrel{w_1}{\Rightarrow} s_2 \stackrel{w_1}{\Rightarrow} \cdots \stackrel{w_1}{\Rightarrow} s_{n+1},$$

with $\forall 1 \leq k < l \leq n+1 : s_k \neq s_l$;

Let A_0 denote the QDD (M, S, Δ, s_0, F);

For all k, $1 \leq k \leq n+1$, compute $A_k = \text{SEND}(j, w_2, \text{RECEIVE}(i, w_1, A_{k-1}))$;

If $L(A_{n+1}) = \emptyset$:

- Return DETERMINIZE$(\cup_{k=0}^{n} A_k)$;

If $L(A_{n+1}) \neq \emptyset$:

- Let $p = 1$;
- While $L(A_{n+1}) \neq L(\text{RECEIVE}(i, w_1^p, A_{n+1}))$:
  $p := p + 1$;
- For all k, $2 \leq k \leq p$, compute $A_{n+k} = \text{SEND}(j, w_2, \text{RECEIVE}(i, w_1, A_{n+k-1}))$;
- Compute $A_{n+p+1} = \text{SEND-STAR}(j, w_2^p, \text{DETERMINIZE}(\cup_{k=n+1}^{n+p} A_k))$;
- Return DETERMINIZE$(\cup_{k=0}^{n+p+1} A_k)$.

}

Figure 4: $(q_i?w_1; q_j!w_2)^*$

- $L_{(q_i?w_1;q_j!w_2)^*}(A) = \{w'' | \exists w' \in L(A), k \geq 0 : w'|_{M_i} = w_1^k w''|_{M_i} \wedge w''|_{M_j} = w'|_{M_j} w_2^k \wedge \forall l \notin \{i, j\} : w''|_{M_l} = w'|_{M_l}\}$.

Algorithms for computing a QDD A' that accepts all possible queue-contents obtained after the execution of a meta-transition of the form $(q_i!w)^*$, $(q_i?w)^*$, or $(q_i?w_1; q_j!w_2)^*$ on a QDD $A = (M, S, \Delta, s_0, F)$ are given in Figures 3 and 4. The correctness of these algorithms is established by the following theorems.

Theorem 5 *Let A be a QDD, let A' denote the automaton returned by SEND-STAR(i, w, A), and let $L(A')$ denote the language accepted by A'. Then A' is a QDD such that $L(A') = L_{(q_i!w)^*}(A)$.*

Proof See the full paper. ■

Theorem 6 *Let A be a QDD, let A' denote the automaton returned by RECEIVE-STAR(i, w, A), and let $L(A')$ denote the language accepted by A'. Then A' is a QDD such that $L(A') = L_{(q_i?w)^*}(A)$.*

Proof See the full paper. ■

Lemma 7 *Let n and A_{n+1} be as defined in the algorithm RECEIVE-SEND-STAR(i, w_1, j, w_2, A), with $i \neq j$. If the language accepted by A_{n+1} is not empty, then there exists p such that $0 < p \leq (n+1)!$, and $L(A_{n+1}) = L(RECEIVE(i, w_1^p, A_{n+1}))$.*

Proof See the full paper. ■

Theorem 8 *Let A be a QDD, let A' denote the automaton returned by RECEIVE-SEND-STAR(i, w_1, j, w_2, A), , with $i \neq j$, and let $L(A')$ denote the language accepted by A'. Then A' is a QDD such that $L(A') = L_{(q_i?w_1;q_j!w_2)^*}(A)$.*

Proof See the full paper. ■

It is worth noticing that, as a corollary of the last theorem, the language $L_{(q_i?w_1;q_j!w_2)^*}(A)$ is *regular*.

6 Experimental Results

Consider again the Alternating-Bit protocol of Example 1. Meta-transitions are added to the protocol description for loops that match either $(q_i!w)^*$, $(q_i?w)^*$, or $(q_i?w_1; q_j!w_2)^*$. Precisely, the meta-transitions $(3, (RtoS?ack1; StoR!msg0)^*, 3)$, $(3, (StoR!msg0)^*, 3)$, $(8, (RtoS?ack0; StoR!msg1)^*, 8)$, $(8, (StoR!msg1)^*, 8)$ are added to the set of transitions of the *Sender*, while the meta-transitions $(1, (StoR?msg1; RtoS!ack1)^*, 1)$ and $(5, (StoR?msg0; RtoS!ack0)^*, 5)$ are added to the set of transitions of the *Receiver*.

We have implemented (in C) a "QDD-package" containing an implementation of the algorithms for manipulating QDDs described in the previous section, and we have combined it with a loop-first search. Starting with the control state $(1, 1)$ and the QDD $(M, \{s_0\}, \{\}, s_0, \{s_0\})$, which corresponds to the queue-content ε for both queues *StoR* and *RtoS*, the execution of the loop-first search for the Alternating-Bit protocol terminates after 5.9 seconds of computation on a SPARC10 workstation. The number of (meta-)transitions executed is 331. The largest QDD constructed during the search contains 21 states, and 52 control states are reachable from the initial state.

Many properties can be checked on the symbolic representation of the state space of the protocol obtained at the end of the search. For instance, it is then straightforward to *prove* that the protocol does not contain any deadlocks, that there are reachable control states where the number of messages in a queue is unbounded, that messages are always delivered in the correct order, etc.

Our tool has also been tested on several variants of the Alternating-Bit protocol, where the transitions labeled by "timeout" are removed from the protocol description, where the *Sender/Receiver* have various number of control states, etc. An interesting variant is the case where queues may lose messages (to model unreliable transmission media). In order to handle this case, it is sufficient to define one additional algorithm SEND-LOSSY(i, w, A), that merely returns $A \cup$ SEND(i, w, A). We also performed experiments on several simple sliding-window protocols [Tan89], with various window sizes. For *all* these examples with infinite state spaces (more than 20 in total), our tool was able to successfully terminate its search within a few minutes of computation. This shows that, at least for this particular though important class of examples, our verification method is very useful and robust.

7 Comparison with Other Work and Conclusions

Although most verification problems are undecidable for arbitrary protocols modeled by communicating finite-state machines, decision procedures have been obtained for the verification of specific properties for limited sub-classes [KM69, RY86, GGLR87, CF87, Fin88, Jer91, SZ91, AJ93, AJ94, CFP96]. These sub-classes do not cover, e.g., the Alternating-Bit Protocol and the properties discussed in the previous section, which were easily verified using a loop-first search and QDDs.

Clearly, a necessary, but not sufficient, condition for the termination of our algorithm is that, for all reachable control states of the protocol, the language of queue-contents associated with that control state can be represented by a QDD. The class of protocols characterized by the above necessary condition is equivalent to the class of protocols for which, for each reachable control state of the protocol, the set of possible queue-contents can be described by a recognizable expression (i.e., a finite union of cartesian products of regular expressions). Indeed, it can be shown that any recognizable language can be represented by a QDD, and that any set of queue-contents represented by a QDD is a recognizable language.

In [Pac87], it is pointed out that several verification problems are decidable for the above class of protocols. However, no method is given for constructing a recognizable expression representing all possible queue-contents for each control state of the protocol. Actually, from [CFP96], it is easy to show that an algorithm for constructing such recognizable expressions, for *any* protocol in the class

defined above, cannot exist. In contrast, our contribution is to provide a practical algorithm which is able to compute such a representation for protocols in the above class, although not for all of them – this is impossible anyway.

In this paper, we have presented algorithms on QDDs for computing the effect of executing three frequent types of meta-transitions. These algorithms were sufficient for analyzing the protocols considered in the previous section. However, it is possible to design algorithms on QDDs for other types of meta-transitions as well. Interesting future work is to characterize precisely the set of meta-transitions that preserve recognizability and to provide a generic algorithm for computing the effect of the execution of any meta-transition in this class. These topics will be addressed in a forthcoming paper.

In [PP91], a verification method based on data-flow analysis is used to generate "flow equations" from the description of a set of communicating finite-state machines. By computing approximations of solutions for these equations, it is possible to show that the original system is free of certain types of errors. In contrast, our algorithm is able to produce an *exact* representation of the state space of the protocol being analyzed. This enables us not only to prove the absence of errors, but also to detect errors and to exhibit to the user sequences of transitions that lead to errors. Note that, obviously, approximations could also be used in our framework, e.g., for simplifying QDDs when they become too complex, or when the search does not seem to stop. For the examples we have considered so far, no approximations were necessary.

The idea of representing states partly explicitly (control part) and partly symbolically (data part) already appeared in [ACD93] for the verification of real-time systems, where dense-time domains are represented by polyhedra. This idea also appeared in [BW94], where the values of integer variables are represented by periodic vector sets. These symbolic representations are quite different from QDDs.

For digital hardware verification [BCM+90], the most commonly used symbolic representation is certainly the Binary Decision Diagram (BDD) [Bry92], which represents a boolean function (with a *finite* domain) as a directed acyclic graph. In [GL96], it is shown how QDDs can be combined with BDDs to improve the efficiency of classical BDD-based symbolic model-checking methods for verifying properties of communication protocols with large *finite* state spaces.

8 Acknowledgments

We wish to thank Michael Merritt and Mark Staskauskas for helpful comments on a preliminary version of this paper.

References

[ACD93] R. Alur, C. Courcoubetis, and D. Dill. Model-checking in dense real-time. *Information and Computation*, 104(1):2–34, May 1993.

[AJ93] P. A. Abdulla and B. Jonsson. Verifying programs with unreliable channels. In *Proceedings of the 8th IEEE Symposium on Logic in Computer Science*, 1993.

[AJ94] P. A. Abdulla and B. Jonsson. Undecidable verification problems for programs with unreliable channels. In *Proc. ICALP-94*, volume 820 of *Lecture Notes in Computer Science*, pages 316–327. Springer-Verlag, 1994.

[BCM+90] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *Proceedings of the 5th Symposium on Logic in Computer Science*, pages 428–439, Philadelphia, June 1990.

[Bry92] R.E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318, 1992.

[BSW69] K. Bartlett, R. Scantlebury, and P. Wilkinson. A note on reliable full-duplex transmissions over half-duplex lines. *Communications of the ACM*, 2(5):260–261, 1969.

[BW94] B. Boigelot and P. Wolper. Symbolic verification with periodic sets. In *Proc. 6th Conference on Computer Aided Verification*, volume 818 of *Lecture Notes in Computer Science*, pages 55–67, Stanford, June 1994. Springer-Verlag.

[BZ83] D. Brand and P. Zafiropulo. On communicating finite-state machines. *Journal of the ACM*, 2(5):323–342, 1983.

[CES86] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, January 1986.

[CF87] A. Choquet and A. Finkel. Simulation of linear FIFO nets having a structured set of terminal markings. In *Proc. 8th European Workshop on Application and Theory of Petri Nets*, pages 95–112, Saragoza, 1987.

[CFP96] G. Cécé, A. Finkel, and S. Purushothaman. Unreliable channels are easier to verify than perfect channels. *Information and Computation*, 124(3):20–31, 1996.

[Fin88] A. Finkel. A new class of analyzable cfsms with unbounded FIFO channels. In *Proc. 8th IFIP WG 6.1 International Symposium on Protocol Specification, Testing, and Verification*, pages 1–12, Atlantic City, 1988. North-Holland.

[GGLR87] M. G. Gouda, E. M. Gurari, T. H. Lai, and L. E. Rosier. On deadlock detection in systems of communicating finite-state machines. *Computers and Artificial Intelligence*, 6(3):209–228, 1987.

[GL96] P. Godefroid and D. E. Long. Symbolic Protocol Verification with Queue BDDs. In *Proceedings of the 11th IEEE Symposium on Logic in Computer Science*, New Brunswick, July 1996.

[Jer91] T. Jeron. Testing for unboundedness of FIFO channels. In *Proc. STACS-91: Symposium on Theoretical Aspects of Computer Science*, volume 480 of *Lecture Notes in Computer Science*, pages 322–333, Hamburg, 1991. Springer-Verlag.

[KM69] R. M. Karp and R. E. Miller. Parallel program schemata. *Journal of Computer and System Sciences*, 3(2):147–195, 1969.

[Liu89] M.T. Liu. Protocol engineering. *Advances in Computing*, 29:79–195. 1989.

[LP81] H. R. Lewis and C. H. Papadimitriou. *Elements of the Theory of Computation.* Prentice Hall, 1981.

[LP85] O. Lichtenstein and A. Pnueli. Checking that finite state concurrent programs satisfy their linear specification. In *Proceedings of the Twelfth ACM Symposium on Principles of Programming Languages*, pages 97–107, New Orleans, January 1985.

[Pac87] J. K. Pachl. Protocol description and analysis based on a state transition model with channel expressions. In *Proc. 7th IFIP WG 6.1 International Symposium on Protocol Specification, Testing, and Verification.* North-Holland, 1987.

[PP91] W. Peng and S. Purushothaman. Data flow analysis of communicating finite state machines. *ACM Transactions on Programming Languages and Systems*, 13(3):399–442, 1991.

[QS81] J.P. Quielle and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *Proc. 5th Int'l Symp. on Programming*, volume 137 of *Lecture Notes in Computer Science*, pages 337–351. Springer-Verlag, 1981.

[Rud87] H. Rudin. Network protocols and tools to help produce them. *Annual Review of Computer Science*, 2:291–316, 1987.

[RY86] L. E. Royer and H. C. Yen. Boundedness, empty channel detection and synchronization for communicating finite automata. *Theoretical Computer Science*, 44:69–105, 1986.

[SZ91] A. P. Sistla and L. D. Zuck. Automatic temporal verification of buffer systems. In *Proc. 3rd Workshop on Computer Aided Verification*, volume 575 of *Lecture Notes in Computer Science*, pages 93–103, Aalborg, July 1991. Springer-Verlag.

[Tan89] A. Tanenbaum. *Computer Neworks*. Prentice Hall, 1989.

[VW86] M.Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proceedings of the First Symposium on Logic in Computer Science*, pages 322–331, Cambridge, June 1986.

A Conjunctively Decomposed Boolean Representation for Symbolic Model Checking

K. L. McMillan

Cadence Berkeley Labs
1919 Addison St., suite 303
Berkeley, CA 94704-1144
mcmillan@cadence.com

Abstract. A canonical boolean representation is proposed, which decomposes a function into the conjunction of a sequence of components, based on a fixed variable order. The components can be represented in OBDD form. Algorithms for boolean operations and quantification are presented allowing the representation to be used for symbolic model checking. The decomposed form has a number of useful properties that OBDD's lack. For example, the size of conjunction of two independent functions is the sum of the sizes of the functions. The representation also factors out dependent variables, in the sense that a variable that is determined by the previous variables in the variable order appears in only one component of the decomposition. An example of verifying equivalence of sequential circuits is used to show the potential advantage of the decomposed representation over OBDD's.

1 Introduction

Symbolic model checking, and related finite-state verification techniques use heuristically compact boolean representations, such as ordered binary decision diagrams (OBDD's), to implicitly represent sets and relations (notably the transition relation of a model, and its set of reachable states). The implicit representation may be compact even thought the number of states or transitions is very large, thus allowing systems with very large state spaces to be verified automatically. However, in many cases the OBDD representation is not compact. To a first approximation, the OBDD representing a set of states can be thought of as a finite state automaton that reads the values of the state variables in some fixed order, and finally accepts or rejects the given valuation. Figuratively speaking, this automaton must "remember" some amount of information about the variables seen so far, in order to decide whether the remaining variable assignments are consistent with those already seen. Hence, to obtain a compact representation, the variable order must be such that the mutual information across any cut through the order is small. This implies that state variables that strongly correlate each other must be nearby in the variable order. Often, however, this is not possible. For example, in a protocol, a state variable representing the contents of a message buffer is likely to be correlated with both the state of the sender and

the state of the receiver. Since all pairs of senders and receivers cannot generally be made close in the variable order, there is no suitable place for the variable representing the message buffer.

In other cases, the relationships between variables are not fixed, but vary according to some global control information. For example, suppose we wish to verify that two hardware implementations of a bounded FIFO queue are equivalent (see figure 1). This can be done by building a single model in which the two implementations run in parallel, and verifying that the outputs always agree. Let one implementation be a "shift register", in which the most recent item is always stored in location 0, and all items shift over when a new item is inserted. Let the other implementation be a "ring buffer", where a "head pointer" points to the oldest item, and the items themselves remain fixed. Given the state of the head pointer, there is a one-to-one correspondence between locations in the two implementations. However, since the head pointer is not fixed, we cannot fix an OBDD variable order that will put related state variables together.

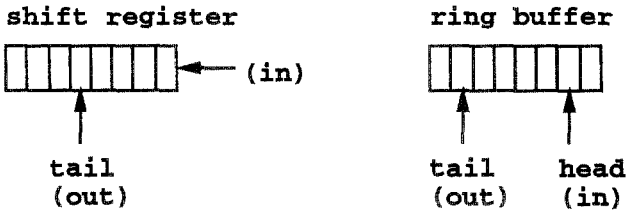


Fig. 1. Two implementations of FIFO queue.

This paper introduces a canonical boolean representation that may be compact in such cases, where OBDD's are not. The intuition behind this representation is that many state variables, such as the message buffers in a protocol, or the data items in the ring buffer, have a property of "conditional independence". That is, once a core set of state variables is fixed, the remaining variables are not mutually correlated. For example, once we the contents of the shift register and the head pointer are fixed, the contents of the ring buffer are determined, and hence uncorrelated. Similarly, the contents of the message buffers in a protocol may provide no mutual information, once the states of the communicating processes are fixed. The representation introduced here decomposes the representation of a boolean function into the conjunction of sequence of components. Each component, which may be represented as an OBDD, fixes the possible values of just one state variable, *given* a feasible assignment to the previous variables. The variable order used for this decomposition may be distinct from the OBDD variable order. The decomposed form has the property that conditionally independent variables are "factored out" into separate components, thus eliminating the need to find a suitable place for these variables in a global OBDD variable order. Among other things, this implies that the conjunction of

functions with independent support is additive (not true for OBDD's), as is the conjunction of components in the transition relation of a state machine.

There are a number of examples of conjunctive forms in the literature. Hu and Dill use a technique of checking conjunctive properties where fixed points are computed in parallel, and each conjunct is used to simplify the other conjuncts at each iteration [HD93]. This can yield a more compact representation than an explicit conjunction, but the original decomposition of the problem must be provided by the user. Also, the representation is not canonical, as it is here. Burch and Long use implicitly conjoined transition relations, but do not decompose the representation of the set of reached states, as we do here [BCL91]. Their representation is also not canonical. Jain also describes a canonical disjunctive representation [JABF92], which has a conjunctive dual. It is not directly related to the current method, however, as is it obtained by dividing the truth table into *ad hoc* regions and using one OBDD for each region.

This paper is organized as follows: Section 2 defines the decomposed representation, and proves some useful theorems about the size of the representation for certain classes of functions. Section 3 introduces algorithms for conjunction, disjunction and existential quantification (projection) on the decomposed form. Section 4 discusses model checking using the above algorithms, and provides performance results for verifying the equivalence of the two FIFO queue implementations mentioned above. We find that as the depth of the queues is increased, the size of the decomposed representation for the reachable states increases quadradically, while the OBDD representation increases exponentially.

In this paper, most of the proofs have been omitted due to space limitations.

2 Conjunctive decompositions

Let f be a boolean function of independent boolean variables $V = (v_1, \ldots, v_n)$. We will use the notation $f^{(i)}$, where $0 \leq i \leq n$, to stand for the projection of f onto $(v_1, \ldots, v_i)$. That is,

$$f^{(i)} = \exists(v_{i+1}, \ldots, v_n).\ f$$

In addition, we will use the notation $f|g$, where f and g are two boolean functions, denote the "generalized cofactor" of f relative to g. This function, which can be read as "f given g", agrees with f whenever g is true [CBM89, TSL+90]. Those values where g is false are mapped to the "nearest" point where g is true, according to a distance measure on truth assignments. Thus, if two functions agree wherever g holds, then their cofactors relative to g are equal:

$$f \wedge g = f' \wedge g \quad \text{iff} \quad f|g = f'|g$$

We will use the projection and cofactor operations to decompose a boolean function f into a vector of boolean functions $(f_1, \ldots, f_n)$, where

$$f_i = f^{(i)}|f^{(i-1)}$$

Intuitively, the component f_i determines the set of possible values of variable v_i, *given* a feasible evaluation of the variables $(v_1, \ldots, v_{i-1})$. We will show that the function f is equal to the conjunction of the components f_i:

$$f = \bigwedge_{i=1}^{n} f_i$$

2.1 Generalized cofactor

If f and g are two boolean functions of boolean variables $(v_1, \ldots, v_{i-1})$, then $f|g$ is a boolean function whose value is obtained for a given truth assignment x by finding the "nearest" truth assignment to x that satisfies g, and evaluating f at this point. For this purpose, the distance between two truth assignments x and y is determined by treating their boolean difference (exclusive-or) as a binary number. To be more precise,

Definition 1. Let A be the set of truth assignments $V \rightarrow \{0,1\}$, and let $W = (w_1, \ldots, w_n)$ be a permutation of V. For any $x, y \in A$, let

$$d(x,y) = \Sigma_{i=1}^{n} 2^{n-i}(x(w_i) \oplus y(w_i))$$

Notice that we have an arbitrary choice of the order W on the variables that defines the distance between truth assignments. Also note for future reference that we have weighted the variables so that w_1 is the most significant, and w_n is the least significant.

Definition 2. Let B be the boolean algebra 2^A. For any $x \in A$, and $g \in B$, where $g \neq 0$, let $x \rightarrow g$ be the unique $y \in g$ minimizing $d(x,y)$.

That is, $x \rightarrow g$ is the nearest point to x that satisfies g. Note that $x \rightarrow g$ is uniquely defined for $g \neq 0$, because the boolean difference between x and any other truth assignment is a unique number. This lets us define the generalized cofactor as follows:

Definition 3. For any $f, g \in B$, and any $x \in A$:

- if $g \neq 0$, then $(f|g)(x) = f(x \rightarrow g)$,
- else $f|g = 0$.

As an example, suppose that $W = (v_1, v_2, v_3)$, that $f = v_3$, $g = (\neg v_1) \wedge (v_2 \vee v_3)$, and that we want to evaluate $f|g$ at the truth assignment $x = (1,0,0)$. The truth assignments satisfying g are $(0,0,1)$, $(0,1,0)$ and $(0,1,1)$, of which the nearest to x is $y = (0,0,1)$, yielding a distance of $d(x,y) = 5$. The value of $(f|g)(x)$ is thus $f(y) = 1$.

We note that generalized cofactor, as defined above, is exactly the "constrain" operator on OBDD's [CBM89] in the special case when the OBDD variable order is W. In the sequel, however, we will not assume that this is the case.

2.2 Properties of generalized cofactor

We will rely on a variety of properties of the generalized cofactor operation in defining the conjunctive decomposition and in constructing algorithms on decompositions. One very important property of $f|g$ is that it agrees with f everywhere that g is true. Another is that, if two function f and f' agree wherever g is true, then $f|g = f'|g$. That is, $f|g$ is independent of the value of f anywhere that g is false. Letting juxtaposition denote conjunction, and | associate to the left, we also have:

Theorem 4. *1.* $fg = f'g$ iff $f|g = f'|g$.
2. $(f|g)g = fg$
3. if $g \neq 0$, *then* $g|g = 1$
4. $f|g|g = f|g$

We also note that generalized cofactor distributes over pointwise operators:

Theorem 5. *For any operator* $\cdot$, *such that* $(f \cdot g)(x) = f(x) \cdot g(x)$:

$$(f \cdot g)|h = (f|h) \cdot (g|h)$$

The following theorem is key to the algorithms on conjunctive decompositions, since it allows us, in certain cases, to cofactor relative to a conjunction of functions without explicitly forming the conjunction:

Theorem 6. *For any* $f, g, h \in B$, *if* $g = g|h$, *then* $f|(gh) = f|h|g$.

The name "generalized cofactor" derives from the following property [TSL+90]:

Theorem 7 Touati, et al.. *For any* $f \in B$ *and* $v_i \in V$,

- $f|v_i = f|_{v_i=1}$
- $f|\bar{v_i} = f|_{v_i=0}$

We say that a function f depends on v_i when $f|_{v_i=0} \neq f|_{v_i=1}$. The support of a function is the set of variables on which it depends. Two functions are said to be independent when their supports are disjoint. When two functions are independent, then cofactoring one by the other has no effect:

Theorem 8. *If* f *and* g *have independent support, then* $f|g = f$.

An immediate corollary of this result and theorem 6 is that cofactoring with respect to two independent functions can be done in either order, without affecting the result:

Corollary 9. *If* g *and* h *have independent support, then* $f|(gh) = f|g|h = f|h|g$.

In addition, we can show that projection distributes over cofactor in the much the same way it distributes over conjunction:

Corollary 10. *If* g *is independent of* v_i, *then* $\exists v_i.(f|g) = (\exists v_i.f)|g$.

2.3 Definition of decomposition

We are now ready to define our canonical conjunctive decomposition of a boolean function:

Definition 11. For all $f \in B$, for all $1 \leq i < n$, let $f_i = f^{(i)}|f^{(i-1)}$

We will refer to the functions $(f_1, \ldots, f_n)$ as the *components* of f (relative to V and W). We now show that a function is equal to the conjunction of its components:

Theorem 12. $f = \wedge_{i=1}^{n} f_i$

Proof. We take as our inductive hypothesis that $f^{(j)} = \wedge_{i=1}^{j} f_i$, for all $1 \leq j \leq n$. For the case where f is identical to false, this clearly holds, since all the components f_i are also false. Otherwise, in the base case we have $f_1 = f^{(1)}|f^{(0)} = f^{(1)}|1 = f^{(1)}$. For the inductive step we have:

$$\begin{aligned} \wedge_{i=1}^{j} f_i &= (\wedge_{i=1}^{j-1} f_i) \wedge f_j && (1) \\ &= f^{(j-1)} \wedge (f^{(j)}|f^{(j-1)}) && (2) \\ &= f^{(j-1)} \wedge f^{(j)} && (3) \\ &= f^{(j)} && (4) \end{aligned}$$

Note equation 3 is a case of theorem 4, part 2. That is, $f^{(j)}|f^{(j-1)}$ agrees with $f^{(j)}$ where $f^{(j-1)}$ is true.

Theorem 12 implies that the vector $(f_1, \ldots, f_n)$ is a canonical representation of f, given a fixed V and W. That is, each function has exactly one decomposition, and no two functions have the same decomposition.

There are a number of useful facts about this representation, independent of the component representation and of the choice of permutation W, that defines the generalized cofactor operation. For example, if a function $f \neq 0$ does not depend on some variable v_i, then the corresponding component f_i is identical to true. That is, if f is independent of v_i, then $f^{(i)} = f^{(i-1)}$. Hence $f_i = f^{(i)}|f^{(i)} = 1$, by theorem 4. More generally, we can show that the the ith component of f constrains only variable v_i. That is:

Theorem 13. *If* $f \neq 0$, *then* $\exists v_i . f_i = 1$.

2.4 Decompositions and disjointness

If two functions f and g have disjoint support, then the components of their conjunction can be obtained by simply taking the conjunction of the corresponding components of f and g, regardless of V or W. Since disjointness implies that every component must be identically true in either f or g or both, it follows that the size of the conjunction is less than or equal to the sum of the sizes of f and g.

Theorem 14. *If* f *and* g *have independent support, then*

- $(fg)_i = f_i$ *when* f *depends on* v_i, *and*

- $(fg)_i = g_i$ *when* g *depends on* v_i, *and*
- *otherwise* $(fg)_i = 1$.

From the above, it follows immediately that the size of fg is bounded by the sum of the sizes of f and g. This result is independent of the underlying representation of the components.

Corollary 15. *If* f *and* g *have independent support, then*

$$\Sigma_i|(fg)_i| \leq \Sigma_i|f_i| + \Sigma_i|g_i|$$

It is worth noting that the OBDD representation [Bry86] has this property only in case the OBDD variable order separates the supports of f and g.

2.5 Decompositions and dependent variables

We now consider the special case where the permutation W is the identity (that is, the order of the components f_i is the same as the order that determines the distance measure for generalized cofactor). In this case, if a given variable v_i is functionally determined by its predecessors $v_1 \ldots v_{i-1}$ in the variable order, then we can show that variable v_i appears only in component f_i.

Definition 16. Given a function f, a variable v_i is *functionally determined* by a set of variables $S \subseteq V$ when any two truth assignments agreeing on S must also agree on v_i. If this condition holds, we write $f : S \rightarrow v_i$.

Theorem 17. *If* $W = (v_1, \ldots, v_n)$ *and* $f : (v_1, \ldots, v_{i-1}) \rightarrow v_i$, *then* f_j *depends on* v_i *only if* $j = i$.

The fact that the decomposed representation is capable of factoring out dependent variables is useful for verifying certain kinds of sequential circuits, as we will observe. It is also a heuristic argument for using $W = V$ in practice.

2.6 Decompositions and conditional independence

The following result generalizes the previous results on independent functions and dependent variables. We will say two variables are *conditionally independent*, relative to a function f, when fixing the value of the preceding variables in the order makes the choice of values of the two variables independent. For example, suppose the function f is $(a \Rightarrow b)(a \Rightarrow c)$. If we fix the value of a, then our choices for b and c become independent. Assuming that the variable order W is (a, b, c), it follows that b and c are conditionally independent. From this we can infer that b occurs only in component f_2, while c occurs only component f_3. That is, conditionally independent variables factor out in the decomposition. In general, we have the following result:

Theorem 18. *Let* $f, g \in B$, *such that* $f^{(i)} = g^{(i)}$, *and* f *and* g *have disjoint support over* $v_{i+1} \ldots v_n$. *Then*

- $(fg)_j = f_j$ *when* f *depends on* v_j, *and*

- $(fg)_j = g_j$ *when* g *depends on* v_j, *and*
- *otherwise* $(fg)_j = 1$.

Once again, the conjunction of f and g requires additive space. Note that the result for disjoint functions (theorem 14) is the special case where $i = 0$, while the fact that dependent variables factor out (theorem 17) is the special case where v_i is independent of later variables because its value is fixed.

3 Algorithms

To use our decomposed form as a representation for symbolic model checking, we need algorithms for computing boolean combinations and for existential quantification (projection) over boolean variables.

3.1 Logical conjunction

We begin with the algorithm for conjunction. It should be noted at the outset that in general it is not the case that $(fg)_i = f_i g_i$ (though this is true for the case when f and g are independent). In general, it may be the case that, though $f^{(i)}$ and $g^{(i)}$ are both true for a given assignment to $(v_1, \ldots, v_i)$, the assignments to the remaining variables that make them true may be different, and hence $(fg)_i$ may be false. Thus $(fg)_i$ may be stronger than $f_i g_i$.

To avoid this problem, we first compute appropriate approximations k_i to $(fg)_i$ for all i. These terms are computed by conjoining the terms $f_i g_i$ in descending sequence, projecting out v_i at each stage. This "early quantification" step is justified by the fact that the remaining terms in the descending sequence do not depend on v_i, and prevents computing an explicit conjunction of all the terms, which would defeat the purpose of a decomposed representation.

Next, we must "normalize" the representation by cofactoring each approximation to k_i by $(fg)^{(i-1)}$. Since we have no direct representation of the latter, we obtain the desired effect by cofactoring each k_i by the preceding components $(fg)_1 \ldots (fg)_{i-1}$ in sequence. This result derives from the following lemma:

Lemma 19. *For any functions* x *and* h,

$$x|(\wedge_{j=1}^{i} h_j) = x|h_1|h_2|\cdots|h_i$$

Proof.

$$\begin{aligned} x|(\wedge_{j=1}^{i} h_j) &= x|(h_i \wedge h^{(i-1)}) && (5)\\ &= x|h^{(i-1)}|h_i && (6)\\ &= x|(\wedge_{j=1}^{i-1} h_j)|h_i && (7) \end{aligned}$$

which by induction gives us the lemma. Equation 6 is a case of theorem 6, while equations 5 and 7 are by theorem 12.

We will state the conjunction algorithm formally in terms of a theorem:

Theorem 20. *Let* $h = fg$ *and let*

$$k_n = f_n g_n \tag{8}$$
$$k_{i-1} = f_{i-1} g_{i-1} \exists v_i . k_i \tag{9}$$

Then

$$h_i = k_i | h_1 | h_2 | \cdots | h_{i-1} \tag{10}$$

A conjunction operation on the decomposed representation involves $O(n)$ conjunction operations on the underlying representation, $O(n)$ one-variable projection operations, and $O(n^2)$ cofactor operations. The latter is unfortunate, but seems to be necessary in order to avoid explicit construction of the terms $(fg)^{(i)}$.

3.2 Logical disjunction

We now consider computing logical disjunction of two functions represented by their components. First, we should note that in general $(f \vee g)_i \neq f_i \vee g_i$. Consider, for example, computing $h = f \vee g$, where $f = \bar{a}\bar{b}\bar{c}\bar{d}\ldots$ and $g = abcd\ldots$. The components of these functions are, respectively, $f_1 = \bar{a}$, $f_2 = \bar{b}$, *etc.*, and $g_1 = a$, $g_2 = b$, *etc.* Thus, the disjunction of f_i and g_i is 1, for every i, which is clearly wrong. The problem here is that because we are forming a disjunction, h_i is "defined" over a potentially larger domain than f_i and g_i. To correct this problem, we would like to broaden the domains of f_i and g_i before taking the disjunction. That is, we would like to compute:

$$f'_i = f^{(i)} | h^{(i-1)}$$
$$g'_i = g^{(i)} | h^{(i-1)}$$

However, we would like to do this without explicitly computing $f^{(i)}$, $g^{(i)}$ and $h^{(i-1)}$. This leads us to the following algorithm:

Theorem 21. *Let* $h = f \vee g$ *and*

$$f'_1 = f_1 \qquad g'_1 = g_1$$
$$f'_{i+1} = f_{i+1}(f'_i | h_i) \qquad g'_{i+1} = g_{i+1}(g'_i | h_i)$$

Then $h_i = f'_i \vee g'_i$.

3.3 Projection

The approach to existential quantification over boolean variables is very similar to the disjunction algorithm. The algorithm is as follows:

Theorem 22. *Let* $h = \exists S.f$, *where* $S \subset V$, *and*

$$f'_1 = f_1 \tag{11}$$
$$f'_{i+1} = f_{i+1}(f'_i | h_i) \tag{12}$$

Then $h_i = \exists S.f'_i$.

The above algorithm is effective in practice for projecting out small numbers of variables. However, if we consider the limiting case, where $S = V$, we see that $h = 1$, and therefore $f'_n = f$, which clearly defeats the purpose of the decomposition. For projecting out a large number of variables, an effective strategy is to successively project out small groups of variables, in descending order. In this way, each step tends to simplify the problem for the next step.

3.4 Implementing the algorithms with OBDD's

Ordered binary decision diagrams (OBDD's) are a particularly effective representation for the components of a function because of the efficient algorithms for conjunction and disjunction [Bry86] and for generalized cofactor [CBM89]. The OBDD representation for a function is determined by a permutation $U = (u_1, \ldots, u_n)$ on the boolean variables. In the special case where $U = W$, there is a quadratic-time algorithm for generalized cofactor on OBDD's. In the case $U = V = W$, we can also show that the size of the decomposed representation of f is never larger than n times the size of the direct OBDD representation of f:

Theorem 23. *If* $U = V = W$, *then* $|f_i|_{OBDD} \leq |f|_{OBDD}$

4 Symbolic model checking and decompositions

In symbolic model checking, we use a boolean formula to represent the transition relation of a model, and we use fixed point iterations to evaluate formulas in certain modal logics relative to this model. The most important operation in these iterations is computing the image of some set of states, relative to the transition relation. The transition relation is represented by using a set of variables $v_1, \ldots, v_n$ to represent the "pre-state", and a corresponding set $v'_1, \ldots, v'_n$ to represent the "post-state". A boolean formula over these variables characterizes the set of transitions. In other words, a set of states is represented thus: $S = \lambda V.\chi_S$, while a transition relation is represented thus: $R = \lambda(V, V').\chi_R$. The forward image of S w.r.t. R is

$$\text{Image}(R, S) = \lambda V'.\exists V.(S(V) \wedge R(V, V'))$$

while the reverse image is

$$\text{Image}(R^{-1}, S) = \lambda V.\exists V'.(S(V') \wedge R(V, V'))$$

Evaluating images thus requires conjunction, projection and variable substitution. The fixed point computations required to compute, for example, the set of states reachable from set S, also use the disjunction operation. Negation is not strictly needed, since all formulas can be put in positive normal form, in which negation applies only to literals.

Thus, we have all of the operations necessary to do symbolic model checking based on the component representation of functions. It is necessary only to

choose appropriate orders V, W and U. We note that transition relations are often of the form $\chi_R = \wedge_{i=1}^{n} C_i(v_i', v_1, \ldots, v_n)$. That is, each post-state variable is typically constrained relative to the pre-state variables, but the post-state variables are independent given a valuation of the prestate variables. In addition, for each i, $\exists v_i'.C_i = 1$. That is, the transition relation does not constrain the pre-state variables in any way. If this is the case, then there is a distinct advantage to using the order $V = (v_1, \ldots, v_n, v_1', \ldots, v_n')$. In this case, by theorem 18, the component in the decomposition corresponding to v_i' is exactly R_i, while all the components corresponding to v_i are equal to 1. That is, the conjunction of the transition relation parts is formed essentially for free. This makes it unnecessary to the "conjunctive partitioning" technique to avoid an explosion in the size of the transition relation [BCL91].

4.1 Example

One of the advantages of the decomposed representation is the fact that conditionally independent variables are "factored out". As an example of this phenomenon, we consider verifying the equivalence of the two FIFO queue implementations of figure 1. The basic technique is to compute the reachable states of the two running in parallel [CBM89]. As mentioned previously, there is no fixed correspondence between locations in the two queues. However, once we fix the shift register contents and the ring buffer "head pointer", the ring buffer data elements become independent (since they are either uninitialized, or determined by the corresponding shift register element). This suggests that in the variable order V control should precede shift register data, which in turn should precede ring buffer data (or the roles of the two implementations could be reversed). In this case, when representing the set of reachable states, each component corresponding to a ring buffer data bit is a linear-size OBDD, which in essence reads the value of the head pointer, then compares the ring buffer bit to the corresponding shift register bit. As a result, the overall size of the representation is quadratic in the number of data bits.

On the other hand, since there is no fixed correspondence between the data bits, there is no interleaving of the bits that will yield a small OBDD for the reachable state set. This is illustrated in the graphs of figures 2–4. In these graphs, the ordinal axis is the number of data bits in each queue (the queues are one bit wide, however essentially the same results apply to wider queues). In the first graph, we see the size of the decomposed representation of the transition relation. This is the same as the size of the conjunctively partitioned transition relation, for reasons mentioned above. The second figure shows the size of the decomposed representation for the largest state set obtained in the reachable states iteration (which happens to be last iteration in all cases). This is well fit to a quadratic curve, as expected. The third figure shows the size of the OBDD representation of the reachable states. Note that the scale here is two orders of magnitude larger than the previous graph. This graph shows the expected exponential explosion, since the OBDD representation must in essence record the entire contents of one queue in order to compare it to the other queue.

It should also be noted here that there exists a compact "free BDD" [GM94] representation for the reachable state set in our example. However using free BDD's would require the user to provide the correct $O(n^2)$ DAG that determines the free BDD "type". Using decompositions, the simple heuristic "control before data" is sufficient.

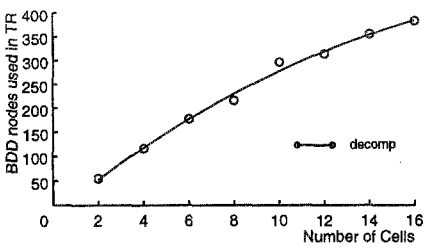


Fig. 2. Space used to represent the transition relation for queue example.

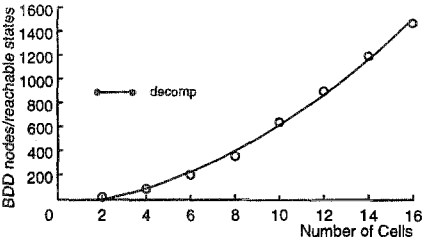


Fig. 3. Space used for decomposed representation of the reachable states for queue example.

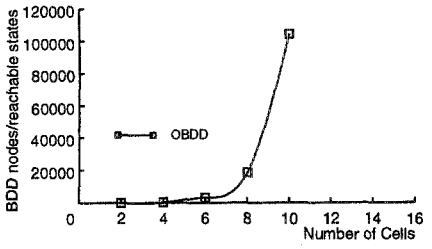


Fig. 4. Space used for OBDD representation of the reachable states for queue example.

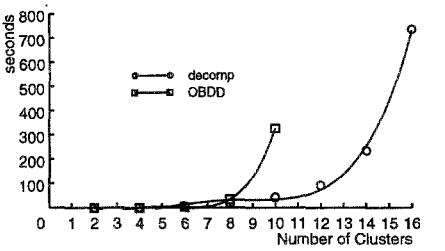


Fig. 5. Time used for computation of the reachable states for queue example.

Finally, figure 5 shows the CPU time in seconds used to compute the reachable state set using both representations. Here, we find the CPU time increasing rapidly in both cases (although the decomposed representation is more efficient as we increase the number of bits). In the decomposition case, it is unclear whether this is an exponential expansion or a fairly high order polynomial (though the difference may be of no practical interest). The algorithms operating on decompositions are not necessarily polynomial, even when measured relative to the result. Therefore, it is possible that exponential time is actually being used. On the other hand, one expects a factor n in the number of iterations due to increasing diameter of the state space. In addition to this, each iteration involves a conjunction, which uses n^2 OBDD operations, each of which is proportional to the transition relation component size ($O(\log n)$) and the state set component size $O(n)$. This would imply at least time proportional to $O(n^5)$, which fits the available data. From a practical point of view, however, it appears that any gains made in space in using the decomposed representation might be

offset by losses in time. The question of improving the time performance of the algorithms (at least heuristically) needs to be addressed.

5 Conclusions

We have seen that a boolean representation conjunctively decomposed using generalized cofactor provides a canonical form that exploits "conditional independence" between variables. This property can provide a more compact representation than OBDD's alone, especially in the case when the correspondence between state variables is not fixed, but varies as a function of control. Algorithms for logical operations and projection on this form were described, making the representation usable for symbolic model checking.

The most important practical problem that remains to be solved regarding decompositions is the time required to apply $O(n^2)$ OBBD operations for each operation on a decomposition (where n is the number of variables). The number of variables could, for example, be reduced by grouping them into many-valued variables, though this could make the representation exponentially larger. Also, tight bounds on the complexity of the algorithms should be obtained.

Acknowledgements: This work benefited greatly from discussions with Robert Kurshan of AT&T Bell Labs.

References

[BCL91] Jerry R. Burch, Edmund M. Clarke, and David E. Long. Symbolic model checking with partitioned transition relations. In A. Halaas and P. B. Denyer, editors, *Proceedings of the IFIP International Conference on Very Large Scale Integration*, Edinburgh, Scotland, August 1991.

[Bry86] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8), 1986.

[CBM89] Olivier Coudert, Christian Berthet, and Jean Christophe Madre. Verification of synchronous sequential machines based on symbolic execution. In Joseph Sifakis, editor, *Automatic Verification Methods for Finite State Systems, International Workshop, Grenoble, France*, volume 407 of *Lecture Notes in Computer Science*. Springer-Verlag, June 1989.

[GM94] J. Gergov and C. Meinel. Efficient boolean manipulation with obdd's can be extended to fbdd's. *IEEE Transactions on Computers*, 43(10):1197–209, Oct. 1994.

[HD93] A. J. Hu and D. L. Dill. Efficient verification with bdds using implicitly conjoined invariants. In C. Courcoubetis, editor, *Computer Aided Verification. 5th International Conference, CAV '93*, pages 3–14, Berlin, Germany, 1993. Springer-Verlag.

[JABF92] J. Jain, J. A. Abraham, J. Bitner, and D. S. Fussell. Probabilistic verification of boolean functions. *Formal Methods in System Design*, 1(1):61–115, July 1992.

[TSL+90] H. J. Touati, H. Savoj, B. Lin, R. K. Brayton, and A. Sangiovanni-Vincentelli. Implicit state enumeration of finite state machines using BDD's. In *ICCAD*, pages 130–133, 1990.

Symbolic Model Checking Using Algebraic Geometry

George S. Avrunin

Department of Mathematics, University of Massachusetts, Amherst, MA 01003-4515
avrunin@math.umass.edu

Abstract. In this paper, I show that methods from computational algebraic geometry can be used to carry out symbolic model checking using an encoding of Boolean sets as the common zeros of sets of polynomials. This approach could serve as a useful supplement to symbolic model checking methods based on Ordered Binary Decision Diagrams and may provide important theoretical insights by bringing the powerful mathematical machinery of algebraic geometry to bear on the model checking problem.

1 Introduction

Symbolic model checking [8, 13] with Ordered Binary Decision Diagrams (OBDDs), or variants of OBDDs, is a widely used and successful technique for verifying properties of concurrent systems, both hardware and software. But there are many systems for which the OBDDs are too large to make model checking feasible and, aside from a few results like McMillan's theorem on bounded width circuits [13] or Bryant's theorem on integer multiplication [5], there is little theoretical guidance to indicate precisely when the OBDD methods are practical.

It therefore seems worthwhile to investigate alternative "symbolic" representations of Boolean sets that could be used for model checking. Such representations, if they are practical at all, would presumably allow efficient model checking of somewhat different classes of systems than OBDDs, and thus supplement existing symbolic model checking methods. Furthermore, an alternative representation might lead to new theoretical insights into the practicality of symbolic model checking, thereby providing guidance to system developers choosing methods for verifying properties of their systems. This is especially true if there is already a substantial body of theory concerning the proposed representation.

In this paper, I show how computational algebraic geometry can provide representations of Boolean sets suitable for symbolic model checking. The basic idea is that any Boolean set can be regarded as the common zeros of a finite set of polynomials with coefficients in the field of two elements. Such a set of polynomials then provides a symbolic representation of the Boolean set. For example, the common zeros of the set of polynomials $\{x_1 + x_2 + \cdots + x_n, x_1x_2\}$ are exactly the points $(a_1, \ldots, a_n)$ for which an even number of the a_i are 1, and at least one of x_1 and x_2 is zero (all the arithmetic is done modulo 2). A *Gröbner basis* is a canonical choice of such a set of polynomials, and there exist algorithms for finding the Gröbner basis corresponding to a particular Boolean set and for carrying out, at the level of Gröbner bases, the manipulations of Boolean sets required for model checking. Thus, Gröbner bases can be used for symbolic model checking in essentially the same way that OBDDs are.

Algebraic geometry is the study of the geometric objects arising as the common zeros of collections of polynomials. It is an old and rich area of mathematics, and one in which there has been enormous activity and progress in the last few years. In particular, algebraic geometers have studied questions related to the action of groups of symmetries and to the mappings that correspond to abstraction techniques, and considerable attention has been given to computational issues. An approach to symbolic model checking making use of methods from algebraic geometry therefore seems to have considerable promise, both as a supplement to existing methods and as a way to bring a large body of powerful mathematical machinery to bear on the model checking problem.

In the next two sections, I sketch some of the necessary background in algebraic geometry and Gröbner basis methods. The fourth section briefly illustrates the ideas with a small example, and the last section contains a discussion of some of the directions for further investigation of this approach.

2 Some Algebraic Geometry

This section contains an extremely brief presentation of the algebraic geometry needed in the sequel. Any standard text will provide the details and proofs omitted here; the interested reader might consult, for example, the books by Cox, Little, and O'Shea [10] and Hartshorne [12].

We start by setting up some machinery for describing sets of polynomials. Let k be a field (for our applications, k will usually be the field of two elements, the integers modulo 2), and let $k[x_1, \ldots, x_n]$ be the ring of polynomials in the variables $x_1, \ldots, x_n$ with coefficients in k, under the standard addition and multiplication of polynomials. That is, a polynomial is a finite k-linear combination of monomials $x_1^{\alpha_1} x_2^{\alpha_2} \ldots x_n^{\alpha_n}$, where the α_i are nonnegative integers, and multiplication of polynomials is defined by setting $x_1^{\alpha_1} x_2^{\alpha_2} \ldots x_n^{\alpha_n} \cdot x_1^{\beta_1} x_2^{\beta_2} \ldots x_n^{\beta_n} = x_1^{\alpha_1+\beta_1} x_2^{\alpha_2+\beta_2} \ldots x_n^{\alpha_n+\beta_n}$ and extending linearly to products of arbitrary polynomials. Note that the multiplication is commutative and that the element $1 = x_1^0 x_2^0 \ldots x_n^0$ is an identity element for multiplication.

The basic structure of polynomial rings (or any commutative rings) is given in terms of subsets called ideals. In this setting, ideals are not subrings in general, but they play a role in commutative ring theory analogous to that played by normal subgroups in the theory of groups. An *ideal* is a nonempty subset of $k[x_1, \ldots, x_n]$ that is closed under addition and closed under multiplication by any element of the ring. If $F = \{ f_\alpha \mid \alpha \in \mathscr{A} \}$ is a set of polynomials in $k[x_1, \ldots, x_n]$ indexed by the (not necessarily finite) set $\mathscr{A}$, the *ideal generated by* F is the set of sums of the form $\sum_{a \in \mathscr{A}} h_a f_a$, where the $h_a \in k[x_1, \ldots, x_n]$ and only finitely many of the h_a are nonzero. We will write $\langle F \rangle$ for the ideal generated by F. When $F = \{f_1, \ldots, f_s\}$ is a finite set, we often write $\langle f_1, \ldots, f_s \rangle$ for $\langle F \rangle$, and we say that F is a *basis* for the ideal $\langle f_1, \ldots, f_s \rangle$. The Hilbert Basis Theorem tells us that every ideal in the ring $k[x_1, \ldots, x_n]$ is generated by some finite set of polynomials.

We can think of the polynomials as k-valued functions on the vector space k^n in the usual way: we evaluate $f(x_1, \ldots, x_n)$ at the point $(a_1, \ldots, a_n)$ by substituting a_1 for x_1, a_2 for x_2, and so on. We say that $(a_1, \ldots, a_n)$ is a *zero of* f if $f(a_1, \ldots, a_n) = 0$.

Let F be a (not necessarily finite) subset of $k[x_1, \ldots, x_n]$. The *variety* defined by F, written $\mathbf{V}(F)$, is the set of points in k^n that are zeros of all the polynomials in F. Thus $\mathbf{V}(F) = \{ (a_1, \ldots, a_n) \in k^n \mid f(a_1, \ldots, a_n) = 0 \text{ for all } f \in F \}$.

As usual, if $F = \{f_1, \ldots, f_s\}$ is a finite set, we sometimes write $\mathbf{V}(f_1, \ldots, f_s)$ rather than $\mathbf{V}(F)$. It is not hard to see that $\mathbf{V}(f_1, \ldots, f_m) = \mathbf{V}\left(\langle f_1, \ldots, f_m \rangle\right)$, so we can think of every variety as being the variety defined by some ideal.

If $V_1 = \mathbf{V}(I_1)$ and $V_2 = \mathbf{V}(I_2)$ are the varieties defined by ideals I_1 and I_2, then $V_1 \cap V_2 = \mathbf{V}\left(\langle I_1, I_2 \rangle\right)$ and $V_1 \cup V_2 = \mathbf{V}(I_1 \cdot I_2)$, where $I_1 \cdot I_2 = \langle f_1 f_2 \mid f_1 \in I_1, f_2 \in I_2 \rangle$. If $I_1 = \langle f_1, \ldots f_r \rangle$ and $I_2 = \langle g_1, \ldots, g_s \rangle$, then $I_1 \cdot I_2 = \langle f_i g_j \mid 1 \leq i \leq r, 1 \leq j \leq s \rangle$.

In general, not every subset of k^n is the variety of some ideal (the varieties are the closed sets of a certain topology on k^n), but each point $(a_1, \ldots, a_n)$ is the variety of the ideal $\langle x_1 - a_1, x_2 - a_2, \ldots, x_n - a_n \rangle$. Since the union of a finite collection of varieties is a variety, any finite set of points is a variety. If k is finite, as will be the case in our application, any subset of k^n is finite, and therefore is a variety.

For the rest of this section, assume that k is the field of two elements.

As just mentioned, we can regard any set of points in k^n as the variety of some ideal. We can then use the ideal, or any basis for the ideal, as a way of encoding the set of points, just as we might use an OBDD. For instance, k^n is the variety of the ideal consisting of the constant polynomial 0, and the empty subset of k^n is the variety of the constant polynomial 1. A somewhat more interesting example is the following.

Choose a positive integer r and let $s = 2^r$. Regard a point $(a_1, \ldots, a_{rs}) \in k^{rs}$ as a list of s numbers between 0 and $s-1$ by treating each block of r coordinates $a_{ri+1}, a_{ri+2}, \ldots, a_{r(i+1)}$ as the binary representation of a nonnegative integer, and let V be the set of points corresponding to lists in which each number from 0 to $s-1$ occurs exactly once. To construct an ideal I such that $V = \mathbf{V}(I)$, let $f_{i,j}$ be the polynomial $(x_{ri+1} + x_{rj+1} + 1)(x_{ri+2} + x_{rj+2} + 1) \cdots (x_{r(i+1)} + x_{r(j+1)} + 1)$. The polynomial $f_{i,j}$ is zero at a point $(a_1, \ldots, a_{rs})$ if and only if $a_{ri+k} \neq a_{rj+k}$ for some k, so if and only if the ith and jth entries in the list corresponding to $(a_1, \ldots, a_{rs})$ are different integers. Then $V = \mathbf{V}(\, f_{i,j} \mid i < j)$. Other examples are given in Section 4.

Note that there will be more than one ideal I defining a given variety. For instance, the ideals $\{0\}$ and $\langle x_1^2 + x_1, \ldots, x_n^2 + x_n \rangle$ both define the variety k^n (since both 0 and 1 satisfy the equation $x^2 + x = 0$ when we are working modulo 2). In order to do symbolic model checking, we need to be able to determine when two ideals represent the same set of points. We first describe how to do this over a larger field. Let $\bar{k}$ be the algebraic closure of k (this is the smallest extension of k in which every polynomial over k has a root, as every polynomial with coefficients in $\mathbb{R}$ has a root in $\mathbb{C}$). Since $k[x_1, \ldots, x_n] \subseteq \bar{k}[x_1, \ldots, x_n]$, we can regard polynomials in $k[x_1, \ldots, x_n]$ as functions on $\bar{k}^n$, and, for a subset F of $k[x_1, \ldots, x_n]$, we define $\overline{\mathbf{V}}(F)$ to be the points in $\bar{k}^n$ where all the elements of F are zero. For an ideal I, the *radical of* I, denoted by $\sqrt{I}$ is the ideal $\{ f \in k[x_1, \ldots, x_n] \mid f^s \in I \text{ for some positive integer } s \}$. If I_1 and I_2 are ideals of $k[x_1, \ldots, x_n]$, then $\overline{\mathbf{V}}(I_1) = \overline{\mathbf{V}}(I_2)$ if and only if $\sqrt{I_1} = \sqrt{I_2}$. (This is Hilbert's Nullstellensatz.) The Gröbner basis methods described in the next section provide good algorithms for determining when $\sqrt{I_1} = \sqrt{I_2}$, so we can determine when two ideals determine the same variety over the algebraic closure of k.

In general, this does not tell us anything about whether $\mathbf{V}(I_1) = \mathbf{V}(I_2)$, but it does settle the question for a certain class of ideals. Let $Z = \{ x_i^2 + x_i \mid i = 1, \ldots, n \}$. As noted above, every point in k^n is a zero of all the elements of Z, so, for any ideal I, $\mathbf{V}(I) = \mathbf{V}(I) \cap \mathbf{V}(Z) = \mathbf{V}(\langle I, Z \rangle)$. This means that every set of points in k^n is the variety defined by some ideal containing the set Z. However, the only elements of $\bar{k}$ satisfying $x^2 + x = 0$ are 0 and 1, the elements of k, so $\overline{\mathbf{V}}(Z) = k^n$ and $\overline{\mathbf{V}}(\langle I, Z \rangle) = \mathbf{V}(I)$. Thus, if we restrict ourselves to ideals containing Z, we can still represent every subset of k^n and we can determine when two ideals represent the same set of points. As we will see in the next section, restricting our representations to ideals containing Z has some other advantages, as well.

3 Gröbner Bases

In this section, we sketch some of the theory of Gröbner bases. Although this theory has roots in the work of Macaulay as early as 1916, it really dates from Buchberger's thesis in 1965 [6]. There are now several good introductions to the subject; the reader seeking more details might consult the book by Cox, Little, and O'Shea [10] mentioned earlier or those by Becker and Weispfenning [4] and Adams and Loustaunau [1].

3.1 Motivation

To understand a little of the motivation for Gröbner bases, consider the problem of determining whether a given polynomial f belongs to an ideal $\langle f_1, \ldots, f_s \rangle$. If we work over a polynomial ring in one variable, the ideal is generated by a single polynomial, the greatest common divisor d of the set $\{f_1, \ldots, f_s\}$. There exist unique polynomials q and r with the degree of r strictly smaller than the degree of d and $f = qd + r$, and then f belongs to the ideal $\langle d \rangle$ if and only if the remainder r is 0. The polynomials d, q, and r are computed by standard algorithms.

For polynomials in more than one variable, the problem is more difficult. First, the ideal $\langle f_1, \ldots, f_s \rangle$ need not be generated by a single polynomial, so we must generalize our division algorithm to compute a remainder of f on division by the set $\{f_1, \ldots, f_s\}$. This is relatively straightforward, but it turns out that the remainder obtained this way is not uniquely determined. To get a unique remainder, which will be 0 if and only if $f \in \langle f_1, \ldots, f_s \rangle$, we need to use a special kind of generating set for the ideal. These generating sets are called Gröbner bases, and they provide the foundation for the algorithmic solution of many problems involving polynomials and ideals.

3.2 Definitions and basic properties

To define Gröbner bases, we need to specify an ordering on the set of monomials that satisfies certain conditions. It is somewhat more convenient to state things in terms of the n-tuples $(\alpha_1, \ldots, \alpha_n)$ rather than the monomials $x_1^{\alpha_1}, \ldots x_n^{\alpha_n}$, so let $\mathbb{N}^n$ be the set of n-tuples of nonnegative integers. There is an obvious isomorphism of semigroups between the set of monomials under the multiplication given in the previous section and $\mathbb{N}^n$ with component-wise addition.

Again, let k be an arbitrary field. A *monomial* or *term* order on $k[x_1, \ldots, x_n]$ is a relation $\succ$ on $\mathbb{N}^n$ (or equivalently on the set of monomials) satisfying the conditions that $\succ$ is a total order, $\succ$ is a well-ordering, and $\alpha \succ \beta$ implies $\alpha + \gamma \succ \beta + \gamma$ for all $\gamma \in \mathbb{N}^n$. The third condition is essentially a compatibility requirement between the order and the multiplication of monomials. We want to use the order to distinguish a leading, or highest, term in each polynomial. The third condition says that, if we multiply a polynomial by a monomial, the leading term of the result will be the product of the monomial and the leading term of the original polynomial.

Two commonly used monomial orders are the *lexicographic* order, in which $\alpha \succ \beta$ if and only if the leftmost nonzero entry in the difference $\alpha - \beta$ is positive, and the *graded reverse lexicographic* order, in which $\alpha \succ \beta$ if and only if $\sum_i \alpha_i > \sum_i \beta_i$ or $\sum_i \alpha_i = \sum_i \beta_i$ and the right-most nonzero entry in $\alpha - \beta$ is negative. Note, however, that each of these orders is defined only up to a permutation of the variables; there are really $n!$ versions of the lexicographic and graded reverse lexicographic orders. There are results indicating that, for many applications, the graded reverse lexicographic order is most efficient [3]. As we will see soon, some special orders, perhaps constructed from the graded reverse lexicographic, are also required for certain operations on ideals that are used in symbolic model checking.

We need some additional notation. For $\alpha = (\alpha_1, \ldots, \alpha_n) \in \mathbb{N}^n$, we write x^α for the monomial $x_1^{\alpha_1} x_2^{\alpha_2} \ldots x_n^{\alpha_n}$. Let $f = \sum_\alpha a_\alpha x^\alpha$ be a polynomial in $k[x_1, \ldots, x_n]$, and let $\succ$ be a monomial order. The *degree* of f, $\deg(f)$, is $\max\{\, \alpha \in \mathbb{N}^n \mid a_\alpha \neq 0 \,\}$. The *leading coefficient* of f, $\mathrm{LC}(f)$,is $a_{\deg(f)}$. The *leading monomial* of f, $\mathrm{LM}(f)$, is $x^{\deg(f)}$, and the *leading term* of f, $\mathrm{LT}(f)$, is $\mathrm{LC}(f) \cdot \mathrm{LM}(f) = a_{\deg(f)} x^{\deg(f)}$.

Fix a monomial order. A finite subset $G = \{g_1, \ldots, g_t\}$ of an ideal I is a *Gröbner basis* for I (with respect to the given order) if and only if, for every $f \in I$, $\mathrm{LT}(f)$ is divisible by one of the $\mathrm{LT}(g_i)$. It is easy to see that every nonzero ideal has a Gröbner basis, and that any Gröbner basis for an ideal is also a basis for the ideal.

Suppose $\succ$ is a fixed monomial order on $k[x_1, \ldots, x_n]$ and $F = \{f_1, \ldots, f_s\}$ is an ordered s-tuple of polynomials. Then we can generalize the division algorithm for polynomials in one variable to show that every $f \in k[x_1, \ldots, x_n]$ can be written as a sum of multiples of the f_i and a polynomial r that is either 0 or a sum of monomials not divisible by any of $\mathrm{LT}(f_1), \ldots, \mathrm{LT}(f_s)$. We say that r is a *remainder* of f on division by F. The polynomial r depends on the way that the set F is indexed.

Buchberger gave an algorithm for constructing a Gröbner basis for a given ideal. The algorithm starts with a set of generators for the ideal. It then constructs an *S-polynomial* for a pair of elements of this set, and adds the remainder of the S-polynomial on division by the generating set to the set. It continues in this fashion until all the remainders are 0; at this point, the set of generators is a Gröbner basis. Various improvements in efficiency can be made by carefully choosing which S-polynomials to compute at a particular stage [7].

If G is actually a Gröbner basis for an ideal I and $f \in k[x_1, \ldots, x_n]$, then the remainder r of f on division by G is uniquely determined (i.e., does not depend on the order in which the elements of the basis are listed), and $f \in I$ if and only if $r = 0$. Buchberger's Gröbner basis algorithm thus yields an algorithm for determining whether a polynomial belongs to a given ideal. As noted in the previous section, we can also use

Gröbner bases to determine whether a polynomial is in the radical of a given ideal.

We say that a Gröbner basis G is *reduced* if the leading coefficients of the elements of G are all 1 and no monomial of an element of G lies in the ideal generated by the leading terms of the other elements of G. The key result is that, for a fixed monomial order, a nonzero ideal has a unique reduced Gröbner basis. The algorithm for finding a Gröbner basis can easily be extended to output this reduced Gröbner basis. Thus, we have an algorithm for determining whether two ideals $\langle f_1, \ldots, f_s \rangle$ and $\langle h_1, \ldots, h_t \rangle$ are equal.

3.3 Projections

Suppose that a concurrent system can be described in terms of n Boolean state variables, and let F be the field of two elements. We then represent the possible states of the system by the elements of the vector space F^n. The transition relation of the system can then be regarded in the usual fashion as a subset T of F^{2n}, where a point $(b_1, \ldots, b_n, b'_1, \ldots, b'_n) \in T$ if and only if there is a transition from the state represented by $(b_1, \ldots, b_n)$ to the one represented by $(b'_1, \ldots, b'_n)$. Suppose we have a set of points $C \subseteq F^n$ corresponding to a formula ϕ. For symbolic model checking, we need to be able to describe the points corresponding to, for instance, the formula $EX\phi$. These are the points $(b_1, \ldots, b_n) \in F^n$ such that there exists a point $(b'_1, \ldots, b'_n) \in C$ with $(b_1, \ldots, b_n, b'_1, \ldots, b'_n) \in T$. In the framework of algebraic geometry, this amounts to finding the projection of a subset of F^{2n} onto the first n coordinates. We can use Gröbner bases, with suitable monomial orders, to accomplish this.

Let R be the polynomial ring $F[x_1, \ldots, x_n, x'_1, \ldots, x'_n]$ in $2n$ variables. We regard R as a ring of Boolean functions on F^{2n}, as usual. Let $I = \langle f_1, \ldots, f_s \rangle$, and assume that the set Z consisting of the polynomials of the form $x_i^2 + x_i$ and $(x'_i)^2 + x'_i$ is contained in $\{f_1, \ldots, f_s\}$. (Recall that adding Z to the generating set of I does not change $\mathbf{V}(I)$.) Let R_1 be the subring consisting of polynomials in the variables $x_1, \ldots, x_n$ and let I_1 be the ideal $I \cap R_1$ of the ring R_1. We can show that any $(b_1, \ldots, b_n) \in V(I_1)$ extends to an element $(b_1, \ldots, b_n, b'_1, \ldots, b'_n) \in \mathbf{V}(I)$. In particular, if we take I to be an ideal with variety $\{ (b_1, \ldots, b_n, b'_1, \ldots, b'_n) \in T \mid (b'_1, \ldots, b'_n) \in C \}$, then $\mathbf{V}(I_1)$ is the projection of this set on the first n coordinates. It is this projection that we need for model checking.

So the problem is to find I_1. Let $\succ$ be a monomial order satisfying the property that any monomial involving one of the x'_i is greater than any monomial involving only $x_1, \ldots, x_n$, and let $G = \{g_1, \ldots, g_s\}$ be a Gröbner basis of I with respect to $\succ$. If I contains Z, it can be shown that $G \cap R_1$ is a Gröbner basis for I_1. So we can find a Gröbner basis for I_1 as long as we can produce a suitable monomial order, and we can do that by, for example, modifying the graded reverse lexicographic order.

3.4 Complexity

It is natural to measure the size of a finite set F of polynomials in terms of the number of variables, the number of polynomials in F, the maximum degree of the polynomials, and the size of their coefficients. Given F, we are interested in these measures for a Gröbner basis for $\langle F \rangle$, as well as for the intermediate sets constructed in finding

a Gröbner basis. In the general case, all of these measures behave fairly badly. For instance, examples are known where the construction of a Gröbner basis for an ideal generated by polynomials of degree less than or equal to d can involve polynomials of degree 2^{2^d} [13]. Over the field of two elements, however, all the coefficients are 0 or 1, and when our ideal includes all the $x_i^2 + x_i$, the only polynomials we have to consider are those in which no variable appears with degree greater than 1. I am not aware of specific complexity results for this case. Of course, just as with OBDDs, there are too many Boolean sets for all of them to have small representations in terms of Gröbner bases, so the interesting question is really one of characterizing the Boolean sets that do have such nice representations and understanding when the model checking process involves only such sets.

It is worth noting that there has been work on dynamic modification of the monomial order as the Gröbner basis calculation proceeds [11].

4 An Example

In this section we show how the machinery described in the preceding sections can be applied to verify a property of a small system. Consider the SMV code shown in Figure 1 (the numbers on the left in the module `prc` are inserted for reference, and are not part of the SMV program). This is the "mutex1" example distributed with SMV, with the fairness declarations deleted for simplicity. This system implements a mutual exclusion protocol.

We begin by describing the state variables. We can use one state variable for `turn` and two state variables for each of `s0` and `s1` to describe the state of the system, so we need 11 state variables for the transition relation (five for the current state, five for the next state, and one to keep track of which process is currently running, as required by the semantics of SMV). Figure 2 shows how we partition the variables. We encode the enumerated variables `s0` and `s1` by setting the corresponding pair of bits to $(0,0)$ for `noncritical`, to $(0,1)$ for `trying`, and to $(1,0)$ for `critical`.

The next step is to find an ideal J such that $\mathbf{V}(J)$ is the transition relation, T. We have to capture the assignments made by the processes `pr0` and `pr1`. Our approach is to find polynomials whose zeros correspond to pairs of states in which the appropriate assignments are made.

Consider first `pr0`. Line (1) tells us that, if the system is in a state where `pr0` is running (i.e., when $x_6 = 0$), and `s0` is `noncritical` (i.e., when $(x_2, x_3) = (0,0)$), the value of `s0` in the next state will be `noncritical` or `trying` (i.e., $(x_2', x_3') = (0,0)$ or $(x_2', x_3') = (0,1)$). So we need to find a set of polynomials whose common zeros are the points $(x_1, \ldots, x_5, x_1', \ldots, x_5', x_6)$ with $x_6 = 0$, $x_2 = 0$, $x_3 = 0$, $x_2' = 0$, and $x_3' = 0$ or 1. Since the condition on x_3' holds at all points, we can use the set $\{ x_6, x_2, x_3, x_2' \}$. For calculations, it seems somewhat more convenient to take the single polynomial

$$f_1 = (x_6 + 1)(x_2 + 1)(x_3 + 1)(x_2' + 1) + 1,$$

which has the same zeros.

```
MODULE main

VAR
s0: {noncritical, trying, critical};
s1: {noncritical, trying, critical};
turn: boolean;
pr0: process prc(s0, s1, turn, 0);
pr1: process prc(s1, s0, turn, 1);

ASSIGN
init(turn) := 0;

SPEC
EF((s0 = critical) & (s1 = critical))


MODULE prc(state0, state1, turn, turn0)

ASSIGN
init(state0) := noncritical;
next(state0) :=
  case
(1)   (state0 = noncritical) : {trying,noncritical};
(2)   (state0 = trying) & (state1 = noncritical): critical;
(3)   (state0 = trying) & (state1 = trying) & (turn = turn0): critical;
(4)   (state0 = critical) : {critical,noncritical};
(5)   1: state0;
  esac;
 next(turn) :=
  case
(6)   turn = turn0 & state0 = critical: !turn;
(7)   1: turn;
  esac;
```

Fig. 1. SMV program for mutual exclusion protocol

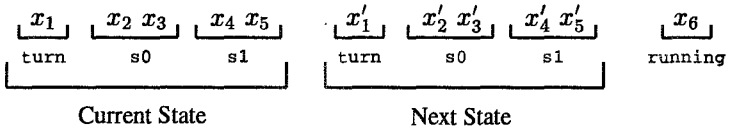


Fig. 2. State variables for transition relation

In a similar fashion, lines (2)–(4) yield polynomials

$$f_2 = (x_6+1)(x_2+1)x_3(x_4+1)(x_5+1)x_2'(x_3'+1)+1$$
$$f_3 = (x_6+1)(x_2+1)x_3(x_4+1)x_5(x_1+1)x_2'(x_3'+1)+1$$
$$f_4 = (x_6+1)x_2(x_3+1)(x_3'+1)+1.$$

Line (5) must be treated a little differently. It asserts that, if `pr0` is running and none of the first four guards in the case statement is true, then `next(s0) = s0`. There are two ways all the guards could fail: `s0 = s1 = trying` but `turn = 1`, and

`s0 = trying` while `s1 = critical`. We will represent each of these conditions with a polynomial:

$$f_{5a} = (x_6 + 1)(x_2 + 1)x_3(x_4 + 1)x_5x_1(x_2' + 1)x_3' + 1$$
$$f_{5b} = (x_6 + 1)(x_2 + 1)x_3x_4(x_5 + 1)(x_2' + 1)x_3' + 1.$$

We note that it would also be possible to represent the negation of the guards on lines (1)–(4) directly, rather than explicitly listing the remaining cases. This approach is illustrated in the treatment of line (7) below.

Lines (6) and (7) describe the possible values of `next(turn)` while `pr0` is running. From line (6), we have

$$f_6 = (x_6 + 1)(x_1 + 1)x_2(x_3 + 1)x_1' + 1,$$

using the fact that, for `pr0`, `turn0` $= 0$.

Line (7) tells us that, while `pr0` is running, `turn` does not change unless the guard of line (6) is satisfied. We want a polynomial that is zero at exactly the points where $x_6 = 0$, the guard of line (6) is false (so $(x_1 + 1)x_2(x_3 + 1) = 0$), and $x_1 = x_1'$. A polynomial that is zero at exactly these points is

$$f_7 = (x_6 + 1)\left((x_1 + 1)x_2(x_3 + 1) + 1\right)(x_1 + x_1' + 1) + 1.$$

The variable `s1` is not assigned while `pr0` is running. The semantics of SMV then imply that `next(s1)` = `s1` if `pr0` is running. We can express this condition with the polynomial

$$f_8 = (x_6 + 1)(x_4 + x_4' + 1)(x_5 + x_5' + 1) + 1.$$

The points $(x_1, \ldots, x_5, x_1', \ldots, x_5', x_6) \in T$ corresponding to pairs of states in which `pr0` is running in the current state are those where one of $f_1 \ldots f_{5b}$ is zero, one of f_6 or f_7 is zero, and f_8 is zero. Since a product of polynomials is zero if and only if at least one of the factors is zero, these are the points where the three polynomials $f_1f_2f_3f_4f_{5a}f_{5b}$, f_6f_7, and f_8 are all zero. In other words, the points in the transition relation with $x_6 = 0$ form the variety of the ideal $I_{\mathtt{pr0}} = \langle f_1f_2f_3f_4f_{5a}f_{5b}, f_6f_7, f_8\rangle$.

In a similar fashion, we construct an ideal $I_{\mathtt{pr1}}$ whose variety is the set of points in T with $x_6 = 1$. If we set $I = I_{\mathtt{pr0}} \cdot I_{\mathtt{pr1}}$ and $J = \langle I, Z\rangle$, where $Z = \{x_1^2 + x_1, \ldots, x_5^2 + x_5, (x_1')^2 + x_1', \ldots, (x_5')^2 + x_5', x_6^2 + x_6\}$, then $T = \mathbf{V}(J)$.

The property we want to check is $EF($`s0 = critical` $\wedge$ `s1 = critical`$)$. Let $\phi =$ (`s0 = critical` $\wedge$ `s1 = critical`). So we want to find the least fixed point of $\tau = \lambda y.\phi \vee EXy$. Given a description of y as a variety, we need to express the points corresponding to $\phi \vee EXy$ as the variety of some ideal. To do this, we need to describe the points satisfying ϕ as a variety, and we need to compute the ideal defining the variety EXy.

The points $(x_1, \ldots, x_5, x_1', \ldots, x_5', x_6)$ for which ϕ holds are those corresponding to system states in which both `s0` and `s1` are `critical`, i.e., those in which $x_2 = x_4 = 1$ and $x_3 = x_5 = 0$. These are the points in the variety of the ideal $I_\phi = \langle x_2(x_3 + 1) + 1, x_4(x_5 + 1) + 1\rangle$.

To find the ideal corresponding to EXy, we first need to specify that the polynomials defining y are zero in the next state. In our setting, this is accomplished by applying a homomorphism of rings that replaces the x_i by the corresponding x'_i. Let $R = \boldsymbol{F}[x_1, \ldots, x_5, x'_1, \ldots, x'_5, x_6]$ and let $\nu\colon R \to R$ be the (k-linear) ring homomorphism mapping each x_i to x'_i, for $i = 1, \ldots, 5$, each x'_i to 0, and x_6 to x_6. If $f \in R_1 = \boldsymbol{F}[x_1, \ldots, x_6]$ is a polynomial in the x_i, $\nu(f)$ is the corresponding polynomial in the variables $x'_1, \ldots, x'_5, x_6$.

Then if y corresponds to the variety $\mathbf{V}(h_1, \ldots, h_s)$, the variety corresponding to EXy is the projection onto the first n coordinates of the variety of the ideal $I_{y'} = \langle T, \nu(h_1), \ldots, \nu(h_s), Z\rangle$. We find the ideal defining this variety using the methods discussed in Section 3.3: We construct a Gröbner basis $G_{y'}$ for $I_{y'}$ with respect to a suitable order, and take the elements of $G_{y'}$ that lie in the subring R_1. If $G_1 = R_1 \cap G_{y'}$, then the variety defined by $\langle G_1\rangle \cdot I_\phi$ corresponds to the points satisfying the formula $\phi \vee EXy$. In this fashion, we can find the least fixed point of $\lambda y.\phi \vee EXy$.

I used the program *Macaulay* [2] to carry out these calculations. *Macaulay* provides facilities for defining rings, ideals, and homomorphisms, and for carrying out a variety of Gröbner basis calculations. Many of these calculations could have been done using other computer algebra systems; *Macaulay* seemed to be the most convenient for these experiments.

The Gröbner basis found by *Macaulay* for the ideal I_μ whose variety is the least fixed point of $\lambda y.\phi \vee EXy$ consists of the six polynomials $x_1^2 + x_1$, $x_2 + 1$, x_3, $x_4 + 1$, x_5, and $x_6^2 + x_6$. (Note that the first and last of these are zero at all points of $\boldsymbol{F}^n$.) The variety $\mathbf{V}(I_\mu)$ consists of the points $(x_1, \ldots, x_5, x'_1, \ldots, x'_5, x_6)$ where $x_2 = 1, x_3 = 0$, $x_4 = 1$, and $x_5 = 0$. These are the points where `s0` and `s1` are both `critical`; this tells us that it is not possible to reach a state where both `s0` and `s1` are `critical` (i.e., where ϕ holds) from a state where at least one is not `critical`. In particular, no state where both `s0` and `s1` are `critical` is reachable from the initial state, since the initial conditions specify that `s0` and `s1` are `noncritical`. We can verify this by expressing the initial conditions as the zeros of an ideal, say $I_{\texttt{init}} = \langle x_2, x_3, x_4, x_5, x_6\rangle$, and computing the ideal of the intersection of the varieties $\mathbf{V}(I_\mu)$ and $\mathbf{V}(I_{\texttt{init}})$. *Macaulay* reports that the constant polynomial 1, which has no zeros, is a Gröbner basis for this ideal, and we see that the intersection is empty. We conclude that $EF\phi$ is false in the initial state.

Alternatively, we could have found the set of reachable states by starting from $I_{\texttt{init}}$, and taken the intersection with this variety at each stage. (This corresponds to running SMV with the `-f` flag.)

Macaulay runs as an interpreter that can be used interactively or can execute scripts. A script to check the property $EF($`s0 = critical` $\wedge$ `s1 = critical`$)$ took about 10 seconds to execute on a PC with a 100 MHz Pentium and 16 MB of memory, running Linux. *Macaulay* allocated 755 KB of memory in the course of this calculation. For comparison, on the same machine SMV took approximately 0.1 seconds to check the same property, and allocated just over 917 KB. SMV, of course, was building OBDDs from the code shown in Figure 1, while for *Macaulay*, I had manually translated this code into the polynomials described above.

5 Discussion

In this paper, I have shown how techniques from computational algebraic geometry can be used for symbolic model checking. This approach may provide a useful supplement to existing methods based on OBDDs, and may also provide important theoretical insights by allowing the application of deep results in algebraic geometry to the model checking problem. Additional research will be needed to determine whether these potential advantages are borne out.

Macaulay, the program I used for the calculations described in the previous section, was intended for use in a much more general setting. It supports, for instance, calculations over fields of characteristic up to about 32,000, rather than just characteristic 2. Its data structures and algorithms are therefore not optimized for the cases used in symbolic model checking. Furthermore, it runs as an interpreter. For that reason, the difference in execution time between *Macaulay* and SMV does not seem to carry much significance for assessing the practicality of these methods. Although some further investigation of the practicality of symbolic model checking using the techniques from algebraic geometry can probably be done using tools like *Macaulay*, more serious study will likely require building a prototype tool designed specifically for that purpose. Examples like the one in the previous section suggest that it should be fairly easy to build a tool that would work directly from specifications given in the SMV input language.

There are several directions in which the framework proposed here might be generalized. For instance, in the example of Section 4, I worked with polynomials over the field of two elements. This has some clear advantages and seems to be the most natural analog of the OBDD approach. Working over the field of order 2^k, however, might allow much more efficient encoding of conditions involving k-bit blocks of state variables. Similarly, working over fields of characteristic greater than 2 would correspond to some of the non-binary generalizations of OBDDs.

It is difficult to predict exactly what theorems of algebraic geometry might be applicable to symbolic model checking, but some general directions can be sketched. For instance, there is a rich collection of invariants of varieties and ideals, including such things as notions of dimension and degree. Many of these invariants are likely to be related to the difficulty of carrying out symbolic model checking. Algebraic geometry also provides good machinery for handling such things as the action of groups on varieties, maps between varieties, and the properties of intersections of varieties. It might therefore provide new ways to understand and take advantage of symmetries of the system being checked, abstraction to simpler systems, or the effects of constraints representing the interface between a subsystem and its environment. Results in these directions might give information about, for instance, the kinds of Boolean sets arising in fixed point calculations and thus even have implications for model checking using OBDDs.

Acknowledgments

This research was partially supported by the National Science Foundation under Grant No. CCR-9407182. I am grateful to David Cox for helpful discussions about Gröbner bases, to Jay Corbett for clarifying many of the details of SMV, for making a lightly

loaded SparcStation available for some of my experimentation and for helpful comments on earlier drafts of this paper, and to Nicholas Schmitt for providing me with several versions of his Gröbner basis package, *Ideal*, which I used in my initial exploration of these ideas.

References

1. W. W. Adams and P. Loustaunau. *An Introduction to Gröbner Bases*, volume 3 of *Graduate Studies in Mathematics*. American Mathematical Society, Providence, RI, 1994.
2. D. Bayer and M. Stillman. *Macaulay: A System for Computation in Algebraic Geometry and Commutative Algebra.* Source and object code available for Unix and Macintosh computers. Contact the authors, or download from `math.harvard.edu` via anonymous ftp., 1982–1994.
3. D. Bayer and M. Stillman. A criterion for detecting m-regularity. *Invent. Math.*, 87:1–11, 1987.
4. T. Becker and V. Weispfenning. *Gröbner Bases*, volume 141 of *Graduate Texts in Mathematics*. Springer-Verlag, New York, 1993.
5. R. E. Bryant. On the complexity of VLSI implementations and graph representations of Boolean functions with application to integer multiplication. *IEEE Trans. Comput.*, 40(2):205–213, Feb. 1991.
6. B. Buchberger. *Ein Algorithmus zum Auffinden der Basiselemente des Restklassenringes nach einem nulldimensionalen Polynomideal.* PhD thesis, University of Innsbruck, 1965.
7. B. Buchberger. Gröbner bases: An algorithmic method in polynomial ideal theory. In N. K. Bose, editor, *Multidimensional Systems Theory*, pages 184–232. D. Reidel, 1985.
8. J. Burch, E. Clarke, K. McMillan, D. Dill, and L. Hwang. Symbolic model checking: 10^{20} states and beyond. In *Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 428–439, 1990.
9. D. Cox, J. Little, and D. O'Shea. *Ideals, Varieties, and Algorithms.* Undergraduate Texts in Mathematics. Springer-Verlag, New York, 1992.
10. P. Gritzmann and B. Sturmfels. Minkowski addition of polytopes: Computational complexity and applications to Gröbner bases. *SIAM Journal on Discrete Mathematics*, 6(2):246–269, 1993.
11. R. Hartshorne. *Algebraic Geometry*, volume 52 of *Graduate Texts in Mathematics.* Springer-Verlag, New York, 1977.
12. E. Mayr and A. Meyer. The complexity of the word problem for commutative semigroups and polynomial ideals. *Adv. in Math.*, 1982.
13. K. L. McMillan. *Symbolic Model Checking.* Kluwer Academic Publishers, Boston, 1993.

A Partition Refinement Algorithm for the π-Calculus (Extended Abstract) *

Marco Pistore[1] and Davide Sangiorgi[2]

[1] Department of Computer Science, University of Pisa, Italy
[2] INRIA - Sophia Antipolis, France.

1 Introduction

Bisimulation is widely used for defining behavioural equivalences on terms of process description languages. It has been extensively studied in CCS, where efficient algorithms and tools for bisimulation checking have been devised. A prominent role among these algorithms is occupied by the *partition refinement algorithm* [10, 6]. It works in two phases: In the first, the state spaces of the processes to be checked (i.e., the set of their derivatives under arbitrary sequences of actions) are separately generated; in the second, a partition refinement procedure is applied to the union of the state spaces, which terminates when the equivalence classes of bisimulation are found. Most important, the same algorithm can be used to obtain a *minimal realisation* of a process P, i.e., a process which has the minimum number of states and transitions among all those bisimilar with P. In the case of CCS, algorithms for bisimilarity generally apply only to *finite-state* processes, syntactically described by disallowing parallel composition within recursive definitions.

In this paper, we study bisimulation checking in the π-calculus, a development of CCS where channel names can be communicated. Name-passing increases the expressiveness of the calculus, but it also dramatically affects the theory — above all the definition of bisimulation and its associated algorithms. In the π-calculus, the syntactic counterpart of CCS finite state processes are the *finite control* processes [3]. Due to the creation of new names, finite control processes can exhibit an infinite-state behaviour.

Three definitions of bisimulation, called *late* [7], *early* [7] and *open* [11], have been proposed for the π-calculus, and vary in the way name instantiations are handled. Here, we focus on open bisimulation, for three main reasons: First, by contrast with the other two, open bisimulation is a full congruence; this can be used, for instance, for compositional minimisations of processes. Secondly, the average complexity of checking open bisimulation is expected to be substantially lower than those of late and early bisimulations [11]. Thirdly, a partition refinement algorithm for open bisimulation presents more difficulties, hence one expects to extract algorithms for late and early bisimulations from it.

Open bisimulation. Differently from early and late bisimulations, where free names of processes are viewed as constants (hence cannot be identified), in open

* Research supported by CNR project "Strumenti per la Verifica di Proprietà Critiche di Sistemi Concorrenti e Distribuiti" and by CNET project "Modélisation de Systèmes Mobiles".

bisimulation free names are viewed as free variables. However, permanent inequalities on names can be imposed by means of *distinctions*, i.e., irreflexive and symmetric relations on names. Distinctions allow us, for instance, to create constants by declaring certain names different from all other names; they are also useful to handle name extrusions — see below. We shall write $\sim_D$ to denote open bisimilarity under distinction D.

We review some aspects of the symbolic characterisation of open bisimulation [11]. Symbolic transitions are of the form $P \xrightarrow{M,\alpha} P'$ where, intuitively, M represents the least condition, in the form of a conjunction of equalities between names, under which action α can occur. A condition M determines an equivalence relation on names; chosen a representative for each equivalence class, M also determines a name substitution σ_M which maps names to their representatives. We write $\emptyset$ for the "true" condition, that is the empty conjunction. Among the possible forms of action α, there is the *bound output* $\overline{a}(b)$, which denotes the emission at a of the *private* name b. Approximately, the clause of bisimilarity on bound outputs says that if $P \sim_D Q$, and $\mathrm{fn}(P,Q)$ is the set of free names in P and Q, then:

$$\begin{array}{l}\text{whenever } P \xrightarrow{M,\overline{a}(b)} P' \text{ with } b \text{ not free in } Q, \text{ there are } N \text{ and } Q' \text{ s.t.} \\ Q \xrightarrow{N,\overline{a}(b)} Q', \quad M \text{ implies } N, \quad \text{and} \\ P' \sim_{D'} Q'\sigma_M, \quad \text{for } D' \stackrel{\text{def}}{=} D\sigma_M \cup \{b\} \times \mathrm{fn}(P,Q)\end{array} \tag{1}$$

Derivative Q' and distinction D are updated according to the name equalities imposed by condition M, and distinction $\{b\} \times \mathrm{fn}(P,Q)$ is added to record the creation of the new name b.

Main problems for a partition algorithm. In (1), three forms of dependencies between P and Q affect the transitions and the derivatives to examine:

Dep.1: The name emitted by P cannot occur free in Q;
Dep.2: The condition M in the transition of P determines a substitution σ_M which is applied onto the derivative of Q;
Dep.3: There is a global distinction, which is updated using informations (the free names) from both processes.

These dependencies prevent us from generating the state spaces of P and Q separately from each other, as requested in the first phase of the partition refinement algorithm. Each dependency introduces a separate problem, which we now discuss in more detail.

Dep.1 is imposed to ensure that the bound name chosen by P can also be chosen by Q, and is necessary because bisimilar processes may have different sets of free names, like

$$P_1 \stackrel{\text{def}}{=} P_2 + [c=e]\overline{a}(b).c(d).\mathbf{0} \qquad \text{and} \qquad P_2 \stackrel{\text{def}}{=} \overline{a}(b).e(d).\mathbf{0} \tag{2}$$

(a matching $[c=e]$ means "if $c = e$ then"). The choice of the bound name could be made locally if *active names* [8], instead of free names, were used. A name x is *active* for a process P if x does affect the behaviour of P, i.e., P and $\nu x P$ are not

bisimilar ($\nu\_$ is the restriction construct). Active names would eliminate Dep.1 because bisimilar processes have the same sets of active names. Unfortunately, computing active names is as difficult as computing bisimilarity. For instance, x is active in $a(y).([y=x]P+Q)$ iff $P\{y/x\}$ and $Q\{y/x\}$ are bisimilar.

Dep.2 arises because conditions M and N on the transitions from P and Q may be logically non-equivalent. This situation happens, for instance, when comparing processes P_1 and P_2 in (2), where transition $P_1 \xrightarrow{[c=e],\overline{a}(b)} c(d).\mathbf{0}$ is matched by transition $P_2 \xrightarrow{\emptyset,\overline{a}(b)} e(d).\mathbf{0}$, for under the condition $[c=e]$ the two derivatives are bisimilar (in this case, they are actually equal). However, this transition of P_1 does not add anything interesting to its behaviour, since covered by transition $P_1 \xrightarrow{\emptyset,\overline{a}(b)} e(d).\mathbf{0}$, which has a logically weaker condition. The first transition can therefore be regarded as *redundant*. Indeed, if redundant transitions were ignored in the bisimulation clauses, then Dep.2 could be removed. But again, determining whether a transition is non-redundant is as difficult as computing bisimilarity.

To avoid Dep.3, distinctions should be made local to processes, and should be locally updated. But the update of the local distinction of a process P has to depend on its free names, otherwise the update might not be sound. As a consequence, since bisimilar processes may have different free names, one must be able to compare processes with different local distinctions. This makes it hard to obtain a *transitive* bisimulation relation and to recover open bisimulation.

Our approach. Computing bisimilarity, active names or non-redundant transitions is of equal difficulty, since, unfortunately, they all depend from each other. Our algorithm will hence compute bisimilarity, active names and non-redundat transitions *at the same time*. Since bisimulation is a maximal fixed-point, whereas active names and non-redundant transitions are minimal fixed-points, the algorithm approximates bisimulation from above, and active names and non-redundant transitions from below. At the beginning, all processes are assumed bisimilar, no name is assumed active and no transition is assumed non-redundant. In each approximation step, the appropriate transitions and derivatives of processes are selected according to the current estimation of active names and non-redundant transitions, and the standard partition refinement algorithm is applied. At the end, active names and non-redundant transitions are updated. In this way, at each step the assumed set of bisimilar processes decreases, whereas the assumed sets of active names and non-redundant transitions increase. This procedure is repeated until a fixed-point is reached.

The update of the distinction which is local to a process P uses its free names. The bisimulation relation computed by the algorithm, and defined on processes with a local distinction, is not transitive; but it enjoys a weak form of transitivity which is enough to prove a characterisation theorem w.r.t. open bisimulation and to apply the partition refinement algorithm (in which transitivity is important).

Our algorithm can be used on finite-control processes to check open bisimulation and to compute minimal realisations of processes.

Related work. The closest work to ours is [8], where a partition refinement

algorithm for early and late bisimulation is obtained which works for finite-control processes without the matching operator. [8] has inspired our work, and has provided us with useful insights. However, our technical development is quite different, because none of the problems in Dep.1–3 arise in [8]. Dep.2 and Dep.3 do not arise because they are specific to open bisimulation; Dep.1 does not pose a serious obstacle because, in late and early bisimulations without matching, active names of processes are trivial to compute: Roughly, a name x is active in a process P if x appears as a free name in a label of a computation of P. Hence active names can be computed by a standard transitive closure procedure.

The *Mobility Workbench* [12] is a tool for mechanically checking open bisimulation on finite control processes. It adopts an *on the fly* [4] approach, as opposed to the partitioning approach (in on-the-fly, the state spaces of processes compared are created at the same time as the candidate bisimulation relation). A disadvantage of on-the-fly is that it cannot be used to give the minimal realisation of a process. Moreover, due to the need of backtracking, in general on-the-fly is less efficient than partitioning both in time and in space (especially in the case of weak bisimulations). However, on-the-fly can be superior on processes which exhibit a limited degree of non-determinism; and it may return an answer even on non-finite-control processes, if not bisimilar.

In [3] decidability of early and late bisimilarity for finite control processes is proved. In particular, it is shown that for every pair of finite control processes only a finite number of names is sufficient for checking bisimilarity; once the number of names if guessed, the state space of both agents can be built and a partition refinement algorithm can be applied. However, this approach can be expensive and, since the number of names can be guessed only for pairs of processes, it does not provide us with minimal realisations. Open bisimulation, and hence Dep.2 and Dep.3, are not considered in [3].

For lack of space, in this short version some technical definitions and all proofs have been suppressed.

2 π-calculus

We briefly review the syntax of the π-calculus, and the definition of open bisimulation. Letters $a, b, \ldots, x, y, \ldots$ range over the infinite ordered set $\mathcal{N}$ of names, and K over the set of process identifiers. The class of processes is built from the operators of inaction, prefixing, matching, parallel composition, restriction, sum, and recursion; a prefix can be a silent prefix, an input, a free output, or a bound output:

$$P, Q := \mathbf{0} \mid \alpha.P \mid [a=b]P \mid P \mid Q \mid \nu a\, P \mid P + Q \mid K\langle \tilde{a} \rangle$$
$$\alpha := \tau \mid a(b) \mid \overline{a}b \mid \overline{a}(b).$$

Each identifier K has an associated arity and a definition of the form $K \stackrel{\text{def}}{=} (\tilde{a})P$. We give sum and parallel composition the lowest syntactic precedence among the operators. In $a(b).P$, $\nu b\, P$, and $\overline{a}(b).P$, all free occurrences of name b in

P are bound. *Free names* (fn), *bound names* (bn) and *names* (n) of processes and prefixes, name substitutions, and alpha conversion are defined as expected. The *extruded names* of an action α, written $\mathrm{en}(\alpha)$, are its bound names if α is a bound output, are the empty set otherwise. We use σ to range over substitutions. Application of a substitution σ to a process P and to an action α are written $P\sigma$ and $\alpha\sigma$, respectively (in $\alpha\sigma$, the bound names of α are not touched).

Conditions are finite conjunctions of matching, like $[a=b][c=d]$. We use M and N to range over conditions, and $\mathrm{n}(M)$ for the names which appear in M. We write $M \rhd N$ if M implies N, i.e., the equalities in M imply those in N; $M \lhd\rhd N$ is a shortcut for "$M \rhd N$ and $N \rhd M$", whereas $M \not\lhd\rhd N$ is a shortcut for "$M \rhd N$ but not $N \rhd M$". Notice that, since names are ordered, there are canonical forms for conditions, which are unique up to $\lhd\rhd$. Given a matching M, we denote by σ_M the substitution which selects the minimal representative out of each equivalence class on names induced by M, i.e., $\sigma_M(a) = \min\{b \mid M \rhd [a=b]\}$.

A *distinction* is a finite symmetric irreflexive relation on $\mathcal{N}$, which expresses permanent inequalities on names, i.e., if (a,b) is in the distinction, then a must be kept separate from b. We use D to range over distinctions and $\mathrm{n}(D)$ for the names which are mentioned in D. A substitution σ *respects* a distinction D if $(a,b) \in D$ implies $\sigma(a) \neq \sigma(b)$; in this case we write $D\sigma$ for the distinction $\{(\sigma(a),\sigma(b)) \mid (a,b) \in D\}$. Similarly, a matching M *respects* a distinction D if σ_M respects D. Sometimes, in the expressions defining distinctions we shall avoid to give all symmetric pairs. If N is a set of names, then $D - N$ is the distinction $\{(a,b) \in D \mid a,b \notin N\}$ and $D \cap N$ is the distinction $\{(a,b) \in D \mid a,b \in N\}$.

In the symbolic transition system for open bisimulation [11], transitions have the form $P \overset{M,\alpha}{\rightsquigarrow} P'$, where M represents the minimal condition on names required by P to perform that action. For lack of space, we omit the transition rules, which can be found, for instance, in [11] or [12].

It will be convenient to use a transition system $P \xrightarrow{M,\alpha} P'$ in which M is in canonical form and substitution σ_M has already been applied to action α and derivative P':

$$\frac{P \overset{N,\alpha}{\rightsquigarrow} P' \qquad M \text{ is the canonical form for } N}{P \xrightarrow{M,\alpha\sigma_M} P'\sigma_M}\ \mathrm{bn}(\alpha) \cap \mathrm{fn}(P) = \emptyset$$

Other notations. If S is a set, then $\wp(S)$ is the powerset of S. If $\mathcal{R}$ is a relation, then we sometimes write $h\ \mathcal{R}\ k$ to mean $(h,k) \in \mathcal{R}$.

Definition 1 (open bisimulation [11]). *Open bisimulation* is the largest set $\{\sim_D\}_D$ of symmetric process relations s.t. for all D and $P \sim_D Q$:

- whenever $P \xrightarrow{M,\alpha} P'$, with $\mathrm{bn}(\alpha) \cap (\mathrm{fn}(Q) \cup \mathrm{n}(D)) = \emptyset$ and M respects D, there are N, β, and Q' such that $Q \xrightarrow{N,\beta} Q'$ and
  - $M \rhd N$, $\alpha = \beta\sigma_M$, and
  - $P' \sim_{D'} Q'\sigma_M$, for $D' \overset{\mathrm{def}}{=} D\sigma_M \cup (\mathrm{en}(\alpha) \times \mathrm{fn}(P,Q))$.

3 Making distinctions local to processes

The goal of this section is to make the indexing distinctions of open bisimulation local to processes. A *constrained process* is a pair $\langle P, D\rangle$, where P is a process and D is a distinction with $\mathrm{n}(D) \subseteq \mathrm{fn}(P)$. The requirement on names is used to keep distinctions "small" and to reduce the number of free names (in the algorithm of Section 6, it will allow us a better re-use of names). The set $\mathcal{CP}$ of constrained processes is ranged over by A, B. Substitutions, free names, bound names are extended to constrained processes as expected. If $A \stackrel{\text{def}}{=} \langle P, D\rangle$, then $\nu a\, A$ abbreviates $\langle \nu a\, P, D - \{a\}\rangle$. We call relations on constrained processes *$\mathcal{CP}$-relations*; they are ranged over by $\mathcal{R}$.

We use $D^M_{P,\alpha}$ to denote the distinction $D\sigma_M \cup (\mathrm{en}(\alpha) \times \mathrm{fn}(P))$. Transitions for constrained processes are defined from those for processes:

$$\frac{P \xrightarrow{M,\alpha} P' \qquad M \text{ respects } D}{\langle P, D\rangle \xrightarrow{M,\alpha} \langle P', D^M_{P,\alpha} \cap \mathrm{fn}(P')\rangle}$$

It makes sense to compare only constrained processes whose distinctions are compatible, that is, identical on common names.

Definition 2. Constrained processes $\langle P, D\rangle$ and $\langle Q, E\rangle$ are *compatible*, written $\langle P, D\rangle \Downarrow \langle Q, E\rangle$, if $D \cap \mathrm{fn}(Q) = E \cap \mathrm{fn}(P)$.

A *compatible* $\mathcal{CP}$-relation is a $\mathcal{CP}$-relation whose pairs are compatible constrained processes.

Theorem 3. *Let $\sim$ be the largest symmetric compatible $\mathcal{CP}$-relation s.t. whenever $\langle P, D\rangle \sim \langle Q, E\rangle$ and $\langle P, D\rangle \xrightarrow{M,\alpha} \langle P', D'\rangle$, with $\mathrm{bn}(\alpha) \cap \mathrm{fn}(Q) = \emptyset$, there are N, β, Q' and E' such that $\langle Q, E\rangle \xrightarrow{N,\beta} \langle Q', E'\rangle$ and*

- *$M \rhd N$, $\alpha = \beta\sigma_M$, and*
- *$\langle P', D'\rangle \sim \langle Q'\sigma_M, (D^M_{P,\alpha} \cup E^M_{Q,\alpha}) \cap \mathrm{fn}(Q'\sigma_M)\rangle$.*

Then $\langle P, D\rangle \sim \langle Q, E\rangle$ implies $P \sim_{D\cup E} Q$ and, vice versa, $P \sim_D Q$ implies $\langle P, D \cap \mathrm{fn}(P)\rangle \sim \langle Q, D \cap \mathrm{fn}(Q)\rangle$.

The proof of Theorem 3 uses a few lemmas for manipulating distinctions, among which the following strong-narrowing law for open bisimulation:

$$P \sim_D Q \text{ implies } P \sim_E Q \text{ for } E \stackrel{\text{def}}{=} (D \cap \mathrm{fn}(P)) \cup (D \cap \mathrm{fn}(Q)).$$

This law, which has a delicate proof, strengthens the standard narrowing law for open bisimulation, where $E \stackrel{\text{def}}{=} D \cap \mathrm{fn}(P, Q)$.

A $\mathcal{CP}$-relation $\mathcal{R}$ is *weak transitive* if $A_1\ \mathcal{R}\ A_2$, $A_2\ \mathcal{R}\ A_3$, and $A_1 \Downarrow A_3$ imply $A_1\ \mathcal{R}\ A_3$.

Proposition 4. *Relation $\sim$ is not transitive, but is weak transitive.*

In the remainder of the paper, with some abuse of notation, we call *open bisimilarity* the relation $\sim$ on constrained processes defined in Theorem 3. When clear from the context, we may call constrained processes simply *processes*.

4 Non-redundant transitions

The goal of this section is to remove the dependency called Dep.2 in the Introduction from the characterisation of open bisimulation in the previous section. We recall that this dependency is caused by "redundant" transitions. Roughly, a transition $A \xrightarrow{M,\alpha} A'$ is redundant for open bisimulation if there is another transition $A \xrightarrow{N,\beta} A''$ which has a strictly weaker condition N and s.t., when M holds, α is equal to β and A' is bisimilar to A''. Redundant transitions can be defined for an arbitrary relation $\mathcal{R}$ in place of bisimilarity.

Definition 5. A transition $\langle P, D\rangle \xrightarrow{M,\alpha} \langle P', D'\rangle$ is *redundant* for a $\mathcal{CP}$-relation $\mathcal{R}$ if there is a transition $\langle P, D\rangle \xrightarrow{N,\beta} \langle P'', D''\rangle$ such that:

- $M \not\triangleleft\triangleright N$, $\alpha = \beta\sigma_M$, and
- $\langle P', D'\rangle \; \mathcal{R} \; \langle P''\sigma_M, D^M_{P,\alpha} \cap \mathrm{fn}(P''\sigma_M)\rangle$.

A transition $\langle P, D\rangle \xrightarrow{M,\alpha} \langle P', D'\rangle$ is *non-redundant* for a $\mathcal{CP}$-relation $\mathcal{R}$, written $\langle P, D\rangle \xrightarrow{M,\alpha} \langle P', D'\rangle \in \mathrm{nr}(\mathcal{R})$, if it is not redundant for $\mathcal{R}$.

Theorem 6. *Relation $\sim$ coincides with the largest symmetric compatible $\mathcal{CP}$-relation $\mathcal{R}$ s.t. if $A \; \mathcal{R} \; B$ then:*

- *whenever $A \xrightarrow{M,\alpha} A' \in \mathrm{nr}(\mathcal{R})$ and $\mathrm{bn}(\alpha) \cap \mathrm{fn}(B) = \emptyset$, there is B' such that $B \xrightarrow{M,\alpha} B' \in \mathrm{nr}(\mathcal{R})$ and $A' \; \mathcal{R} \; B'$.*

5 Active names and the iterative approach

We now will make the choice of bound names of matching transitions local to processes (i.e., removing the dependency called Dep.1 in the Introduction). A local choice cannot be based on the free names because bisimilar processes may have different sets of free names. In place of free names, we shall use the *active names*, which are the same in bisimilar processes. The active names of a process are the smallest subset of free names which affect its behaviour. For instance, a is active in $a(b).\,\mathbf{0}$ and $[a=b]\overline{b}c.\,\mathbf{0}$, but it is not in $\nu b\,(\overline{b}a.\,\mathbf{0})$ and $[a=b]\overline{a}c.\,\mathbf{0} + \overline{b}c.\,\mathbf{0}$. As for non-redundant transitions, it is convenient to define active names on a generic $\mathcal{CP}$-relation.

Definition 7. For a $\mathcal{CP}$-relation $\mathcal{R}$, the function $\mathrm{an}_{\mathcal{R}} : \mathcal{CP} \to \wp(\mathcal{N})$, mapping a constrained process onto the set of its *active names w.r.t.* $\mathcal{R}$, is the least fixed-point of the monotone function $\psi : (\mathcal{CP} \to \wp(\mathcal{N})) \to (\mathcal{CP} \to \wp(\mathcal{N}))$ defined thus:

$$\psi(f)(A) = \bigcup_{\{M,\alpha,A' \mid A \xrightarrow{M,\alpha} A' \in \mathrm{nr}(\mathcal{R})\}} \mathrm{fn}(M,\alpha) \cup (f(A') - \mathrm{bn}(\alpha)).$$

If $x \in \mathrm{an}_{\mathcal{R}}(A)$, then we say that name x is *active* w.r.t. $\mathcal{R}$ in process A; otherwise we say that x is *inactive* w.r.t. $\mathcal{R}$ in A.

That is, $\mathrm{an}_{\mathcal{R}}$ is the least function which satisfies the condition: whenever $A \xrightarrow{M,\alpha} A' \in \mathrm{nr}(\mathcal{R})$, it holds that $\mathrm{fn}(M,\alpha) \cup (\mathrm{an}_{\mathcal{R}}(A') - \mathrm{bn}(\alpha)) \subseteq \mathrm{an}_{\mathcal{R}}(A)$.

Proposition 8. *$x \in \mathrm{an}_{\sim}(A)$ iff $A \not\sim \nu x\, A$.*

In the *normalised* transitions below, the bound name in a transition of a process A is imposed to be the first inactive name in A; since the first inactive name may also occur free, in rule `Norm2` its free occurrences are redenominated to avoid accidental identifications.

Definition 9. The *normalised transitions for a $\mathcal{CP}$-relation $\mathcal{R}$*, which are of the form $A \overset{M,\alpha}{\longmapsto}_{\mathcal{R}} A'$, are defined from the two following inference rules, where $v \stackrel{\mathrm{def}}{=} \min\{\mathcal{N} - \mathrm{an}_{\mathcal{R}}(A)\}$ and $y \stackrel{\mathrm{def}}{=} \min\{\mathcal{N} - \mathrm{fn}(A)\}$:

$$\texttt{Norm1}\ \frac{A \xrightarrow{M,\alpha} A' \qquad \mathrm{bn}(\alpha) \subseteq \{v\}}{A \overset{M,\alpha}{\longmapsto}_{\mathcal{R}} A'}$$

$$\texttt{Norm2}\ \frac{A \xrightarrow{M,\alpha} A' \qquad \mathrm{bn}(\alpha) = \{y\} \qquad y \neq v}{A \overset{M,\beta}{\longmapsto}_{\mathcal{R}} A'\{v/y\,y/v\}} \text{ where } \beta \stackrel{\mathrm{def}}{=} \begin{cases} a(v) & \text{if } \alpha = a(y) \\ \overline{a}(v) & \text{if } \alpha = \overline{a}(y) \end{cases}$$

The two inference rules `Norm1-2` define an injective mapping from normalised transitions to plain transitions, because each normalised transition is inferred from a unique plain transition. Thus, if $A \xrightarrow{M,\alpha} A'$ is the image of $A \overset{M,\beta}{\longmapsto}_{\mathcal{R}} A''$ under the injection, we write $A \overset{M,\beta}{\longmapsto}_{\mathcal{R}} A'' \in \mathrm{nr}(\mathcal{R})$ if $A \xrightarrow{M,\alpha} A' \in \mathrm{nr}(\mathcal{R})$.

Definition 10. Let $\mathcal{R}$ be a $\mathcal{CP}$-relation. Function $\Psi_{\mathcal{R}} : \wp(\mathcal{CP} \times \mathcal{CP}) \to \wp(\mathcal{CP} \times \mathcal{CP})$ is defined thus: $(A,B) \in \Psi_{\mathcal{R}}(\mathcal{S})$ if

- $A\ \mathcal{R}\ B$;
- $\mathrm{an}_{\mathcal{R}}(A) = \mathrm{an}_{\mathcal{R}}(B)$;
- whenever $A \overset{M,\alpha}{\longmapsto}_{\mathcal{R}} A' \in \mathrm{nr}(\mathcal{R})$, there is B' such that $B \overset{M,\alpha}{\longmapsto}_{\mathcal{R}} B' \in \mathrm{nr}(\mathcal{R})$ and $A'\ \mathcal{S}\ B'$; and the vice versa, with the role of A and B exchanged.

For each $\mathcal{R}$, function $\Psi_{\mathcal{R}}$ is monotone, so it has a greatest fixed-point and we can define a function $\Phi : \wp(\mathcal{CP} \times \mathcal{CP}) \to \wp(\mathcal{CP} \times \mathcal{CP})$, which maps a relation $\mathcal{R}$ onto the greatest-fixed-point of $\Psi_{\mathcal{R}}$.

Functional Φ appears suited to extracting a partition refinement algorithm because derivatives and transitions of processes are computed locally. Notice that in the definition of Φ, clause $A \Downarrow B$ (which appears in the definition of efficient open bisimulation) has been omitted because relation $\Downarrow$ is not transitive, as witnessed by processes

$$A \stackrel{\mathrm{def}}{=} \langle \nu x\, (\overline{x}a.\,\overline{a}b.\,\mathbf{0}), \emptyset \rangle \qquad B \stackrel{\mathrm{def}}{=} \langle \mathbf{0}, \emptyset \rangle \qquad C \stackrel{\mathrm{def}}{=} \langle \nu x\, (\overline{x}a.\,\overline{a}b.\,\mathbf{0}), (a,b) \rangle$$

where $A \Downarrow B$ and $B \Downarrow C$, but $A \Downarrow C$ does not hold. If we added clause $A \Downarrow B$, then function Φ would not preserve transitivity, and hence we could not use Φ to define a partition refinement algorithm.

Unfortunately, Φ is not monotone. Therefore, we do not know whether Φ has a maximal fixed-point and, even if it existed, we could not use Tarsky's theorem to compute it. As a consequence, to obtain an algorithmic characterisation of open bisimulation in terms of Φ, one has to provide a specific proof that the iterated applications of Φ on the universal relation $\mathcal{CP} \times \mathcal{CP}$ are convergent, and that the limit contains open bisimulation. We define:

$$\Phi^0 \stackrel{\text{def}}{=} \mathcal{CP} \times \mathcal{CP}$$
$$\Phi^i \stackrel{\text{def}}{=} \Phi(\Phi^{i-1}) \text{ if } i \text{ is an ordinal successor}$$
$$\Phi^i \stackrel{\text{def}}{=} \bigcap_{j<i} \Phi^j \text{ if } i \text{ is an ordinal limit.}$$

Theorem 11. *Let* $\sim_{\text{alg}} \stackrel{\text{def}}{=} \lim_i \Phi^i$. *Then* $\sim$ *coincides with* $\sim_{\text{alg}} \cap \Downarrow$.

Corollary 12. $P \sim_D Q$ *if and only if* $\langle P, D \cap \text{fn}(P)\rangle \sim_{\text{alg}} \langle Q, D \cap \text{fn}(Q)\rangle$.

Each iteration of the algorithm in the next section precisely corresponds to an application of function Φ.

6 The algorithm

The algorithm we propose for open bisimulation is based on the characterisation in Corollary 12. The main steps are sketched in Table 1. A partition on processes can be viewed as a relation in which all processes in the same class are related; in this way, we can talk of non-redundant transitions, active names and normalised transitions w.r.t. a partition. Some comments on the steps in the table:

1 Generate the saturated state graphs (S_P, T_P) and (S_Q, T_Q) for processes $\langle P, D \cap \text{fn}(P)\rangle$ and $\langle Q, D \cap \text{fn}(Q)\rangle$.
2 Initialize $\mathcal{P}$ to be the singleton partition on $S_P \cup S_Q$ (i.e., all processes in the same class).
3 Repeat the following steps until partition $\mathcal{P}$ becomes stable:
  3.1 Set *NonRed* to be the subset of transitions in $T_P \cup T_Q$ which are non-redundant for $\mathcal{P}$.
  3.2 Compute the active names w.r.t. $\mathcal{P}$, for each process in $S_P \cup S_Q$.
  3.3 If necessary, refine the partition $\mathcal{P}$ so that processes in the same class have the same set of active names.
  3.4 Compute the normalised transitions for $\mathcal{P}$ generated by the transitions in *NonRed*.
  3.5 Apply the Paige and Tarjan refinement algorithm [10] on partition $\mathcal{P}$ using, as transitions, the normalised ones computed in Step 3.4. Redefine $\mathcal{P}$ to be the resulting partition.
4 Check if $\langle P, D \cap \text{fn}(P)\rangle$ and $\langle Q, D \cap \text{fn}(Q)\rangle$ are in the same class.

Table 1. Schema of the algorithm to check $P \sim_D Q$

Step 1: The *saturated state graph* of a process A_0 is the pair (S,T) where S and T are, respectively, the minimal set of constrained processes and of transitions between processes in S such that $A_0 \in S$ and such that S and T are closed under the following operations:

Sat-trans if $A \in S$ and $A \xrightarrow{M,\alpha} A'$ with $\mathrm{bn}(\alpha) \subseteq \{\min\{\mathcal{N} - \mathrm{fn}(A)\}\}$, then $A' \in S$ and $A \xrightarrow{M,\alpha} A' \in T$;

Sat-nonred if $\langle P, D\rangle \xrightarrow{M,\alpha} \langle P', D'\rangle \in T$ and $\langle P, D\rangle \xrightarrow{N,\beta} \langle P'', D''\rangle \in T$ with $M \rhd N$ and $\alpha = \beta\sigma_M$, then $\langle P''\sigma_M, D^M_{P,\alpha} \cap \mathrm{fn}(P''\sigma_M)\rangle \in S$;

Sat-bunch Let $y \stackrel{\mathrm{def}}{=} \min\{\mathcal{N} - \mathrm{fn}(A)\}$; if $A \xrightarrow{M,\alpha} A' \in T$ with $\mathrm{bn}(\alpha) = \{y\}$, then $A'\{v/y\, y/v\} \in S$, for all $v < y$ (where $<$ is the strict order assumed on $\mathcal{N}$).

Sat-nonred and **sat-bunch** are necessary, respectively, for the run-time computation of non-redundant transitions and for the targets of normalised transitions. Note that **sat-trans** requires a single instantiation of bound names of actions. In **sat-bunch**, v is *strictly* below $\min\{\mathcal{N} - \mathrm{fn}(A)\}$, since the latter name, which is surely inactive for A, has already been considered in **sat-trans**.

Step 3: Each cycle corresponds to an application of function Φ of Section 5.

Step 3.1: Following Definition 5, to compute the non-redundant transitions quickly, for each transition $\langle P, D\rangle \xrightarrow{M,\alpha} \langle P', D'\rangle$ we can keep a list of the processes $\langle P''\sigma_M, D^M_{P,\alpha} \cap \mathrm{fn}(P''\sigma_M)\rangle$ such that $\langle P, D\rangle \xrightarrow{N,\beta} \langle P'', D''\rangle$ with $M \rhd N$ and $\alpha = \beta\sigma_M$; then the given transition is non-redundant for $\mathcal{P}$ if and only if none of the processes in the list is in the equivalence class of $\langle P', D'\rangle$.

Step 3.2: Following Definition 7, the active names can be efficiently computed via a transitive-closure algorithm (the non-redundant transitions have already been computed in Step 3.1).

Step 3.4: The normalised transitions are generated applying inference rules `Norm1` and `Norm2` (Definition 9) to each transition in *NonRed*. (The derivatives of normalised transitions are in $S_P \cup S_Q$, so no new process needs to be added.)

The algorithm terminates if the saturated state spaces, produced from the input processes in Step 1, are finite. This is the case for finite-control processes.[1].

The algorithm can be used to produce the minimal realisation of a process P w.r.t. a distinction D — for this case it suffices to generate only the saturated state space of $\langle P, D \cap \mathrm{fn}(P)\rangle$ in Step 1. If we take processes with normalised transitions only (normalised transitions for processes are defined as for constrained processes — just replace A with P and A' with P' in Definition 9), then the process returned by the algorithm (i.e., the one extracted from the final partition, where transitions are those computed in the last execution of Step 3.4) is

[1] Some garbage collection of restrictions is needed, i.e., if $x \notin \mathrm{fn}(P)$ then $\nu x\, P$ should be replaced by P.

minimal in the number of states, normalised transitions, and the number of free names among all processes in the relation $\sim_D$ with P.

7 Conclusions

We presented various characterisations of open bisimulation. From the last one, we have extracted a partition refinement algorithm. It can be used on finite control processes for checking bisimilarity and for producing minimal realisations.

Complexity. Paige-Tarjan algorithm has a worst-case running time $O(m \log n)$, on the number m of transitions and n of states; in our case, m and n are the number of transitions and of states obtained after initialisation (Step 1 in Table 1). The Paige-Tarjan routine may be applied several times (Step 3 in Table 1), but at most it runs n times, since at each iteration at least one split of the partition is made. Hence, we get a worst case running time $O(mn \log n)$.

However, the algorithm is exponential w.r.t. the syntactic length of the processes. The exponentiality is caused by the expansion of the parallel component of processes, as in CCS, and by value communication, as in data-dependent programs [5]. Surprisingly, the degree of exponentiation in the π-calculus is similar to that of CCS [3, 8]. Against this exponential bounds, the possibility of minimising process representations, offered by the algorithm, becomes important: As it happens in CCS, large-scale examples become tractable if process subcomponents are first minimised.

Active names and non-redundant transitions. The algorithm presented makes no attempts at uncovering active names and non-redundant transitions in initialisation. This may cause a large addition of states in the saturation procedure (operations sat-nonred and sat-bunch in Step 1) and a high number of iterations of the Paige-Tarjan routine.

However, a quick analysis of the processes could often produce reasonable estimates of active names and non-redundant transitions. For instance, all transitions with a true (i.e., empty) condition are non-redundant, and all free names in their labels are active. We believe that this direction can lead to significant improvements.

In non-trivial processes, like those used in the specification and the implementation of the handover protocol in the GSM Public Land Mobile Network [9], all free names are active, and all transitions are non-redundant. It would be useful to find syntactic characterisations of classes of processes with this property. An example is the fragment of language without parallel composition and matching. One could then envisage a two-speed algorithm, the first speed (faster) to be used when active names and non-redundant transitions can be quickly computed in initialisation. (Indeed, the speeds could even be three, the intermediate one to be used when active names are known but non-redundant transitions are not.) Again, the applicability of the first speed can be enhanced by first minimising the representation of process subcomponents whose active names and non-redundant transitions are hard to compute.

Extensions. The algorithm can be extended to the polyadic π-calculus (where tuples of names are communicated). It can also be adapted to early and late bisimulations, and to weak and branching bisimulations. In the case of early and late bisimulations, the algorithm would become simpler, since Dep.2 and Dep.3 are absent, but it would be less efficient, due to the heavy use of names in the input clause of these bisimulations [7, 11].

Implementations. We plan to produce an implementation of the algorithm and to study its integration with the MWB and other tool sets like JACK [1] or AUTO/GRAPH [2].

Minimisation. The algorithm performs minimisation of processes on the normalised transition system. Roughly, normalised means that the bound name of transitions is forced to be the first inactive in the source process.

It would be interesting to see whether there are minimal forms and minimisation algorithms with more relaxed conditions on the choice of bound names. The challenge is that the minimal forms must be canonical, i.e., syntactically identical for bisimilar processes.

Acknowledgements. We are very grateful to Ugo Montanari, for several helpful conversations especially in the initial stages of development of this work. We also benefited from comments by Amar Bouali, Robert de Simone and the anonymous referees.

References

1. A. Bouali, S. Gnesi and S. Larosa. The Integration Project for the JACK Environment. *Bullettin of the EATCS* 54, 1994.
2. G. Boudol, V. Roy, R. de Simone and D. Vergamini. Process algebra and Systems of Communicating Processes. In *Proc. Automatic Verification Methods for Finite State Systems*, LNCS 407, 1989.
3. M. Dam. On the decidability of process equivalences for the π-calculus. SICS Research Report RR:94-20, 1994.
4. J.-C. Fernandez and L. Mounier. "On-the-fly" verification of behavioural equivalences and preorders. In *Proc. CAV'91*, LNCS 575. Springer Verlag, 1994.
5. B. Jonsson and J. Parrow. Deciding bisimulation equivalences for a class of non-finite-state programs. *Information and Computation*, 107:272–302, 1993.
6. P. C. Kanellakis and S. A. Smolka. CCS expressions, finite state processes, and three problems of equivalence. *Information and Computation*, 86:43–68, 1990.
7. R. Milner, J. Parrow and D. Walker. A calculus of mobile processes (parts I and II). *Information and Computation*, 100:1–77, 1992.
8. U. Montanari and M. Pistore. Checking bisimilarity for finitary π-calculus. In *Proc. CONCUR'95*, LNCS 962. Springer Verlag, 1995.
9. F. Orava and J. Parrow. An algebraic verification of a mobile network. *Formal Aspects of Computing*, 4:497–543, 1992.
10. R. Paige and R. E. Tarjan. Three partition refinement algorithms. *SIAM Journal on Computing*, 16(6):973–989, 1987.
11. D. Sangiorgi. A theory of bisimulation for π-calculus. In *Proc. CONCUR'93*, LNCS 715. Springer Verlag, 1993.
12. B. Victor and F. Moller. The Mobility Workbench — A tool for the π-calculus. In *Proc. CAV'94*, LNCS 818. Springer Verlag, 1994.

Polynomial Time Algorithms for Testing Probabilistic Bisimulation and Simulation

Christel Baier

Fakultät für Mathematik & Informatik
Universität Mannheim, 68131 Mannheim, Germany
baier@pi1.informatik.uni-mannheim.de

Abstract. Various models and equivalence relations or preorders for probabilistic processes are proposed in the literature. This paper deals with a model based on labelled transition systems extended to the probabalistic setting and gives an $\mathcal{O}(n^2 \cdot m)$ algorithm for testing probabilistic bisimulation and an $\mathcal{O}(n^5 \cdot m^2)$ algorithm for testing probabilistic simulation where n is the number of states and m the number of transitions in the underlying probabilistic transition systems.

1 Introduction

Transition systems have proved to be very useful for modelling concurrent processes. A variety of widely accepted equivalence relations and preorders for such systems support the use of transition systems for the design and verification of concurrent systems. In this context, testing equivalences and preorders become important and have been studied e.g. in [3, 4, 8, 11, 17]. For instance, (strong) bisimulation can be decided in time $\mathcal{O}(m \cdot \log n)$ [22], weak bisimulation in time $\mathcal{O}(n^3)$ [3, 17] and strong and weak simulation in time $\mathcal{O}(n^4 \cdot m)$ [4] where n is the number of states and m the number of transitions of the underlying transition system.

In recent years, many researchers have focussed on reasoning about probabilistic distributed transition systems, see e.g. [15, 18, 23, 25, 28, 29, 30]. A lot of work has been done to extend those models and methods which have been successful for the non-probabilistic case to probabilistic systems. In the literature a variety of models for probabilistic processes has been proposed, most of them based on transition systems. Two kinds of models can be distinguished: on the one hand, models that replace the concept of non-determinism by probabilistic choice, e.g. [5, 13, 18, 26, 28], on the other hand, models which distinguish between non-deterministic and probabilistic choice, e.g. [6, 12, 16, 25, 27, 30]. As pointed out in [27], the distinction between non-determinism and probabilistic choice is essential for concurrent probabilistic systems since some states of a concurrent system are inherently non-deterministic.

Several kinds of equivalences and preorders for probabilistic processes are proposed: [5, 16, 30, 28] consider testing preorders for probabilistic processes. Probabilistic bisimulation for processes whose behaviour are described by "determinsitic" probabilistic transition systems are introduced in [18]. [25] extends

probabilistic bisimulation to non-deterministic probabilistic transition systems and defines a notion of probabilistic simulation which refines Milners notion of a simulation for non-probabilistic transition systems [21]. [15] defines an alternative notion of a simulation which relates a process given by a probabilistic transition system and a specification which is given by a "generalized" probabilistic transition system.

Various authors presented model-checking-algorithms for the verification of probabilistic processes e.g. [1, 6, 13, 14, 19, 23, 24, 27]. But – as far as the author knows – algorithms for testing probabilistic (bi-)simulation are missing until now. In this paper we present algorithms for testing probabilistic simulation and bisimulation in the sense of [18, 25]. The main idea of testing simulation is to reduce the question of whether a state s of a probabilistic transition system simulates a state s' to a maximum flow problem in a suitable network. Using the $\mathcal{O}(n^3)$ algorithm of Malhotra et al [20] to determine the maximum flow we get an $\mathcal{O}(n^5 \cdot m^2)$ algorithm for testing probabilistic simulation where n is the number of states and m the number of transitions. The idea for testing bisimulation is similar to the non-probabilistic case [17, 22]: the algorithm for testing probabilistic bisimulation is based on refinement steps which split a given partition of states into a finer one. The resulting time complexity of our algorithm is $\mathcal{O}(n^2 \cdot m)$.

The remainder of the paper is organized as follows: Section 2 introduces the notions of a probabilistic transition system, probabilistic bisimulation and simulation. Section 3 presents the algorithm for testing probabilistic simulation, section 4 the algorithm for deciding probabilistic bisimulation. Section 5 contains some concluding remarks.

2 Probabilistic transition systems

In this section we present the notions of probabilistic transition systems, bisimulation and simulation. Our model of probabilistic transition systems is closely related to those of [16, 30], to the "simple probabilistic automata" of [25] and "concurrent Markov chains" considered e.g. in [6, 12, 27].

A distribution on a finite set S is a function $\mu : S \to [0,1]$ such that $\sum_{s \in S} \mu(s) = 1$. We extend a distribution μ to a function which assigns to each subset U of S the probability $\mu(U) = \sum_{s \in U} \mu(s)$. In what follows, we suppose Act to be a nonempty and finite set of actions. A *probabilistic transition system* is a pair $\mathcal{S} = (S, \to)$ where S is a finite set of states and $\to$ a finite transition relation, i.e. $\to$ is a finite subset of $S \times \mathrm{Act} \times \mathcal{D}(S)$ where $\mathcal{D}(S)$ denotes the set of distributions on S. We write $s \xrightarrow{\alpha} \mu$ instead of $(s, \alpha, \mu) \in \to$. Informally, the outgoing transitions $s \xrightarrow{\alpha} \mu$ represent the non-deterministic alternatives in the state s. It is convenient to suppose that a scheduler resolves the non-deterministic choices. A transition $s \xrightarrow{\alpha} \mu$ asserts that in state s the action α can be performed and with probability $\mu(t)$ the state t is reached afterwards, i.e. every transition represents a probabilistic choice. (Finite-state) probabilistic processes can be described by a probabilistic transition system and

an initial state (or alternatively a distribution on the possible initial states). In what follows a transition system means a probabilistic transition system. By a non-probabilistic transition system we mean a transition system where for all transitions $s \xrightarrow{\alpha} \mu$: there is a state t with $\mu(t) = 1$. Following [18, 25] we define (probabilistic) bisimulation and simulation:

Definition 1. Let $(S, \rightarrow)$ be a transition system. A *bisimulation* on S is an equivalence relation R on S such that for all $(s, s') \in R$: If $s \xrightarrow{\alpha} \mu$ then there is a transition $s' \xrightarrow{\alpha} \mu'$ with $\mu(A) = \mu'(A)$ for all $A \in S/R$. Here S/R denotes the set of equivalence classes w.r.t. R. Two states s_1 and s_2 are called *bisimilar* (denoted by $s_1 \sim s_2$) iff there exists a bisimulation which contains (s_1, s_2).

An alternative description of bisimulation is based on weight functions for distributions [15]:

Definition 2. Let S be a finite set, $R \subseteq S \times S$ and $\mu, \mu' \in \mathcal{D}(S)$. A *weight function* for (μ, μ') w.r.t. R is a function $\delta : S \times S \rightarrow [0, 1]$ which satisfies:

1. For all $s, s' \in S$: $\sum_{s' \in S} \delta(s, s') = \mu(s), \quad \sum_{s \in S} \delta(s, s') = \mu'(s')$
2. If $\delta(s, s') > 0$ then $(s, s') \in R$.

Let $(S, \rightarrow)$ be a transition system and R an equivalence relation on S. Then R is a bisimulation if and only if for all $(s, s') \in R$: Whenever $(s, s') \in R$ and $s \xrightarrow{\alpha} \mu$ then there exists a transition $s' \xrightarrow{\alpha} \mu'$ and a weight function for (μ, μ') w.r.t. R. Intuitively, the weight function δ shows how to split the probabilities $\mu(s)$ and $\mu'(s')$, s, $s' \in S$, so that the relation R is preserved: we "combine" the $\delta(s, s')$-part of s and s'. As in the non-probabilistic case, simulation is defined as "uni-directional bisimulation": in the above characterization of bisimulation we drop the requirement that R is an equivalence relation.

Definition 3. Let $(S, \rightarrow)$ be a transition system. A *simulation* for $(S, \rightarrow)$ is a subset R of $S \times S$ such that for all $(s, s') \in R$: Whenever $s \xrightarrow{\alpha} \mu$ then there exists a transition $s' \xrightarrow{\alpha} \mu'$ and a weight function δ for (μ, μ') w.r.t. R. We say s *implements* s' (denoted by $s \sqsubseteq s'$) iff there exists a simulation which contains (s, s').

In the non-probabilistic case this notion of a simulation agrees with Milners notion of a simulation [21]. This is because the only weight function for (μ, μ') where μ, μ' are distributions with $\mu(s) = \mu'(s') = 1$ is $\delta(u, u') = 0$ if $(u, u') \neq (s, s')$ and $\delta(s, s') = 1$. Hence if $(S, \rightarrow)$ is a non-probabilistic transition system and $R \subseteq S \times S$ then R is a simulation in the sense of Definition 3 if and only if R is a simulation in the sense of Milner. It is clear that $\sqsubseteq$ is a preorder whose kernel $\sim_{\text{sim}} = \sqsubseteq \cap \sqsubseteq^{-1}$ is coarser than bisimulation equivalence, i.e. $s \sim s'$ implies $s \sim_{\text{sim}} s'$. As in the non-probabilistic case, $\sim_{\text{sim}}$ does not coincide with bisimulation.

Example 4. Let $(S, \rightarrow)$ be the transition system where $S = \{s_0, \ldots, s_5\}$ and

$$s_0 \xrightarrow{\alpha} \mu,\ s_5 \xrightarrow{\alpha} \mu',\ s_2 \xrightarrow{\beta} \rho,\ s_3 \xrightarrow{\beta} \rho,\ s_3 \xrightarrow{\gamma} \rho,\ s_4 \xrightarrow{\alpha} \rho.$$

Here $\rho(s_1) = 1$, $\mu(s_1) = \mu(s_2) = \mu(s_3) = 1/3$ and $\mu'(s_1) = 1/4$, $\mu'(s_3) = 17/24$ and $\mu'(s_4) = 1/24$. Then

$$s_1 \sqsubseteq s_2 \sqsubseteq s_3,\ s_1 \sqsubseteq s_4 \sqsubseteq s_0 \sqsubseteq s_5.$$

The weight function δ for (μ, μ') w.r.t. $\sqsubseteq$ is given by: $\delta(s_1, s_1) = 1/4$, $\delta(s_1, s_3) = \delta(s_1, s_4) = 1/24$, $\delta(s_2, s_3) = \delta(s_3, s_3) = 1/3$. □

The result of Milner [21] that in every (image-)finite non-probabilistic transition system bisimulation can be approximated by "finitary bisimulation" carries over to the probabilistic case. If $(S, \rightarrow)$ is a transition system then we define inductively equivalence relations $\sim_n$ on S: $\sim_0 = S \times S$ and $s \sim_{n+1} s'$ if and only if: Whenever $s \xrightarrow{\alpha} \mu$ then there is a transition $s' \xrightarrow{\alpha} \mu'$ with $\mu(A) = \mu'(A)$ for all $A \in S/\sim_n$ and vice versa. Similarly, we define "finitary simulation": $s \sqsubseteq_0 s'$ for all states s, s' and $s \sqsubseteq_{n+1} s'$ iff whenever $s \xrightarrow{\alpha} \mu$ then there exists a transition $s' \xrightarrow{\alpha} \mu'$ and a weight function δ for (μ, μ') w.r.t. $\sqsubseteq_n$. As shown in [2]:

Lemma 5. *Let $(S, \rightarrow)$ be transition systems and s, $s' \in S$. Then*

(a) $s \sqsubseteq s'$ if and only if $s \sqsubseteq_n s'$ for all $n \geq 0$.
(b) $s \sim s'$ if and only if $s \sim_n s'$ for all $n \geq 0$.

3 Testing simulation

We present an $\mathcal{O}(n^5 \cdot m^2)$ algorithm for testing simulation where n is the number of states and m the number of transitions in the underlying transition system. The results of this section yield also an $\mathcal{O}(n^5 \cdot m^2)$ algorithm for testing bisimulation. In section 4 we improve the costs and give an $\mathcal{O}(n^2 \cdot m)$ algorithm for testing bisimulation. Lemma 6 shows that for a (finite) transition systems there is a natural number N which is polynomial in the size of the underlying transition system such that $\sqsubseteq = \sqsubseteq_N$. Our algorithm successively computes the relations $\sqsubseteq_0$, $\sqsubseteq_1$,..., $\sqsubseteq_N$. We show that the relation $\sqsubseteq_{j+1}$ can be derived from $\sqsubseteq_j$ by solving maximum flow problems in suitable networks.

Lemma 6. *Let $(S, \rightarrow)$ be a transition system, n the number of states in S and $N = n^2$. Then $\sim = \sim_N$ and $\sqsubseteq = \sqsubseteq_N$.*

Proof. We only show $\sqsubseteq = \sqsubseteq_N$. We have $\sqsubseteq_0 \supseteq \sqsubseteq_1 \supseteq \ldots$ and $s \sqsubseteq s'$ iff $s \sqsubseteq_j s'$ for all j (Lemma 5). Since $\sqsubseteq_0 = S \times S$ contains N elements there exists j with $0 \leq j \leq N$ and $\sqsubseteq_{j+1} = \sqsubseteq_j$. Then $\sqsubseteq_j = \sqsubseteq_i$ for all $i \geq j$ and hence $\sqsubseteq = \sqsubseteq_j = \sqsubseteq_N$. □

Lemma 6 tells us that in order to compute the simulation preorder $\sqsubseteq$ for finite transition systems one has to compute the relation $\sqsubseteq_{n^2}$. We do this by successively computing the relations $\sqsubseteq_j$, $j = 0, 1, \ldots, N$. In order to compute the

relation $\sqsubseteq_{j+1}$ (where $\sqsubseteq_j$ is already computed) we need an algorithm which tests whether or not a weight function for given distributions w.r.t. $\sqsubseteq_j$ exists. We present a polynomial time algorithm which tests whether a weight function for distributions μ, μ' w.r.t. a given relation R exists. The idea of the algorithm is to reduce the problem of finding a weight function to a maximum flow problem in networks. Algorithms to compute the maximum flow are given in [7, 10, 20]. For further details about maximum flow problems see e.g. [9].

A network is a tuple $\mathcal{N} = (N, E, \bot, \top, c)$ where (N, E) is a finite directed graph – where N denotes the set of nodes, $E \subseteq N \times N$ the set of edges – with two specified nodes $\bot$ (the source) and $\top$ (the sink) and a capacity c, i.e. a function c which assigns to each edge $(v, w) \in E$ a non-negative number $c(v, w)$. A flow function f is a function which assigns to edge e a real number $f(e)$ such that

1. For all edges e: $0 \leq f(e) \leq c(e)$
2. Let $in(v)$ be the set of incoming edges to node v and $out(v)$ the set of outgoing edges from node v. Then for each node $v \in N \setminus \{\bot, \top\}$:

$$\sum_{e \in in(v)} f(e) = \sum_{e \in out(v)} f(e)$$

The flow $\mathcal{F}(f)$ of f is given by

$$\mathcal{F}(f) = \sum_{e \in out(\bot)} f(e) - \sum_{e \in in(\bot)} f(e).$$

The maximum flow in $\mathcal{N}$ is the supremum over the flows $\mathcal{F}(f)$ where f is a flow function in $\mathcal{N}$.

Let S be a finite sets, R a subset of $S \times S$ and let μ, $\mu' \in \mathcal{D}(S)$. Let $S' = \{t' : t \in S\}$ where t' are pairwise distinct "new" states (i.e. $t' \notin S$). We choose new elements $\bot$ and $\top$ not contained in $S \cup S'$, $\bot \neq \top$. We associate with (μ, μ') the following network $\mathcal{N}(\mu, \mu', R)$: The nodes are the elements of S and S' and $\bot$ (the source) and $\top$ (the sink), i.e. $N = \{\bot, \top\} \cup S \cup S'$. The edges are

$$E = \{(s, t') : (s, t) \in R\} \cup \{ (\bot, s) : s \in S \} \cup \{ (t', \top) : t \in S \}.$$

The capacities $c(e) \in [0, 1]$ are given by: $c(\bot, s) = \mu(s)$, $c(t', \top) = \mu'(t)$ and $c(s, t') = 1$.

Lemma 7. *The following are equivalent:*

(i) There exists a weight function δ for (μ, μ') w.r.t. R.
(ii) The maximum flow in $\mathcal{N}(\mu, \mu', R)$ is 1.

Proof. (i) $\Longrightarrow$ (ii): For each flow function f in $\mathcal{N}(\mu, \mu', R)$:

$$\mathcal{F}(f) = \sum_{s \in S} f(\bot, s) \leq \sum_{s \in S} c(\bot, s) = \sum_{s \in S} \mu(s) = 1.$$

Let δ be a weight function for (μ, μ') w.r.t. R. Then we define a flow function f as follows: $f(\bot, s) = \mu(s)$, $f(t', \top) = \mu'(t)$, $f(s, t') = \delta(s, t)$. Then $\mathcal{F}(f) = 1$.

Hence the maximum flow of $\mathcal{N}(\mu,\mu',R)$ is 1.
(ii) $\Longrightarrow$ (i): Let f be a flow function with $\mathcal{F}(f)=1$. Since $f(\bot,s)\leq c(\bot,s)=\mu(s)$ and since

$$\sum_{s\in S} f(\bot,s) \;=\; \mathcal{F}(f) \;=\; 1 \;=\; \sum_{s\in S}\mu(s)$$

we get $f(\bot,s)=\mu(s)$ for all $s\in S$. Similarly, we get $f(t',\top) = \mu'(t)$ for all $t\in S$. Let $\delta(s,t)=f(s,t')$ for all $(s,t)\in R$ and $\delta(s,t)=0$ if $(s,t)\notin R$. Then

$$\sum_{t\in S}\delta(s,t) \;=\; \sum_{t\in S} f(s,t') \;=\; f(\bot,s) \;=\; \mu(s)$$

and similarly $\sum_{s\in S}\delta(s,t) = \mu'(t)$. Hence δ is a weight function for (μ,μ') w.r.t. R. □

With Lemma 7 we get an algorithm which tests whether a weight function for distributions μ, μ' w.r.t. a relation R exists: We apply an algorithm for finding the maximum flow F in $\mathcal{N}(\mu,\mu',R)$. The maximum flow in $\mathcal{N}(\mu,\mu',R)$ can be computed e.g. with the $\mathcal{O}(n^3)$ algorithm of Malhotra et al [20] where n is the cardinality of S.

Algorithm 1.
Input: *a finite set S, distributions μ, $\mu'\in\mathcal{D}(S)$ and $R\subseteq S\times S$*
Output: *a weight function δ for (μ,μ') w.r.t. R if there exists one, "No" otherwise.*
Method: *Compute the maximum flow F of the network $\mathcal{N}(\mu,\mu',R)$ and a flow function f with $\mathcal{F}(f)=F$. If $F<1$ then answer "No" else answer "Yes" and return*

$$\delta(s,t) \;=\; \begin{cases} 0 & : \text{if } (s,t)\in S\times S\setminus R \\ f(s,t') & : \text{if } (s,t)\in R. \end{cases}$$

Lemma 6 and Algorithm 1 yield an algorithm for testing simulation:

Algorithm 2. *for testing probabilistic simulation*
Input: *a transition system $(S,\rightarrow)$*
Output: *the simulation preorder $R = \{(s,t)\in S\times S \;:\; s\sqsubseteq t\}$*
Method: *Let $N = n^2$ where n is the number of states of S and let $R_0 = S\times S$.*
For $j=1,\ldots,N$ do:
begin $R_j := R_{j-1}$
For all $(s,t)\in R_{j-1}$ do
begin For all transitions $s\stackrel{\alpha}{\rightarrow}\mu$ do:
If there does not exist a transition $t\stackrel{\alpha}{\rightarrow}\mu'$
such that Algorithm 1 yields a weight function
for (μ,μ') w.r.t. R_{j-1} then $R_j := R_j\setminus\{(s,t)\}$.
end
end
Return $R := R_N$.

It is clear that $R_j = \sqsubseteq_j$ and hence $R = \sqsubseteq_N = \sqsubseteq$. The time complexity of the algorithm is $\mathcal{O}(n^5 \cdot m^2)$ where m is the number of transitions and n the number of states. Algorithm 2 can be implemented in space $\mathcal{O}(n^2 + m)$ because the maximum flow problem (and hence Algorithm 1) can be solved in space $\mathcal{O}(n + m)$ and the representation of the sets R_j needs $\mathcal{O}(n^2)$ space. Similar to Algorithm 2, an $\mathcal{O}(n^5 \cdot m^2)$ algorithm for testing bisimulation can be given. In the next section we improve the time complexity giving an $\mathcal{O}(n^2 \cdot m)$ algorithm.

4 Testing bisimulation

Following the idea of [17] which gives an $\mathcal{O}(n \cdot m)$ algorithm for testing (non-probabilistic) bisimulation we present a method for deciding probabilistic bisimulation that works with refinement steps of partitions on the states. Given a transition system $(S, \rightarrow)$ we start with the trivial partition $X_0 = \{S\}$. Then we successively refine the partition X_k by substituting $B \in X_k$ by the set of equivalence classes w.r.t. the relation $s \equiv s'$ iff

1. Whenever $s \xrightarrow{\alpha} \mu$ then there exists a transition $s' \xrightarrow{\alpha} \mu'$ with $\mu(B) = \mu'(B)$ for all $B \in X_k$.
2. Whenever $s' \xrightarrow{\alpha} \mu'$ then there exists a transition $s \xrightarrow{\alpha} \mu$ with $\mu(B) = \mu'(B)$ for all $B \in X_k$.

At most after n refinement steps the partition X_k cannot be refined. Then X_k is the set of bisimulation equivalence classes.

Definition 8. A *partition* of a transition system $(S, \rightarrow)$ is a set X consisting of pairwise disjoint subsets B of S with $\bigcup_{B \in X} B = S$ and such that for all $B \in X$ and $s \in B$: the bisimulation equivalence class $[s]$ of s is contained in B.

In what follows, we shortly write $\mu(X)$ to denote the vector $(\mu(B))_{B \in X}$. If $s \in S$ then we define $X(s) = \{ (\alpha, \mu(X)) : s \xrightarrow{\alpha} \mu \}$. Each partition X is associated with an equivalence relation $\equiv_X$ on S: $s \equiv_X s'$ iff $X(s) = X(s')$. Having a partition X we split the elements of X into the equivalence classes w.r.t. $\equiv_X$: We define

$$\mathcal{J}(X) = \bigcup_{B \in X} B / \equiv_X .$$

Lemma 9. *Let $(S, \rightarrow)$ be a transition system and X a partition.*

(a) $S/\sim$ is a partition with $\mathcal{J}(S/\sim) = S/\sim$.
(b) $\mathcal{J}(X)$ is a partition.
(c) If $\mathcal{J}(X) = X$ then $X = S/\sim$.

Proof. (a) is clear. Let X be a partition of $(S, \rightarrow)$. It is clear that the sets $B \in \mathcal{J}(X)$ are pairwise disjoint and that the union of them is S. Each $B \in X$ can be written as disjoint union of bisimulation equivalence classes. This is because

$s \in B$ implies $[s] \subseteq B$. Hence whenever μ, μ' are distributions with $\mu(A) = \mu'(A)$ for all $A \in S/\sim$ then

$$\mu(B) \;=\; \sum_{A \in B/\sim} \mu(A) \;=\; \sum_{A \in B/\sim} \mu'(A) \;=\; \mu'(B)$$

for all $B \in X$. Hence $s \sim s'$ implies $s \equiv_X s'$. Therefore: If $C \in \mathcal{J}(B)$, $s \in C$ then C is the equivalence class of s w.r.t. $\equiv_X$ and hence contains $[s]$. We conclude that $\mathcal{J}(X)$ is a partition of $(S, \to)$. If $\mathcal{J}(X) = X$ then $\equiv_X$ is a bisimulation. Hence $s \equiv_X s'$ implies $s \sim s'$. Therefore $s \equiv_X s'$ iff $s \sim s'$ and hence $\mathcal{J}(X) = S/\sim$. □

Lemma 10. *Let $(S, \to)$ be a transition system with n states and m transitions and let X be a partition of $(S, \to)$. Then $\mathcal{J}(X)$ can be computed in time $\mathcal{O}(n \cdot m)$ and space $\mathcal{O}(n \cdot m)$.*

Proof. For fixed $B \in X$ and $\alpha \in \mathrm{Act}$ let $\mathcal{L}_{B,\alpha}$ be the set of all pairs $(\mathbf{p}, L)$ where L is a nonempty subset of B and $\mathbf{p} = (p_C)_{C \in X}$ a real vector such that $s \in L$ if and only if there exists a transition $s \xrightarrow{\alpha} \mu$ with $\mu(X) = \mathbf{p}$. Let $\mathcal{L}_B$ be the set of all pairs (α, L) where $\alpha \in \mathrm{Act}$ and $(\mathbf{p}, L) \in \mathcal{L}_{B,\alpha}$ for some $\mathbf{p}$. Then $s \equiv_X s'$ if and only if:

$$\text{Whenever } (\alpha, L) \in \mathcal{L}_B \text{ then } s \in L \text{ iff } s' \in L.$$

The idea of computing $B/\equiv_X$ is to calculate first the sets $\mathcal{L}_{B,\alpha}$, $\alpha \in \mathrm{Act}$, and then to derive the equivalence classes of B w.r.t. $\equiv_X$.

Computation of $\mathcal{L}_{B,\alpha}$. For each $\alpha \in \mathrm{Act}$ and $B \in X$ we construct a tree $T_{B,\alpha}$ by successively inserting nodes and edges. The edges of $T_{B,\alpha}$ are labelled by real numbers $p \in [0,1]$. Each leaf v has depth l and is labelled by an element $(\mathbf{p}(v), L(v)) \in \mathcal{L}_{B,\alpha}$.
Let $X = \{B_1, \ldots, B_l\}$. We start with $T_{B,\alpha}$ to be a tree of depth 0, i.e. a tree consisting of its root. Then for each transition $s \xrightarrow{\alpha} \mu$ where $s \in B$ we traverse the tree starting at the root. Reaching a node v of depth k we do:

- If $k < l$ and there is an outgoing edge from v leading to the node w labelled by $\mu(B_{k+1})$ then we pass the edge $v \to w$ and continue to travel through $T_{B,\alpha}$ with node w.
- If $k < l$ and there is no outgoing edge from v labelled by $\mu(B_{k+1})$ then we insert a new node w and an edge from v to w labelled by $\mu(B_{k+1})$. In the case $k+1 < l$ we continue to travel through $T_{B,\alpha}$ with node w. If $k+1 = l$ then w is a leaf and we define $L(w) = \{s\}$ and $\mathbf{p}(w) = \mu(X)$.
- If v is a leaf of depth l then we insert s into the set $L(v)$.

It is easy to see that the leaves of $T_{B,\alpha}$ represent the elements of $\mathcal{L}_{B,\alpha}$. Hence $\mathcal{L}_B$ is the set of all pairs $(\alpha, L(v))$ where v is a leaf in $T_{B,\alpha}$.

Complexity. First we observe that the tuples $\mu(X)$ (where μ ranges over all distributions s.t. $s \xrightarrow{\alpha} \mu$ is a transition) can be computed in $\mathcal{O}(n \cdot m)$ time: For each distribution μ we set $a_B = 0$ for all $B \in X$. Then for all states $s \in S$: If $s \in B$ then we replace a_B by $a_B + \mu(s)$. Finally $\mu(X) = (a_B)_{B \in X}$. The representation of the tuples $\mu(X)$ needs $\mathcal{O}(n \cdot m)$ space.

The construction of $T_{B,\alpha}$ needs $\mathcal{O}(m_{B,\alpha} \cdot l)$ steps where $m_{B,\alpha}$ is the number of transitions $s \xrightarrow{\alpha} \mu$, $s \in B$. Since $\sum_B \sum_\alpha m_{B,\alpha} = m$ and since the cardinality l of X is bounded by n we get: Ranging over all $B \in X$ and $\alpha \in \mathrm{Act}$ the construction of all trees $T_{B,\alpha}$, $B \in X$, $\alpha \in \mathrm{Act}$, takes $O(n \cdot m)$ steps. The set of paths from the root to a leaf in $T_{B,\alpha}$ is bounded by $m_{B,\alpha}$. Since l is the depth of the leaves $T_{B,\alpha}$ has at most $m_{B,\alpha} \cdot l + 1$ nodes. Hence, all trees $T_{B,\alpha}$ together have $\mathcal{O}(m \cdot n)$ nodes and $\mathcal{O}(m)$ leaves. The representation of the sets $L(v)$ needs $\mathcal{O}(|B|)$ space (where v is a leaf of a tree $T_{B,\alpha}$). Since $|B| \leq n$ the representation of all trees $T_{B,\alpha}$ together needs $\mathcal{O}(n \cdot m)$ space.

Computation of $B/\equiv_X$. We construct a binary tree T_B by successively inserting nodes and edges. Each leaf v has depth r and is labelled by a subset $C(v)$ of B. Let (α_i, L_i), $i = 1, \ldots, r$, be an enumeration of the elements of $\mathcal{L}_B$. (Note that $\alpha_i = \alpha_j$, $i \neq j$ is possible.) We start with a tree of depth 0, a tree consisting of its root. For each $s \in B$ we traverse the tree in the following way: If we have reached a node v of depth $k - 1$, $k \leq r$ then:

- If v has a left son w and $s \in L_k$ then we go to w.
- If v does not have a left son and $s \in L_k$ then we create a new left son w of v and go to w. If $k = r - 1$ then we set $C(w) = \{s\}$.
- If v has a right son w and $s \notin L_k$ then we go to w.
- If v does not have a right son and $s \notin L_k$ then we create a new right son w of v and go to w. If $k = r - 1$ then we set $C(w) = \{s\}$.

If we have reached a node v of depth r then we insert s into the set $C(v)$ of states associated with v.

Then we have: If v is a leaf and $v_0, v_1, \ldots, v_r = v$ the unique path from the root v_0 to v then $C(v) = L'_1 \cap L'_2 \cap \ldots \cap L'_r$ where $L'_i = L_i$ if v_i is the left son of v_{i-1} and $L'_i = B \setminus L_i$ if v_i is the right son of v_{i-1}. Let $\mathbf{p}_i = \mathbf{p}(v)$ where v is the leaf in T_{B,α_i} with $(\alpha_i, L_i) = (\alpha, L(v))$. Then for all $s \in B$: $s \in C(v)$ if and only if $X(s) = \{ (\alpha_i, \mathbf{p}_i) : L_i = L'_i \}$. Hence, if $s, s' \in B$ then $s \equiv_X s'$ if and only if $s, s' \in C(v)$ for some leaf v in T_B. We conclude:

$$B/\equiv_X = \{ C(v) : v \text{ is a leaf in } T_B \}$$

Complexity. The computation of T_B needs $\mathcal{O}(|B| \cdot r)$ steps. It is clear that the cardinality r of $\mathcal{L}_B$ is bounded by m. Hence we have the time complexity $\mathcal{O}(|B| \cdot m)$ for the construction of T_B. Each leaf in T_B has depth $r \leq m$. Since the leaves of T_B correspond to the equivalence classes w.r.t. $\equiv_X$ T_B has at most $|B|$ leaves. Since T_B is binary it has at most $|B| \cdot r + 1$ nodes. Hence, all trees T_B, $B \in X$, have $\mathcal{O}(n \cdot m)$ nodes. Ranging over all v, the sets $C(v)$ can be represented in space $\mathcal{O}(n)$. Hence we get the time complexity $\mathcal{O}(n \cdot m)$ for computing the trees T_B, $B \in X$ and the space complexity $\mathcal{O}(n \cdot m)$ for their representation. □

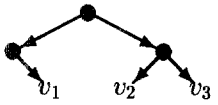
Algorithm 3. *for testing probabilistic bisimulation*
Input: *a transition system* $(S, \to)$
Output: *the set* $R = S/\sim$ *of bisimulation equivalence classes*
Method: *Let* $X := \{S\}$.
Repeat
$Y := X$; $X := \mathcal{J}(X)$;
until $Y = X$;
Return $R := X$.

It is clear that the algorithm returns a partition R with $\mathcal{J}(R) = R$. By Lemma 9: R is the set of bisimulation equivalence classes. If the loop is performed n times then X consists of n one-element sets and hence $\mathcal{J}(X) = X$. Hence the loop is performed at most n times. By Lemma 10 the time complexity is $\mathcal{O}(n^2 \cdot m)$, the space complexity $\mathcal{O}(n \cdot m)$.

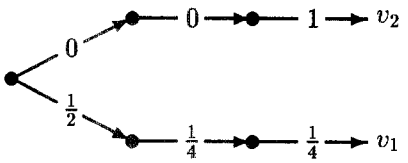
Example 11. Let $(S, \to)$ be given by: $S = \{s_1, s_2, s, t, u\}$ and

$$s_1 \xrightarrow{\alpha} \mu,\ s_2 \xrightarrow{\alpha} \mu,\ s_1 \xrightarrow{\alpha} \mu_1,\ s_2 \xrightarrow{\alpha} \mu_2,\ s \xrightarrow{\alpha} \mu,\ t \xrightarrow{\beta} \mu$$

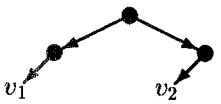
where $\mu(u) = 1$, $\mu_1(s_1) = \mu_1(s_2) = \mu_1(t) = \mu_1(u) = 1/4$ and $\mu_2(s_1) = 1/2$, $\mu_2(t) = \mu_2(u) = 1/4$. Initially we deal with the partition $\{S\}$ and compute $\mathcal{J}(\{S\})$ with the help of Lemma 10: The trees $T_{S,\alpha}$ and $T_{S,\beta}$ consist of a single edge labelled by 1. Their leaves $v_{S,\alpha}$ and $v_{S,\beta}$ are labelled by $(1, \{s_1, s_2, s\})$ and $(1, \{T\})$ respectively. This yields $\mathcal{L}_S = \{(\alpha, \{s_1, s_2, s\}), (\beta, \{t\})\}$ and the tree T_S



where $C(v_1) = \{s_1, s_2, s\}$, $C(v_2) = \{t\}$ and $C(v_3) = \{u\}$. Hence $\mathcal{J}(\{S\}) = \{B_1, B_2, B_3\}$ where $B_i = C(v_i)$. Next we compute $\mathcal{J}(\{B_1, B_2, B_3\})$. Since B_2 and B_3 consist of a single element we only have to consider B_1. The tree $T_{B_1,\alpha}$ can be depict as follows:



where $L(v_1) = \{s_1, s_2\}$ and $L(v_2) = \{s_1, s_2, s\}$. This yields the tree T_{B_1}:



where $C(v_1) = \{s_1, s_2\}$, $C(v_2) = \{s\}$. We obtain the partition X which consists of $\{s_1, s_2\}$, $\{s\}$, $\{t\}$ and $\{u\}$. The next step yields $\mathcal{J}(X) = X$ and hence $X = S/\sim$. □

5 Concluding remarks

We gave an algorithm for testing probabilistic bisimulation in time $\mathcal{O}(n^2 \cdot m)$. Compared with the non-probabilistic case where the best known algorithm for deciding bisimilarity has the time complexity $\mathcal{O}(m \cdot \log n)$ [22] the cost of our algorithm seem to be acceptable. It is an open problem whether the time complexity of our algorithm can be improved in a similar way as the $\mathcal{O}(m \cdot \log n)$ algorithm of [22] improves the $\mathcal{O}(n \cdot m)$ algorithm of [17]. The algorithm which is implemented in the Concurrency Workbench [4] tests non-probabilistic simulation in time $\mathcal{O}(n^4 \cdot m)$. It works similar to the bisimulation equivalence algorithm of [17]. It is an open question whether our $\mathcal{O}(n^5 \cdot m^2)$ result can be improved by a partioning technique. Our methods applied to "deterministic" probabilistic transition systems yield time complexity $\mathcal{O}(n^7)$ for deciding simulation and time complexity $\mathcal{O}(n^3)$ for deciding bisimulation. (In "deterministic" transition systems, for every state s and action α there is at most one outgoing transition labelled by α. Hence, for fixed action set, the total number m of transitions is $\mathcal{O}(n)$.)

In this paper we only considered strong (bi-)simulation which does not abstract from internal actions. It would be interesting if the algorithms presented here can be modified to check weak (bi-)simulation.

References

1. R. Alur, C. Courcoubetis, D. Dill: Verifying Automata Specifications of Probabilistic Real-Time Systems, Proc. REX Workshop, Mook, The Netherlands, Real-Time: Theory in Practice, J.W. de Bakker, C. Huizing, W.P. de Roever, C. Rozenberg (eds.), Lecture Notes in Computer Science 600, pp 27-44, 1991.
2. C. Baier, M. Kwiatkowska: Domain Equations for Probabilistic Processes, submitted for publication.
3. T. Bolognesi, S. Smolka: Fundamental Results for the Verification of Observational Equivalence: a Survey, Protocol Specification, Testing and Verification, Elsevier Science Publishers, IFIP, pp 165-179, 1987.
4. R. Cleaveland, J. Parrow, B. Steffen: A Semantics-Based Verification Tool for Finite State Systems, Protocol Specification, Testing and Verification IX, Elsevier Science Publishers, IFIP, pp 287-302, 1990.
5. R. Cleaveland, S. Smolka, A. Zwarico: Testing Preorders for Probabilistic Processes, Proc. ICALP 1992, Lecture Notes in Computer Science 623, Springer-Verlag, pp 708-719, 1992.
6. C. Courcoubetis, M. Yannakakis: Verifying Temporal Properties of Finite-State Probabilistic Programs, Proc. 29th Annual Symp. on Foundations of Computer Science, pp 338-345, 1988.
7. E. Dinic: Algorithm for Solution of a Problem of Maximal Flow in a Network with Power Estimation, Soviet. Math. Dokl., Vol. 11, pp 1277-1280, 1970.
8. R. Enders, T. Filkorn, D. Taubner: Generating BDDs fir Symbolic Model checking in CCS, Distributed Computing, Vol. 6, pp 155-164, 1993.
9. S. Even: Graph Algorithms, Computer Science Press, 1979.
10. L. Ford, D. Fulkerson: Flows in Networks, Princeton University Press, 1962.

11. J. Groote, F. Vaandrager: An Efficient Algorithm for Branching Bisimulation and Stuttering Equivalence, Proc. 17th International Colloqium Warwick, Automata, Languages and Programming, Lecture Notes in Computer Science 443, pp 626-638, 1990.
12. H. Hansson: Time and Probability in Formal Design of Distributed Systems, Ph.D.Thesis, Uppsala University, 1994.
13. H. Hansson, B. Jonsson: A Logic for Reasoning about Time and Probability, Formal Aspects of Computing, Vol. 6, pp 512-535, 1994.
14. S. Hart, M. Sharir: Probabilistic temporal logic for finite and bounded models, Proc. 16th ACM Symposium on Theory of Computing, pp 1-13, 1984.
15. B. Jonsson, K.G. Larsen: Specification and Refinement of Probabilistic Processes, Proc. 6th IEEE Symp. on Logic in Computer Science, 1991.
16. B. Jonsson, W. Yi: Compositional Testing Preorders for Probabilistic Processes, Proc. 10th IEEE Symp. on Logic in Computer Science, pp 431-443, 1995.
17. P. Kannelakis, S. Smolka: CCS Expressions, Finite State Processes and Three Problems of Equivalence, Proc. 2nd ACM Symposium on the Pronciples of Distributed Computing, pp 228-240, 1983.
18. K. Larsen, A. Skou: Bisimulation through Probabilistic Testing, Information and Computation, Vol. 94, pp 1-28, 1991.
19. D. Lehmann, S. Shelah: Reasoning with Time and Chance, Information and Control, Vol. 53, pp 165-198, 1982.
20. V. Malhotra, M. Pramodh Kumar, S. Maheshwari: An $\mathcal{O}(|V^3|)$ Algorithm for Finding Maximum Flows in Networks, Computer Science Program, Indian Institute of Technology, Kanpur 208016, 1978.
21. R. Milner: Communication and Concurrency, Prentice Hall, 1989.
22. R. Paige, R. Tarjan: Three Partition Refinement Algorithms, SIAM Journal of Computing, Vol. 16, No. 6, pp 973-989, 1987.
23. A. Pnueli, L. Zuck: Verification of Multiprocess Probabilistic Protocols, Distributed Computing, Vol. 1, No. 1, pp 53-72, 1986.
24. A. Pnueli, L. Zuck: Probabilistic Verification, Information and Computation, Vol. 103, pp 1-29, 1993.
25. R. Segala, N. Lynch: Probabilistic Simulations for Probabilistic Processes, Proc. CONCUR 94, Theories of Concurrency: Unification and Extension, Lecture Notes in Computer Science 836, Springer-Verlag, pp 492-493, 1994.
26. R. van Glabbeek, S. Smolka, B. Steffen, C. Tofts: Reactive, Generative, and Stratified Models for Probabilistic Processes, Proc. 5th IEEE Symposium on Logic in Computer Science, pp 130-141, 1990.
27. M. Vardi: Automatic Verification of Probabilistic Concurrent Finite-State Programs, Proc. 26th Symp. on Foundations of Computer Science, pp 327-338, 1985.
28. S. Yuen, R. Cleaveland, Z. Dayar, S. Smolka: Fully Abstract Characterizations of Testing Preorders for Probabilistic Processes, Probabilistic Simulations for Probabilistic Processes, Proc. CONCUR 94, Theories of Concurrency: Unification and Extension, Lecture Notes in Computer Science 836, Springer-Verlag, pp 497-512, 1994.
29. W. Yi: Algebraic Reasoning for Real-Time Probabilistic Processes with Uncertain Information, Formal Techniques in Real Time and Fault Tolerant Systems, Lecture Notes in Computer Science 863, Springer-Verlag, pp 680-693, 1994.
30. W. Yi, K. Larsen: Testing Probabilistic and Nondeterminsitic Processes, Protocol, Specification, Testing and Verification XII, Elsevier Science Publishers, IFIP, pp 47-61, 1992.

Pushdown Processes: Games and Model Checking

Igor Walukiewicz
BRICS[1,2]
Department of Computer Science
University of Aarhus
Ny Munkegade
DK-8000 Aarhus C, Denmark
e-mail: igw@mimuw.edu.pl

(Extended abstract)

Abstract

Games given by transition graphs of pushdown processes are considered. It is shown that if there is a winning strategy in such a game then there is a winning strategy which is realized by a pushdown process. This fact turns out to be connected with the model checking problem for pushdown automata and the propositional μ-calculus. It is show that this model checking problem is DEXPTIME–complete.

1 Introduction

Pushdown processes are, at least in this paper, just another name for pushdown automata. The different name is used to underline the fact that we are mainly interested in the graph of configurations of a pushdown process and not in the language it recognises. This graph can be considered as a transition system. In general such a transition system may not be regular, i.e., may not be an unwinding of a finite transition system. Given a priority function mapping states of the automaton to natural numbers, such a transition system defines a two player parity game. In the game moves of the players alternate. In a move a player picks a configuration reachable from the current one. The result of a game is a finite or an infinite path. The path is finite if one of the players cannot make a move; in this case the other player wins. If the path is infinite we find the smallest priority such that a state of this priority appears infinitely often on the path. Player I wins if this priority is even.

[1]Basic Research in Computer Science,Centre of the Danish National Research Foundation.
[2]On leave from: Institute of Informatics, Warsaw University,
Banacha 2, 02-097 Warsaw, POLAND

Pushdown processes are a generalisation of regular process which correspond to finite automata or regular transition systems. It is stated in [6] that the extra expressive power of pushdown processes may be of use for describing hierarchically structured systems, such as multi-level cashes, or wide area networks. Considering pushdown games is interesting at least for two reasons. First, as we will show here, there is a connection with model checking. The second reason is the problem of synthesis of correct programs (see for example [11]). The conditions of a game may be seen as a specification, and the two players as a program and environment respectively. In this approach a winning strategy is identified with a reactive program satisfying the specification. Hence it is important to know whether a strategy can be implemented as, for example, a regular or pushdown process.

Pushdown processes are a strict generalisation of processes from so called basic process algebra BPA (see [5] for a short survey). The decidability of the model checking for pushdown processes and the propositional μ-calculus follows from [14]. An elementary model checking procedure for the alternation free fragment of the μ-calculus was given in [3]. We are not aware of any such elementary decision procedure for the whole μ-calculus. BPA is a subclass of process algebra (PA) [1]. For the other interesting subclass of PA, namely, basic parallel processes, the model checking problem is undecidable [9]. The question whether pushdown games have pushdown strategies was posed in [16].

The main results of this paper are the following.

1. We show that if there is a winning strategy in a pushdown game G then there is a pushdown winning strategy in G.

2. We give a model checking algorithm for pushdown processes and the whole μ-calculus which runs in time $\mathcal{O}(2^{cn^3m})$ where m is the size of a pushdown process, n the size of a formula and c is some constant.

3. We show that there exists a formula α such that the model checking problem for pushdown processes and this particular formula α is DEXPTIME–hard.

Let us mention that the restriction to parity games is not essential for the result 1 to hold. One can use standard methods of translating Muller, Rabin or Streett conditions into parity conditions to obtain appropriate result for these kind of conditions.

The plan of the paper is as follows. We start with a preliminary section where we recall definitions of pushdown automata and the propositional μ-calculus. In the following section we present some facts about games with parity conditions. Next we prove that if there is a winning strategy on a pushdown tree then there is one realized by a pushdown automaton. In the last section we consider the model checking problem. The proofs are omitted in this abstract.

Acknowledgement: I would like to thank Damian Niwinski for his helpful comments.

2 Preliminaries

2.1 Pushdown processes

The set of finite sequences over Σ will be denoted Σ^* and the set of finite nonempty sequences over Σ will be denoted Σ^+. The empty sequence is denoted by ε.

For a given finite set Σ_s, let $Com(\Sigma_s) = \{pop\} \cup \{push(z) : z \in \Sigma_s\}$ be the set of *stack commands* over Σ_s.

A pushdown automaton (over one letter alphabet) is a tuple:

$$\mathcal{A} = \langle Q, \Sigma_s, q_0 \in Q, \bot \in \Sigma_s, \delta : \Sigma_s \times Q \to \mathcal{P}(Com(\Sigma_s) \times Q) \rangle \qquad (1)$$

where Q is a finite set of *states* and Σ_s is a finite *stack alphabet.* State q_0 is the initial state of the automaton and $\bot$ is the initial stack symbol. A *configuration* of an automaton is a pair (s, q) with $s \in \Sigma_s^+$ and $q \in Q$. The initial configuration is $(\bot, q_0)$. We assume that $\bot$ can be neither put nor removed from the stack. We will sometimes write $(z, q) \to (z', q')$ if $(z', q') \in \delta(z, q)$. Let $\to^+$, $\to^*$ denote respectively the transitive closure of $\to$ and the reflexive and transitive closure of $\to$. We will use q to range over states and z to range over letters of the stack alphabet.

Definition 1 (Pushdown tree) A pushdown automaton $\mathcal{A}$ as in (1) defines a tree $T_{\mathcal{A}} \subseteq (\Sigma_s^+ \times Q)^+$ as follows:

- the root of the tree is $(\bot, q_0)$,
- for every node $(s_0, q_0), \ldots, (s_i, q_i)$, if $(s_i, q_i) \to (s, q)$ then the node has a son $(s_0, q_0), \ldots, (s_i, q_i), (s, q)$.

We call (s_i, q_i) the *label* of the node $(s_0, q_0), \ldots, (s_i, q_i)$.

Remark: In our definition of a pushdown automaton we have assumed that the automaton can put at most one symbol on the stack in one move. This is done only for convenience of the presentation. The main results also hold for the more general form of automata which can push many symbols on the stack in one move. Of course we can simulate pushing more symbols on the stack by extending the alphabet and the set of states but the simulating automaton will be in general much bigger. (We are not interested in the language equivalence but in isomorphism of induced pushdown trees.)

Remark: The assumption that automata do not have an input alphabet is not essential as in the problems we will consider we will allow states to have "properties" which can be used to simulate behaviour of an automaton with input alphabet.

2.2 Propositional μ-calculus

Let $Prop = \{p_1, p_2, \ldots\}$ be a set of propositional constants and let $Var = \{X, Y, \ldots\}$ be a set of variables. Formulas of the μ-calculus over these sets can

be defined by the following grammar:

$$F := Prop \mid \neg Prop \mid Var \mid F \vee F \mid F \wedge F \mid \langle\,\rangle F \mid [\,]F \mid \mu Var.F \mid \nu Var.F$$

Note that we allow negations only before propositional constants. As we will be interested in closed formulas this is not a restriction. In the following, $\alpha, \beta, \ldots$ will denote formulas.

Formulas are interpreted in *transition systems* of the form $\mathcal{M} = \langle S, R, \rho \rangle$, where: S is a nonempty set of states, $R \subseteq S \times S$ is a binary relation on S and $\rho : Prop \to \mathcal{P}(S)$ is a function assigning to each propositional constant a set of states where this constant holds.

For a given model $\mathcal{M}$ and an assignment $V : Var \to \mathcal{P}(S)$, the set of states in which a formula φ is true, denoted $\| \varphi \|_V^{\mathcal{M}}$, is defined inductively as follows:

$$\| p \|_V^{\mathcal{M}} = \rho(p) \qquad \| \neg p \|_V^{\mathcal{M}} = S - \rho(p) \qquad \| X \|_V^{\mathcal{M}} = V(X)$$

$$\| \alpha \vee \beta \|_V^{\mathcal{M}} = \| \alpha \|_V^{\mathcal{M}} \cup \| \beta \|_V^{\mathcal{M}} \qquad \| \alpha \wedge \beta \|_V^{\mathcal{M}} = \| \alpha \|_V^{\mathcal{M}} \cap \| \beta \|_V^{\mathcal{M}}$$

$$\| \langle\,\rangle \alpha \|_V^{\mathcal{M}} = \{ s : \exists s'.R(s,s') \wedge s' \in \| \alpha \|_V^{\mathcal{M}} \}$$

$$\| [\,] \alpha \|_V^{\mathcal{M}} = \{ s : \forall s'.R(s,s') \Rightarrow s' \in \| \alpha \|_V^{\mathcal{M}} \}$$

$$\| \mu X.\alpha(X) \|_V^{\mathcal{M}} = \bigcap \{ S' \subseteq S : \| \alpha \|_{V[S'/X]}^{\mathcal{M}} \subseteq S' \}$$

$$\| \nu X.\alpha(X) \|_V^{\mathcal{M}} = \bigcup \{ S' \subseteq S : S' \subseteq \| \alpha \|_{V[S'/X]}^{\mathcal{M}} \}$$

here $V[S'/X]$ is the assignment such that, $V[S'/X](X) = S'$ and $V[S'/X](Y) = V(Y)$ for $Y \neq X$. We will write $\mathcal{M}, s, V \models \varphi$ when $s \in \| \varphi \|_V^{\mathcal{M}}$. We will write $\mathcal{M}, s \models \varphi$ if for every assignment V we have $\mathcal{M}, s, V \models \varphi$.

A model checking problem is to decide whether for a given model $\mathcal{M}$, state s and formula α without free variables, the relation $M, s \models \alpha$ holds. Here we will be interested in the case when M is a pushdown tree and s is the root of it.

3 Parity games and canonical strategies

In this section we recall the notion of parity games and give an explicit description of winning strategies in such games. It turns out that a strategy in such a game induces an assignment of tuples of ordinals to nodes of the game. We call these tuples of ordinals *signatures*. In this way we have means to compare different strategies by comparing signatures they induce. It turns out that there exists canonical, or the least possible, signature assignment.

Most of the material presented here comes from [17]. The notion of signature was proposed by Streett and Emerson [15]. The proof of the existence of memoryless strategies in parity games was given independently by Mostowski [13] and by Emerson and Jutla [7]. Klarlund [10] proves a more general fact that a player has a memoryless strategy in a game if the has a strategy and his winning conditions are given as a Rabin condition.

Let $G = \langle V = V_I \cup V_{II}, E, \Omega : V \to Ind \rangle$ be a bipartite graph with vertices labeled by *priorities* from *Ind* which is a finite subset of natural numbers. A game from some vertex $v_1 \in V_I$ is played as follows: first player I chooses a vertex $v_2 \in V_{II}$, s.t. $E(v_1, v_2)$ then player II chooses a vertex $v_3 \in V_I$, s.t. $E(v_2, v_3)$ and so on ad infinitum unless one of the players cannot make a move. If a player cannot make a move he looses. The result of an infinite play is an infinite path $v_1, v_2, v_3, \ldots$ This path is *winning* for player I if in the sequence $\Omega(v_1), \Omega(v_2), \Omega(v_3), \ldots$ the smallest priority appearing infinitely often is even. The play from vertices of V_{II} is defined similarly but this time player II starts.

Strategy σ for player I is a function assigning to every sequence of vertices $\vec{v}$ ending in a vertex from V_I a vertex $\sigma(\vec{v}) \in V_{II}$ such that $E(v, \sigma(\vec{v}))$. A strategy is called *memoryless* iff $\sigma(\vec{v}) = \sigma(\vec{v}')$ whenever $\vec{v}$ and $\vec{v}'$ end in the same vertex. A strategy is *winning* iff it guarantees a win for player I whenever he follows the strategy. Similarly we define a strategy for player II.

Suppose we have a propositional constant I which holds in the vertices from which player I is to move, i.e., in vertices from V_I. Let us assume that the range of Ω is $\{1, \ldots, n\}$ and suppose that for every $i \in \{1, \ldots, n\}$ we have the propositional constant i which holds in the vertices of priority i. Consider the formula:

$$\varphi_I(Z_1, \ldots, Z_n) = \big(I \Rightarrow \bigwedge_{i \in \{1,\ldots,n\}} (i \Rightarrow \langle\,\rangle Z_i)\big) \wedge \big(\neg I(x) \Rightarrow \bigwedge_{i \in \{1,\ldots,n\}} (i \Rightarrow [\,] Z_i)\big)$$

We will be interested in the set:

$$\mathcal{W}_I = \| \mu Z_1 . \nu Z_2 \ldots \mu Z_{n-1} . \nu Z_n . \varphi_I(Z_1, \ldots, Z_n) \|^G$$

(μ is used to close variables with odd indices and ν is used for even indices).

Definition 2 When applied to n-tuples of ordinals symbols $=$, $<$, $\leq$ stand for corresponding relations in the lexicographical ordering. For every $i \in \{1, \ldots, n\}$ we use $=_i$ to mean that both arguments are defined and when truncated to first i positions the two vectors are equal; similarly for $<_i$ and $\leq_i$.

Definition 3 (Signature) A signature is a n-tuple of ordinals. An assignment $\mathcal{S}$ of signatures to nodes in some set $S \subseteq T$ will be called *consistent* if for every $v \in S \cap V_I$ there is a son $w \in S$ such that:

$$\mathcal{S}(w) \leq_{\Omega(v)} \mathcal{S}(v) \text{ and it is strictly smaller if } \Omega(v) \text{ is odd.} \tag{2}$$

similarly if $v \in S \cap V_{II}$ then for all w such that $E(v, w)$ we have $w \in S$ and the condition (2) holds.

Definition 4 (Canonical signatures) We extend the syntax of the formulas by allowing constructions of the form $\mu^\tau Z.\alpha(Z)$, where τ is an ordinal and $\alpha(Z)$ is a formula from the extended syntax. The semantics is defined as follows:

$$\| \mu^0 Z.\alpha(Z) \|_V^{\mathcal{M}} = \emptyset \qquad \| \mu^{\tau+1} Z.\alpha(Z) \|_V^{\mathcal{M}} = \| \alpha(Z) \|_{V[\|\mu^\tau Z.\alpha(Z)\|_V^{\mathcal{M}}/Z]}^{\mathcal{M}}$$

$$\| \mu^\tau Z.\alpha(Z) \|_V^{\mathcal{M}} = \bigcup_{\rho < \tau} \| \mu^\rho Z.\alpha(Z) \|_V^{\mathcal{M}} \quad (\tau \text{ a limit ordinal})$$

By Knaster-Tarski theorem $\| \mu Z.\alpha(Z) \|_V^{\mathcal{M}} = \bigcup_\tau \| \mu^\tau Z.\alpha(Z) \|_V^{\mathcal{M}}$.

We define a notion of the *canonical signature*, $Sig(v)$, of a vertex $v \in S_I$ (we will write $Sig_G(v)$ if the game is not clear from the context). This is the smallest in the lexicographical ordering sequence of ordinals $(\tau_1, \ldots, \tau_n)$ such that:

$$v \in \| \varphi_I(P_0, \ldots, P_n) \|^G$$

where:

$$P_i = \nu Z_i.\mu Z_{i+1} \ldots \nu Z_n.\varphi_I(P_0, \ldots, P_{i-1}, Z_i, \ldots, Z_n) \quad \text{for } i \text{ even}$$
$$P_i = \mu^{\tau_i} Z_i.\nu Z_{i+1} \ldots \nu Z_n.\varphi_I(P_0, \ldots, P_{i-1}, Z_i, \ldots, Z_n) \quad \text{for } i \text{ odd}$$

As for an even i the ordinal τ_i is not used, the definition implies that $\tau_i = 0$ for every even i. We prefer to have this redundancy rather than to calculate right indices each time.

Fact 5 Canonical signature assignment is the least consistent signature assignment. That is, for every consistent signature assignment $\mathcal{S}$, whenever for some node v, $\mathcal{S}(v)$ is defined then $Sig(v)$ is defined and $Sig(v) \leq \mathcal{S}(v)$.

Definition 6 (Canonical strategy) A *canonical strategy* is a strategy taking for each node $v \in \mathcal{W}_I \cap V_I$ a son which has the smallest possible canonical signature.

Remark: Despite the name, canonical strategies may not be uniquely determined because a node may have many sons with the same signature.

Fact 7 Suppose w is a node reached from v when player I uses a canonical strategy and let p be the minimum of priorities of states appearing in the labels between v and w (not including w). We have that $Sig(w) \leq_p Sig(v)$ and it is strictly smaller if p is odd.

Theorem 8
The set $\mathcal{W}_I$ is the set of nodes from which player I has a winning strategy. A canonical strategy is winning and memoryless.

4 Pushdown strategies in pushdown games

Let $\mathcal{A}$ be a pushdown automaton as in (1). For simplicity of the presentation let us assume that the set Q of states of $\mathcal{A}$ is partitioned into two sets Q_I and Q_{II}. We also assume that transitions from states in Q_I lead only to states in Q_{II} and vice versa. More formally we require that for every q, q', z, z': whenever $(push(z'), q')$ or (pop, q') is in $\delta(z, q)$ then: $q \in Q_I$ iff $q' \in Q_{II}$.

The automaton $\mathcal{A}$ defines a pushdown tree $T_{\mathcal{A}}$ which we will take as a graph of the game. To have a game we will also need a priority function. It is an important point to decide which priority functions to allow. If we allowed arbitrary such

functions then the whole advantage of the fact that the graph is generated by a pushdown automaton would be gone. It seems that a reasonable choice is to allow only functions associated with states of the automaton. That is, we start by giving a priority function $\Omega : Q \to \mathbb{N}$ and then for every vertex v of T we consider the state q appearing in the label of v and let $\Omega(v) = \Omega(q)$. This choice of the method of assigning priorities is motivated by the fact that we are interested in the winning conditions definable in S1S.

Next we should clarify what we mean by a pushdown strategy. We would like to say that a pushdown strategy is a strategy realised by a pushdown automaton in a sense that this automaton reads moves of player *II* and outputs moves of player *I*. Such an automaton must have the property that while reading a (possibly infinite) sequence of moves of player *II* it outputs a sequence of moves of player *I* such that the path of $T_{\mathcal{A}}$ designated by these moves is winning for player *I*. We will not formalise this notion of pushdown strategy here as it would require several definitions for which we have no space. We will content ourselves with a weaker definition given in the theorem below. Let us just remark that the strategy automaton given in the proof can be used to construct a strategy automaton as defined above.

Theorem 9
If there is a winning strategy for player I in $T_{\mathcal{A}}$ then there is a winning pushdown strategy, i.e., there is a pushdown automaton $\mathcal{B}$ such that $T_{\mathcal{B}}$ is isomorphic to a winning strategy in $T_{\mathcal{A}}$.

Let us try to explain an idea of the construction of the pushdown strategy for player *I*. In some sense one may consider a pushdown strategy as a strategy operating with a stack of strategies for regular graphs. Whenever a new element is pushed on a stack, player *I* is suspended and a new player *I* is started which has only partial information about the history of the play up to this point. Suppose we are in some position (s, q) and player *I* decides that the best move for him would be to push z' on the stack and change the state to q'. At this moment this player *I* is suspended and a new player *I* starts to play. He will play until z' is taken out from the stack. The main question is what the new player *I* should know about the current position of the play. Because the canonical strategy is memoryless it would be enough for him to know only how the arena of the game looks from his current position. In turn this is determined by the label of the node, which is (sz', q'). Unfortunately we cannot afford to let the player know so much because the size of the stack is potentially unbounded. On the other hand the new payer *I* will play only until z' is popped and the stack becomes s again. Hence the part of the tree where the new player *I* is playing does not depend on s but only on the letter z' on the top of the stack and the current state q'. What depends on s is the rest of the play when the new player is finished. Hence it should be enough for the new player *I* if the old player *I* told him which states are safe. In other words what are the states such that if the new player *I* finishes in one of them then the old player *I* is able to carry on and win. This set of states should depend on the lowest priority of a state met

from the moment the old player I was suspended. So we will not have just one set of states but a vector $\vec{A} = \{A^p\}_{p\in\{1,\ldots,n\}}$ of sets of states. Each A^p is a set of states in which the new player I can finish provided p is the smallest priority of a state from the moment when the old player I was suspended. Apart from $\vec{A}$ the new player should also know the current state q' and the current symbol z' on the top of the stack. We will also use a variable θ to store the lowest priority of a state we came across. This amount of information is bounded so we have a basis for construction of a pushdown automaton realizing the strategy.

Let us now start with the formal definitions. As in the previous section we assume that $\{1,\ldots,n\}$ is the range of Ω. We will use $\vec{A}$ to range over n element vectors of sets of states and θ to range over $\{1,\ldots,n\}$. We also use z to range over stack symbols and q to range over states.

Definition 10 (Sub-game) For every $\vec{A}, z, \theta, q$ we define the game $G(\vec{A}, z, \theta, q)$ as follows. The arena of the game is a subtree of $T_{\mathcal{A}}$ starting from a node with a configuration $(\perp z, q)$. Every node labeled with a configuration $(\perp, q')$, for some q', is marked winning or loosing. We mark the node *winning* if $q' \in A^{\min(p,\theta)}$, where p is the lowest priority of a state appearing on the path to the node (counting q but not q'). Otherwise we mark the node loosing. Whenever a play reaches a marked node, player I wins if this node is marked winning otherwise player II is the winner. If a play is infinite, player I wins iff the obtained path is winning (as defined at the beginning of Section 3).

Remark: In our definition of the game we did not have the concept of marking but we allowed vertices with no sons, and had the rule that a player looses if he cannot make a move. Hence we can simulate marking of vertices with cutting the paths. We find the metaphor of markings more useful here.

Definition 11 (Signature, Hint) Suppose that player I has a winning strategy in a game $G(\vec{A}, z, \theta, q)$. Define $Sig(\vec{A}, z, \theta, q)$ to be the canonical signature of the root of the game.

If $q \in Q_I$ then let v be a son of the root which has the smallest canonical signature (if there is more than one such son then fix one arbitrary). If v is labeled by $(\perp, q')$ then let $Hint(\vec{A}, z, \theta, q) = (pop, q')$ otherwise v is labeled by $(\perp zz', q')$ and let $Hint(\vec{A}, z, \theta, q) = (push(z'), q')$.

Definition 12 (Update function) Define $Up(\vec{A}, z, q, \theta)$ to be the sequence of sets $\vec{A}_1 = \{A_1^p\}_{p\in\{1,\ldots,n\}}$, where each A_1^p is the set of states q' such that:

$$Sig(\vec{A}, z, \min(\Omega(q), p, \theta), q') \leq_{\min(\Omega(q),p)} Sig(\vec{A}, z, \theta, q)$$

in case $\min(\Omega(q), p)$ is even and

$$Sig(\vec{A}, z, \min(\Omega(q), p, \theta), q') <_{\min(\Omega(q),p)} Sig(\vec{A}, z, \theta, q)$$

otherwise.

Definition 13 (Strategy automaton) Let

$$\mathcal{B} = \langle Q, \mathcal{P}(Q)^n \times \Sigma_s \times \{1, \ldots, n\}, q_0, (\emptyset, \ldots, \emptyset, \bot, n), \delta_\beta \rangle$$

Before defining the transition relation let us introduce an abbreviation. We introduce new automata operation $repmin(\theta')$ which means: if on the top of the stack there is some triple $\vec{A}z\theta$, replace it with $\vec{A}z\theta_1$ where $\theta_1 = \min(\theta, \theta')$. We also introduce a semicolon operation, so $\delta_\mathcal{B}(\vec{A}z\theta, q, a) = (pop, q'); repmin(\theta')$ means that first $\vec{A}z\theta$ is removed from the stack and the state is changed to q'; then, possibly, the third component of the triple currently at the top of the stack is changed. Hence if we had a configuration $(s\vec{A}_1 z_1 \theta_1 \vec{A}z\theta, q)$ then after this operation we obtain the configuration $(s\vec{A}_1 z_1 \min(\theta_1, \theta'), q')$.

Let us now proceed with the definition of $\delta_\mathcal{B}$:

- If $q \in Q_I$ then:
  - $\delta_\mathcal{B}(\vec{A}z\theta, q) = \{(pop, q'); repmin(\min(\theta, \Omega(q)))\}$ if $Hint(\vec{A}, z, q, \theta) = (pop, q')$.
  - $\delta_\mathcal{B}(\vec{A}z\theta, q) = \{repmin(\Omega(q)); push(\vec{A}'z'n, q')\}$ if $\vec{A}' = Up(\vec{A}, z, \theta, q)$ and $Hint(\vec{A}, z, q, \theta) = (push(z'), q)$.
- If $q \in Q_{II}$ then:
  - $(pop, q'); repmin(\min(\theta, \Omega(q))) \in \delta_\mathcal{B}(\vec{A}z\theta, q)$ if $(pop, q') \in \delta_\mathcal{A}(z, q)$.
  - $repmin(\Omega(q)); push(\vec{A}'z'n, q') \in \delta_\mathcal{B}(\vec{A}z\theta, q, a)$ if $\vec{A}' = Up(\vec{A}, z, \theta, q)$ and $(push(z'), q') \in \delta_\mathcal{A}(z, q)$.

Theorem 9 follows from the following lemmas:

Lemma 14 If player I can win in $G(\vec{A}, z, \theta, q)$ and

$$repmin(\Omega(q)); push(\vec{A}_1 z_1 n, q_1) \in \delta_\mathcal{B}(\vec{A}z\theta, q)$$

then $Sig(\vec{A}_1, z_1, n, q_1) \leq_{\Omega(q)} Sig(\vec{A}, z, \theta, q)$ and it is strictly smaller if $\Omega(q)$ is odd.

Lemma 15 Let $(sz\vec{A}\theta, q)$ be a configuration reachable from the initial one. Suppose $(sz\vec{A}\theta, q) \rightarrow^+ (sz\vec{A}\theta', q')$ and $sz\vec{A}$ is always in the stack during this derivation. Let p be the minimum of the priorities of the states appearing in the derivation (not counting the last one). We have: (i) $\theta' = \min(p, \theta)$ and (ii) $Sig(\vec{A}, z, \theta', q') \leq_p Sig(\vec{A}, z, \theta, q)$ and it is strictly smaller if p is odd.

Lemma 16 The strategy defined by the automaton $\mathcal{B}$ is winning.

Remark: The automaton $\mathcal{B}$ is exponentially larger than $\mathcal{A}$. One can show that in general the strategy automaton must be exponentially larger, although it is not clear that the exponent must be $\mathcal{O}(n|Q|)$ as it is in the case of $\mathcal{B}$. This situation is different from the situation for parity games on finite transition systems where no memory is need.

5 Model checking for pushdown trees

Here we consider a problem of checking whether the root of a given pushdown tree satisfies a given formula of the propositional μ-calculus. First we reduce this problem to the problem of finding a winning strategy in some pushdown game. Next we use results from the previous section to show how one can solve this later problem. Finally we show the lower bound on the complexity of the model checking problem.

The reduction of the model checking to establishing existence of a winning strategy follows from a fairly standard arguments [8]. In that paper Emerson, Jutla and Sistla show how to reduce the model checking problem over finite transition systems to establishing existence of a winning strategy in a finite game. In our case the argument is essentially the same but we must also observe that in the resulting game the priority function Ω depends only on the states in the current configuration.

Theorem 17
For a given pushdown automaton $\mathcal{A}$ and a μ-calculus formula φ one can construct a pushdown automaton $\mathcal{C}$ and a priority function Ω, such that: $T_{\mathcal{A}} \models \varphi$ iff there is a winning strategy for player I in the game $T_{\mathcal{C}}$ with the priority function Ω. The size of $\mathcal{C}$ is linear in sizes of both $\mathcal{A}$ and φ.

5.1 Establishing existence of winning strategies

Let $\mathcal{A}$ be a pushdown automaton as in (1) and let $\Omega : Q \to \{1, \ldots, n\}$ be an indexing function. These define the game on $T_{\mathcal{A}}$. Here we are concerned with the problem: given $\mathcal{A}$ and Ω establish whether there exists a winning strategy for player I in $T_{\mathcal{A}}$. We will reduce this problem to the problem of establishing existence of a winning strategy in a game on some finite graph. Let $\mathcal{A}$ and Ω be fixed for the rest of this subsection.

Before we begin let us try to give some intuitions behind the construction of a finite game $\mathcal{M}_{\mathcal{A}}$. For every $\vec{A}, z, \theta, q$ and $p \in \{1, \ldots, n\}$ we will have in $\mathcal{M}_{\mathcal{A}}$ a node $Check(\vec{A}, z, \theta, p, q)$. There will be strategy for player I from this node iff there is a strategy for player I in the game $G(\vec{A}, z, \theta, q)$ (see Definition 10); we will explain the role of p later. If $pop(q')$ move is possible from (z, q) then for it to be a good choice for player I it should be the case that $q' \in A^{\min(\Omega(q),\theta)}$. If $(push(z_1), q_1)$ is possible then the checking is more complicated as we do not have a stack. We will use universal branching instead. We will have a node $Move((\vec{A}, z, \theta, q), (?, z_1, q_1))$ with the intended meaning that the next planed move is $(push(z_1), q_1)$ and that one has to guess $\vec{A}_1$. We will also have nodes $Move((\vec{A}, z, \theta, q), (\vec{A}_1, z_1, q_1))$ where $\vec{A}_1$ is already guessed and from which it is necessary to check whether it was guessed correctly. We divide the future play into two parts which we consider separately. We check what happens until z_1 is popped from the stack and simultaneously we check what happens after this event. The first task is started from the node $Push(\vec{A}_1, z_1, n, q_1)$ the

other one from nodes $Check(\vec{A}, z, \min(\theta, p''), p'', q'')$ where p'' intuitively represents the lowest priority which was met while z_1 was on the stack and q'' is a state from $A_1^{p''}$.

Definition 18 (Game $\mathcal{M}_\mathcal{A}$) Let $\mathcal{M}_\mathcal{A}$ be a game on a finite graph defined as follows. For every $\vec{A}, \vec{A}_1$, z, z_1, θ, q, q_1 and $p \in \{1, \ldots, n\}$ we have nodes:

$$\begin{array}{ll} Check(\vec{A}, z, \theta, p, q) & Push(\vec{A}, z, \theta, q) \\ Move((\vec{A}, z, \theta, q), (?, z_1, q_1)) & Move((\vec{A}, z, \theta, q), (\vec{A}_1, z_1, q_1)) \\ Pop(q) & Err(q) \end{array}$$

Here '?' is a special symbol. We have the following transitions between the nodes:

$$\begin{array}{l} Check(\vec{A}, z, \theta, p, q) \to Pop(q') \quad \text{if } (pop, q') \in \delta(z, q) \text{ and } q' \in A^{\min(\Omega(q), \theta)} \\ Check(\vec{A}, z, \theta, p, q) \to Err(q') \quad \text{if } (pop, q') \in \delta(z, q) \text{ and } q' \notin A^{\min(\Omega(q), \theta)} \\ Check(\vec{A}, z, \theta, p, q) \to Move((\vec{A}, z, \theta, q), (?, z_1, q_1)) \quad \text{if } (push(z_1), q_1) \in \delta(z, q) \end{array}$$

and exactly the same transitions from $Push(\vec{A}, z, \theta, q)$, moreover we have:

$$\begin{array}{l} Move((\vec{A}, z, \theta, q), (?, z_1, q_1)) \to Move((\vec{A}, z, \theta, q), (\vec{A}_1, z_1, q_1)) \\ Move((\vec{A}, z, \theta, q), (\vec{A}_1, z_1, q_1)) \to Push(\vec{A}_1, z_1, n, q_1) \\ Move((\vec{A}, z, \theta, q), (\vec{A}_1, z_1, q_1)) \to Check(\vec{A}, z, \min(\theta, p), p, q'') \\ \qquad\qquad \text{if } p \leq \Omega(q) \text{ and } q'' \in A_1^p \end{array}$$

The set V_I of nodes where Player I makes a move consists of nodes:

$$\begin{array}{l} Check(\vec{A}, z, \theta, p, q) \text{ and } Push(\vec{A}, z, \theta, q) \quad \text{for } q \in Q_I \\ Move((\vec{A}, z, \theta, q), (?, z_1, q_1)) \quad \text{for arbitrary } q \in Q_I \cup Q_{II} \end{array}$$

In the remaining nodes player II makes a move. Priority function Ω_M is defined by:

$$\Omega_M(Check(\vec{A}, z, \theta, p, q)) = p \qquad \Omega_M(Push(\vec{A}, z, \theta, q)) = \Omega(q)$$
$$\Omega_M(Move((\vec{A}, z, \theta, q), (?, z_1, q_1))) = \Omega_M(Move((\vec{A}, z, \theta, q), (\vec{A}_1, z_1, q_1))) = n + 1$$

Player I wins in the game $\mathcal{M}_\mathcal{A}$ if either: (i) after finitely many steps player II cannot make a move or a node labeled $Pop(q)$, for some q, is reached; or (ii) the game is infinite and the infinite path $\mathcal{P}$ which is the result of the play is winning for I. Otherwise player II is the winner.

Theorem 19
Player I has a winning strategy in the game $T_\mathcal{A}$ iff he has a winning strategy in the game $\mathcal{M}_\mathcal{A}$ from the node $Check((\emptyset, \ldots, \emptyset), \bot, n, n, q_0)$.

Let us remark here that the theorem does not imply that there is a finite strategy on a pushdown tree. In order to use the strategy in $\mathcal{M}_\mathcal{A}$ to play in $T_\mathcal{A}$ we need a stack.

Finally let us put Theorems 17 and 19 together and calculate the complexity of the model checking algorithm. The size of the game $\mathcal{M}_{\mathcal{A}}$ is $\mathcal{O}(k2^{cmn})$ where: k is the size of the stack alphabet, m is the number of states of $\mathcal{A}$, n is the cardinality of the range of the priority function Ω, and c is a constant. The task of establishing existence of a winning strategy in $\mathcal{M}_{\mathcal{A}}$ is equivalent to checking whether the specific μ-calculus formula holds. Hence any model checking algorithm will solve the problem. Using currently known algorithms we obtain that the whole problem can be solved in time $\mathcal{O}((k2^{cmn})^n)$ (or $\mathcal{O}((k2^{cmn})^{1+n/2})$ if using [12]). This is the estimation only for the problem of establishing existence of a winning strategy. Putting it together with the reduction from the previous subsection we obtain that for a given automaton with m states and k stack symbols and a formula of size n_1 with alternation depth n_2 we have an algorithm working in time $\mathcal{O}((k2^{cmn_1n_2})^{n_2})$.

5.2 The lower bound

Finally we show a deterministic exponential time lower bound on the model checking problem for pushdown automata and (non alternating) μ-calculus. It follows from a quite standard reduction by simulating alternating linear space bounded Turing machines. The simulating automaton is very similar to the one described by Chandra, Kozen and Stockmeyer in [4]. Given an alternating linear space bounded machine M and a word w we construct a pushdown automaton which acts as follows. First it puts the initial configuration of M on the stack. If the initial state is existential, player I chooses which possible move of M to simulate, otherwise player II chooses the move. Simulating the move is done by putting a new configuration by player I on the stack. Proceeding this way, the game eventually arrives to a point when a configuration with an accepting state is pushed on the stack. At the same moment we have also all the preceding configurations on the stack. In this position player II is allowed to make a guess about correctness of this sequence of configurations. He may try to show that player I cheated and there are two subsequent configurations on the stack such that one is not reachable from the other in the move of M which was chosen at that point. Player I wins if player II is not able to do this. We have the following:

Fact 20 There exists a formula α (without alternations) such that the problem "given a pushdown automaton $\mathcal{A}$, is α satisfied in the root of $T_{\mathcal{A}}$" is DEXPTIME–hard.

Remark: This argument does not work for BPA processes. Indeed the complexity result from [2] shows that the the model checking problem for the μ-calculus without alternations is polynomial when a formula is fixed.
Remark: We conjecture that model checking is exponential also in the second parameter. That is, there exists a fixed pushdown process $\mathcal{A}$ such that the problem: "given a formula α, is α satisfied in the root of $T_{\mathcal{A}}$" is DEXPTIME–hard.

References

[1] J. Bergstra and J. Klop. Process theory based on bisimulation semantics. volume 354 of *LNCS*, 1988.

[2] O. Burkart and B. Steffen. Model checking for context-free processes. In *CONCUR '92*, volume 630 of *LNCS*, pages 123–137, 1992.

[3] O. Burkart and B. Steffen. Pushdown processes: Parallel composition and model checking. In *CONCUR '94*, volume 836 of *LNCS*, 1994.

[4] A. K. Chandra, D. C. Kozen, and L. J. Stockmeyer. Alternation. *Journal of the ACM*, 28(1):114–133, 1981.

[5] S. Christiensen and H. Huttel. Deciding issues for infinite-state processes – a survey. *Bulletin of EATCS*, 51:156–166, October 1993.

[6] E. Clarke, O. Grumberg, and D. Long. Verification tools for finite-state concurrent systems. In *A Decade of Concurrency*, volume 803 of *LNCS*, pages 124–175. Springer-Verlag, 1993.

[7] E. A. Emerson and C. Jutla. Tree automata, mu-calculus and determinacy. In *Proc. FOCS 91*, 1991.

[8] E. A. Emerson, C. Jutla, and A. Sistla. On model-checking for fragments of μ-calculus. In *CAV'93*, volume 697 of *LNCS*, pages 385–396, 1993.

[9] J. Esparza and A. Kiehn. On the model checking problem for branching time logics and basic parallel processes. In *CAV '95*, volume 939 of *LNCS*, pages 353–366, 1995.

[10] N. Klarund. Progress measures, immediate determinacy and a subset construction for tree automata. In *IEEE LICS*, pages 382–393, 1992.

[11] H. Lescow. On polynomial–size programs winning finite–state games. In *CAV '95*, volume 939 of *LNCS*, pages 239–252, 1995.

[12] D. E. Long, A. Browne, E. M. Clarke, S. Jha, and W. R. Marrero. An improved algorithm for the evaluation of fixpoint expressions. In *CAV'94*, volume 818 of *LNCS*, pages 338–350, 1994.

[13] A. W. Mostowski. Games with forbidden positions. Technical Report 78, University of Gdansk, 1991.

[14] D. Muller and P. Schupp. The theory of ends, pushdown automata and second-order logic. *Theoretical Computer Science*, 37:51–75, 1985.

[15] R. S. Street and E. A. Emerson. An automata theoretic procedure for the propositional mu-calculus. *Information and Computation*, 81:249–264, 1989.

[16] W. Thomas. On the synthesis of strategies in infinite games. In *STACS '95*, volume 900 of *LNCS*, pages 1–13, 1995.

[17] I. Walukiewicz. Monadic second order logic on tree-like structures. In *STACS '96*, LNCS, pages 401–414, 1996.

Module Checking

Orna Kupferman[1] and Moshe Y. Vardi[2]

[1] Bell Laboratories, 600 Mountain Avenue, Murray Hill, NJ 07974, U.S.A.
Email: ok@research.att.com
[2] Rice University, Department of Computer Science, P.O. Box 1892, Houston, TX 77251-1892, U.S.A.
Email: vardi@cs.rice.edu, URL: http://www.cs.rice.edu/~vardi

Abstract. In computer system design, we distinguish between closed and open systems. A *closed system* is a system whose behavior is completely determined by the state of the system. An *open system* is a system that interacts with its environment and whose behavior depends on this interaction. The ability of temporal logics to describe an ongoing interaction of a reactive program with its environment makes them particularly appropriate for the specification of open systems. Nevertheless, model-checking algorithms used for the verification of closed systems are not appropriate for the verification of open systems. Correct model checking of open systems should check the system with respect to arbitrary environments and should take into account uncertainty regarding the environment. This is not the case with current model-checking algorithms and tools. In this paper we introduce and examine the problem of *model checking of open systems* (*module checking*, for short). We show that while module checking and model checking coincide for the linear-time paradigm, module checking is much harder than model checking for the branching-time paradigm. We prove that the problem of module checking is EXPTIME-complete for specifications in CTL and is 2EXPTIME-complete for specifications in CTL*. This bad news is also carried over when we consider the program-complexity of module checking. As good news, we show that for the commonly-used fragment of CTL (universal, possibly, and always possibly properties), current model-checking tools do work correctly, or can be easily adjusted to work correctly, with respect to both closed and open systems.

1 Introduction

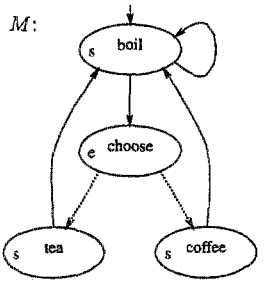
In computer system design, we distinguish between *closed* and *open* systems [HP85]. A *closed system* is a system whose behavior is completely determined by the state of the system. An *open system* is a system that interacts with its environment and whose behavior depends on this interaction. As an example to closed and open systems, we can think of two drink-dispensing machines. One machine, which is a closed system, repeatedly boils water, makes an *internal nondeterministic* choice, and serves either coffee or tea. The second machine, which is an open system, repeatedly boils water, asks the environment to choose between coffee and tea, and *deterministically* serves a drink according to the *external* choice [Hoa85]. Both machines induce the same infinite tree of possible executions. Nevertheless, while the behavior of the first machine is determined by internal choices solely, the behavior of the second machine is determined also by external choices, made by its environment. Formally, in a closed system, the environment can not modify any of the system variables. In contrast, in an open system, the environment can modify some of the system variables.

Designing correct open systems is not an easy task. The design has to be correct with respect to any environment, and often there is much uncertainty regarding the environment [FZ88]. Therefore, in the context of open systems, formal specification and verification of the design has great importance. Traditional formalisms for specification of systems relate the initial state and the final state of a system [Flo67, Hoa69]. In 1977, Pnueli suggested *temporal logics* as a suitable formalism for reasoning about the correctness of *nonterminating systems* [Pnu77]. The breakthrough that temporal logics brought to the area of specification and verification arises from their ability to describe an *ongoing interaction* of

a *reactive module* with its environment [HP85]. This ability makes temporal logics particularly appropriate for the specification of open systems.

Two possible views regarding the nature of time induce two types of temporal logics [Lam80]. In *linear* temporal logics, time is treated as if each moment in time has a unique possible future. Thus, linear temporal logic formulas are interpreted over linear sequences and we regard them as describing the interaction of the system with its environment along a single computation. In *branching* temporal logics, each moment in time may split into various possible futures. Accordingly, the structures over which branching temporal logic formulas are interpreted are infinite trees, and they describe the possible interactions of a system with its environment. In both paradigms, we can describe the design in some formal model, specify its required behaviour with a temporal logic formula, and check formally that the model satisfies the formula. Hence the name *model checking* for the verification methods derived from this viewpoint.

We model finite-state closed systems by *programs*. We model finite-state open systems by *reactive programs* (*modules*, for short). A module is simply a program with a partition of the states into two sets. One set contains *system states* and corresponds to locations where the system makes a transition. The second set contains *environment states* and corresponds to locations where the environment makes a transition[3]. Consider the module M presented on the right. It has three system states (*boil*, *tea*, and *coffee*), and it has one environment state (*choose*). It models the second drink-dispensing machine described above. When M is in the system state *boil*, we know exactly what its possible next states are. It can either stay in the state *boil* or move to the state *choose*. In contrast, when M is in the environment state *choose*, there is no certainty with respect to the environment and we can not be sure that both *tea* and *coffee* are possible next states. For example, it might be that for some users of the machine, coffee is not a desirable option. If we ignore the partition of M's states to system and environment states and regard it as a program P, then it models the first drink-dispensing machine described above.



To see the difference between the semantics of programs and modules, let us consider two questions. Is it always possible for the first machine to eventually serve tea? This is equivalent to asking whether P satisfies the CTL formula $AGEF\,tea$, and the answer is yes. Is it always possible for the second machine to eventually serve tea? Here, the answer is no. Indeed, if the environment always choose coffee, the second machine will never serve tea. Suppose now that we check with current model-checking tools whether it is always possible for the second machine to eventually serve tea, what will be the answer? Unfortunately, model-checking tools do not distinguish between closed and open systems. They regard M as a program and answer yes.

As discussed in [MP92], when the specification is given in linear temporal logic, there is indeed no need to worry about uncertainty with respect to the environment; since all the possible interactions of the system with its environment have to satisfy a linear temporal logic specification in order for M to satisfy the specification, the program P and the module M satisfy exactly the same linear temporal logic formulas. From the example above we learn that when the specification is given in branching temporal logic, we do need to take into account the uncertainty about the environment. There is a need to define a different model-checking problem for open systems, and there is a need to adjust current model-checking tools to handle open systems correctly.

[3] A similar way for modelling open systems is suggested in [LT88, Lar89]. There, Larsen and Thomsen use Modal Transition Systems, where some of the transitions are *admissible* and some are *necessary*, in order to specify processes loosely, allowing a refinement ordering between processes.

We now specify formally the problem of *model checking of open systems* (*module checking*, for short). As with usual model checking, the problem has two inputs. A module M and a temporal logic formula ψ. For a module M, let V_M denote the unwinding of M into an infinite tree. We say that M satisfies ψ iff ψ holds in all the trees obtained by pruning from V_M subtrees whose root is a successor of an environment state. The intuition is that each such tree corresponds to a different (and possible) environment. We want ψ to hold in every such tree since, of course, we want the open system to satisfy its specification no matter how the environment behaves. For example, an environment for the second drink-dispensing machine is an infinite line of thirsty people waiting for their drinks. Since each person in the line can either like both coffee and tea, or like only coffee, or like only tea, there are many different possible environments to consider. Each environment induces a different tree. For example, an environment in which all the people in line do not like tea, induces a tree that has the left subtree of all its *choose* nodes pruned. Similarly, environments in which the first person in line like both coffee and tea induce trees in which the first *choose* node has two successors[4].

We examine the *complexity* of the module-checking problem for linear and branching temporal logics. Recall that for the linear paradigm, the problem of module checking coincides with the problem of model checking. Hence, the known complexity results for LTL model checking remain valid. As we have seen, for the branching paradigm these problems do not coincide. We show that the problem of module checking is much harder. In fact, it is as hard as satisfiability. Thus, CTL module checking is EXPTIME-complete and CTL* module checking is 2EXPTIME-complete, both worse than the PSPACE complexity we have for LTL. Keeping in mind that CTL model checking can be done in linear time [CES86] and CTL* model checking can be done in polynomial space [EL85], this is really bad news. We also show that for CTL and CTL*, the program complexity of module checking (i.e., the complexity of this problem in terms of the size of the module, assuming the formula is fixed), is PTIME-complete, worse than the NLOGSPACE complexity we have for LTL. As the program complexity of model checking for both CTL and CTL* is NLOGSPACE [BVW94], this is bad news too.

As a consolation for the branching-time paradigm, we show that from a practical point of view, our news is not *that* bad. We show that in the absence of existential quantification, module checking and model checking do coincide. Thus, $\forall$CTL module checking can be done in linear time, and its program complexity is NLOGSPACE. More consolation can be found in "possibly" and "always possibly" properties. These classes of properties are considered an advantage of the branching paradigm. While being easily specified using the CTL formulas $EF\xi$ and $AGEF\xi$, these properties can not be specified in LTL [EH86]. We show that module checking of the formulas $EF\xi$ and $AGEF\xi$ can be done in linear time (though the problems are PTIME-complete).

2 Preliminaries

The logic *CTL** combines both branching-time and linear-time operators. Formulas of CTL* are defined with respect to a set AP of atomic propositions. A *path quantifier*, either E ("for some path") or A ("for all paths"), can prefix a *path formula* composed of an arbitrary combination of the linear-time operators F ("eventually"), G ("always"), X ("next time"), and U ("until").

The semantics of CTL* is defined with respect to a *program* $P = \langle AP, W, R, w_0, L\rangle$, where AP is the set of atomic propositions, W is a set of states, $R \subseteq W \times W$ is a transition relation that must be total (i.e., for every $w \in W$ there exists $w' \in W$ such that $R(w, w')$), w_0 is an initial state, and $L : W \to 2^{AP}$ maps each state to a set of atomic propositions true in this state. A *path* of P is an infinite sequence $w_0, w_1, \ldots$ of states such that for every $i \geq 0$, we have $R(w_i, w_{i+1})$. The notation

[4] Readers familiar with game theory can view module checking as solving an *infinite game* between the system and the environment. A correct system is then one that has a winning strategy in this game.

$P \models \varphi$ indicates that the formula φ holds at state w_0 of the program P. A formal definition of the relation $\models$ can be found in [Eme90].

The logic *CTL* is a restricted subset of CTL* in which the temporal operators must be immediately preceded by a path quantifier. Thus, for example, the CTL* formula $\varphi = AGF(p \wedge EXq)$ is not a CTL formula. Adding a path quantifier, say A, before the F temporal operator in φ results in the formula $AGAF(p \wedge EXq)$, which is a CTL formula. The logics $\forall$*CTL* and $\forall$*CTL**, known as the *universal fragments* of CTL and CTL*, respectively, allow only universal quantification of path formulas. Thus, all the occurrences of the path quantifier E should be under an odd number of negations. The formula φ above is therefore not a $\forall$CTL* formula. Changing the path quantifier E in φ to the path quantifier A results in the formula $AGF(p \wedge AXq)$, which is a $\forall$CTL* formula. The logic *LTL* is a linear temporal logic. Its syntax does not allow any path quantification. Formulas of LTL are interpreted over paths in a program. The notation $P \models \psi$ indicates that the LTL formula ψ holds in all the paths of the program P.

A *closed system* is a system whose behavior is completely determined by the state of the system. We model a closed system by a program. An *open system* is a system that interacts with its environment and whose behavior depends on that interaction. We model an open system by a *module* $M = \langle AP, W_s, W_e, R, w_0, L\rangle$, where AP, R, w_0, and L are as in programs, W_s is a set of *system states*, W_e is a set of *environment states*, and we often use W to denote $W_s \cup W_e$.

For each state $w \in W$, let $succ(w)$ be the set of w's R-successors; i.e., $succ(w) = \{w' : R(w, w')\}$. Consider a system state w_s and an environment state w_e. Whenever a module is in the state w_s, all the states in $succ(w_s)$ are possible next states. In contrast, when the module is in state w_e, there is no certainty with respect to the environment transitions and not all the states in $succ(w_e)$ are possible next states. The only thing guaranteed is that not all the environment transitions are impossible, since the environment can never be blocked. For a state $w \in W$, let $step(w)$ denote the set of the possible sets of w's next successors during an execution. By the above, $step(w_s) = \{succ(w_s)\}$ and $step(w_e)$ contains all the nonempty subsets of $succ(w_e)$.

An *infinite tree* is a set $T \subseteq \mathbb{N}^*$ such that if $x \cdot c \in T$ where $x \in \mathbb{N}^*$ and $c \in \mathbb{N}$, then also $x \in T$, and for all $0 \leq c' < c$, we have that $x \cdot c' \in T$. In addition, if $x \in T$, then $x \cdot 0 \in T$. The elements of T are called *nodes*, and the empty word ϵ is the *root* of T. Given an alphabet Σ, a *Σ-labeled tree* is a pair $\langle T, V\rangle$ where T is a tree and $V : T \to \Sigma$ maps each node of T to a letter in Σ. A module M can be unwound into an infinite tree $\langle T_M, V_M\rangle$ in a straightforward way. When we examine a specification with respect to M, it should hold not only in $\langle T_M, V_M\rangle$ (which corresponds to a very specific environment that does never restrict the set of its next states), but in all the trees obtained by pruning from $\langle T_M, V_M\rangle$ subtrees whose root is a successor of a node corresponding to an environment state. Let $exec(M)$ denote the set of all these trees. Formally, $\langle T, V\rangle \in exec(M)$ iff the following holds:

- $\epsilon \in T$ and $V(\epsilon) = w_0$.
- For all $x \in T$ with $V(x) = w$, there exists $\{w_0, \ldots, w_n\} \in step(w)$ such that $T \cap \mathbb{N}^{|x|+1} = \{x \cdot 0, x \cdot 1, \ldots, x \cdot n\}$ and for all $0 \leq c \leq n$ we have $V(x \cdot c) = w_c$.

Intuitively, each tree in $exec(M)$ corresponds to a different behaviour of the environment. Note that a single environment state with more than one successor suffices to make $exec(M)$ infinite. We will sometimes view the trees in $exec(M)$ as 2^{AP}-labeled trees, taking the label of a node x to be $L(V(x))$. Which interpretation is intended will be clear from the context.

Given a module M and a CTL* formula ψ, we say that M satisfies ψ, denoted $M \models_\tau \psi$, if all the trees in $exec(M)$ satisfy ψ. The problem of deciding whether M satisfies ψ is called *module checking*[5]. We use $M \models \psi$ to indicate that when we regard M as a program (thus refer to all its states as system

[5] A different problem where a specification is checked to be correct with respect to any environment is discussed in [ASSSV94]. There, all the states of the module are system states, and the formula should hold in all compositions that contain the module as a component.

states), then M satisfies ψ. The problem of deciding whether $M \models \psi$ is the usual model-checking problem [CE81, QS81]. Let $A \mapsto B$ denote that A implies B but B does not necessarily imply A. It is easy to see that

$$M \models_r \psi \mapsto M \models \psi \mapsto M \not\models_r \neg\psi.$$

Indeed, $M \models_r \psi$ requires all the trees in $exec(M)$ to satisfy ψ. On the other hand, $M \models \psi$ means that the tree $\langle T_M, V_M \rangle$ satisfies ψ. Finally, $M \not\models_r \neg\psi$ only tells us that there exists some tree in $exec(M)$ that satisfies ψ.

We can define module checking also with respect to linear-time specifications. We say that a module M satisfies an LTL formula ψ iff $M \models_r A\psi$.

3 Module Checking for Branching Temporal Logics

We have already seen that for branching temporal logics, the model checking problem and the module checking problem do not coincide. In this section we study the complexity of CTL and CTL* module checking. We show that not only the problems do not coincide but also their complexities do not coincide, and in a very significant manner.

Theorem 1.

(1) *The module-checking problem for CTL is EXPTIME-complete.*
(2) *The module-checking problem for CTL* is 2EXPTIME-complete.*

Proof (sketch): We start with the upper bounds. Consider a CTL formula ψ and a set $\mathcal{D} \subset \mathbb{N}$ with a maximal element k. Let $\mathcal{A}_{\mathcal{D},\neg\psi}$ be a Büchi tree automaton that accepts exactly all the tree models of $\neg\psi$ with branching degrees in $\mathcal{D}$. By [VW86b], such $\mathcal{A}_{\mathcal{D},\neg\psi}$ of size $O(2^{k\cdot|\psi|})$ exists.

Given a module $M = \langle AP, W_s, W_e, R, w_0, L \rangle$, we define a Büchi tree automaton $\mathcal{A}_M$ that accepts the set of all trees in $exec(M)$. Intuitively, $\mathcal{A}_M$ guesses which subtrees of $\langle T_M, V_M \rangle$ are pruned. Formally, $\mathcal{A}_M = \langle 2^{AP}, \mathcal{D}, W, \delta, w_0, W \rangle$ where $\mathcal{D}$ and δ are as follows.

- $\mathcal{D} = \bigcup_{w \in W_s} \{|succ(w)|\} \cup \bigcup_{w \in W_e} \{1, \ldots, |succ(w)|\}$.
- For every $w \in W$, $\sigma \in 2^{AP}$, and $d \in \mathcal{D}$, we have $\langle w_1, \ldots, w_d \rangle \in \delta(w, \sigma, d)$ iff $L(w) = \sigma$ and $\{w_1, \ldots, w_d\} \in step(w)$.

Since the acceptance condition only requires $\mathcal{A}_M$ not to get stuck (note that δ is partial), it is easy to see that $\mathcal{L}(\mathcal{A}_M) = exec(M)$. Since for every environment state w, the set $step(w)$ considers all possible subsets of $succ(w)$, the size of $\mathcal{A}_M$ is exponential in $\max_{w \in W_e}\{|succ(w)|\}$, thus exponential in the size of M.

By the definition of satisfaction, we have that $M \models_r \psi$ iff all the trees in $exec(M)$ satisfy ψ. In other words, if no tree in $exec(M)$ satisfies $\neg\psi$. This can be checked by testing $\mathcal{L}(\mathcal{A}_M) \cap \mathcal{L}(\mathcal{A}_{\mathcal{D},\neg\psi})$ for emptiness. Equivalently, we have to test $\mathcal{L}(\mathcal{A}_M \times \mathcal{A}_{\mathcal{D},\neg\psi})$ for emptiness. By [VW86b], the nonemptiness problem of Büchi tree automata can be solved in quadratic time, which gives us an algorithm of time complexity $2^{O(|M|+k\cdot|\psi|)}$. We can, however, do better. By [VW86a], the number of states in the automaton $\mathcal{A}_{\mathcal{D},\neg\psi}$ is $2^{O(|\psi|)}$ and is independent of k. Also, the automaton $\mathcal{A}_M$ has the same number of states as M. The fact that the sizes of these automata are exponential in k and M originates from a special structure where all subsets of a certain tuple in the transition relation are possible tuples too. Therefore, the algorithm in [VW86b] can be implemented to test $\mathcal{L}(\mathcal{A}_M \times \mathcal{A}_{\mathcal{D},\neg\psi})$ for emptiness in time polynomial in $|M| * 2^{|\psi|}$.

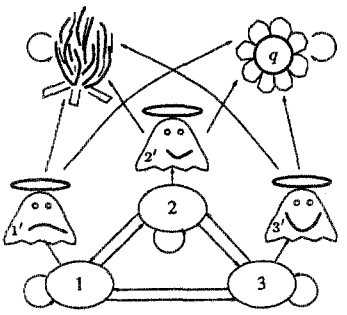
The proof is similar for CTL*. Here, following [ES84, EJ88], we have that $\mathcal{A}_{\mathcal{D},\neg\psi}$ is a Rabin tree automaton with $2^{k\cdot 2^{|\psi|}}$ states and $2^{|\psi|}$ pairs. By [EJ88, PR89], and again, using the restricted structures

of the automata $\mathcal{A}_{\mathcal{D},\neg\psi}$ and $\mathcal{A}_M$, checking the emptiness of $\mathcal{L}(\mathcal{A}_M \times \mathcal{A}_{\mathcal{D},\neg\psi})$ can then be done in time $|M|^{O(|\psi|)} * 2^{2^{O(|\psi|)}}$.

It remains to prove the lower bounds. To get an EXPTIME lower bound for CTL, we reduce CTL satisfiability, proved to be EXPTIME-complete in [FL79], to CTL module checking. Given a CTL formula ψ, we construct a module M and a CTL formula φ such that the size of M is quadratic in the length of ψ, the length of φ is linear in the length of ψ, and ψ is satisfiable iff $M \not\models_r \neg\varphi$.

Consider a CTL formula ψ. For simplicity, let us first assume that ψ has a single atomic proposition q. Let n be the number of existential quantifiers in ψ plus 1. By the sufficient branching-degree property of CTL, ψ is satisfiable iff there exists a $\{\emptyset, \{q\}\}$-labeled tree of branching degree n that satisfies ψ [Eme90]. Let P_n be a clique with n states. By the above, ψ is satisfiable iff there exists a possibility to label an unwinding of P_n such that the resulted $\{\emptyset, \{q\}\}$-labeled tree satisfies ψ. This simple idea, due to [Kup95], is the key to our reduction. We define a module M_n such that each tree in $exec(M_n)$ corresponds to a $\{\emptyset, \{q\}\}$-labeling of $\langle T_{P_n}, V_{P_n}\rangle$. We then define φ such that there exists a tree satisfying φ in $exec(M_n)$ iff there exists a $\{\emptyset, \{q\}\}$-labeling of $\langle T_{P_n}, V_{P_n}\rangle$ that satisfies ψ. It follows that ψ is satisfiable iff $M \not\models_r \neg\varphi$. Let $[n] = \{1, \ldots, n\}$, $[n]' = \{1', \ldots, n'\}$, and let $M_n = \langle AP, W_s, W_e, R, w, L\rangle$, where,

- $AP = \{ghost, q\}$.
- $W_s = [n]$.
- $W_e = [n]' \cup \{heaven, hell\}$.
- $R = \{\langle i,j\rangle : i,j \in [n]\} \cup \{\langle i,i'\rangle : i \in [n]\} \cup ([n]' \times \{heaven, hell\}) \cup \{\langle heaven, heaven\rangle\} \cup \{\langle hell, hell\rangle\}$.
- $w = 1$.
- For all $i \in [n]$, we have $L(i) = \emptyset$ and $L(i') = \{ghost\}$. Also, $L(heaven) = \{q\}$ and $L(hell) = \emptyset$.



The reactive module M_3

That is, the system states of M_n induce the clique P_n. In addition, each system state has a ghost: an environment state with two successors, one labeled with q and one not labeled with q. Intuitively, the ability of the ghost i' to take an environment transition to heaven in M_n, corresponds to the ability of a node associated with the state i in $\langle T_{P_n}, V_{P_n}\rangle$ to be labeled with q. Thus, each tree in $exec(M_n)$ indeed corresponds to a $\{\emptyset, \{q\}\}$-labeling of $\langle T_{P_n}, V_{P_n}\rangle$. We now have to define φ such that whenever the formula ψ refers to q, the formula φ will refer to $EXEXq$. Indeed, since *heaven* is the only state labeled with q, then a system state satisfies $EXEXq$ iff the transition of its ghost to heaven is enabled. In addition, path quantification in φ should be restricted to computations of P_n. That is, to paths that never meet a ghost. To do this, we define a function f : CTL* formulas $\rightarrow$ CTL* formulas such that $f(\xi)$ restricts path quantification to paths that never visit a state labeled with *ghost*. We define f inductively as follows.

- $f(q) = q$.
- $f(\neg\xi) = \neg f(\xi)$.
- $f(\xi_1 \vee \xi_2) = f(\xi_1) \vee f(\xi_2)$.
- $f(E\xi) = E((G\neg ghost) \wedge f(\xi))$.
- $f(A\xi) = A((Fghost) \vee f(\xi))$.
- $f(X\xi) = Xf(\xi)$.
- $f(\xi_1 U \xi_2) = f(\xi_1) U f(\xi_2)$.

For example, $f(EqU(AFp)) = E((G\neg ghost) \wedge (qU(A((Fghost) \vee Fq))))$. We can now define φ as $f(\psi)$ with $EXEXq$ replacing q. Note that we first apply f and only then do the replacement. When ψ is a CTL formula, the formula $f(\psi)$ is not necessarily a CTL formula. Still, it has a restricted syntax: its path formulas have either a single linear-time operator or two linear-time operators connected by a Boolean operator. By [BG94], formulas of this syntax have a linear translation to CTL.

When ψ has more than one atomic proposition, the reduction is very similar. Then, for ψ over $\{q_1, \ldots, q_m\}$, we have m heavens, one for each atomic proposition, and we associate with each system state m ghosts, again, one for each atomic propositions. We can now replace a proposition q_i in ψ with $EXEXq_i$ in φ. The obtained module has $n + nm + m + 1$ states and it has $n^2 + 3nm + m + 1$ transitions.

The proof is the same for CTL*. Here, we do a reduction from satisfiability of CTL*, proved to be 2EXPTIME-hard in [VS85]. □

We note that the problem of CTL module checking is EXPTIME-complete (and the one for CTL* is 2EXPTIME-complete) even when we restrict ourselves to modules in which all states are environment states. To see this, note we could have defined M_n as the clique P_n, adding a transition from each state to heaven. We could then force each node of a tree in $exec(M_n)$ to have as children at least its n successors in P_n (this can be enforced by the formula, having $[n]$ as atomic propositions, and having formulas like $AG(1 \rightarrow EX2 \wedge EX3)$ conjuncted with the original formula), and replace q in ψ with EXq in φ. The price of using only environment states is that now the length of φ is quadratic in the length of ψ.

Moreover, module checking for CTL is EXPTIME-complete even for modules of a fixed size. To see this, note that the size of M_n depends on the number of atomic propositions in ψ and on the minimum branching degree of models of ψ. Proving that the satisfiability problem for CTL is EXPTIME-hard, Fisher and Lander reduce acceptance of a word x by a linear-space alternating Turing machine to satisfiability of a CTL formula ψ_x [FL79]. A somewhat different reduction, which considers a fixed Turing machine that accepts an EXPTIME-complete problem, results in ψ_x of length polynomial in $|x|$, but with a fixed number of atomic propositions, which, if satisfiable, has models with branching degree 2. Such ψ_x induces, for all x, modules of a fixed size.

4 The Program Complexity of Module Checking

When analyzing the complexity of model checking, a distinction should be made between complexity in the size of the input structure and complexity in the size of the input formula; it is the complexity in size of the structure that is typically the computational bottleneck [LP85]. In this section we consider the *program complexity* [VW86a] of module checking; i.e., the complexity of this problem in terms of the size of the input module, assuming the formula is fixed. It is known that the program complexity of LTL, CTL, and CTL* model checking is NLOGSPACE [VW86a, BVW94]. This is very significant since it implies that if the system to be checked is obtained as the product of the components of a concurrent program (as is usually the case), the space required is polynomial in the size of these components rather than of the order of the exponentially larger composition.

We have seen that for CTL and CTL*, module checking is much harder than model checking. We now claim that when we consider program complexity, module checking is still harder.

Theorem 2. *The program complexity of CTL and CTL* module checking is PTIME-complete.*

Proof (sketch): Since the algorithms given in the proof of Theorem 1 are polynomial in the size of the module, membership in PTIME is immediate.

We prove hardness in PTIME by reducing the Monotonic Circuit Value Problem (MCV), proved to be PTIME-hard in [Gol77], to module checking of the CTL formula EFp. In the MCV problem,

we are given a monotonic Boolean circuit α (i.e., a circuit constructed solely of AND gates and OR gates), and a vector $\langle x_1, \ldots, x_n \rangle$ of Boolean input values. The problem is to determine whether the output of α on $\langle x_1, \ldots, x_n \rangle$ is 1.

Let us denote a monotonic circuit by a tuple $\alpha = \langle G_\forall, G_\exists, G_{in}, g_{out}, T \rangle$, where $G_\forall$ is the set of AND gates, $G_\exists$ is the set of OR gates, G_{in} is the set of input gates (identified as $g_1, \ldots, g_n$), $g_{out} \in G_\forall \cup G_\exists \cup G_{in}$ is the output gate, and $T \subset G \times G$ denotes the acyclic dependencies in α, that is $\langle g, g' \rangle \in T$ iff the output of gate g' is an input of gate g.

Given a monotonic circuit $\alpha = \langle G_\forall, G_\exists, G_{in}, g_{out}, T \rangle$ and an input vector $\mathbf{x} = \langle x_1, \ldots, x_n \rangle$, we construct a module $M_{\alpha,\mathbf{x}} = \langle \{0,1\}, G_\forall, G_\exists \cup G_{in}, R, g_{out}, L \rangle$, where

- $R = T \cup \{\langle g, g \rangle : g \in G_{in}\}$.
- For $g \in G_\forall \cup G_\exists$, we have $L(g) = 1$. For $g_i \in G_{in}$, we have $L(g_i) = x_i$.

Clearly, the size of $M_{\alpha,\mathbf{x}}$ is linear in the size of α. Intuitively, each tree in $exec(M_{\alpha,\mathbf{x}})$ corresponds to a decision of α as to how to satisfy its OR gates (we satisfy an OR gate by satisfying any nonempty subset of its inputs). It is therefore easy to see that $M_{\alpha,\mathbf{x}} \models_r EF0$ iff there exists no $V \in exec(M_{\alpha,\mathbf{x}})$ such that $V \models AG1$, which holds iff the output of α on x is 0. □

Recall that for a CTL formula ψ, checking that a module M satisfies ψ reduces to testing emptiness of the automaton $\mathcal{A}_M \times \mathcal{A}_{\mathcal{D},\neg\psi}$. Checking nonemptiness of a Büchi tree automaton can be reduced to calculating a μ-calculus expression of alternation depth 2 [Rab69, VW86b]. As such, it can be implemented, using symbolic methods, in tools that handle fixed-point calculations (e.g., SMV [BCM+90, McM93]).

5 Pragmatics

How bad is our news? In this section we show that from a pragmatic point of view, it is not *that* bad. We show that in the absence of existential quantification, module checking and model checking coincide, and that in the case where there is only a limited use of existential quantification, module checking can still be done in linear time.

5.1 Module Checking for Universal Temporal Logics

Lemma 3. *For universal branching temporal logics, the module checking problem and the model checking problem coincide.*

Proof: Given a module M and a $\forall$CTL* formula ψ, we prove that $M \models_r \psi$ iff $M \models \psi$. Assume first that $M \models_r \psi$. Then, all trees in $exec(M)$ satisfy ψ. Thus, in particular, $\langle T_M, V_M \rangle$ satisfies ψ and $M \models \psi$. Assume now that $M \models \psi$. The relation $\{\langle w, w \rangle : w \in W\}$ is a simulation relation between any tree in $exec(M)$ and M. Therefore, by [GL94], all trees in $exec(M)$ satisfy ψ, and $M \models_r \psi$. □

Theorem 4 now follows from the known complexity results for $\forall$CTL and $\forall$CTL* model checking [CES86, SC85, BVW94].

Theorem 4.

(1) *The module-checking problem for $\forall$CTL is in linear time.*
(2) *The module-checking problem for $\forall$CTL* is PSPACE-complete.*
(3) *The program complexity of module checking for $\forall$CTL and $\forall$CTL* is NLOGSPACE-complete.*

It follows from the above theorem that the module-checking problem for LTL is PSPACE-complete and its program complexity is NLOGSPACE-complete.

5.2 Module Checking of "Possibly" and "Always Possibly" Properties

We have seen that, for each fixed CTL formula ψ, checking that a module M satisfies ψ can be checked in time polynomial in the size of M. Sometimes, we can do even better. Some CTL formulas have a special structure that enables us to module-check them in time linear in the size of M. In this section we show that "possibly" and "always possibly" properties, by far the most popular properties specified in CTL and not specifiable in $\forall$CTL, induce such formulas.

Consider the CTL formula $EF send$. The formula states that it is possible for the system to eventually send a request. We call properties of this form *possibly properties*. Consider now the CTL formula $AGEF send$. The formula states that in all computations, it is always possible for the system to eventually send a request. We call properties of this form *always possibly properties*. It is easy to see that possibly and always possibly properties can not be specified in linear temporal logics, nor in universal branching logics [EH86].

Theorem 5. *Module checking of possibly and always possibly properties can be done in linear running time.*

Proof (sketch): We describe an efficient algorithm that module-checks these properties. For simplicity, we assume that system and environment states are labeled with atomic propositions s and e, respectively. Consider a module $M = \langle AP, W_s, W_e, R, w_0, L\rangle$ and a propositional assertion ξ. By definition, $M \models_r EF\xi$ iff there exists no tree $\langle T, V\rangle \in exec(M)$ all of whose nodes satisfy $\neg\xi$. We say that a state $w \in W$ is *safe* iff such a tree $\langle T, V\rangle$ can not have w as its root. We check that $M \models_r EF\xi$ by checking that w_0 is safe. In order to be safe, a state w should satisfy one of the following:

1. $w \models \xi$,
2. w is a system state that has a safe successor, or
3. w is an environment state all of whose successors are safe.

Consider the monotone function $f : 2^W \to 2^W$ where $f(y) = \xi \vee (s \wedge EXy) \vee (e \wedge AXy)$. It can be shown that w is safe iff w is in the least fixed-point of f. Therefore, we have that w is safe iff $w \models \mu y.\xi \vee (s \wedge EXy) \vee (e \wedge AXy)$. Hence,

$$M \models_r EF\xi \Leftrightarrow M \models \mu y.\xi \vee (s \wedge EXy) \vee (e \wedge AXy).$$

Now, $M \models_r AGEF\xi$ iff there exists no tree $\langle T, V\rangle \in exec(M)$ such that $\langle T, V\rangle$ has a subtree $\langle T', V'\rangle$ all of whose nodes satisfy $\neg\xi$. We can therefore check that $M \models_r AGEF\xi$ by checking that all the reachable states in M are safe. Hence,

$$M \models_r AGEF\xi \Leftrightarrow M \models \nu z.[\mu y.\xi \vee (s \wedge EXy) \vee (e \wedge AXy)] \wedge AXz.$$

So, we reduced module checking of possibly and always possibly properties to model checking of an alternation-free μ-calculus formula. As the latter can be done in linear running time [Cle93], we are done.

□

Again, as our algorithms involve at most two simple fixed-point computations, they can be easily implemented symbolically.

What about the space complexity of checking these properties? Is there a nondeterministic algorithm that can check always possibly properties in logarithmic space? As the formula we used proving Theorem 2 is $EF\xi$, the answer for possibly properties is no. Unsurprisingly, this is also the answer for the more complicated always possibly properties, as we claim in the theorem below.

Theorem 6. *Module checking of possibly and always possibly properties is PTIME-complete.*

Proof (sketch): Membership in PTIME follows from Theorem 5. To prove hardness in PTIME, we do the same reduction we did for CTL. For $EF\xi$, we need no change. For $AGEF\xi$ we do the following change. Instead a self loop, each state associated with an input gate now has a transition to the initial state g_{out}. Let us call the resulted module $M'_{\alpha,x}$. It is easy to see that $M'_{\alpha,x} \models_r AGEF0$ iff there exists no $V \in exec(M'_{\alpha,x})$ such that $V \models EFAG1$, which holds iff the output of α on x is 0. □

6 Discussion

The discussion of the relative merits of linear versus branching temporal logics is almost as early as these paradigms [Lam80]. We mainly refer here to the linear temporal logic LTL and the branching temporal logic CTL. One of the beliefs dominating this discussion has been "while specifying is easier in LTL, model checking is easier for CTL". Indeed, the restricted syntax of CTL limits its expressive power and many important behaviors (e.g., strong fairness) can not be specified in CTL. On the other hand, while model checking for CTL can be done in time $O(|P| * |\psi|)$ [CES86], it takes time $O(|P| * 2^{|\psi|})$ for LTL [LP85]. Since LTL model checking is PSPACE-complete [SC85], the latter bound probably cannot be improved. The attractive complexity of CTL model checking have compensated for its lack of expressive power and branching-time model-checking tools that can handle systems with more than 10^{120} states [Bro86, McM93, CGL93] are incorporated into industrial development of new designs [BBG+94].

If we examine the history of this discussion more closely, we found that things are not that simple. On the one hand, the inability of LTL to quantify computations existentially is considered by many a serious drawback. In addition, the introduction of fair-CTL [CES86] and of many other extensions to CTL [Lon93, BBG+94, BG94], have made CTL a basis for specification languages that maintain the efficiency of CTL model checking and yet overcome many of its expressiveness limitations. On the other hand, the computational superiority of CTL is also not that clear. For example, comparing the complexities of CTL and LTL model checking for concurrent programs, both are in PSPACE [VW86a, BVW94]. As shown in [Var95, KV95], the advantage that CTL enjoys over LTL disappears also when the complexity of modular verification is considered.

In this work we questioned the computational superiority of the branching-time paradigm further. We showed that when reasoning about open systems, the complexity of CTL model checking is actually higher than that of LTL. Our results are summarized in the table below. All the complexities in the table denote tight bounds.

	model checking	module checking	program complexity of model checking	program complexity of module checking	satisfiability
LTL	PSPACE [SC85]	PSPACE	NLOGSPACE [VW86b]	NLOGSPACE	PSPACE [SC85]
CTL	linear-time [CES86]	EXPTIME	NLOGSPACE [BVW94]	PTIME	EXPTIME [FL79]
CTL*	PSPACE [EL85]	2EXPTIME	NLOGSPACE [BVW94]	PTIME	2EXPTIME [EJ88, VS85]
∀CTL	linear-time [CES86]	linear-time	NLOGSPACE [BVW94]	NLOGSPACE	PSPACE [KV95]
$EF\xi$ $AGEF\xi$	linear-time [CES86]	linear-time	NLOGSPACE [BVW94]	PTIME	NPTIME [GJ79]

Acknowledgments. We are grateful to Martin Abadi and Pierre Wolper for fruitful discussions on the verification of reactive systems.

References

[ASSSV94] A. Aziz, T.R. Shiple, V. Singhal, and A.L. Sangiovanni-Vincentelli. Formula-dependent equivalence for compositional CTL model checking. In *Proc. 6th Conf. on Computer Aided Verification*, volume 818 of *Lecture Notes in Computer Science*, pages 324–337, Stanford, CA, June 1994. Springer-Verlag.

[BBG+94] I. Beer, S. Ben-David, D. Geist, R. Gewirtzman, and M. Yoeli. Methodology and system for practical formal verification of reactive hardware. In *Proc. 6th Workshop on Computer Aided Verification*, volume 818 of *Lecture Notes in Computer Science*, pages 182–193, Stanford, June 1994.

[BCM+90] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *Proceedings of the 5th Symposium on Logic in Computer Science*, pages 428–439, Philadelphia, June 1990.

[BG94] O. Bernholtz and O. Grumberg. Buy one, get one free !!! In *Proceedings of the First International Conference on Temporal Logic*, volume 827 of *Lecture Notes in Artificial Intelligence*, pages 210–224, Bonn, July 1994. Springer-Verlag.

[Bro86] M.C. Browne. An improved algorithm for the automatic verification of finite state systems using temporal logic. In *Proceedings of the First Symposium on Logic in Computer Science*, pages 260–266, Cambridge, June 1986.

[BVW94] O. Bernholtz, M.Y. Vardi, and P. Wolper. An automata-theoretic approach to branching-time model checking. In D. L. Dill, editor, *Computer Aided Verification, Proc. 6th Int. Conference*, volume 818 of *Lecture Notes in Computer Science*, pages 142–155, Stanford, June 1994. Springer-Verlag, Berlin.

[CE81] E.M. Clarke and E.A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Proc. Workshop on Logic of Programs*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71. Springer-Verlag, 1981.

[CES86] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, January 1986.

[CGL93] E.M. Clarke, O. Grumberg, and D. Long. Verification tools for finite-state concurrent systems. In J.W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *Decade of Concurrency – Reflections and Perspectives (Proceedings of REX School)*, Lecture Notes in Computer Science, pages 124–175. Springer-Verlag, 1993.

[Cle93] R. Cleaveland. A linear-time model-checking algorithm for the alternation-free modal μ-calculus. *Formal Methods in System Design*, 2:121–147, 1993.

[EH86] E.A. Emerson and J.Y. Halpern. Sometimes and not never revisited: On branching versus linear time. *Journal of the ACM*, 33(1):151–178, 1986.

[EJ88] E.A. Emerson and C. Jutla. The complexity of tree automata and logics of programs. In *Proceedings of the 29th IEEE Symposium on Foundations of Computer Science*, White Plains, October 1988.

[EL85] E.A. Emerson and C.-L. Lei. Temporal model checking under generalized fairness constraints. In *Proc. 18th Hawaii International Conference on System Sciences*, Hawaii, 1985.

[Eme90] E.A. Emerson. Temporal and modal logic. *Handbook of theoretical computer science*, pages 997–1072, 1990.

[ES84] E.A. Emerson and A. P. Sistla. Deciding branching time logic. In *Proceedings of the 16th ACM Symposium on Theory of Computing*, Washington, April 1984.

[FL79] M.J. Fischer and R.E. Ladner. Propositional dynamic logic of regular programs. *J. of Computer and Systems Sciences*, 18:194–211, 1979.

[Flo67] R.W. Floyd. Assigning meaning to programs. In *Proceedings Symposium on Applied Mathematics*, volume 19, 1967.

[FZ88] M.J. Fischer and L.D. Zuck. Reasoning about uncertainty in fault-tolerant distributed systems. In M. Joseph, editor, *Proc. Symp. on Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 142–158. Springer-Verlag, 1988.

[GJ79] M. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-completeness*. W. Freeman and Co., San Francisco, 1979.

[GL94] O. Grumberg and D.E. Long. Model checking and modular verification. *ACM Trans. on Programming Languages and Systems*, 16(3):843–871, 1994.

[Gol77] L.M. Goldschlager. The monotone and planar circuit value problems are log space complete for p. *SIGACT News*, 9(2):25–29, 1977.

[Hoa69] C.A.R. Hoare. An axiomatic basis of computer programming. *Communications of the ACM*, 12(10):576–583, 1969.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[HP85] D. Harel and A. Pnueli. On the development of reactive systems. In K. Apt, editor, *Logics and Models of Concurrent Systems*, volume F-13 of *NATO Advanced Summer Institutes*, pages 477–498. Springer-Verlag, 1985.

[Kup95] O. Kupferman. Augmenting branching temporal logics with existential quantification over atomic propositions. In *Computer Aided Verification, Proc. 7th Int. Workshop*, pages 325–338, Liege, July 1995.

[KV95] O. Kupferman and M.Y. Vardi. On the complexity of branching modular model checking. In *Proc. 6th Conferance on Concurrency Theory*, pages 408–422, Philadelphia, August 1995.

[Lam80] L. Lamport. Sometimes is sometimes "not never" - on the temporal logic of programs. In *Proceedings of the 7th ACM Symposium on Principles of Programming Languages*, pages 174–185, January 1980.

[Lar89] K.G. Larsen. Modal specifications. In *Automatic Verification Methods for Finite State Systems, Proc. Int. Workshop, Grenoble*, volume 407, pages 232–246, Grenoble, June 1989. Lecture Notes in Computer Science, Springer-Verlag.

[Lon93] D.E. Long. *Model checking, abstraction and compositional verification*. PhD thesis, Carnegie-Mellon University, Pittsburgh, 1993.

[LP85] O. Lichtenstein and A. Pnueli. Checking that finite state concurrent programs satisfy their linear specification. In *Proceedings of the Twelfth ACM Symposium on Principles of Programming Languages*, pages 97–107, New Orleans, January 1985.

[LT88] K.G. Larsen and G.B. Thomsen. A modal process logic. In *Proceedings of the 3th Symposium on Logic in Computer Science*, Edinburgh, 1988.

[McM93] K.L. McMillan. *Symbolic model checking*. Kluwer Academic Publishers, 1993.

[MP92] Z. Manna and A. Pnueli. Temporal specification and verification of reactive modules. 1992.

[Pnu77] A. Pnueli. The temporal logic of programs. In *Proc. 18th IEEE Symposium on Foundation of Computer Science*, pages 46–57, 1977.

[PR89] A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *Proceedings of the Sixteenth ACM Symposium on Principles of Programming Languages*, Austin, Januery 1989.

[QS81] J.P. Queille and J. Sifakis. Specification and verification of concurrent systems in Cesar. In *Proc. 5th Int'l Symp. on Programming*, volume 137, pages 337–351. Springer-Verlag, Lecture Notes in Computer Science, 1981.

[Rab69] M.O. Rabin. Decidability of second order theories and automata on infinite trees. *Transaction of the AMS*, 141:1–35, 1969.

[SC85] A.P. Sistla and E.M. Clarke. The complexity of propositional linear temporal logic. *J. ACM*, 32:733–749, 1985.

[Var95] M.Y. Vardi. On the complexity of modular model checking. In *Proceedings of the 10th IEEE Symposium on Logic in Computer Science*, June 1995.

[VS85] M.Y. Vardi and L. Stockmeyer. Improved upper and lower bounds for modal logics of programs. In *Proc 17th ACM Symp. on Theory of Computing*, pages 240–251, 1985.

[VW86a] M.Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proceedings of the First Symposium on Logic in Computer Science*, pages 322–331, Cambridge, June 1986.

[VW86b] M.Y. Vardi and P. Wolper. Automata-theoretic techniques for modal logics of programs. *Journal of Computer and System Science*, 32(2):182–221, April 1986.

Automatic Verification of Parameterized Synchronous Systems* (Extended Abstract)

E. Allen Emerson and Kedar S. Namjoshi

Department of Computer Sciences,
The University of Texas at Austin, U.S.A.

Abstract. Systems with an arbitrary number of homogeneous processes occur in many applications. The *Parameterized Model Checking Problem* (PMCP) is to determine whether a temporal property is true of every size instance of the system. We consider systems formed by a synchronous parallel composition of a single control process with an arbitrary number of homogeneous user processes, and show that the PMCP is decidable for properties expressed in an indexed propositional temporal logic. While the problem is in general PSPACE-complete, our initial experimental results indicate that the method is usable in practice.

1 Introduction

Systems with an arbitrary number of homogeneous processes occur in many contexts, especially in protocols for data communication, cache coherence, and classical synchronization problems. Current verification work on such systems has focussed mostly on verifying correctness for instances with a small number of processes. This does not indicate whether larger size instances are error-free, and so does not guarantee correctness in general. We are thus interested in methods that verify correctness for arbitrary size instances. Even though sometimes there is indeed a specific upper bound on the number of processes in a system, verifying such large size instances is intractable because of state explosion.

The general problem, then, is the *Parameterized Model Checking Problem* (PMCP): to determine whether a temporal property is true of every size instance of the the system. This is known to be undecidable in general [AK 86, Su 88]; however, it is decidable algorithmically for restricted classes [GS 92, EN 95], and there are methods with some degree of automation [Lu 84, ShG 89, KM 89, WL 89, V 93, CGJ 95]. This previous work (with the exception of [KM 89]) was oriented toward asynchronous systems.

We propose a fully automated approach to the PMCP for synchronous systems. We consider synchronous systems with a unique control process and an arbitrary number of homogeneous user processes. Each system is thus parameterized by the number of user processes. The processes are specified by labeled transition graphs, in which guards on each transition check the state of the control process as well as certain conditions on the global state. The correctness properties are expressed in an indexed propositional branching temporal logic, and are of the following types:

* This work was supported in part by NSF grant CCR 9415496 and SRC Contract 95-DP-388. The authors can be reached at `emerson, kedar@cs.utexas.edu` and at `http://www.cs.utexas.edu/users/{emerson, kedar}`.

1. Over the control process : formulae of the form $\mathsf{A}h$ and $\mathsf{E}h$, where h is a linear-time formula with atomic propositions over control process states,
2. Over all user processes: $\bigwedge_i \mathsf{A}h(i)$, and $\bigwedge_i \mathsf{E}h(i)$, where $h(i)$ is a linear-time formula with atomic propositions over control process states, and over user process states indexed with i.
3. Over every distinct pair of user processes : $\bigwedge_{i \neq j} \mathsf{A}h(i,j)$, and $\bigwedge_{i \neq j} \mathsf{E}h(i,j)$, where $h(i,j)$ is a linear-time formula with atomic propositions over control process states, and over user process states indexed with either i or j.

We show that the PMCP for the first type of formulae is decidable for this class of systems, and is PSPACE-complete. This decidability result is based on constructing an abstract graph in which *every* computation of *every* size instance of the system is represented by some path in the graph. However, the abstract graph may have "bad" paths that do not correspond to computations of any size instance. The heart of the algorithm is a method for identifying good paths in the abstract graph. This algorithm can be implemented in space polynomial in the size of the control and user processes. We show by a generic reduction that the PMCP is PSPACE-hard. As a result of the symmetry inherent in the system, the PMCP for the other types of formulae reduces to the PMCP for the first type. We have implemented this algorithm in SMV [McM92] and used it to check correctness of a bus arbitration protocol. Our initial experimental results indicate that the algorithm should be useful in practice.

Section 2 defines the system model and the logic used for expressing correctness properties. Section 3 describes the abstract graph representation, and Section 4 the algorithm for the PMCP for formulae of type (1). Section 5 shows the reduction of the PMCP for formulae of types (2) and (3) to the PMCP for formulae of type (1). Section 6 describes our implementation of the algorithm, and the application to the bus protocol. Section 7 concludes the paper with a discussion of related work.

2 The system model and logic

We refer to the collection of system instances formed by control process C and copies of a generic user process U as a (C,U) *family*. The control and user processes are specified as finite-state labeled transition graphs. We use the terms "process" and "labeled transition graph" interchangeably. For a process P, let S_P denote its set of states, R_P its transition relation, and ι_P its initial state[2].

The system *instance* of size n is a synchronous parallel composition of C with n copies of process U, and is denoted as $C \parallel U^n = C \parallel U_1 \parallel U_2 \ldots \parallel U_n$. U_i is the ith copy of U, which is obtained from U by uniformly subscripting the states of U with i as shown in the example below [3]:

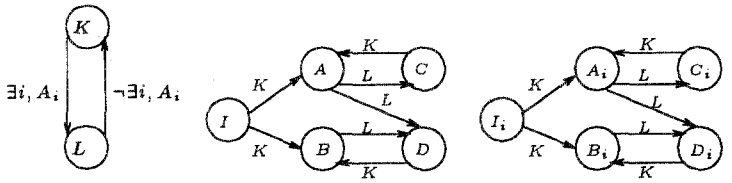


FIG 1a: The control process FIG 1b: The generic user process FIG 1c: The ith user process

[2] The results of this paper carry over for processes with a *set* of initial states.

[3] In this example, C has initial state K, and U has initial state I. Atomic propositions are identified with state names.

Thus for all i, j, U_i and U_j are isomorphic up to re-indexing. Transitions in both C and U_i are labeled with *guards*. Every guard is a boolean combination of *users conditions*, which have the form $(\exists i\ \mathcal{E}(i))$, where $\mathcal{E}(i)$ is a boolean expression formed from atomic propositions over the states of C, and over the states of U_i. [4]

$\mathcal{G}_n$ denotes the global state transition graph of the instance of size n. A state s of $\mathcal{G}_n$ is written as an $(n+1)$-tuple $(c, u_1, \ldots, v_n)$, where c is the local state of C, and the $(i+1)$'th component of the tuple is the local state of U_i (for $i \in [1..n]$). The $(i+1)$'th component of s is denoted by $s(i)$. The initial state of $\mathcal{G}_n$ is $(\iota_C, (\iota_U)_1, \ldots, (\iota_U)_n)$. A transition (s,t) is in $\mathcal{G}_n$ iff

1. A transition of C from $s(0)$ to $t(0)$ is enabled in s, and
2. For all $i \in [1..n]$, a transition of U_i from $s(i)$ to $t(i)$ is enabled in s.

where a transition in a process is said to be enabled in a global state iff the corresponding guard is true when evaluated in that global state. We write $s \models g$ iff guard g is true in the global state s. $s \models (\exists i\ \mathcal{E}(i))$ iff for some $k \in [1..n]$, $\mathcal{E}(k)$ is true given the propositions that hold at $s(0)$ (the control state), and $s(k)$ (the state of process U_k). Boolean operators are handled in the standard manner. For a global state s, and state $a \in S_U$, we let $\#a(s) = |\{i \mid i \in [1..n] \wedge s(i) = a_i\}|$ (i.e., $\#a(s)$ is the number of user processes with local state a of the generic user process).

PLTL is the standard propositional linear temporal logic built up from atomic propositions, boolean connectives, and temporal operators G (always), F (sometime), X (next time), and U (until) [Pn 77, MP 92]. *CTL** is a branching temporal logic which extends PLTL by allowing the path quantifiers A (for all fullpaths) and E (for some fullpath). Many interesting correctness properties of parameterized systems can be expressed in one of the following forms:

1. Over the control process : formulae of the form $\mathsf{A}h$ and $\mathsf{E}h$, where h is a linear-time formula with atomic propositions over control process states,
2. Over all user processes: $\bigwedge_i \mathsf{A}h(i)$, and $\bigwedge_i \mathsf{E}h(i)$, where $h(i)$ is a linear-time formula with atomic propositions over control process states, and over states of U indexed with i.
3. Over all distinct pairs of user processes : $\bigwedge_{i \neq j} \mathsf{A}h(i,j)$, and $\bigwedge_{i \neq j} \mathsf{E}h(i,j)$, where $h(i,j)$ is a linear-time formula with atomic propositions over control process states, and over states of U indexed with either i or j.

The formal semantics of these logics is defined in the usual way [Em 90, BCG 89, ES 95], and we write $M, s \models f$ to mean that formula f is true in structure M at state s.

3 The abstract model

For a given (C, U) family, we construct an abstract process $\mathcal{A}$ which includes all computations of every size instance of the family. Intuitively, a state (c, S) of $\mathcal{A}$ *represents* any global state in which the control process is in state c, there is at least one user process in every user state in S, and no user process is in a

[4] There are two interesting special cases : (a) The guards in U_i involve only propositions over states of C. The control process may then be viewed as controlling the execution of the user processes. (b) The control process is a copy of the user process, and can be written as U_0. Then $C \parallel U^n$ is isomorphic to U^{n+1}. Our method applies in general, but often finds interesting application in these special cases.

state in $S_U \backslash S$. Transitions from a state (c, S) represent transitions enabled from global states that are represented by (c, S). Each such transition has a label which represents moves of individual processes.

Formally, let $\Lambda = 2^{S_U \times S_U} \backslash \{\emptyset\}$ be the set of edge labels. $\mathcal{A}$ is defined by a labeled transition graph, where

1. $S_{\mathcal{A}} = S_C \times (2^{S_U} \backslash \{\emptyset\})$ is the set of states,
2. $R_{\mathcal{A}} \subseteq S_{\mathcal{A}} \times \Lambda \times S_{\mathcal{A}}$ is the set of transitions,
3. $\iota_{\mathcal{A}} = (\iota_C, \{\iota_U\})$ is the initial state.

To make the correspondence between global states and abstract states precise, we define families of abstraction functions $\{\phi_i\}, \{\psi_i\}$, where $\phi_n : S_{\mathcal{G}_n} \to S_{\mathcal{A}}$, and $\psi_n : S_{\mathcal{G}_n} \times S_{\mathcal{G}_n} \to \Lambda$. For a state $s \in S_{\mathcal{G}_n}$, $\phi_n(s) = (s(0), \{a \mid (\exists i \in [1..n]\, s(i) = a_i)\})$, and for a pair (s,t), $\psi_n(s,t) = \{(a,b) \mid \exists i \in [1..n]\, s(i) = a_i \wedge t(i) = b_i\}$. Then (c, S) represents $s \in \mathcal{G}_n$ iff $(c, S) = \phi_n(s)$.

For a guard g, and state (c, S) of $\mathcal{A}$, we define $(c, S) \,||\!-\, g$ as $(c, u_1, \ldots, v_k) \models g$, where $S = \{u \ldots v\}$ (for some ordering $u \ldots v$ of the elements of S) and $|S| = k$. The following proposition relates $\models$ and $||\!-$:

Proposition 1. *For any n, and any $s \in \mathcal{G}_n$, if $(c, S) = \phi_n(s)$, then for every guard expression g, $s \models g$ iff $(c, S) \,||\!-\, g$.* □

The set of transitions is defined as follows: A tuple $((c, S), X, (c', S')) \in R_{\mathcal{A}}$ iff

1. $(\exists p\ c \xrightarrow{p} c' \in R_C \wedge (c, S) \,||\!-\, p)$ (A transition from c to c' is enabled for the control process),
2. $(\forall a, b\ (a,b) \in X \Rightarrow a \in S \wedge b \in S' \wedge (\exists q\ a \xrightarrow{q} b \in R_U \wedge (c, S) \,||\!-\, q))$. (For every pair (a,b) in X, there is an enabled transition from a to b in the user process).
3. X is total on S, and X^{-1} is total on S', (Every state in S has a successor in S', and every state in S' has a predecessor in S).

Definition 2. A *path* in $\mathcal{G}_n$ is a sequence of states such that adjacent states are in the global transition relation of $\mathcal{G}_n$. □

Definition 3. A *path* in $\mathcal{A}$ is a sequence starting at a state, with alternating states and transition labels such that for every $s, s' \in S_{\mathcal{A}}$ and $X \in \Lambda$, sXs' occurs in the sequence only if $(s, X, s') \in R_{\mathcal{A}}$. □

Define a family of functions $\{\gamma_i\}$ such that γ_n maps from paths in $\mathcal{G}_n$ to paths in $\mathcal{A}$ by $(\gamma_n(\sigma))_{2i} = \phi_n(\sigma_i)$, and $(\gamma_n(\sigma))_{2i+1} = \psi_n(\sigma_i, \sigma_{i+1})$ for all $i \in \mathbf{N}$.

Proposition 4. *For every path σ in $\mathcal{G}_n$, $\gamma_n(\sigma)$ is a path in $\mathcal{A}$.* □

It follows from Proposition 4 that if $\mathcal{A}$ satisfies a linear temporal formula over all paths, then so does every size instance of the family. However, if the formula is false for some path in $\mathcal{A}$, it does not follow that it is false for some instance, as not every path in $\mathcal{A}$ arises from a corresponding path in some instance; those that do are called "good".

Definition 5. A path ρ in $\mathcal{A}$ is *good* iff $\exists n\, \exists \sigma \in \mathcal{G}_n\ \gamma_n(\sigma) = \rho$. □

Definition 6. A path σ' in $\mathcal{G}_i$ *covers* a path σ in $\mathcal{G}_j$ $(i \geq j)$ iff $\gamma_i(\sigma') = \gamma_j(\sigma)$, and for every $k \in \mathbf{N}$, $a \in U$, $\#a(\sigma'_k) \geq \#a(\sigma_k)$. □

Lemma 7. *(Covering Lemma) For $n' \geq n$, every path in $\mathcal{G}_n$ has a covering path in $\mathcal{G}_{n'}$.*

Proof

Let σ be a path in $\mathcal{G}_n$. Define σ' in $\mathcal{G}_{n'}$ by the following: $\sigma'_k(0) = \sigma_k(0)$, and for $i \in [1..n']$, $\sigma'_k(i) = a_i$, where a is such that if $i \bmod n \neq 0$, then $\sigma_k(i \bmod n) = a_{i \bmod n}$, and and if $i \bmod n = 0$, then $\sigma_k(n) = a_n$.

It follows that $\phi_{n'}(\sigma'_k) = \phi_n(\sigma_k)$, and that $\#a(\sigma'_k) \geq \#a(\sigma_k)$ for all $a \in U$. To complete the proof, we need to show that $\psi_{n'}(\sigma'_k, \sigma'_{k+1}) = \psi_n(\sigma_k, \sigma_{k+1})$. Since (σ_k, σ_{k+1}) is a transition of $\mathcal{G}_n$, there exist guards $p, q_1, \ldots q_n$, such that $\sigma_k(0) \xrightarrow{p} \sigma_{k+1}(0)$ is the transition of the control process, and $\sigma_k(i) \xrightarrow{q_i} \sigma_{k+1}(i)$ is the transition of process U_i, for $i \in [1..n]$. As $\phi_{n'}(\sigma'_k) = \phi_n(\sigma_k)$, from Proposition 1, transition q_i is enabled for user processes with indices $j = i\,(mod\ n)$ in σ'_k. The resulting state is σ'_{k+1}. It follows that $\psi_{n'}(\sigma'_k, \sigma'_{k+1}) = \psi_n(\sigma_k, \sigma_{k+1})$, and so σ' is a path of $\mathcal{G}_{n'}$ that covers σ. □

Lemma 8. *Every finite path of $\mathcal{A}$ is good.*

Proof

The proof is by induction on the number of states in the path. Suppose the path is a single state s. Let $s = (c, S)$, and let $n = |S|$. Consider the state $r = (c, u_1, \ldots v_n)$ in $\mathcal{G}_n$, where $S = \{u \ldots v\}$. As $\phi_n(r) = s$, the claim is true of paths with one state. Suppose that it is true for all paths with at most m states, for $m \geq 1$, and let ρ be a path with $m + 1$ states. Then, $\rho = \rho' X t$, where if s is the last state in ρ', then $(s, X, t) \in R_{\mathcal{A}}$. By inductive hypothesis, for some n', there is a path $\sigma' \in \mathcal{G}_{n'}$ such that $\gamma_{n'}(\sigma') = \rho'$. Let r' be the last state in σ'.

For each $a \in U$, let $m_a = |\{b \mid (a, b) \in X\}|$. If for some a, $m_a > \#a(r')$, one can construct a path covering σ' such that if u is the final state on that path, then $m_a \leq \#a(u)$. Repeating this construction for each user state a for which it is necessary, we obtain, for some n, a path σ in $\mathcal{G}_n$ such that σ covers σ', and for every a, $m_a \leq \#a(r)$, where r is the last state on σ.

As $m_a \leq \#a(r)$ for each a, one can associate at least one index $i \in [1..n]$ with each pair (a, b) in X. For every pair (a, b) in X, there is an enabled transition from a to b in the user process. Thus, there is a state $u \in \mathcal{G}_n$ generated by performing the enabled transition from a_i to b_i in each process U_i where index i is associated with the pair (a, b), and the enabled transition for the control process. It is easy to verify that $\phi_n(u) = t$, and hence, σu is a path in $\mathcal{G}_n$ such that $\gamma_n(\sigma u) = \rho$. □

4 Verifying properties of the control process

The properties of the control process are of the form Ah or Eh, where h is a linear-time temporal formula with atomic propositions over the states of C. To model-check such a property, we follow the automata-theoretic approach of [VW 86] : To determine if $M, \iota_M \models \mathsf{E}h$, construct a Büchi automaton $\mathcal{B}_h$ for h, and check that the language of the product Büchi automaton of M and $\mathcal{B}_h$ is non-empty (cf. [LP85]). The check for the property Ah is easily reduced to that for the earlier case by noting that $M, \iota_M \models \mathsf{A}h$ iff $M, \iota_M \not\models \mathsf{E}\neg h$.

We say that formula Ah is *universal* iff it is true for every size instance of the family. To determine if Ah is universal, we model check it over the abstract graph, by constructing a Büchi automaton $\mathcal{B}$ for $\neg h$, and forming the product Büchi automaton $\mathcal{M}$ of $\mathcal{A}$ and $\mathcal{B}$. $\mathcal{B}$ accepts a computation σ labeled with propositions

over states of C iff there is a run of $\mathcal{B}$ on σ such that a "green" state of $\mathcal{B}$ is entered infinitely often. An *accepting* path in $\mathcal{M}$ is one which starts in an initial state, and along which a green state occurs infinitely often. For a path δ in $\mathcal{M}$, let $\delta_{\mathcal{A}}$ be its projection on $\mathcal{A}$. A path in $\mathcal{M}$ is *good* iff its projection on $\mathcal{A}$ is a good path in $\mathcal{A}$.

Theorem 9. *Formula* Ah *is not universal iff there is an accepting good path in* $\mathcal{M}$.

Proof

Suppose δ is an accepting good path in $\mathcal{M}$. As $\delta_{\mathcal{A}}$ is good, for some n, there is a path in $\mathcal{G}_n$ that matches $\delta_{\mathcal{A}}$ on the sequence of states of C, and is hence accepted by $\mathcal{B}$. Therefore, Ah is false in $\mathcal{G}_n$, and hence is not universal.

In the other direction, if Ah is not universal, then for some n, there is a path σ in $\mathcal{G}_n$ from the initial state that is accepted by $\mathcal{B}$. From Lemma 4, $\gamma_n(\sigma)$ is a path in $\mathcal{A}$, which is good by construction. The sequence of states of C in $\gamma_n(\sigma)$ is the same as in σ, hence there is a run of $\mathcal{B}$ on $\gamma_n(\sigma)$ that forms an accepting good path in $\mathcal{M}$. □

4.1 Finding accepting good paths in $\mathcal{M}$

From Theorem 9, to determine if Ah is not universal, we have to check if there is an accepting good path in $\mathcal{M}$. The following lemmas provide the basis for a PSPACE algorithm to check universality.

For a cycle δ in $\mathcal{M}$, we say that δ is good iff the infinite path δ^{ω} is good.

Lemma 10. *There is an accepting good path in* $\mathcal{M}$ *iff there are finite paths* α *and* β *in* $\mathcal{M}$, *such that*

1. α *is a path from the initial state to a green state* s, *and*
2. β *is a good cycle starting at* s. □

Intuitively, a cycle in $\mathcal{M}$ is good if, starting at some global state which maps to a state in the cycle, there is no transition in that cycle that causes the count of processes in a specific local state to be "drained" (i.e. decreased monotonically) as the sequence of transitions along the cycle is executed repeatedly. For example, a self-loop with the transition label $\{(a, b)\}$ will decrease the count of processes in state a with every execution of the transition, while one with transition label $\{(a, b), (b, a)\}$ may not. Notice that in the latter case, there is a cycle $a \rightarrow b \rightarrow a$ in the transition label considered as a graph. This presence of cycles in the transition labels is the intuition behind the characterization of good cycles of $\mathcal{M}$.

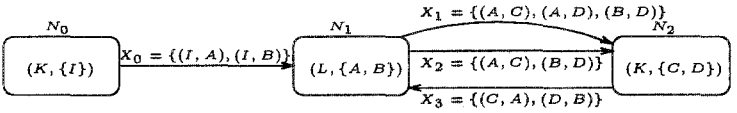


FIG 2 : A portion of the abstract graph for the example in FIG 1.

To determine if such cycles are present, we resolve a cycle in $\mathcal{M}$ into a "threaded graph" (cf. [ES 95]) which shows explicitly which local user state in an abstract state is driven into which other local user state in the next abstract state. This information is obtained from the transition label. The threaded graph is defined below:

Definition 11. Threaded Graph

Let δ be a finite path in $\mathcal{M}$ with m states. Let the ith state of δ be called s_i, and the ith transition be called X_i. For a state $s = ((c, S), u)$ of $\mathcal{M}$, let $Ustates(s) = S$. Define H_δ to be the following graph :

$V(H_\delta) = \{(x, i) \mid i \in [1..m] \wedge x \in Ustates(s_i)\}$

$E(H_\delta) = \{((x, i), (y, i+1)) \mid i \in [1..m-1] \wedge (x, y) \in X_i\}$

If δ is a cycle, define G_δ to be the graph where $V(\mathcal{G}_\delta) = V(H_\delta)$, and $E(\mathcal{G}_\delta) = E(H_\delta) \cup \{((x, m), (x, 1)) \mid x \in Ustates(s_1)\}$. Note that for a cycle δ, $s_1 = s_m$.

A graph is *isolated* iff its edge set is empty. For any directed graph G, let $maxscc(G)$ be the graph representing the decomposition of G into its maximal strongly connected components (scc's).

$V(maxscc(G)) = \{C \mid C \text{ is a maximal strongly connected component of } G\}$

$E(maxscc(G)) = \{(C, D) \mid \exists s \in C, t \in D\ (s, t) \in E(G)\}$

We refer to vertices of $maxscc(G)$ as max-scc's. It is a fact that $maxscc(G)$ is acyclic for any graph G. For any max-scc D in $maxscc(G)$, define max-scc C to be *above* D if there is a path in $maxscc(G)$ from C to D. □

The following figure shows the threaded graphs for the cycles $N_1 \xrightarrow{X_1} N_2 \xrightarrow{X_3} N_1$ and $N_1 \xrightarrow{X_2} N_2 \xrightarrow{X_3} N_1$ in figure 2:

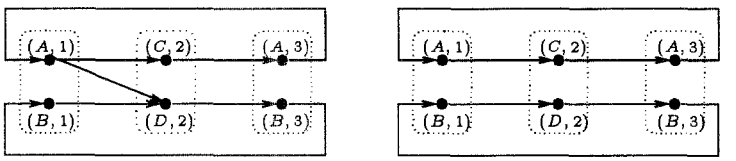


FIG 3a : Threaded graph G_δ for the first cycle FIG 3b : Threaded graph G_δ for the second cycle

Lemma 12. *δ is a good cycle in $\mathcal{M}$ iff $maxscc(G_\delta)$ is isolated.*

Proof Sketch

(LHS $\Rightarrow$ RHS): Suppose that $maxscc(G_\delta)$ is not isolated but δ is good. Hence, there are max-scc's C and D such that some pair of vertices (x, i) in C and (y, j) in D is connected in G_δ. For any n, consider an infinite path σ in $\mathcal{G}_n$ such that $\gamma_n(\sigma) = \delta_{\mathcal{A}}^\omega$. We say that a process with index $l \in [1..n]$ and local state a_l is in component F at the kth state in σ iff $(a, k) \in F$.

Let m be the number of states in δ. Starting with the ith transition in σ, at every mth successive transition, at least one of the processes in C, say one with index l, must change its local state from x_l to y_l. Thus, the count of processes in components above D decreases at each such step. As the max-scc decomposition is acyclic, this number cannot increase. Thus, eventually, the number of processes in components above D must become negative, which is impossible as σ is infinite. Hence, δ is not good.

(RHS $\Rightarrow$ LHS): Suppose that $maxscc(G_\delta)$ is isolated. For each max-scc of G_δ, construct a cycle in G_δ that includes each edge in that component at least once. For each $a \in U$, let m_a be the number of occurrences of the vertex $(a, 1)$ in the set of cycles. Let $n = \Sigma_{a \in U} m_a$. We will construct a path σ in $\mathcal{G}_n$ such that $\gamma_n(\sigma) = \delta_{\mathcal{A}}^\omega$. The idea behind the construction is to allot a set of processes for each constructed cycle, and to ensure that each transition of every process is along the cycle that it is alloted to.

The inductive assumption is that at the ith step ($i < m$), a path σ' has been constructed such that $\gamma_n(\sigma')$ is the prefix of $\delta_{\mathcal{A}}$ up to the ith state, and if s is

the last state of σ', then $\#a(s)$ is the number of occurrences of (a,i) in the set of constructed cycles. Hence, after m steps, the last state s_m is a permutation of the first state s_1. Repeating the construction at most n times produces a path σ with last state identical to s_1, and such that $\gamma_n(\sigma) = \delta_{\mathcal{A}}^k$. Thus, $\gamma_n(\sigma^\omega) = (\delta_{\mathcal{A}}^k)^\omega = \delta_{\mathcal{A}}^\omega$, and so δ is a good cycle. □

For a finite path α with m states in $\mathcal{A}$ define $\overline{\alpha}$ to be the relation over $S_U \times S_U$ where $(a,b) \in \overline{\alpha}$ iff there is a path from $(a,1)$ to (b,m) in H_α. We say that relation R is cyclic iff for every edge in the graph of R, there is there is a cycle in the graph that includes that edge.

Lemma 13. *For a cycle δ in $\mathcal{M}$, $maxscc(G_\delta)$ is isolated iff $\overline{\delta}$ is cyclic.* □

Theorem 14. *Formula Ah is not universal iff there is a finite path in $\mathcal{M}$ from an initial state to a green state and a cycle δ from that state such that $\overline{\delta}$ is cyclic.*

Proof Follows from Theorem 9 and Lemmas 12, 13. □

Let L be the maximum length of a guard in C and U processes. Note that $L \leq |C| + |U|$.

Theorem 15. *There is a nondeterministic algorithm to decide if a temporal property over computations of C is not universal which uses space $O(|S_U|^2 + log(|S_C||S_B|) + L)$. The algorithm uses space logarithmic in the size of $\mathcal{M}$.*

Proof

By Theorem 14, a property Ah is not universal iff there is a finite path in $\mathcal{M}$ to a green state and a following cycle δ from that state such that $\overline{\delta}$ is cyclic. The algorithm "guesses" a path to a green state, and a cycle δ from it, recording only the current state of $\mathcal{M}$, and $\overline{\delta'}$ for the prefix δ' of δ that has been examined. As $\overline{(\alpha; X; s)} = \overline{\alpha} \circ X$, $\overline{\delta}$ can be computed incrementally.

Recording a state of $\mathcal{M}$ takes space $(log(|S_C||S_B|) + |S_U|)$. Computing a successor state can be done in space proportional to $(log|S_B| + log|S_C| + log|S_U| + L)$ (as this requires checking if $(c,S) \,||- p$ for guards p). Storing $\overline{\delta'}$ takes space $|S_U|^2$, and checking if $\overline{\delta}$ is cyclic can be done within the same space bound. Thus, the overall space usage is $O(|S_U|^2 + log(|S_C||S_B|) + L)$. □

Remark. There are two special cases where the algorithm can be optimized. If the user processes are deterministic, every cycle δ in $\mathcal{M}$ is good (as G_δ must be isolated). If the correctness property is a safety property, the algorithm need check only finite accepting paths, which are good by Lemma 8. In both cases, the check for good cycles can be eliminated, which is a substantial saving. □

A reduction from a generic PSPACE Turing Machine shows that checking if AG¬*accept* is not universal is PSPACE-hard.

Theorem 16. *Deciding if a property over computations of C is not universal is PSPACE-complete.*

Corollary 17. *Deciding if a property over computations of C is universal is PSPACE-complete.*

The algorithm given above for determining if a property is not universal is nondeterministic and uses polynomial space. So, using Savitch's construction, there is a deterministic algorithm with time complexity $O(2^{k(|S_U|^2 + log(|S_C||S_B|) + L)^2})$ for some k. We present a "natural" deterministic algorithm with the same worst case time complexity in $|S_U|$. Let $K = |S_{\mathcal{M}}| \times 2^{|S_U|^2}$. The algorithm follows from this observation:

Proposition 18. *If ρ is a finite path in $\mathcal{M}$ from s to t of length greater than K, then there is a path δ from s to t in $\mathcal{M}$ of length at most K such that $\overline{\rho} = \overline{\delta}$.*

Proof

Define an equivalence relation on states s of ρ by $s_i \equiv s_j$ iff $s_i = s_j$ and $\overline{X_0 \circ X_1 \ldots X_{i-1}} = \overline{X_0 \circ X_1 \ldots X_{j-1}}$. Clearly there are at most K equivalence classes. So if the length of ρ is greater than K, there must be distinct indices i and j such that $s_i \equiv s_j$. Assume that $i < j$. Then the path ρ' formed by appending the suffix from s_j to the prefix up to s_i is a path in $\mathcal{M}$ that is shorter than the path ρ, and is such that $\overline{\rho'} = \overline{\rho}$. Repeating this construction a finite number of times produces a path δ with the desired properties. □

Theorem 19. *There is a deterministic algorithm to determine if a property is not universal with exponential worst case time complexity in $|S_U|$.*

Proof Sketch

From Proposition 18, it suffices to look for cycles (in Theorem 14) of length at most K. This can be done using an iterative squaring of the transition relation of $\mathcal{M}$, with overall time complexity exponential in $|S_U|$. □

5 Symmetry reduction

Let π be a permutation over the set $\{0 \ldots n\}$ that fixes 0. For a state $s = (c, u_1, \ldots, v_n)$ in $\mathcal{G}_n$, the permuted state $\pi(s)$ is defined by $(\pi(s))(i) = a_i$ iff $s(\pi^{-1}(i)) = a_{\pi^{-1}(i)}$, for $i \in [0..n]$. For example, the state (c, u_1, v_2, w_3) under the permutation $\pi = \{(1 \to 2), (2 \to 3), (3 \to 1)\}$ becomes (c, w_1, u_2, v_3). As $\phi_n(\pi(s)) = \phi_n(s)$, from Proposition 1, the truth value of any guard is the same in both s and $\pi(s)$. Hence there is complete symmetry among the user processes in any size instance of a (C, U) family, and the PMCP for formulae of type (2) and (3) reduces that for formulae of type (1). The following lemmas are based on those in [ES 93, CFJ 93] (cf. [ID 93]) Let $f(i)$ be a *CTL** formula with propositions over the states of C and over the states of U indexed with i, and let $f(i,j)$ be a *CTL** formula with propositions over the states of C and over the states of U indexed with either i or j.

Lemma 20. *For $n \geq 1$, $\mathcal{G}_n, \iota_{\mathcal{G}_n} \models \bigwedge_i f(i)$ iff $\mathcal{G}_n, \iota_{\mathcal{G}_n} \models f(1)$.*

Lemma 21. *For $n \geq 2$, $\mathcal{G}_n, \iota_{\mathcal{G}_n} \models \bigwedge_{i \neq j} f(i,j)$ iff $\mathcal{G}_n, \iota_{\mathcal{G}_n} \models f(1,2)$.*

Let $C|U$ be the process where $S_{C|U} = S_C \times S_U$, and $(c,u) \stackrel{p' \wedge q'}{\to} (c',u') \in R_{C|U}$ iff $c \stackrel{p}{\to} c' \in R_C$ and $u \stackrel{q}{\to} u' \in R_U$, and p' (similarly q') is p (q) with every global condition $(\exists i\ \mathcal{E}(i))$ replaced with $\mathcal{E}(0) \vee (\exists i\ \mathcal{E}(i))$, where propositions labeled by 0 refer to the state of U in $C|U$.

Theorem 22. *A property of the form $\bigwedge_i \mathsf{A}h(i)$ is universal for a (C,U) family iff $\mathsf{A}h(0)$ is universal for the control process in the family $(C|U, U)$.*

Theorem 23. *A property of the form $\bigwedge_{i \neq j} \mathsf{A}h(i,j)$ is universal for a (C,U) family iff $\mathsf{A}h(0,0')$ is universal for the control process in the family $((C|U|U), U)$*

6 Applications

We have implemented this algorithm to verify a bus arbitration protocol based on the SAE J1850 draft standard [SAE 92] for automobile applications. This is a protocol where many microcontrollers can transmit symbols along a shared single-wire bus in a car. As a consequence of this restriction, symbols are encoded by the width of a pulse. Nodes on the bus may begin transmitting different messages simultaneously; only the node with the highest priority message should complete transmission after the arbitration process. Symbol 0 has priority over symbol 1, and priority between messages over the alphabet $\{0, 1\}$ is determined lexicographically. The microcontrollers are modeled as user processes, and the bus as the control process. The property which we have verified, using the result in Theorem 23, is that whenever two users begin simultaneous transmission of symbols 0 and 1 respectively, the user transmitting 1 continues transmission unless it loses arbitration. Hence, messages with lower priority cannot prevail over higher priority messages.

We implemented the algorithm by generating SMV [McM92] code to describe the abstract process transitions, given a description of the next-state relation of the user and control processes. Since the correctness property is a safety property, we were able to simplify the implementation as described following Theorem 15. Each user process has about 50 states, while the control process together with the automaton for the property has about 400 states. Verification took less than a minute on a SPARC 5. We emphasize that this establishes correctness of the bus protocol for an arbitrary number of attached microcontrollers.

7 Conclusions and Related Work

A variety of positive results on the PMCP have been obtained previously. All of them, however, possess certain limitations, which is perhaps not surprising since the PMCP is undecidable in general (cf. [AK 86],[Su 88]). Many of the methods are only partially automated, requiring human ingenuity to construct, e.g., a process invariant or closure process (cf. [CG 87], [BCG 89], [KM 89], [WL 89]). Some could be fully automated but do not appear to have a clearly defined class of protocols on which they are guaranteed to succeed (cf. [ShG 89], [V 93], [CGJ 95]).

Abstract graphs (for asynchronous systems) were considered in [ESr 90] for synthesis, [V 93] for automatic but incomplete verification, and in [CG 87], where they are called process closures. Interestingly, [CG 87] show (in our notation) that if, for some k, $C \parallel U^k \parallel \mathcal{A}$ is appropriately bisimilar to $C \parallel U^{k+1} \parallel \mathcal{A}$, then it suffices to model check instances of size at most k to solve the PMCP. However, they do not show that such a cutoff k always exists, and their method is not guaranteed to be complete. Pong and Dubois [PD 95] propose a similar abstract graph construction for verification of safety properties of cache coherence protocols. They consider a synchronous model with broadcast actions. Although sound for verification, their method appears to be incomplete. Lubachevsky [Lu 84] makes an interesting early report of the use of an abstract graph similar to a "region graph" for parameterized asynchronous programs using *Fetch-and-Add* primitives; however, while it caters for (partial) automation, the completeness of the method is not established and it is not clear that it can be made fully automatic.

Our approach, in contrast, is a fully automated, sound and complete one (i.e., always generates a correct "yes" or "no" answer to the PMCP). Another such approach appears in [GS 92]. They also consider systems with a single control

process and an arbitrary number of user processes, but with asynchronous CCS-type interactions. Unfortunately, their algorithm has exponential space (double exponential time) worst case complexity.

Our framework thus differs from [GS 92] in these significant respects: (a) the parallel composition operator is synchronous; (b) we permit guards testing "everywhere" conditions (i.e., of the form $\forall i\, \mathcal{E}(i)$); (c) it is more tractable (PSPACE vs. EXPSPACE)[5]. Partial synchrony can also be handled in our framework. These factors permit us to represent a wider range of concurrent systems. For example, the bus protocol described in Section 6 relies on the ability to test everywhere conditions, which are not permitted in [GS 92]. There is a noteworthy limitation in the modeling power of our present framework. Because of the covering lemma (Lemma 7), an algorithm for mutual exclusion cannot be implemented in our model (cf. [GS 92]'s control process-free model), even with the control process. We suspect it is possible to overcome this restriction, and are working on it.

Finally, it is interesting to note that we can show that for fully asynchronous computation (interleaving semantics), the PMCP for our model becomes undecidable. This is shown by a simple simulation of a two counter machine by a (C, U) family. Essentially, the zero-test of a two counter machine can be expressed as an everywhere condition, and increments can be encoded because precisely one process fires at each step in the computation.

Acknowledgements. We would like to thank Carl Pixley of Motorola for suggesting the bus protocol example, and the referees for bringing [PD 95] to our attention.

References

[AK 86] Apt, K., Kozen, D. Limits for automatic verification of finite-state concurrent systems. *IPL* 15, pp. 307-309.

[BCG 89] Browne, M. C., Clarke, E. M., Grumberg, O. Reasoning about Networks with Many Identical Finite State Processes, *Information and Computation*, vol. 81, no. 1, pp. 13–31, April 1989.

[CE 81] Clarke, E.M., Emerson, E.A. Design and Synthesis of Synchronization Skeletons using Branching Time Temporal Logic. *Workshop on Logics of Programs*, Springer-Verlag LNCS 131.

[CES 86] Clarke, E.M., Emerson, E.A., and Sistla, A.P., Automatic Verification of Finite-State Concurrent Systems using Temporal Logic, *ACM Trans. Prog. Lang. and Sys.*, vol. 8, no. 2, pp. 244-263, April 1986.

[CFJ 93] Clarke, E.M., Filkorn, T., Jha, S. Exploiting Symmetry in Temporal Logic Model Checking, 5th CAV, Springer-Verlag LNCS 697.

[CG 87] Clarke, E.M., Grumberg, O. Avoiding the State Explosion Problem in Temporal Logic Model Checking Algorithms, PODC 1987.

[CGJ 95] Clarke, E.M., Grumberg, O., Jha, S. Verifying Parameterized Networks using Abstraction and Regular Languages. CONCUR 95.

[Em 90] Emerson, E.A., Temporal and Modal Logic, in Handbook of Theoretical Computer Science, vol. B, (J. van Leeuwen, ed.), Elsevier/North-Holland, 1991.

[5] On the other hand, for their model of computation with all user processes but no control process, there is a polynomial time algorithm [GS 92]. We believe that our PSPACE-completeness result is not an insurmountable barrier to practical utility, given BDD-based implementations, as suggested in section 6.

[EN 95] Emerson, E.A., Namjoshi, K.S. Reasoning about Rings. *Proc. ACM Symposium on Principles of Programming Languages*, 1995.

[ES 93] Emerson, E.A., Sistla, A.P. Symmetry and Model Checking, 5th CAV, Springer-Verlag LNCS 697.

[ES 95] Emerson, E.A., Sistla, A.P. Utilizing Symmetry when Model Checking under Fairness Assumptions: An Automata-theoretic approach. CAV 1995.

[ESr 90] Emerson, E.A., Srinivasan, J. A decidable temporal logic to reason about many processes. PODC 1990.

[GS 92] German, S.M., Sistla, A.P. Reasoning about Systems with Many Processes. *J.ACM*, Vol. 39, Number 3, July 1992.

[HB 95] Hojati, R., Brayton, R. Automatic Datapath Abstraction in Hardware Systems, CAV 1995.

[ID 93] Ip, C., Dill, D. Better verification through symmetry. Proc. 11th Intl. Symp. on Computer Hardware Description Languages and their Applications.

[KM 89] Kurshan, R.P., McMillan, K. A Structural Induction Theorem for Processes, PODC 1989.

[LSY 94] Li, J., Suzuki, I., Yamashita, M. Fair Petri Nets and structural induction for rings of processes. *Theoretical Computer Science*, vol. 135(2), 1994. pp. 337-404.

[LP85] Litchtenstein, O., and Pnueli, A., Checking That Finite State Concurrent Programs Satisfy Their Linear Specifications, POPL 85, pp. 97-107.

[Lo 93] Long, D. Model Checking, Abstraction, and Compositional Verification. Ph.D. Thesis, Carnegie-Mellon University, 1993.

[Lu 84] Lubachevsky, B. An Approach to Automating the Verification of Compact Parallel Coordination Programs I. *Acta Informatica* 21, 1984.

[MP 92] Manna, Z., Pnueli, A. Temporal Logic of Reactive and Concurrent Systems: Specification, Springer-Verlag, 1992.

[McM92] McMillan, K., Symbolic Model Checking: An Approach to the State Explosion Problem, Ph.D. Thesis, Carnegie-Mellon University, 1992.

[Pn 77] Pnueli, A. The Temporal Logic of Programs. FOCS 1977.

[PD 95] Pong, F., Dubois, M. A New Approach for the Verification of Cache Coherence Protocols. *IEEE Transactions on Parallel and Distributed Systems*, August 1995.

[RS 85] Reif, J., Sistla, A. P. A multiprocess network logic with temporal and spatial modalities. *JCSS* 30(1), 1985.

[RS 93] Rho, J. K., Somenzi, F. Automatic Generation of Network Invariants for the Verification of Iterative Sequential Systems. CAV 1993, LNCS 697.

[SAE 92] SAE J1850 Class B data communication network interface. Society of Automotive Engineers, Inc., 1992.

[ShG 89] Shtadler, Z., Grumberg, O. Network Grammars, Communication Behaviours and Automatic Verification. Springer-Verlag, LNCS 407.

[Su 88] Suzuki, I. Proving properties of a ring of finite state machines. *IPL* 28, pp. 213-214.

[Va9?] Vardi, M. An Automata-theoretic Approach to Linear Temporal Logic, Proceedings of Banff Higher Order Workshop on Logics for Concurrency, F. Moller, ed., Springer-Verlag LNCS, to appear.

[VW 86] Vardi, M., Wolper, P. An Automata-theoretic Approach to Automatic Program Verification, Proc. IEEE LICS, pp. 332-344, 1986.

[V 93] Vernier, I. Specification and Verification of Parameterized Parallel Programs. Proc. 8th Intl. Symp. on Computer and Information Sciences, Istanbul, Turkey, pp. 622-625.

[WL 89] Wolper, P., Lovinfosse, V. Verifying Properties of Large Sets of Processes with Network Invariants. Springer-Verlag, LNCS 407.

HORNSAT, Model Checking, Verification and Games*

(Extended Abstract)

Sandeep K. Shukla[1] Harry B. Hunt III[1]
Daniel J. Rosenkrantz[1]

Department of Computer Science
University at Albany – State University of New York
Albany, NY 12222
Email: {sandeep,hunt,djr}@cs.albany.edu

Abstract. We develop a HORNSAT-based methodology for verification of finite state systems. This general methodology leads naturally to algorithms, that are *local* [25, 19], *on the fly* [28, 11, 13, 5] and *incremental* [24]. It also leads naturally to *diagnostic* behavioral relation checking [7] algorithms. Here we use it to develop model checking algorithms for various fragments of modal mu-calculus. We also use our methodology to develop a uniform game theoretic formulations of all the relations in the linear time/branching time hierarchy of [27]. As a corollary, we obtain natural sufficient conditions on a behavioral relation ρ, for ρ to be polynomial time decidable for finite state transition systems.

1 Introduction

We consider a number of problems related to the verification of finite state systems which include model checking for various fragments of modal mu-calculus [15] and checking behavioral relations [10] with diagnostic information. We outline a methodology for solving these problems, based upon efficient *local* reductions to satisfiability problems for simple variants of HORN formulas. We use our methodology to develop *local, on the fly* and *incremental* algorithms and to generate diagnostic information for these problems. Our algorithms are asymptotically as efficient as other specific algorithms in the literature for the problems considered. The desirability of *local, on the fly,* and *incremental* verification algorithms and algorithms for generating diagnostic information has been widely discussed [28, 5, 7, 13, 17, 18, 11, 10, 1, 24, 25, 8, 7]. However, previous algorithms proposed in the literature have only some of these advantages and only apply to some of the verification problems considered here. Our uniform methodology combines all these advantages in the same solution. Another advantage of our methodology is that efficient data structures and algorithms for the appropriate satisfiability problems for HORN formulas already exist in the literature [12, 3].

Our methodology is based upon efficient *local* reductions of the problems considered to the minimal and maximal satisfiability problems, for weakly positive

* This research was supported by NSF Grants CCR-90-06396 and CCR-94-06611.

and weakly negative [21] Horn formulas. We call these satisfiability problems minimal-HORNSAT and maximal-NHORNSAT respectively. In fact, restricted forms of these Horn formulas are enough for some of the problems. In Sections 2-4, we outline our (N)HORNSAT-based algorithm for model checking, for the alternation-free modal mu-calculus. We show that this algorithm is a simplification of the algorithms in [19, 1] involving solutions of systems of Boolean equations. (Recall that [19] involves *consistent* and *factual* solutions of Boolean equation systems and [1] involves maximal and minimal fixed points of Boolean equation systems.)

In Section 5, we use our (N)HORNSAT-based methodology to define a class of games, that includes the *characteristic games* for each of the behavioral relations in the linear-time/branching-time hierarchy of [27]. As a corollary, we get natural sufficient conditions, for a behavioral relation on finite state processes to be polynomial time decidable.

In [22], we show in details, how our (N)HORNSAT-based methodology can be used to develop efficient algorithms for diagnostic behavioral relation checking and model checking for the modal mu-calculus.

The main advantages of our methodology may be summarized as follows. First, it shows that the underlying combinatorics for a number of verification problems and their proposed solutions is essentially very simple. Second, it turns out that an efficient verifier can be based on an implementation whose core consists of a solver for (N)HORNSAT which runs in linear time, which can run *on the fly* for space efficiency, and can run *incrementally* (e.g., using simple modifications of the incremental HORNSAT algorithms given in [3]). Third, the fact that efficient solutions for HORNSAT and its variants already exist in the literature [12, 3] and that many important verification problems are reducible to those variants of HORNSAT makes the implementation of verification tools easier. Moreover, it relieves the designer of the verifier from the obligation of reinventing complex data structures which already exist in the literature on HORNSAT. Many model checking algorithms in the literature involved inventing complex new data structures, whereas existing efficient data structures for solving variants of HORNSAT are sufficient to obtain the same efficiency. Moreover, this approach leads to modular design, because the efficient implementation of HORNSAT solver can be delegated to a different designer. In [16] a data structure for a linear time algorithm for determining functional dependencies in relational databases [4] was reused to obtain a model checking algorithm for CTL. It is interesting to note that functional dependency is also reducible to HORNSAT, and in [3, 2] the same kinds of data structures are used to solve them in linear time. [2] In [1] the model checking problem for mu-calculus was reduced to finding fixed points of system of Boolean equations; and complex graph-based data structures were invented for efficiency. Our results show that

[2] However, (N)HORNSAT captures the essence of these problems more directly and intuitively. Moreover, efficient data structures for solving (N)HORNSAT are easily implementable. Also, HORNSAT based methods are directly implementable in DATALOG.

the full power of Boolean equations are not needed to solve these problems. Fourth, we identified many easiness results in the area of model checking and verification as a consequence of the corresponding easy instances of NHORN-SAT. For example, after characterizing special cases of HORNSAT which have NC algorithms, we could strengthen the results in [29] by characterizing cases when the model checking problem is in NC.

2 Satisfiability Problem for (N)HORNSAT

We consider special instances of CNF satisfiability problems, namely HORNSAT, where each clause contains at most one positive literal, and NHORNSAT, where each clause contains at most one negative literal. We are interested in finding maximal and minimal satisfying assignment (if one exists) respectively.

An instance of the problem is presented as a pair (X, C), where $X = \{x_1, x_2, ..., x_n\}$, a finite set of propositional variables which take Boolean values, and $C = \{C_1, C_2, ..., C_m\}$, a set of clauses with one of the restrictions discussed above. Note that if an instance has a satisfying assignment, such an assignment can be represented as an element of an n-dimensional Boolean lattice $\{0,1\}^n$. If we consider $0 < 1$, then with a component-wise extension of the ordering, and a component-wise $\wedge$ and $\vee$ as *meet* and *join* operation, we get a complete lattice. For an instance of a satisfiability problem h, we denote the set of all satisfying assignments as $SAT(h) \subseteq \{0,1\}^n$. An element $x \in SAT(h)$ is *minimal*, if no other $y \in SAT(h)$ is less than x in the ordering of $\{0,1\}^n$. Dually, an element $x \in SAT(h)$ is *maximal*, if no other $y \in SAT(h)$ is greater than x in the ordering of $\{0,1\}^n$. We call the problem of finding the maximal satisfying assignment for an NHORNSAT instance as the maximal-NHORNSAT problem, and the problem of finding the minimal satisfying assignment for a HORNSAT instance as the minimal-HORNSAT problem.

A linear time algorithm for minimal-HORNSAT appears in [12]. Dually the maximal-NHORNSAT is also solvable in linear time.

In some of our applications we have a special type of HORNSAT or NHORN-SAT instances. Here we discuss that special type of NHORNSAT, called *rooted NHORNSAT*. The corresponding cases and algorithms for HORNSAT are very similar.

Definition 1. Given a clause C_k of the form $x_j \Rightarrow \bigvee_{i \in I} x_i$, where I is an index set possibly empty (note that the disjunction $\bigvee_{i \in I} x_i = true$ when $I = \phi$.), we call x_j the **head** of clause C_k, denoted as $head(C_k) = x_j$, and $\bigvee_{i \in I} x_i$ the **tail** of C_k. Any variable x_i appearing in $tail(C_k)$, is called a disjunct in the tail.

Note that for a clause of the form $C_k = \overline{x_j}$, $head(C_k) = x_j$ and $tail(C_k) = false$. Similarly, for a clause of the form $C_k = x_j$, $head(C_k) = true$ and $tail(C_k) = x_j$.

Definition 2. An instance of a *rooted NHORNSAT* problem is of the form (X, C, x_1) where (X, C) is an NHORNSAT instance and the clauses in C are

ordered. Also. $C_1 = x_1$ (a single positive literal clause), where $x_1 \in X$. Furthermore, for each clause C_k, if $head(C_k) = x_j$ then there must be a clause $C_l (l < k)$ preceding C_k, such that x_j is a disjunct in $tail(C_l)$. Also for a single literal clause $C_k = x_p$ $(k > 1)$, x_p must also be a disjunct in $tail(C_l)$ for some $l < k$. and x_p cannot be the head of any clause.

The correctness of our (N)HORNSAT based methodology for model checking can be demonstrated easily by showing the following. There is a *local* reduction (see the proof sketch of Theorem 3 below) between the (N)HORNSAT based methodology and the methodologies in [19, 1] based upon systems of simple Boolean equations. The (N)HORNSAT based approach has the advantage that efficient algorithms and data structures for (N)HORNSAT are already available in the literature [12, 3]. The soundness and completeness of our methodology follow easily from the following theorem and its extensions to the results in [1].

Theorem 3. *The* factuality problem *and the* consistency problem *of system of* simple *Boolean equations described in [19] and the class of minimal-HORNSAT and maximal-NHORNSAT problems we consider, are* locally *and efficiently interreducible.*

Proof sketch: [3] Given a system of *simple* Boolean equations, if we are interested in factuality [19], we replace

an equation of the form $x = true$ by a single literal clause x,
an equation of the form $x = false$ by a single negated literal clause $\overline{x}$,
an equation of the form $x = x_1 \wedge x_2$ by a clause $x \Leftarrow x_1 \wedge x_2$, and
an equation of the form $x = x_1 \vee x_2$ by two clauses $x \Leftarrow x_1$ and $x \Leftarrow x_2$.

It is easy to prove that the variables which are assigned a value 1 in the minimal satisfying assignment for this HORNSAT instance are the factual variables of the original Boolean equational system. Since, we are considering minimal-HORNSAT, the implications can replace the equalities. Given this, duality implies that the consistency problem of [19] can be reduced efficiently and locally to the maximal-NHORNSAT problem.

Similarly, the problems of finding the least and greatest fixed points of the Boolean equations of [1] can be reduced to minimal-HORNSAT and maximal-NHORNSAT respectively. Details are omitted due to lack of space.

3 On the Fly, Local and Incremental Model Checking

Local Model Checking : A *local* model checking algorithm does not explore all the states of the finite state system, if not required. It tries to explore only a

[3] Given a set E of Boolean equations over a set of Boolean variables in V, the *factuality problem* is to find $F \subseteq V$ such that $x \in F$ if and only if x is set to *true* in every model of E. The *consistency* problem is to find $C \subseteq V$, such that $x \in C$ if and only if there exists a model of E in which x is set to *true*.

minimal set of states and determines whether certain properties are true in those states in order to infer that a given property is true in a given state. The tableau based methods in [18, 25, 6] are examples of such *local* algorithms for model checking. Our (N)HORNSAT based method achieves this objectives naturally. Given a fix point formula Φ, and a state s^* of a finite transition system, suppose we want to determine if s^* satisfies Φ. We generate (N)HORN formulas roughly as follows: We use a Boolean variable Y_s^ϕ, and create clauses such that s satisfies ϕ if and only if Y_s^ϕ is *true* in the (maximal) minimal satisfying assignment of the (N)HORNSAT instance.

On the Fly Model Checking : In [28, 11, 5, 16, 13] *on the fly* model checking and behavioral relation checking have been emphasized. In an *on the fly* algorithm the state space is constructed on demand, hence the verification takes place together with the construction of the state space. In our (N)HORNSAT based approach, *on the fly* algorithm is obtained naturally because of the existing *on the fly* or *online* algorithms for (N)HORNSAT [3] and some minor improvements on them. Our reduction to (N)HORNSAT can be done in NLOGSPACE and *on the fly* algorithm for HORNSAT works in $O(q)$ amortized time, where q is the size of each new clause generated. Since the size of the (N)HORNSAT instance created is linear in the product of the size of the transition system and the specification in the case of model checking, and product of the sizes of the two transition systems in case of relational checking, we might use in the worst case, linear space and linear time in those measures. For on the fly behavioral relation checking this is an improvement over [13] which requires quadratic time in these measures for behavioral relation checking. However, in most cases, counter examples are found after constructing substantially less number of clauses.

Incremental Model Checking : In [24], an incremental algorithm for model checking alternation free mu-calculus was developed. The basic idea was the following. When transitions are added or deleted from the transition system, an incremental algorithm exploits the information available from the previous runs of the model checking algorithm. It carries out minimal computation so that the model checking problem with respect to the changed transition system is solved in time $O(\Delta)$, where Δ is a measure of changes in the transition system. It has been pointed out [24] that in the worst case, this may not be possible. However, in the best case and more importantly, in many pragmatic situations the incremental computation could be linear in the size of the modification. Since the online algorithm for HORNSAT [3] is incremental and since the modification in the transition system will be reflected in the changes in the corresponding (N)HORNSAT instance, we can now directly obtain incremental algorithms for all the problems considered in this paper.

Note: The equational syntax of modal mu-calculus used in the subsequent sections is taken from [10]. Due to lack of space, the syntax and semantics could not be discussed and the readers are referred to [10, 22].

4 Model Checking Fragments of Modal Mu-Calculus

Our methodology can be extended to apply to full Mu-Calculus [15, 6], by using the model checking algorithm for the alternation-free fragment as a subroutine, as in [9] with the same efficiency as in [9]. Here we, illustrate our methods through its application to the *unnested single fixed point* fragment (which is similar to the Hennessy-Milner Logic with recursion [17, 18]) and to the alternation-free mu-calculus, as discussed in [10].

Model Checking for Single Fix point Mu-Calculus to (N)HORNSAT
For each state $s \in \mathcal{S}$ of the given finite state system $\mathcal{T}$ and each variable X_i of the equational specification, we associate a boolean variable $Y_s^{X_i}$. Recall, in the single fixpoint calculus, there is a single block of equations which is either a *max* block or a *min* block.

We consider the case when the block is a max block $B = max\{E\}$ where $E = \{X_1 = \Phi_1, ..., X_n = \Phi_n\}$. A dualization will hold for *min* blocks.

Here, the model checking problem is to determine if $s^* \in \|X_i\|_{\|B\|_e}$, for a given transition system $\mathcal{T} = \langle \mathcal{S}, Act, \rightarrow \rangle$, for an initial environment e, and $s^* \in \mathcal{S}$.

The reduction proceeds as follows:

1. Create a variable $Y_{s^*}^{X_i}$ and put the variable $Y_{s^*}^{X_i}$ in a queue.
2. For each variable of the form $Y_s^{X_j}$ on the queue, such that X_j appears in the left-hand side of an equation $\hat{e}$ in B

   (i) If $\hat{e}$ is $X_j = A$ where A is atomic, then create a clause Y_s^A if A is true at s else create a clause $\overline{Y_s^A}$. (This information is obtained from the valuation map associated with the model.) Put the variable Y_s^A in the queue if this variable was never on the queue before.
   (ii) If $\hat{e}$ is $X_j = X_p \vee X_q$, then create the clause $Y_s^{X_j} \rightarrow Y_s^{X_p} \vee Y_s^{X_q}$ and put the variables $Y_s^{X_p}$ and $Y_s^{X_q}$ into the queue, if these variables were never on the queue before.
   (iii) If $\hat{e}$ is $X_j = X_p \wedge X_q$, then create two clauses $Y_s^{X_j} \rightarrow Y_s^{X_p}$ and $Y_s^{X_j} \rightarrow Y_s^{X_q}$ and put the variables $Y_s^{X_p}$ and $Y_s^{X_q}$ into the queue, if they were never on the queue before.
   (iv) If $\hat{e}$ is $X_j = \langle a \rangle X_p$, then create a clause of the form $Y_s^{X_j} \rightarrow \bigvee_{s' \in a(s)} Y_{s'}^{X_p}$ where $a(s) = \{s' \mid \exists s' : s \xrightarrow{a} s'\}$. When $a(s)$ is empty, the disjunction is equivalent to false. Put the variables $Y_{s'}^{X_p}$ on the queue if they were never on the queue before.
   (v) If $\hat{e}$ is $X_j = [a]X_p$, then create clauses of the form $Y_s^{X_j} \rightarrow Y_{s'}^{X_p}$ for each $s' \in a(s)$ where $a(s) = \{s' \mid \exists s' : s \xrightarrow{a} s'\}$. Put the variables $Y_{s'}^{X_p}$ on the queue if they were never on the queue before. When $a(s)$ is empty, create the single literal clause $Y_s^{X_j}$.

3. If $Y_s^{X_j}$ is in the queue and if X_j does not appear on the left hand side in B, then if $s \in e(X_j)$, add a single literal clause $Y_s^{X_j}$ else add the clause $\overline{Y_s^{X_j}}$.

This will produce an NHORNSAT instance, of the size linear in the product of the size of the transition system and equational block B. We now state the theorem stating the correctness of the reduction. The correctness of the model checking algorithm obtained this way follows from the discussions in section 2.

Let $s \in \mathcal{S}$ is a state in the given finite state transition system $\mathcal{T} = \langle \mathcal{S}, Act, \rightarrow \rangle$. Let X_i be a variable in the equational block used in specifying a property using the syntax of [10] and let the initial environment be e. Suppose the block specifying the formula is a max block, $B = max\{E\}$ where $E = \{X_1 = \Phi_1, ..., X_n = \Phi_n\}$.

Theorem 4. *If h is the instance of NHORNSAT produced by the algorithm described above from the given model checking problem (if $s^* \in ||X_i||_{\|B\|_e}$), then h is satisfiable and in the maximal satisfying assignment of h, $Y_{s^*}^{X_i} = 1$, if and only if $s^* \in ||X_i||_{\|B\|_e}$.*

The dual of the above theorem holds for min blocks. Which means that in the minimal solution of the HORNSAT instance produced in that case, $Y_{s^*}^{X_i} = 1$ if and only if $s^* \in ||X_i||_{\|B\|_e}$. This gives us a linear time algorithm for the problem.

Alternation free mu calculus : Now we generalize the algorithm in the previous section, to obtain a (N)HORNSAT based algorithm for the model checking of alternation free mu-calculus. A linear time algorithm for the same problem was presented in [10]. Their algorithm needed to invent an efficient data structure to obtain the linear time algorithm. Our method brings out the fact that the essential data structure necessary to obtain the linear time algorithm for model checking is in fact the same as in [12] for the linear time algorithm for HORNSAT/NHORNSAT

Given a Transition system $\mathcal{T}$, a valuation map υ, an initial environment e, a blockset $\mathcal{B}$, the model checking problem is to decide if $s^* \in ||X_i||_{\|\mathcal{B}\|_e}$, for a given state s^* in the transition system and a given variable X_i appearing on the left hand side of some equation in some block B_l in $\mathcal{B}$.

Briefly, the steps in the (N)HORNSAT based version of the algorithm for model checking alternation free mu-calculus are as follows:

1. Create a variable $Y_{s^*}^{X_i}$ and put the variable $Y_{s^*}^{X_i}$ in the queue associated with the block B_l where X_i appears on the left hand side.
2. Expand the variables in the queue associated with each block, in the reverse topological order, [4] with the following rules:
If the block is a max block then use the methods described in the previous subsection and if the block is a min block use a dual approach. Keep the NHORN or HORN clauses for each block separated. If new variable $Y_s^{X_j}$ is generated and X_j belongs to a different block B , put that variable in the queue associated with block B.

[4] Given $\mathcal{B}$, the block set, topologically sort the blocks in B with respect to the variable dependency relation depicted in block graph. Let $B_1, B_2, ..., B_m$ be the set of blocks in the topologically sorted order.

If the a variable $Y_s^{X_j}$ in the queue for a block B is already expanded then remove it from the queue otherwise expand it.

3. Start solving the minimal-HORNSAT/maximal-NHORNSAT instances corresponding to each block in the topological order. Let h_B be the HORNSAT/ NHORNSAT instance corresponding to block B. Suppose a variable $Y_s^{X_j}$ was assigned a value 1 in the solution of a h_B (where X_j appears on the left hand side in B) then add a clause $Y_s^{X_j}$ in the (N)HORNSAT instances corresponding to the blocks which had to put this variable in the queue of the block B (This information can be read off the block graph also). If $Y_s^{X_j}$ was assigned a value 0 in the solution of a h_B (where X_j appears on the left hand side in B) then add a clause $\overline{Y_s^{X_j}}$ in the (N)HORNSAT instances corresponding to the blocks which put this variable in the queue of the block B. Then continue solving the next block HORNSAT instance.

Suppose the block B corresponding to X_i, is a max block. (A Dual strategy holds for the min blocks). The maximal-NHORNSAT instance for the block B is satisfiable and $Y_{s^*}^{X_i} = 1$, in the maximal satisfying assignment, if and only if $s^* \in \|X_i\|_{\|B\|_e}$.

Note that this algorithm produces a sequence of HORNSAT and NHORNSAT instances and it is local and it can be made into an On the fly algorithm by noting that one can use the on the fly algorithm for each HORNSAT instance. We state the theorem about the correctness and efficiency of the algorithm sketched above with out proof.

Theorem 5. *The algorithm for model checking alternation free mu-calculus obtained by reducing the problem to a sequence of minimal-HORNSAT and maximal-NHORNSAT problems runs in time linear in the product of the sizes of the transition system and the block set specifying the property. Hence the HORNSAT based algorithm is as efficient as the algorithm in [10].*

We also have developed HORNSAT based methods to capture the tableau based local model checking in [8] and [25]. Details will appear in a future version of this paper.

5 Game for rooted (N)HORNSAT and Stirling Games

In [23] we show that many relational problems are also directly, locally, and naturally reducible to rooted NHORNSAT. Hence, given a two-player game for *rooted* NHORNSAT, we can easily associate games to all these relations as well. However, our objective is to obtain a sufficient characterization of various process algebraic behavioral relations, which helps us identifying whether a particular relation ρ, between finite transition systems is polynomial time decidable. In what follows, through a game theoretic formulation (similar to [26] where a characteristic game for bisimulation was defined,) we fulfill this objective. Such a natural sufficient characterization is really useful in identifying a polynomial time decidable relation when the definitions of the relations are complicated. [5]

[5] In [14], J. F. Groote who originally defined 2-nested simulation and $k-$nested simulation conjectured that deciding these relations must be NP-hard. However, by our

Game for *rooted* NHORNSAT: Game for an instance of a rooted NHORNSAT instance $h = (X, C, x_1)$ is a two player game G_h in which player I (the *spoiler*) wants to show that the instance h is not satisfiable and Player II (the *duplicator*) wants to show otherwise. The game proceeds in rounds. The spoiler opens the game by choosing a clause C_i such that $head(C_i) = x_1$. Duplicator reciprocates by choosing x_{ij} such that x_{ij} is a disjunct in $tail(C_i)$. In subsequent rounds, the spoiler chooses a clause C_k such that $head(C_k) = x_{ij}$ where x_{ij} was the duplicator's choice in the previous round. The duplicator has to reciprocate by choosing a disjunct in the tail of C_k. The game continues until one of the player loses. The duplicator loses if it does not have such a disjunct to choose (i.e, when the spoiler has chosen a clause of the form $\overline{x_l}$ in its last move), the spoiler loses when the game continues for ever (which is not possible in a finite size NHORNSAT instance) or when the spoiler chooses a clause chosen earlier. The following theorem states that the game we defined above, is indeed characteristic for rooted-NHORNSAT.

Theorem 6. *Given an instance $h = (X, C, x_1)$ of the* rooted *NHORNSAT problem, the duplicator has a winning strategy* [6] *in the corresponding game if and only if h is satisfiable.*

Stirling Class of Games: Now we describe a class of two player games called the *Stirling Class.* In this class, player I (the *duplicator* or *prover*) and player II (the *spoiler* or *disprover*) plays on two Finite transition systems. Each game in the class has the following components:

> Two Finite Transition systems $T_1 = 1$ and $T_2 = 2$; Two languages $R_1 \subseteq A^*$ and $R_2 \subseteq A^*$; Two total relations $m_2 \subseteq R_1 \times A^*$ and $m_2 \subseteq R_2 \times A^*$; A set of (*winning positions*) $\Gamma \subseteq S_1 \times S_2$; A set of starting positions $\Sigma \subseteq \Gamma \subseteq S_1 \times S_2$; A set $M \subseteq \{1, 2\}$ which denotes the indices of the coordinate of a position that spoiler can play on. In each round the duplicator plays on the other coordinate; and, A positive integer r denoting the number of rounds allowed in the game. This is crucial for some of the games.

The game starts in a position $\langle s, t \rangle \in \Sigma$. A *play* of the game is a finite or infinite length sequence of the form $\langle s_0^1, s_0^2 \rangle, ..., \langle s_i^1, s_i^2 \rangle, ...$ The *spoiler* wants to show that there is a *difference* between the two transition systems (the kind of difference it wants to show depends on the relation the game corresponds to). The *duplicator* wants to show that such a distinction attempted by the *spoiler* is not possible. A *partial play* in a game is a prefix of a *play* of the game. Let π_j be a *partial play* $\langle s_0^1, s_0^2 \rangle, ..., \langle s_j^1, s_j^2 \rangle$. The next pair $\langle s_{j+1}^1, s_{j+1}^2 \rangle$ is determined by the following move rule:

- The Spoiler picks a triple $\langle i, x, u \rangle$ such that $i \in M$ and $x \in R_i$ and $s_j^i \stackrel{x}{\Longrightarrow}_i u$. and $u = s_{j+1}^i$. (Note that $\Longrightarrow_i$ denotes an extended step in the transition system T_i).

---

characterization it is easy to see that they are polynomial time decidable. Moreover, many other relations such as $\frac{m}{2}$−nested relations[20] were shown to be polynomial time decidable this way.

[6] For the definition of winning strategy, see next subsection

- Let the choice of the spoiler in the move be $\langle i, x, u\rangle$ and let $i' \neq i$. Then the Duplicator picks a pair $\langle y, u'\rangle$ such that $(x, y) \in m_{i'}$ and $s_j^{i'} \stackrel{y}{\Longrightarrow}_{i'} u'$ and $u' = s_{j+1}^{i'}$.

Extending a partial play π_j to π_{j+1} by the above move rule is called a *round* of the game. Hence a play can be thought of as a sequence of rounds.

The duplicator wins the game if either in the last position of the play, there is no further allowable move by none (when $M = \{1, 2\}$) or there is no further allowable move by the spoiler(when $|M| = 1$), depending on the cardinality of the set M. Duplicator also wins, if in the play a position is repeated. In both cases, the spoiler has failed to expose a distinction between the transition systems. The spoiler wins, if in the last position of the play is not a winning position which means the spoiler has been able to force the duplicator to a non winning position of the game or if in the last position, the spoiler has an allowable move but the duplicator does not have a matching move. A *strategy* for a player is a set of rules which tells him/her how to make a move depending on the partial play and opponent's move so far.

A strategy is a *winning strategy* for a player, if playing with that strategy, that player wins against all possible strategies of the opponent.

Definition 7. A game G in Stirling class is called a characteristic game for a relation R between two finite state processes, if the following condition holds. Let the game G be played on two transition systems T_1 and T_2 and the duplicator has a history free winning strategy if and only if T_1 and T_2 are related by the relation R.

Here, we illustrate characteristic games for *bisimulation*, *weak bisimulation*, and *Failure equivalence*. We assume in the following that all the games are being played on $T_1 = 1$ and $T_2 = 2$. [7]

Characteristic Game for Bisimulation : $Bsim - game$ is a game in Stirling class with the following parameters: $R_1 = R_2 = A$, $m_1, m_2 = \iota$, $\Gamma = S_1 \times S_2$, $\Sigma = \{\langle s_1, s_2\rangle\}$, $M = \{1, 2\}$, $r = \mid S_1 \mid * \mid S_2 \mid +1$.

Characteristic Game For Weak Bisimulation: $WeakBsim - game$ is a game in Stirling class with the following parameters: $R_1 = R_2 = \tau^* A \tau^*$, $m_1(a) = \tau^* a \tau^*, m_2(a) = \tau^* a \tau^* \forall a \in A$, $\Gamma = S_1 \times S_2$, $\Sigma = \{\langle s_1, s_2\rangle\}$, $M = \{1, 2\}$, $r = \mid S_1 \mid * \mid S_2 \mid +1$.

Characteristic Game For Failure Equivalence: $Failure - game$ is a game in Stirling class with the following parameters: $R_1 = R_2 = A^*$, $m_1, m_2 = \iota$, $\Gamma = \{\langle s, t\rangle \mid s \in S_1, t \in S_2 \wedge Failures(s) = Failures(t)\}$, $\Sigma = \{\langle s_1, s_2\rangle\}$, $M = \{1, 2\}$, $r = 1$.

For each relation R, in the linear-time/branching time hierarchy, and its characteristic game G_R, the following theorem can be proved easily.

Theorem 8. *Let T_1, T_2 be two transition systems and let G_R be the instance of the characteristic game for a relation R, such that the game is played on T_1 and*

[7] Note that ι denotes the identity relation.

T_2. The duplicator has a winning strategy for this instance of the game G_R if and only if R holds between the given two transitions systems.

For a certain subclass of Stirling class, the problem whether the duplicator has a winning strategy is directly reducible to rooted NHORNSAT problem. Hence, for any behavioral relation, whose characteristic game is in this subclass, the problem of checking that relation between two finite state transition systems is reducible to the rooted NHORNSAT problem. This leads to a polynomial time algorithm for the problem of checking that relation, provided one can create the instance of the game from the instance of the relational problem in polynomial time. For all the games in Stirling Class, given that the transition systems are represented as finite state systems, the transformation to game instance is polynomial time, provided that the winning positions can be decided in polynomial time. Hence, we get a sufficiency condition as to under what condition a behavioral relation between finite state processes is polynomial time decidable.

A Subclass of Stirling Class We now briefly give a sufficient characterization as to when a game in Stirling Class is reducible to an instance of *rooted* NHORNSAT in polynomial time.

1. R_1 and R_2 are finite and explicitly enumerated. For example, in bisimulation game $R_1 = R_2 = A$, where A is the set of action symbols.
2. The representation of the set of winning positions is either by an explicit listing or is a polynomial time decidable set.

Acknowledgements: We wish to thank Rajeev Alur, S. S. Ravi, and Moshe Vardi for helpful discussions.

References

1. H. R. Andersen. Model checking and boolean graphs. *Theoretical Computer Science*, 126(1):3–30, 1994.
2. G. Ausiello, A. D'Atri, and D. Sacca. Graph algorithms for functional dependency manipulation. *Journal of Association for Computing Machinery*, 30(4):752–766, Oct 1983.
3. G. Ausiello and G. F. Italiano. On-line algorithms for polynomially solvable satisfiability problems. *Journal of Logic Programming*, 10:69–90, 1991.
4. C. Beeri. On the membership problem for functional and multivalued dependencies in relational databases. *ACM Transactions on Database Systems*, 5:241–259, 1980.
5. G. Bhat, R. Cleaveland, and O. Grumberg. Efficient on-the-fly model checking for ctl. In *Proceedings of IEEE Symposium on Logic In Computer Science' 95*, 1995.
6. J. C. Bradfield. *Verifying Temporal Properties of Systems*. Birkhauser, 1992.
7. U. Celikkan and R. Cleaveland. Generating diagnostic information for behavioral preorders. In *Proceedings of Computer Aided Verification: 1992, Lecture Notes in Computer Science 663*, pages 370–383, 1992.
8. R. Cleaveland. Tableau-based model checking in the propositional mu-calculus. *Acta Informatica*, 27:725–747, 1990.
9. R. Cleaveland, M. Klein, and B. Steffen. Faster model checking for modal mu-calculus. In *Proceedings of Computer Aided Verification: 1992, Lecture Notes in Computer Science 663*, pages 410–422, 1992.

10. R. Cleaveland and B. Steffen. Computing behavioural relations, logically. In *ICALP*, pages 127–138, 1991.
11. C. Courcoubetis, M. Y. Vardi, P. Wolper, and M. Yannakakis. Memory efficient algorithms for the verification of temporal properties. *Formal Methods in System Design*, 1:275–288, 1992.
12. W.F. Dowling and J.H. Gallier. Linear time algorithm for testing the satisfiability of propositional horn formulae. *Journal of Logic Programming*, 3:267–284, 1984.
13. J. C. Fernandez and L. Mounier. On the fly verification of behavioral equivalences and preorders. In *The 3rd International Workshop on Computer Aided Verification 1991, Lecture Notes in Computer Science 575*, pages 181–191, 1991.
14. J. F. Groote. Private communications. 1996.
15. D. Kozen. Results on the propositional mu-calculus. *Theoretical Computer Science*, 27, 1983.
16. O. Kupferman, M. Y. Vardi, and P. Wolper. An automata-theoretic approach to branching time model checking. *Draft*, 1995.
17. K. G. Larsen. Proof systems for hennessy milner logic with recursion. In *CAAP'88 Lecture Notes in Computer Science 299*, 1988.
18. K. G. Larsen. Proof systems for satisfiability in hennessy-milner logic with recursion. *Theoretical Computer Science*, 72:265–288, 1990.
19. K. G. Larsen. Efficient local correctness checking. In *CAV 92, Lecture Notes in Computer Science 663*, pages 30–43, 1992.
20. X. Liu. Specification and decomposition in concurrency. Technical report, Department of Mathematics and Computer Science, Aalborg University, Denmark, 1992.
21. Thomas J. Schaefer. The complexity of satisfiability problems. In *Tenth Annual Symposium on Theory of Computing*, 1978.
22. S. K. Shukla, H. B. Hunt III, and D. J. Rosenkrantz. Hornsat, model checking, verification, and games. Research Report TR-95-8, Department of Computer Science, SUNY Albany, 1995.
23. S. K. Shukla, D. J. Rosenkrantz, H. B. Hunt III, and R. E. Stearns. A hornsat based approach to the polynomial time decidability of simulation relations for finite state processes. *DIMACS workshop on Satisfiability Problem: Theory and Practice*, 1996.
24. O. Sokolsky and S. A. Smolka. Incremental model checking in the modal mu-calculus. In *Proceedings of CAV'94*, 1994.
25. C. Stirling and D. Walker. Local model checking in the modal mu-calculus. *Theoretical Computer Science*, 89:161–177, 1991.
26. Colin Stirling. Modal and temporal logics for processes. In *Notes for Summer School in Logic Methods in Concurrency*, pages Department of Computer Science, Aarhus University, 1993.
27. R.J. van Glabbeek. The linear time - branching time spectrum. Technical Report CS-R9029, Computer Science Department, CWI, Centre for Mathematics and Computer Science, Netherlands, 1990.
28. M. Vardi and P. Wolper. An automata theoretic approach to automatic program verification. In *Proceedings of LICS 1986*, pages 332–344, 1986.
29. S. Zhang, O. Sokolsky, and S. A. Smolka. On the parallel complexity of model checking in the modal mu-calculus. In *Proceedings of LICS 1994*, 1994.

Verifying the SRT Division Algorithm Using Theorem Proving Techniques

E. M. Clarke* S. M. German** X. Zhao*

Abstract. We verify the correctness of an SRT division circuit similar to the one in the Intel Pentium processor. The circuit and its correctness conditions are formalized as a set of algebraic relations on the real numbers. The main obstacle to applying theorem proving techniques for hardware verification is the need for detailed user guidance of proofs. We overcome the need for detailed proof guidance in this example by using a powerful theorem prover called Analytica. Analytica uses symbolic algebra techniques to carry out the proofs in this paper fully automatically.

1 Introduction

Proving the correctness of arithmetic operations has always been an important problem. The importance of this problem has been recently underscored by the highly-publicized division error in the Pentium processor [14]. Some people have estimated that this error cost Intel almost 500 million dollars [1]. In this paper, we verify a division circuit [16] that is similar to the one used in the Pentium. The circuit uses a radix four SRT division algorithm that looks ahead to find the next quotient digit in parallel with the generation of next partial remainder. An 8-bit ALU estimates the next remainder's leading bits. A quotient digit look-up table generates the next quotient digit depending on the leading bits of the estimated remainder and the leading bits of the divisor.

In our approach to verification, we formalize the circuit and its correctness conditions as a set of algebraic relations over the real numbers [9]. These algebraic relations correspond closely to the bit-level structure of the circuit, and could have been generated mechanically from a hardware description. Most of the hardware for the SRT algorithm can be described by linear inequalities. This led us to experiments [9] in which we proved properties of the SRT hardware using the Maple symbolic algebra system and its Simplex algorithm package.

We now have a fully automatic approach, where the correctness of the circuit is proved using a powerful theorem prover called Analytica [6] that we have developed. Analytica is the first theorem prover to use symbolic computation techniques in a major way. It is written in the Mathematica programming language and runs in the interactive environment provided by this system [18].

This research was sponsored in part by the National Science Foundation under Grant No. CCR-9217549, by the Semiconductor Research Corporation under Contract No. 94-DJ-294, and by the Wright Laboratory, Aeronautical Systems Center, Air Force Materiel Command, USAF, and the Advanced Research Projects Agency (ARPA) under Grant No. F33615-93-1-1330.

* School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, USA
** IBM Watson Research Center, P.O. Box 218, Yorktown Heights, NY 10598, USA
Email: emc@cs.cmu.edu, german@watson.ibm.com, xzhao@cs.cmu.edu

Compared to Analytica, most theorem provers require significant user interaction. The main problem is the large amount of domain knowledge that is required for even the simplest proofs. Our theorem prover, on the other hand, is able to exploit the mathematical knowledge that is built into the symbolic computation system and is highly automatic.

The work that is most closely related to ours is by Verkest *et al* [17], who have verified a nonrestoring division algorithm and hardware implementation using the Boyer Moore theorem prover [5]. The circuit they consider is much simpler than the one we verify. The main difficulty in verifying our circuit is in showing that the estimation circuit and the quotient lookup table give the correct quotient digits. In contrast, their circuit computes the quotient in radix 2, and does not speed up the computation by estimating the partial remainders. Another project by Leeser *et al* [11] verifies a radix 2 square root algorithm and hardware implementation. This work is similar to [17] and does not involve the design features that make fast division circuits difficult to verify. Although we prove the correctness of a relatively complicated circuit, our use of symbolic computation techniques allows us to carry out the proof automatically.

Due to space limitations, the paper in this volume of conference proceedings is abridged. A more complete version is available on request from the authors. The complete paper has several appendices. In one appendix, we develop the specification of the SRT circuit in greater detail than we can here, and discuss the convergence of the quotient calculation. Other appendices shows the input to the theorem prover and part of the generated proof.

2 The SRT Division Algorithm and Circuit

2.1 Floating-Point Numbers and Floating Division

Under the IEEE arithmetic standard, a normalized floating point number has the form $sign \cdot significand \cdot 2^{exponent}$, where *sign* is one bit representing ± 1, the *significand* is a rational number in the range $1 \leq significand < 2$, and *exponent* is an integer. Certain values, such as 0, have special representations under the standard. Hardware circuits for floating-point arithmetic are usually organized into two parts: a normalization circuit and an arithmetic core, which performs arithmetic operations on the significands of the normalized numbers. The circuit that we consider in this paper is the core of a floating point division circuit. A separate circuit handles the signs and exponents.

There are several ways to interpret the arithmetic operation performed by the hardware of the core. One way is to consider it as an operation on scaled integers. In this paper, we interpret signals in the division core as arbitrary real numbers, and develop our proof using algebraic theory that holds for all the reals, not just the values that can be represented in a certain number of bits. One advantage of our approach is that our specification and correctness proof are independent of the hardware word length; that is, we prove the correctness of the SRT division circuit for all word lengths $n > 8$ bits, *without* having to induct on word length. Note that this approach is sound but may not yield a proof

in all cases. It is possible, for example, to design a floating-point circuit whose correctness depends on the fact that only a finite set of values is represented.

2.2 Long Division

The idea of the division algorithm is to compute a sequence of quotient digits $q_0, q_1, \ldots, q_{m-1}$, such that the significand of the quotient is the numeral $q_0.q_1 \cdots q_{m-1}$. In order to compute the quotient digits, the algorithm computes a sequence of *partial remainders* p_i according to the recurrence

$$\begin{aligned} p_0 &= \mathit{Dividend}, \\ p_{j+1} &= r \cdot (p_j - q_j \cdot \mathit{Divisor}), \text{ for } j = 0, \ldots, m-1, \end{aligned} \tag{1}$$

where r is the radix of the representation of the quotient.

The running time of the division algorithm depends on the number of iterations of (1) and the time needed for each iteration. The number of iterations needed to compute the quotient to a given number of bits b of accuracy depends on the radix r. If the quotient is represented in radix 2, b iterations will be needed, because each iteration produces only one bit of the quotient.

In practice, radix 4 is often used in hardware division circuits because only $b/2$ iterations are needed and the calculations on each iteration can be performed quickly in hardware. Each iteration involves two multiplications and a subtraction, assuming q_j is known. In radix $r = 4$, both of the multiplications can be implemented by fast hardware that simply shifts one of the operands to the left. For example, the multiplication by r can be computed by shifting two bits to the left. Also, the multiplication by q_j can be done by shifting when the value of q_j is 0, 1, or 2. In the case that $q_j = 3$, there is a potential problem because multiplication by 3 cannot be done in this way. We will see, however, that the SRT algorithm uses a representation of the quotient digits that avoids this problem.

The subtraction operation in (1) dominates the time needed for each iteration. For double precision arguments, a 64 bit subtraction must be performed on each cycle.

The basic idea of the SRT algorithm [2] is to arrange the computation so that the quotient digit selection can be done in parallel with the long subtraction operation. Referring to the basic recurrence (1), it is clear that the choice of q_j depends on the value of p_j.

In order to carry out quotient selection concurrently with the computation of p_j, the SRT algorithm allows the choice of the quotient digit at each step to be *inexact*. In simple long division, the quotient digit q_j is chosen at each stage so that $0 \leq q_j \leq r - 1$ and $0 \leq p_j - q_j \cdot \mathit{Divisor} < \mathit{Divisor}$. At each step of the computation, there is a unique choice of q_j that will keep the next partial remainder in the desired range.

The SRT algorithm computes an estimate of p_j while the full subtraction is in progress. The estimated value of p_j is used to select a quotient digit, but the estimate is not precise enough to guarantee that the exact quotient digit will

be selected. Intuitively, the algorithm selects a quotient digit that is either "just right" or "too big" by 1.

If the quotient digit chosen at a given stage is "too big," the value computed for p_{j+1} will be negative. In order to make the computation converge, the algorithm will choose a *negative quotient digit* at the next iteration. Negative quotient digits are written with an overbar, for example $\overline{2}$ has the value -2. A number containing negative digits can be converted to one without negative digits by subtracting the negative digits. As an example, in $.21\overline{1} = .203$, the negative digit can be removed by the subtraction $.210 - .001$. In an implementation of the SRT algorithm, it is straightforward to provide hardware that performs the conversion.

For radix 4 calculations, division can be defined using quotient digits $\overline{3}$, $\overline{2}$, $\overline{1}$, 0, 1, 2, 3. Observe, however, that all radix 4 numbers can be represented using only $\overline{2}$, $\overline{1}$, 0, 1, 2. For instance, $.3 = 1.\overline{1}$. This observation allows hardware implementations to avoid the problem of multiplying the divisor by three; see the next section for details.

2.3 Structure and Operation of the Division Circuit

The division circuit has four full-width registers: The Divisor register holds the value of the divisor, the Remainder register holds the value of the partial remainder, and the registers QPOS and QNEG hold the value of the quotient. The q register holds one digit of the quotient. The outputs of the q register are qdigit (2 bits), for the absolute value of the quotient digit, and qsign (1 bit), for the sign. The DALU is a full width adder/subtracter, which is used to compute the partial remainders. The GALU is an 8-bit wide adder/subtracter, which computes an estimate of the partial remainder. QUO LOGIC is a block of combinational logic. Given the leading bits of the divisor and the estimate of the partial remainder from the GALU, QUO LOGIC outputs the next digit of the quotient. At several places, the circuit shifts a signal by one or two bits to the left in order to multiply it by two or four. This operation is shown in the diagram as a box with the operation $<< 1$ or $<< 2$. Throughout the paper, we use roman typeface for names of signals and *italics* for the values of signals.

The division circuit operates in two phases: an initialization phase followed by the main calculation phase. The initialization phase begins by setting the Remainder register to hold the dividend, setting the Divisor register to hold the divisor, and setting the QPOS and QNEG registers to zero. After these initializations have been done, the initialization phase uses the GALU and the quotient selection logic to compute the first quotient digit and store it in the q register. This completes the initialization phase.

The calculation phase performs one cycle of the division circuit for each digit of the quotient. At the beginning of the jth cycle, Remainder holds p_j, Divisor holds the divisor, and the q register holds q_j. The DALU receives p_j on its A input. The other input to DALU is the signal md, which is controlled by the MUX. The inputs of the MUX are the values 0, *Divisor*, and $2 \cdot$ *Divisor*. Under control of qdigit, the MUX sets the line md to *qdigit* $\cdot$ *Divisor*. The signal qsign

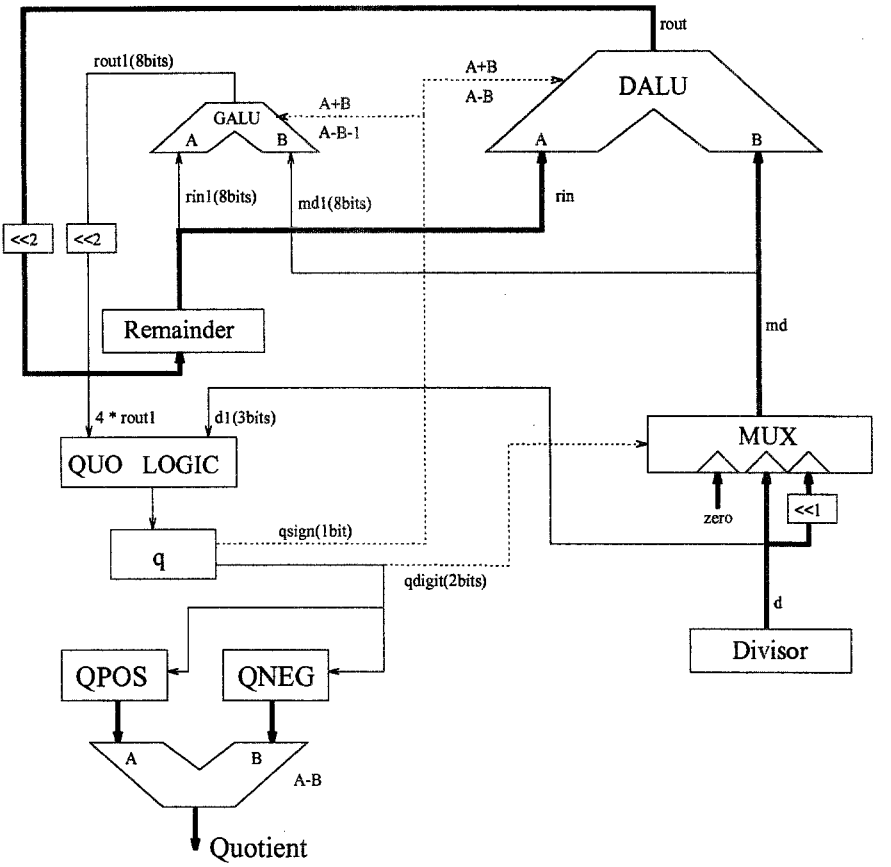


Fig. 1. The division circuit

controls whether DALU adds or subtracts its inputs: DALU performs subtraction if qsign is +; otherwise it does an addition. The result is that DALU computes the value $p_j - q_j \cdot \mathit{Divisor}$ and outputs this value on rout. The signal rout is shifted two bits to the left and stored in the Remainder register for the next cycle.

The GALU essentially computes the leading 8 bits of rout. The A (resp. B) input to GALU receives the leading 8 bits of the A (resp. B) input to DALU, and qsign switches GALU between addition and subtraction. The output of GALU is routed through QUO LOGIC to select the next quotient digit.

The value of the quotient is computed using the registers QPOS and QNEG. QPOS holds all of the positive quotient digits and QNEG holds all of the negative digits. On each cycle, these registers are updated as follows: Both registers are shifted two bits to the left. If the digit in the q register is positive, then the value of qdigit (2 bits) is stored in the low order bits of QPOS and the two low-order bits of QNEG are set to zero. If the digit is negative, then the value of qdigit (i.e

the absolute value of the digit) is stored in the low-order bits of QNEG and the low-order bits of QPOS are set to zero. When all of the quotient digits have been computed, the values of QPOS and QNEG are routed to an ALU to compute $QPOS - QNEG$. The output of this ALU is the quotient. The reason for storing the positive and negative digits in separate registers is to keep the cycle time of the circuit short. Adding a full-width ALU on the inner cycle of the circuit would slow it down.

```
    g1  (4 * rout1 -- 7 bits)
    g2
    g3  1  0  1  0  1  0  1  0  1  0  1  0  1  0  1  0  1  0  1  0  1  0
        .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .
    g4  0  1  1  0  0  1  1  0  0  1  1  0  0  1  1  0  0  1  1  0  0  1
    g5  1  1  1  0  0  0  0  1  1  1  1  0  0  0  0  1  1  1  1  0  0  0
    g6  0  0  0  1  1  1  1  1  1  1  1  0  0  0  0  0  0  0  0  1  1  1
    g7  1  1  1  1  1  1  1  1  1  1  1  0  0  0  0  0  0  0  0  0  0  0

1.000  -- -- -- -- -2 -2 -2  A -1 -1  0  0  1  1  2  2  2 -- -- -- -- --
1.001  -- -- -- -- -2 -2 -2  B -1 -1  0  0  1  1  C  2  2  2 -- -- -- --
1.010  -- -- -- -2 -2 -2 -2 -1 -1  D  0  0  1  1  1  2  2  2  2 -- -- --
1.011  -- -- -2 -2 -2 -2  B -1 -1  D  0  0  1  1  1  2  2  2  2 -- -- --
1.100  -- -- -2 -2 -2 -2 -1 -1 -1  0  0  0  E  1  1  C  2  2  2  2 -- --
1.101  -- -2 -2 -2 -2 -2 -1 -1 -1  0  0  0  0  1  1  1  2  2  2  2  2 --
1.110  -2 -2 -2 -2 -2  B -1 -1 -1  0  0  0  0  1  1  1  2  2  2  2  2 --
1.111  -2 -2 -2 -2 -2 -1 -1 -1 -1  0  0  0  0  1  1  1  1  2  2  2  2  2

(d1 -- 4 bits)                          A = -(2 - g2 * g1)
                                        B = -(2 - g2)
                                        C = 1 + g2
                                        D = -(1 - g2)
                                        E = g2
```

Table 1. The quotient prediction table for the division circuit

The Quotient Selection Table The quotient selection logic for QUO LOGIC is represented in tabular form in Table 1. QUO LOGIC receives two inputs: an estimate of the partial remainder from GALU and the first four bits of the divisor, and selects one of the digits $\overline{2}$, $\overline{1}$, 0, 1, 2. In the table, the GALU input is $g_7\, g_6\, g_5\, g_4 \cdot g_3\, g_2\, g_1$; note g_7 is the most significant bit. The table does not list the input values for the least significant bits $g_2\, g_1$. The reason is that for most values of the inputs, the quotient digit can be determined using only the five leading bits of the GALU output. The bits $g_2\, g_1$ are needed only near boundaries where the value of the quotient digit changes. The output in these cases is given by the lettered formulas A, B, C, D, E.

For input combinations that cannot be reached on executions of the division circuit, the table has no entry, indicated by --. It is important to verify both that the computation stays within the marked area in the table, and that the quotient selections in this part are correct.

3 Analytica

In this section, we describe a new approach to mechanical theorem proving that involves combining an automatic theorem prover with a symbolic computation system. The theorem prover, which we call *Analytica*, is able to exploit the mathematical knowledge that is built into this symbolic computation system. In addition, it can guarantee the correctness of certain steps that are made by the symbolic computation system and, therefore, prevent common errors like division by an expression that may be zero.

Analytica is written in the Mathematica programming language and runs in the interactive environment provided by this system [18]. Since we wanted to generate proofs that were similar to proofs constructed by humans, we have used a variant of the sequent calculus in the inference phase of our theorem prover. However, quantifiers are handled by skolemization instead of explicit quantifier introduction and elimination rules. Although inequalities play a key role in all of analysis, Mathematica is only able to handle very simple inequalities. We have implemented the Sup-Inf method of Bledsoe [4] to handle linear inequality systems. In addition, we have developed a technique that is able to handle a large class of non-linear inequalities as well. This technique is more closely related to the BOUNDER system developed at MIT [13] than to the traditional Sup-Inf method.

Analytica consists of four different phases: skolemization, simplification, inference, and rewriting. When a new formula is submitted to Analytica for proof, it is first skolemized to a quantifier free form. Then, in the simplification phase, a large number of rules are used to simplify the atomic formulas (i.e. equations and inequalities) with respect to the current *proof context*. If the formula reduces to true, the current branch of the inference tree terminates with success. If not, the theorem prover matches the formula against the conclusions of the available inference rules, and attempts to prove the formula by backwards chaining.

If Analytica is attempting to prove a goal and no inference rule is applicable, then Analytica tries to use rewriting to convert the goal into another equivalent form. If the formula can be rewritten, then the simplification, inference, and rewriting phases are applied to the new formula. Backtracking will cause the entire inference tree to be searched before the proof of the original goal formula terminates with failure.

Analytica contains several methods for handling both linear and non-linear inequalities. One method is based on computing upper and lower bounds for expressions. There are three main ways to obtain upper and lower bounds:

1. Obtain bounds from context information.
2. Obtain bounds from the monotonicity of some function.
3. Use some known bound on the value of a function.

The above technique is explained in more detail in the full paper. This technique can be shown to be complete for linear inequalities and can also be used to prove many of the nonlinear inequalities that arise in practice. However, the

overhead required for non-linear inequalities makes the algorithm very inefficient for linear inequalities. Consequently, we have incorporated Bledsoe's Sup-Inf method [4, 15] into Analytica for handling linear inequalities. The Sup-Inf method is treated as a special tactic in the inference phase and is applied before the more complicated inequality reasoning tactic. The Sup-Inf method provides a decision procedure for universally quantified formulas containing linear inequalities.

4 Proof of the correctness of the SRT algorithm

4.1 Axioms for the circuit

First we need to find a way represent each component of the circuit by logic expressions.

- rin1 is the leading 8 bits of rin (2 bits before binary point and 6 bits after):

$$rin1 \leq rin < rin1 + 2^{-6}$$

- md1 is the leading 8 bits of md (2 bits before binary point and 6 bits after):

$$md1 \leq md < md1 + 2^{-6}$$

- d1 is the leading 4 bits of d (constant 1 before binary point and 3 bits after):

$$d1 \leq d < d1 + 2^{-3}$$

  Thus, it d1 can only have the 8 binary values 1.000, 1.001, 1.010, 1.011, 1.100, 1.101, 1.110 and 1.111. This limitation on the range of d1 is expressed by the following formula.

$$d1 = 1 \vee d1 = \frac{9}{8} \vee d1 = \frac{5}{4} \vee d1 = \frac{11}{8} \vee d1 = \frac{3}{2} \vee d1 = \frac{13}{8} \vee d1 = \frac{7}{4} \vee d1 = \frac{15}{8}$$

- The MUX:

$$md = \begin{cases} 0 & \text{when } qdigit = 0 \\ d & \text{when } qdigit = 1 \\ 2d & \text{when } qdigit = 2 \end{cases}$$

- The GALU:

$$rout1 = \begin{cases} rin1 + md1 & \text{when } qsign \\ rin1 - md1 - 2^{-6} & \text{when } \neg qsign \end{cases}$$

- The DALU:

$$rout = \begin{cases} rin + md & \text{when } qsign \\ rin - md & \text{when } \neg qsign \end{cases}$$

- The QPOS:

$$next(QPOS) = \begin{cases} 4 \cdot QPOS & \text{when } qsign \\ 4 \cdot QPOS + qdigit & \text{when } \neg qsign \end{cases}$$

- The QNEG:

$$next(QNEG) = \begin{cases} 4 \cdot QNEG + qdigit & \text{when } qsign \\ 4 \cdot QNEG & \text{when } \neg qsign \end{cases}$$

- The Quotient:

$$Quotient = QPOS - QNEG$$

- The Remainder:

$$next(rin) = 4 \cdot rout$$

- The QUO LOGIC:
This is the hardest part in formalizing the circuit. To reduce the number of cases in the proof, we have represented each row of the quotient prediction table as a *boundary value list* $\{b_1, b_2, b_3, b_4, b_5, b_6\}$. For a given value of $d1$, we choose b_6 to be the minimal positive value for $(4 \cdot rout1)$ that is not covered by the table. For example, when $d1 = 1$, this minimal value has binary representation 0011.0. Consequently, $b_6 = 3$. Similarly, we choose b_1, b_2, b_3, b_4 and b_5 to be the minimal values for $(4 \cdot rout1)$ that gives quotient values $-2, -1, 0, 1$ and 2, respectively. When $d1 = 1$, the minimal value for $(4 \cdot rout1)$ with quotient -2 has binary representation 1100.1. Therefore, $b_1 = -7/2$. The boundary value list for each of the 8 possible values for $d1$ are shown below:

$\{-7/2, -13/8, -1/2, 1/2, 3/2, 3\}$, when $d1 = 1$;
$\{-7/2, -7/4, -1/2, 1/2, 7/4, 7/2\}$, when $d1 = 9/8$;
$\{-4, -2, -3/4, 1/2, 2, 4\}$, when $d1 = 5/4$;
$\{-9/2, -9/4, -3/4, 1/2, 2, 4\}$, when $d1 = 11/8$;
$\{-9/2, -5/2, -1, 3/4, 9/4, 9/2\}$, when $d1 = 3/2$;
$\{-5, -5/2, -1, 1, 5/2, 5\}$, when $d1 = 13/8$;
$\{-11/2, -11/4, -1, 1, 5/2, 5\}$, when $d1 = 7/4$;
$\{-11/2, -3, -1, 1, 3, 11/2\}$, when $d1 = 15/8$;

From the definition of the boundary values, we know that the following holds:

1. when $b_1 \leq 4 \cdot rout1 < b_2$, $q = -2$;
2. when $b_2 \leq 4 \cdot rout1 < b_3$, $q = -1$;
3. when $b_3 \leq 4 \cdot rout1 < b_4$, $q = 0$;
4. when $b_4 \leq 4 \cdot rout1 < b_5$, $q = 1$;
5. when $b_5 \leq 4 \cdot rout1 < b_6$, $q = 2$;
6. when $4 \cdot rout1 < b_1$, out of table and we define $q = -3$;
7. when $4 \cdot rout1 \geq b_6$, out of table and we define $q = 3$;

Let $\{b_1, b_2, b_3, b_4, b_5, b_6\}$ represent the row in the quotient prediction table that corresponds to $d1$. The QUO LOGIC is given by:

$$next(qsign) \equiv (4 \cdot rout1 < b_3)$$

$$next(qdigit) \equiv \begin{cases} 3 & \text{when } 4 \cdot rout1 < b_1 \vee 4 \cdot rout1 \geq b_6 \\ 2 & \text{when } b_1 \leq 4 \cdot rout1 < b_2 \vee b_5 \leq 4 \cdot rout1 < b_6 \\ 1 & \text{when } b_2 \leq 4 \cdot rout1 < b_3 \vee b_4 \leq 4 \cdot rout1 < b_5 \\ 0 & \text{when } b_3 \leq 4 \cdot rout1 < b_4 \end{cases}$$

4.2 Correctness of the circuit

Since the initialization phase of the circuit is simple, we will not discuss it here. The correctness of the main caculation phase of the circuit depends on two invariants:

(1) $next(Quotient \cdot d + rout) = 4 \cdot (Quotient \cdot d + rout)$
(2) $-\frac{2}{3} \cdot d \leq rout < \frac{2}{3} \cdot d$

The first invariant says that $(Quotient \cdot d + rout)$ remains constant with respect to left shifting by 2 bits. Since the initial value of this expression is the *dividend* and the computation takes 34 cycles, $(Quotient \cdot d + rout)$ equals the *dividend* left shifted by 68 bits after the computation finishes. This is the expected bevavior of the division operation. The second invariant guanrantees that the computation will never overflow. Detailed discussion of these invariants can be found in the complete version of this paper; we give a brief version here.

We want to show that $dividend = quotient \cdot divisor + remainder$, where *remainder* converges to 0. Referring to the equations for rin and rout, observe that the value placed in the Remainder register is effectively multiplied by $r = 4$ on each cycle of the circuit. Intuitively, after j cycles, the value of the actual remainder is $Remainder_j \cdot 4^{-j} = rout_j \cdot 4^{1-j}$. Similarly, the actual value of the quotient is $Quotient_j \cdot 4^{1-j}$.

The circuit computes a correct result if two conditions hold. First, the equation $Dividend = (Quotient_j \cdot d + rout_j) \cdot 4^{1-j}$ must hold. The factor 4^{1-j} represents the scaling of the quotient and remainder. Second, the value of the remainder as represented by $rout_j \cdot 4^{1-j}$ must converge to 0.

The above equation is proved by induction on j. For the base case, show that the equation holds for $j = 0$ in the initial state of the circuit. For the inductive case, we use the fact that the value of the dividend does not change. Thus we can prove the inductive case by showing that $Quotient_{j+1} \cdot d + rout_{j+1} = 4 \cdot (Quotient_j \cdot d + rout_j)$ holds on all iterations of the circuit (invariant 1).

It remains to show that the scaled value of the partial remainder converges to 0. Since d remains constant, it follows from invariant (2) that $rout_j \cdot 4^{1-j}$ converges to 0 by a factor of 1/4 on each cycle of the circuit.

Analytica is able to prove the following theorems:

- The loop invariant for $Quotient \cdot d + rout$ always holds.

$$next(Quotient \cdot d + rout) = 4 \cdot (Quotient \cdot d + rout)$$

- The GALU gives the correct estimate for the remainder.

$$rout1 \leq rout < rout1 + 2^{-5}$$

- The remainder never falls outside of the defined part of the quotient table.

$$-\frac{2}{3} \cdot d \leq rout < \frac{2}{3} \cdot d \Longrightarrow$$
$$next(qdigit) = 1 \vee next(qdigit) = 2 \vee (next(qdigit) = 0 \wedge \neg next(qsign))$$

- The loop invariant for $-\frac{2}{3} \cdot d \leq rout < \frac{2}{3} \cdot d$ always holds.

$$-\frac{2}{3} \cdot d \leq rout < \frac{2}{3} \cdot d \Longrightarrow -\frac{2}{3} \cdot d \leq next(rout) < \frac{2}{3} \cdot d$$

All theorems are proven by Analytica. The last theorem is the most interesting. The exact statement of the theorem is given below.

```
Prove[imp[and[d1 <= d < d1 + 2^(-3),
              or[d1 == 8/8, d1 == 9/8, d1 == 10/8, d1 == 11/8,
                 d1 == 12/8, d1 == 13/8, d1 == 14/8, d1 == 15/8],
              rout1 <= rout < rout1+2^(-5),
              -2/3 d <= rout < 2/3 d],
         -2/3 d <= next[rout] < 2/3 d]];
```

Notice that there are some additional conjuncts in the hypothesis part. The first two hypotheses are axioms about the values of d1. The third conjunct, relating rout and rout1, states that GALU gives a correct estimate for the remainder. Analytica proves the theorem about the GALU separately, so we can assume it as a hypothesis in this proof. The whole input required by Analytica and part of the proof it generates are shown in the complete paper.

5 Conclusion

In this paper, we investigate a radix-4 SRT division algorithm similar to the one used in the Intel Pentium processor. We have built a formal model for the circuit and proven the correctness of the model using our theorem prover Analytica.

The main obstacle to wider use of theorem proving techniques for hardware verification is the need for detailed user guidance when using most theorem provers. Therefore, it is significant that Analytica is able to prove the correctness of this circuit automatically.

In other research, we have developed a *word level model checker* [7] that can verify arithmetic circuits. Although word level model checking works extremely well for many circuits, there are still serious restrictions on the application of this technique. For example, it can only handle circuits that maintain the exact value of the data and would not be applicable for a circuit that involves rounding.

Theorem provers, on the other hand, can be applied to a wider range of problems and are particularly useful for reasoning at a high level of abstraction (architectural level verification). For instance, in this paper, we used a theorem prover to show that the division circuit is correct for all word lengths greater than 8 bits. Finite-state methods such as model checking usually verify a circuit only for a single word length. However, circuit verification by theorem proving techniques usually requires some user interaction, while model checking is largely automatic.

In the future, we intend to combine automatic theorem proving and model checking. There has already been some work in this direction [10, 12]. This combination of approaches should make it possible to handle much larger circuits

than is currently the case. In proving some property of a circuit, the specification will be decomposed into sub-goals. Each sub-goal is verified using a decision procedure or the model checker. Then the theorem prover is used to combine the proofs of the sub-goals.

References

1. APT Data Services. Pentium bug fiasco costs Intel dear. *Computer Business Review*, January 3, 1995.
2. D. E. Atkins. Higher-radix division using estimates of the divisor and partial remainders. *IEEE Transactions on Computers*, C-17(10):925–934, October 1968.
3. W. W. Bledsoe. The UT natural deduction prover. Technical Report ATP-17B, Mathematical Dept., University of Texas at Austin, 1983.
4. W. W. Bledsoe, P. Bruell, and R. Shostak. A prover for general inequalities. Technical Report ATP-40A, Mathematical Dept., University of Texas at Austin, 1979.
5. R. S. Boyer and J. S. Moore. *A Computational Logic Handbook*. Academic Press, 1988.
6. E. M. Clarke and X. Zhao. Analytica: A theorem prover for Mathematica. *The Journal of Mathematica*, 3(1), 1993.
7. E. M. Clarke, M. Khaira and X. Zhao. Word Level Symbolic Model Checking – Avoiding the Pentium FDIV Error. Design Automation Conference, June, 1996.
8. J. H. Gallier. *Logic for Computer Science: Foundations of Automatic Theorem Proving*. Harper & Row, 1986.
9. S. M. German. Towards automatic verification of arithmetic hardware. *Lecture notes*, March 1995.
10. J. Joyce and C. Seger. The HOL-Voss system: model-checking inside a general-purpose theorem prover. In *Proceedings of the 6th International Workshop on Higher Order Logic Theorem Proving and its Applications, HUG '93, LNCS 780*. Springer Verlag, 1993.
11. J. O'Leary, M. Leeser, J. Hickey, and M. Aagaard. Non-restoring integer square root: a case study in design by principled optimization. In *Proceedings of the Theorem Provers in Circuit Design '94, LNCS 901*. Springer Verlag, 1995.
12. S. Rajan, N. Shankar, and M. K. Srivas. An integration of model checking with automated proof checking. In *Proceedings of the Seventh Workshop on Computer-Aided Verification*, 1995.
13. E. Sacks. Hierarchical inequality reasoning. Technical report, MIT Laboratory for Computer Science, 1987.
14. H. P. Sharangpani and M. L. Barton. Statistical analysis of floating point flaw in the Pentium processor(1994). Technical report, Intel Corporation, November 1994.
15. R. Shostak. On the sup-inf method for proving Presburger formulas. *Journal of the Association for Computing Machinery*, 24:529–543, 1977.
16. G. S. Taylor. Compatible hardware for division and square root. In *Proceedings of the the 5th IEEE Symposium on Computer Arithmetic*, May 1981.
17. D. Verkest, L. Claesen, and H. De Man. A proof of the nonrestoring division algorithm and its implementation on an ALU. *Formal Methods in System Design*, 4:5–31, January 1994.
18. S. Wolfram. *Mathematica: A System for Doing Mathematics by Computer*. Wolfram Research Inc., 1988.

Modular Verification of SRT Division *

H. Rueß, N. Shankar, and M.K. Srivas

Computer Science Laboratory, SRI International
Menlo Park, CA 94025
{ruess,shankar,srivas}@csl.sri.com

Abstract. We describe a formal specification and verification in PVS for the general theory of SRT division, and for the hardware design of a specific implementation. The specification demonstrates how attributes of the PVS language (in particular, predicate subtypes) allow the general theory to be developed in a readable manner that is similar to textbook presentations, while the PVS `table` construct allows direct specification of the implementation's quotient look-up table. Verification of the derivations in the SRT theory and for the data path and look-up table of the implementation are highly automated and performed for arbitrary, but finite precision; in addition, the theory is verified for general radix, while the implementation is specialized to radix 4. The effectiveness of the automation derives from PVS's tight integration of rewriting with decision procedures for equality, linear arithmetic over integers and rationals, and propositional logic. This example demonstrates that the resources of an expressive specification language and of a general-purpose theorem prover are not inimical to highly automated verification in this domain, and can contribute to clarity, generality, and reuse.

1 Introduction

The SRT division algorithm is one of the most popular methods for implementing floating-point division and related operations in high-performance arithmetic units. Even though the theory of SRT division has been extensively studied [Atk68], the design of dividers still remains a serious challenge [OF94], and it is easy to make mistakes in its implementation—as was illustrated by the much publicized FDIV error in the Intel Pentium chip. As Pratt [Pra95] points in his analysis, it is unlikely testing alone would have caught that error as it was due to five wrong entries in the quotient look-up table in a region of the table that was thought to be unreachable. Hence, formal verification can play an essential role in the design and debugging of arithmetic circuits.

In this paper, we present a mechanized verification of a general SRT division algorithm that can be used for performing floating-point divisions and an implementation of it based on the circuit given in [Tay81]. This circuit implements the IEEE floating-point standard, and its kernel consists of a fixed-point iteration.

* Supported in part by ARPA under Arpa Order A721, by NASA under contract NAS1-20334, and by NSF Grant No. CCR-930044. We gratefully acknowledge the valuable guidance and help given by John Rushby, Sam Owre, Ed Clarke, and Steve German.

The verification of this kernel was performed in the interactive theorem proving system PVS [ORSvH95]. Since our goal was to perform the verification so that as much of the initial set-up effort can be reused in the verification of other similar circuits, we took a modular approach that separates concerns about general facts of the SRT theory from a specific circuit implementation and a look-up table. Furthermore, we develop clear interfaces between these parts, so that each of the verifications can be done separately.

More precisely, the formalization and verification of the SRT divider proceeds in two steps. First, we formalize textbook knowledge about SRT dividers at an algorithmic level and verify its correctness. The formalization at this level does not use a specific data path to compute the partial remainder nor a specific look-up table. It characterizes a set of semantic constraints a look-up table ought to satisfy and a recurrence relation a partial remainder computation circuit ought to preserve. In the second step, we specify a data path circuit (bit-vector signals over time) to compute the partial remainder and define a specific look-up table, both of which are based on the implementation given in [Tay81]. We then show that the data path circuit and the look-up table meet the constraints characterized in step one. Both steps of the verification are performed for arbitrary, but finite precision, which appears as a parameter to the specification. The first step of the verification is applicable to arbitrary radixes, while the second step assumes a radix-4 implementation, since it uses a look-up table for radix-4.

2 Related Work

Claesen et al. [VCM94] and Leeser and O'Leary [LO95] have used theorem provers to verify a non-restoring divider and a radix-2 subtractive square root algorithm, respectively. The circuits verified in both of these efforts are not based on the SRT method and hence do not contain the kinds of optimizations used in SRT division. Recently, German and Clarke [Ger95, CG95] performed a verification of Taylor's SRT divider circuit considered in this paper by manually deriving a set of inequalities that the circuit imposes on the data path signals and then showing, in the MAPLE symbolic algebraic system, that two main SRT correctness invariants are preserved by the data path inequalities. This work provided the main impetus for our work. Clarke et al. [CGZ96] have independently mechanized their verification in the ANALYTICA theorem prover. Our work not only mechanizes all the steps in the verification of the SRT circuit, but also formalizes the general SRT theory correctness and develops a modular framework which can be used to verify other similar circuits. While their specification interprets signals in the circuit as arbitrary real numbers, we interpret signals as parameterized finite, but arbitrary-length bit-vectors.

Methods based on ordered BDDs and symbolic model checking are not well-suited for verifying multipliers and dividers since BDD graphs for such operations grow exponentially with the word size [Bry94]. However, Bryant [Bry95] has used BDDs to check the relation that one iteration of the SRT circuit must preserve for the circuit to correctly divide. To do the verification, he needed to

construct a gate-level representation of a *checker-circuit* (much larger than the verified circuit) to describe the desired behavior of the verified circuit, which is not the ideal level of specification.

While Bryant's BMDs can be used to verify multipliers against their number-theoretic specification [Bry94], they cannot be used for SRT verification, because they cannot efficiently check inequalities over bit-vectors. But Clarke and Zhao [CZ95] have recently extended the symbolic model-checking algorithm used in SMV to express and verify word-level properties on numbers. They use an extension of BDDs called *hybrid decision diagrams* to represent integer functions and check relations on them. The word-level model-checker can be used to check if finite-sized arithmetic circuits satisfy desired number-theoretic properties. They have used the word-level model checker to verify Taylor's SRT circuit by checking if a state transition model of the circuit satisfied the main SRT invariants. Both [CZ95] and [CGZ96] are only applicable for fixed-sized data paths.

3 An Overview of PVS

The PVS system combines an expressive specification language with a productive, interactive proof checker that has a reasonable amount of theorem proving capabilities, and has been used for reasoning in domains as diverse as microprocessor verification, protocol verification, and algorithm and architectures concerning fault-tolerance [ORSvH95]. The PVS specification language builds on classical typed higher-order logic with the usual base types, function type constructor, dependent types, and abstract data types. A distinctive feature of PVS are *predicate subtypes* `{x:A | P(x)}`. These subtypes consist of exactly those elements `a` of type `A` satisfying predicate `P(a)`. Predicate subtypes are used to explicitly constrain the domain and ranges of operations in a specification and to define partial functions. In general, type-checking with predicate subtypes is undecidable, and the type-checker generates *type correctness conditions* (TCCs) corresponding to predicate subtypes.

Proofs in PVS are presented in a sequent calculus. The atomic commands of the PVS prover component include induction, quantifier instantiation, automatic conditional rewriting, simplification using arithmetic and equality decision procedures and type information, and propositional simplification using binary decision diagrams. PVS has an LCF-like strategy language for combining inference steps into more complicated proof strategies.

4 SRT Division

SRT dividers [McS61, Rob58, Toc58] speed up nonrestoring division and are widely used in high-speed floating point units. The quotient is represented in radix-r form and one digit of it is calculated in each iteration. To obtain fast algorithms, SRT division represents quotient digits using a *redundant digit set*

$[-a, \ldots, a]$ so that there can by multiple choices for the most significant quotient digit for a given partial remainder and divisor. The redundancy can be used to correct small errors in one iteration in subsequent iterations. It also allows the quotient digit to be computed in parallel with the partial remainder using an approximation of the partial remainder. The common choices $r = 4$ and $a = 2$ lead to adequate and efficient circuits, since multiplication by 0, 1, and 2 is easy.

The presentation of fundamental concepts about SRT division covers the basic recurrence, the conditions under which the computation converges to a reasonable result, and the quotient selection criterion both in exact and approximate forms. These general arithmetic facts about SRT division are presented in terms of their PVS formalization and are parameterized with respect to algorithm-specific details such as the radix and the set of quotient digits. In Sections 5 and 6 we instantiate these arithmetic facts to verify the correctness of a specific high-speed radix-4 SRT circuit on the bit-level. Note also, that we restrict ourselves in this paper to the verification of fixed-point division kernels of IEEE compliant floating-point division.

Subtractive Division Algorithms. Given two normalized fractions p and d of the form $1.xx \ldots xx_2$, the digit recurrence

$$\begin{aligned} p_0 &= p \\ p_{i+1} &= r * (p_i - q_i * d) \text{ with the constraint } |p_{i+1}/d| \leq r * \rho \\ &\qquad \text{where } \rho = r * a/(r-1) \end{aligned}$$

computes the value of p/d by producing one quotient digit q_i and a new partial remainder p_{i+1} in each iteration i. The constraint on the partial remainder is needed to guarantee convergence of the algorithm. The above characteristics of subtractive division algorithms are formalized in 1 for arbitrary radices `r:upfrom[2]` and sets of quotient digits `subrange[-a,a]` such that `a:posnat` and `r/2 <= a < r - 1`.

1

```
p_new, p: VAR rational; q: VAR subrange[-a, a]; d: VAR posrat

recurrence?(p_new, p, q, d): bool = (p_new = r * (p - q * d))
rho: rational = a / (r - 1)
p_over_d_bound?(d, p): bool = (-r * rho <= p / d & p / d  <= r * rho)
```

Convergence. The function `valq(i + 1, q)` in 2 computes the radix-`r` fixed-point value of the accumulated quotient digits `q(0).q(1) ...q(i)`, and Theorem `convergence` states that `q(0).q(1)...q(i)` is an approximation to the infinite precision fraction `p(0) / d` within an error bound.

2

```
i, j, k: VAR nat;                    d: VAR posrat
p      : VAR sequence[rational]; q: VAR sequence[subrange[-a, a]]

val(i, q): RECURSIVE rational =
 IF i = 0 THEN 0 ELSE  q(i - 1) * 1/r^(i - 1) + val(i - 1, q) ENDIF
MEASURE i

lemma1: LEMMA
 (FORALL j: recurrence?(p(j + 1), p(j), q(j), d))
   IMPLIES p(0) / d - val(i, q) = 1/r^i * (p(i) / d)

convergence: THEOREM
 ((FORALL j: recurrence?(p(j + 1), p(j), q(j), d)) AND
  (FORALL k: p_over_d_bound?(d, p(k))))
   IMPLIES LET residue = p(0) / d - val(i + 1, q) IN
            -1/r^i * rho <= residue & residue <= 1/r^i * rho
```

This theorem is an immediate consequence of the invariant `lemma1` and the given bound on `p(i + 1) / d`, and `lemma1` is proven automatically in PVS with the general-purpose induction strategy `induct-and-simplify` and some basic facts from the library about rational numbers.

Quotient Selection. The hard part in each iteration is to determine a quotient digit `q(i)` such that the next partial remainder `p(i + 1)` also satisfies the boundary constraint `p_over_d_bound?`. By substituting the recurrence relation defining the new partial remainder into the bound contraint on the partial remainder, one can obtain the condition `legitimate?` that characterizes a selection interval of legitimate choices of quotient digits.

3

```
q: VAR subrange[-a, a]; d: VAR posrat; p: VAR rat

legitimate?(q, d, p): bool = q - rho <= p/d  &  p/d <= q + rho

lemma2: LEMMA recurrence?(p_new, p, q, d) IMPLIES
               (p_over_d_bound?(d, p_new) IFF legitimate?(q, d, p))
```

Note that the boundaries of this interval depend on the divisor `d`, and Figure 1 graphically displays the region for legitimately selecting quotient digits -2 through 2 for the choices `r = 4` and `a = 2` (thus `rho = 2/3`). The region for legitimately selecting `q = 1`, for example, is bound by the dashed lines `5/3 * d` and `1/3 * d`.

For specific interpretations, say `r = 4` and `a = 2`, the combination of decision procedures with rewriting on known facts about real and rational numbers (`grind :theories "real_props"`) discharges the proof obligation `lemma2` in 3 automatically. In the general case, however, where `r` and `a` are uninterpreted, the proof of this fact involves solving non-linear inequalities, and the

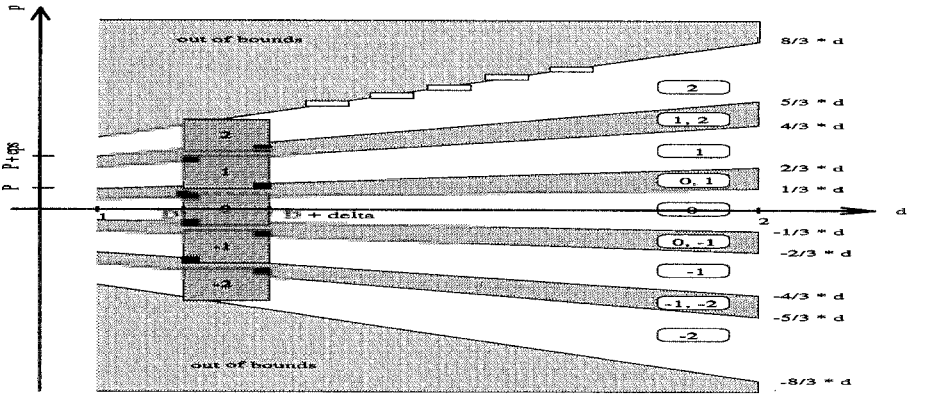


Fig. 1. pd-plot for `r = 4` and `a = 2`

PVS prover needs two interactions to guide the manipulation of these non-linear inequalities.

Redundancy. Shaded regions in Figure 1 indicate pairs `(d, p)` for which selection intervals for the quotient digit `q` overlap. This redundancy permits calculating `q` from truncated versions `P:rational` and `D:posrat` of the partial remainder and divisor respectively.

4

```
D: VAR posrat; P: VAR rational

P_bound_by_D?(D, P): bool =
 -eps - r * rho * (D + delta) < P & P < r * rho * (D + delta)

lemma3: LEMMA (P <= p & p < P+eps & D <= d & d < D + delta AND
  p_over_d_bound?(d,p)) IMPLIES P_bound_by_D?(D, P)
```

Let `delta, eps:posrat` be two arbitrary positive rational numbers. Assuming that `P` and `D` underestimate `p` and `d`, respectively, the constraint `p_over_d_bound?` imposed by the algorithm on the partial remainder, imposes a corresponding bound on `P` as a function of `D`. This constraint is defined and proved (by `lemma3`) in 4. Note that if negative numbers are represented in 2-complement form, which is what we assume in the circuit we verify later, truncation (after the binary point) always produces a number less than the actual value.

Inspection of the *pd-plot* in Figure 1 reveals that the legitimacy of quotient selection for the marked corners of the shaded rectangles suffices to show the legitimacy of selecting this quotient digit for all `(d, p)` pairs in the rectangle. Consequently, the constraint `lookup_legitimate?` in 5 on lookup tables guarantees the legitimacy of quotient selection as shown in `lemma4`. The combination of the PVS decision procedures with facts from the library about rational numbers proves `lemma4` automatically when `r` and `a` are instantiated with specific

numeric values; otherwise manual guidance is needed to deal with non-linear equalities.

5

```
lookup_legitimate?(q, D, (P: rational | P_bound_by_D?(D, P))): bool =
 COND
  q = a      -> (a - rho) * (D + delta) <= P,
  0<q & q<a  -> (q - rho) * (D + delta) <= P & (P + eps) <= (q+rho)*D,
  q = 0      -> -rho * D <= P & (P + eps) <= rho * D,
 -a<q & q<0  -> (q - rho) * D <= P & (P + eps) <= (q + rho) *(D+delta),
  q = -a     -> (P + eps) <= (-a + rho) * (D + delta)
 ENDCOND

 lemma4: LEMMA
  (P <= p & p < P + eps  AND D <= d & d < D + delta AND
   p_over_d_bound?(d, p) AND lookup_legitimate?(q, D, P))
    IMPLIES legitimate?(q, d, p)
```

Quotient Prediction. A significant reduction of the overall cycle time is obtained by computing the next partial remainder `p(i + 1)` and predicting a next quotient digit `q(i + 1)` in parallel. In this case, the approximation `P(i)` used in iteration i, (under)estimates the next partial remainder `p(i + 1)`. Note that `P(i)` can be computed faster than `p(i + 1)`, since most of the time taken to compute `p(i + 1)` is a full-precision addition, and the computation of `P(i)` only involves a limited-precision adder.

It is a simple matter of combining the results in 3, 4, 5 to prove the statement `invariant` in 6 for SRT dividers with quotient prediction where the non-trivial part of the induction step involves the chain of implications

```
   legitimate?(q(i), d, p(i))
⇒ remainder_bound?(d, p(i + 1))
⇒ estimation_bound?(D, P(i))
⇒ lookup_legitimate?(q(D, P(i)), D, P(i))
⇒ legitimate?(q(i + 1), d, p(i + 1))
```

6

```
invariant: THEOREM
 (p_over_d_bound?(d, p(0))) AND (legitimate?(q(0), d, p(0))) AND
  (FORALL j: recurrence?(p(j + 1), p(j), q(j), d) AND
    P(j) <= p(j+1) & p(j+1) < P(j)+eps & D <= d & d < D+delta AND
     (P_bound_by_D?(D,P(j)) IMPLIES lookup_legitimate?(q(j+1),D,P(j))))
  IMPLIES
   (p_over_d_bound?(d, p(i)) AND legitimate?(q(i), d, p(i)))
```

Altogether, to prove the correctness of a specific SRT divider circuit it suffices to show that *1)* the arithmetic interpretations of the computed sequences of partial remainders and quotient digits satisfy the recurrence relation `recurrence?`,

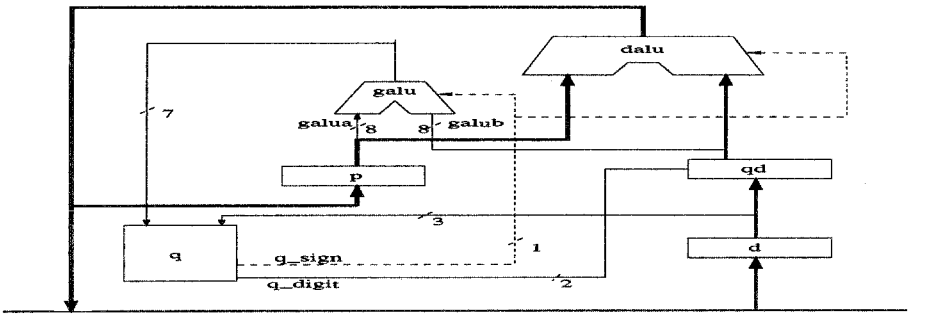


Fig. 2. The data path for the division circuit.

2) there are constants `delta` and `eps` such that the divisor and the partial remainders are bound by under-estimators in the sense described above, and *3)* the quotient selection logic satisfies the `lookup_legitimate?` predicate. Whenever these conditions hold, theorem `invariant` in [6], and consequently theorem `convergence` in [2], is applicable.

5 Modeling The Data Path

Now, the data path of an SRT division circuit with `r = 4` and `a = 2` as described by Taylor [Tay81] is specified and proven to be correct by applying the general SRT theory developed in Section 4.

The signals of the circuit in Figure 2 are declared as uninterpreted constants of signals of bit-vectors of various fixed lengths, and the uninterpreted constant `N:posnat`, where `N > 8`, determines the width of the data paths for the divisor and the partial remainders; examples of signal declarations and their interpretation functions are listed in [7].

7
```
d: signal[bvec[N]]; d(i): rational = fp[1,N-1].val(d(i))
P: signal[bvec[7] ; P(i): rational = fp2c[4, 3].val(P(i))
```

The divisor signal `d` has a fixed-point interpretation with 1 leading and `N-1` residual bits, and the estimation `P` of the next partial remainder has a 2-complement fixed-point interpretation with 4 leading bits and 3 residual bits. Note also that overloading the name of the bit-vector signal with its arithmetic interpretation mimics a specification style often found in textbooks about computer arithmetic.

The inputs to the quotient selection unit `q` are the three bit truncation of the divisor `d` and the seven bit approximation `P` of the next partial remainder.

8
```
lookup((D: bvec[3]),
       (P: bvec[7] | P_bound_by_D?(1+fp[0,3].val(D),fp2c[4,3].val(P))))
:{ q: subrange(-2, 2) |
    lookup_legitimate?(q, 1 + fp[0,3].val(D), fp2c[4, 3].val(P)) }
```

Here, predicate subtypes serve as a specification of a set of quotient look-up tables by means of domain and range constraints, and a specific implementation of these constraints is proven correct in Section 6. The behavior of the circuit is specified by equality and inequality (to capture the effect of truncation) constraints on the inputs and outputs of the `dalu` and `galu` components which are omitted here for lack of space.

From these formalizations, the `grind` strategy proves the lemmas in [9] about Taylor's division circuit in Figure 2.

9

```
taylor_lemma1: LEMMA recurrence?(p(t + 1), p(t), q(t), d(t))
taylor_lemma2: LEMMA galu(t) <= dalu(t) & dalu(t) < galu(t) + 2 *ulp(6)
taylor_lemma3: LEMMA P(t) <= p(t + 1) & p(t + 1) < P(t) + 3/16
```

Together with the constraint on `D` with respect to `d`, this accomplishes Step 2 with `delta = 1/8` and `eps = 3/16` mentioned in Section 4. Now it is a simple matter of instantiating the theorem `invariant` in [6] and `convergence` in [2] to obtain the invariant results in [10] for this specific circuit.

10

```
taylor_invariant: LEMMA
 p_over_d_bound?(d(0), p(i)) AND legitimate?(q(i), d(0), p(i))

taylor_convergence: THEOREM
 LET residue =  p(0) / d(0) - val(i + 1, q) IN
  - 2 / (3 * 4^i) <= residue & residue <= 2 / (3 * 4^i)
```

6 The Look-Up Table.

The legitimacy constraint `lookup_legitimate?` (see [8]) on quotient look-up tables permits different implementations, and Taylor [Tay81] develops a particularly compact one. This table computes the next quotient digit from the truncation `D:bvec[3]` of the divisor to the three leading bits and the estimation `P:bvec[7]` of the next partial remainder. Bits 6 down to 2 of `P` are used as a table index and the remaining bits are used in some cases to compute the resulting value.

The formalization of the resulting table `q(D, P)` (shown in Appendix A) uses the `TABLE` construct of the PVS specification language [ORS95]. This construct was added to the PVS specification language in order to provide visually appealing two-dimensional tabular specifications in the manner advocated by Parnas and others [Par95]. It proved adequate to express the look-up table of this SRT circuit in a concise and perspicuous way. In particular, blank entries in the look-up table cause the type-checker to generate TCCs which ensure that viable arguments `D`, `P` never point to such a blank entry. Furthermore, the table construct requires that the look-up is functional and ensures this by generating *disjointness* and *coverage* TCCs. From the fact

```
(FORALL (D, (P: bvec[7] | estimation_bound?(valD(D), valP(P)))):
  lookup_legitimate?(q(D, P), valD(D), valP(P)))
```
11

one concludes that the given table q indeed satisfies the constraint given for look-up tables in Section 5. A simple case split on the different values of D and P followed by unfolding definitions, term rewriting, and calls to the decision procedures proves the theorem in 11. The type correctness conditions generated by the type-checker for the look-up table are proven with similar strategies.

In the course of proving the consistency of the look-up table, PVS has proven helpful as a debugging tool and came up with precise *counterexamples*.[2] By injecting, for example, a wrong value 0 at a certain position in the look-up table and rerunning the proof above, the PVS prover returns an unsolved subgoal that yields an immediate counterexample. Note that the 5 missing entries in the look-up table of initial releases of the Pentium floating-point unit were also right at the upper boundary of the legitimate selection region for q = 2 as depicted in Figure 1 by the blank rectangles.

7 Summary and Conclusions

We have shown how PVS can be used to specify and prove correctness of a non-trivial SRT division algorithm and its hardware implementation in a modular way. This modular approach not only structures the specifications and the proof in a nice way but also has the advantage that slight variations of this particular circuit design and look-up table can be verified by just redoing one part of the proof. Moreover, parts of the theory can be reused for verifying other similar division, and perhaps even square-root, circuits.

This verification exercise demonstrates the value of efficient decision procedures and the use of an expressive specification language in mechanized verification. The concepts of predicate subtypes, overloading, and tables of the PVS specification language proved to be very useful for expressing the high-level designs of this arithmetic circuit in a concise and natural way. Such high-level descriptions reduce the possibility of introducing errors in initial design specification and can also serve as design documents. The tight integration of decision procedures with rewriting strategies of PVS proved to be a useful workhorse, since the circuit specific theorems and the correctness of the table implementation are proven in a fairly automatic way. In most proof obligations that involve non-linear equalities, however, the PVS prover must be manually guided to construct the proofs.

This case study also suggests some improvements to the implementation of PVS. The correctness proof of the table implementation in Section 6 takes 3

[2] Even though the original design of Taylor's look-up table in [Tay81] proved to be correct, we still managed to accidentally inject errors in the initial PVS transcriptions.

hours. This is unreasonably slow, since the proof basically involves small case analysis followed by the evaluation of ground predicates. The incorporation of an efficient notion of evaluation into the proving process could drastically reduce the time for doing this and many other hardware-related proofs. In the future we plan to extend this case study to the verification of related circuits and operations, such as square root, and investigate other concepts like IEEE compliant rounding [Min95].

References

[Atk68] D.E. Atkins. Higher-radix Division Using Estimates of the Divisor and Partial Remainders. *IEEE Transactions on Computers*, C-17(10):925–934, October 1968.

[Bry94] R.E. Bryant. Verification of Arithmetic Functions with Binary Moment Diagrams. Technical Report CMU-CS-94-160, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, 1994.

[Bry95] R.E. Bryant. Bit-Level Analysis of an SRT Divider Circuit. Technical Report CMU-CS-95-140, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, April 1995.

[CG95] E.M. Clarke and S.M. German. Personal Communication, 1995.

[CGZ96] E.M. Clarke, S.M. German, and X. Zhao. Verifying the SRT Division Algorithm using Theorem Proving Techniques. Submitted to CAV'96, 1996.

[CZ95] E.M. Clarke and X. Zhao. Word Level Symbolic Model Checking: A New approach for Verifying Arithmetic Circuits. Technical Report CMU-CS-95-161, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, April 1995.

[Ger95] S.M. German. Towards Automatic Verification of Arithmetic Hardware. Lecture notes, March 1995.

[LO95] M. Leeser and J. O'Leary. Verification of a Subtractive Radix-2 Square Root Algorithm and Implementation. In *Proc. of ICCD'95*, pages 526–531. IEEE Computer Society Press, 1995.

[McS61] O.L. McSorley. High-speed Arithmetic in Binary Computers. In *Proc. of IRE*, pages 67–91, 1961.

[Min95] P.S. Miner. Defining the IEEE-854 floating-point standard in PVS. NASA Technical Memorandum 110167, NASA Langley Research Center, Hampton, VA, June 1995.

[OF94] S.F. Oberman and M.J. Flynn. Design Issues in Floating-Point Division. Technical Report CSL-TR-94-647, Dept. of Computer Science, Stanford University, Stanford, CA 94305-2140, December 1994.

[ORS95] S. Owre, J. Rushby, and N. Shankar. Analyzing Tabular and State-Transition Specification in PVS. Technical Report CSL-95-12, Computer Science Laboratory, SRI International, Menlo Park CA 94025 USA, June 1995.

[ORSvH95] S. Owre, J. Rushby, N. Shankar, and F. von Henke. Formal Verification for Fault-Tolerant Architectures: Prolegomena to the Design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, February 1995.

[Par95] D. L. Parnas. Using mathematical models in the inspection of critical softwa re. In Michael G. Hinchey and Jonathan P. Bowen, editors, *Ap-*

plications of Formal Methods, International Series in Computer Science, chapter 2, pages 17–31. Prentice Hall, 1995.

[Pra95] V. Pratt. Anatomy of the Pentium Bug. In P.D. Mosses, M. Nielsen, and M.I. Schwartzbach, editors, *TAPSOFT'95: Theory and Practice of Software Development*, number 915 in Lecture Notes in Computer Science, pages 97–107. Springer Verlag, May 1995.

[Rob58] J.E. Robertson. A new Class of Digital Division Methods. In *IRE Trans. on Electron. Computers*, volume EC-7, pages 218–222, 1958.

[Tay81] G.S. Taylor. Compatible Hardware For Division and Square Root. In *Proceedings of the 5th Symposium on Computer Arithmetic*, pages 127–134. IEEE Computer Society Press, 1981.

[Toc58] K.D. Tochter. Techniques of Multiplication and Division for Automatic Bin ary Computers. In *Quart. J. Mech. Appl. Match*, volume Part 3, pages 364–384, 1958.

[VCM94] D. Verkest, L. Claesen, and H. De Man. A Proof of the Nonrestoring Division Algorithm and its Implementation on an ALU. *Formal Methods in System Design*, 3:5–31, January 1994.

A Implementation of the Lookup Table

```
q(D, (P | P_bound_by_D?(...,...))): subrange(-2,2) =
 LET a= -(2-P(1)*P(0)), b= -(2-P(1)), c= 1+P(1), d= -(1-P(1)), e= P(1)
 IN TABLE bv2pattern(P^(6,2), bv2pattern(D)
                  |[ 000| 001| 010| 011| 100| 101| 110| 111]|
               %-----------------------------------------------%
               |01010|    |    |    |    |    |    |    2 ||
               |01001|    |    |    |    |    2 |    2 |  2 ||
               |01000|    |    |    |    2 |    2 |    2 |  2 ||
               |00111|    |    |  2 |  2 |  2 |  2 |    2 |  2 ||
               |00110|    |  2 |  2 |  2 |  2 |  2 |    2 |  2 ||
               |00101|  2 |  2 |  2 |  2 |  2 |  2 |    2 |  1 ||
               |00100|  2 |  2 |  2 |  2 |  c |  1 |    1 |  1 ||
               |00011|  2 |  c |  1 |  1 |  1 |  1 |    1 |  1 ||
               |00010|  1 |  1 |  1 |  1 |  1 |  1 |    1 |  1 ||
               |00001|  1 |  1 |  1 |  1 |  e |  0 |    0 |  0 ||
               |00000|  0 |  0 |  0 |  0 |  0 |  0 |    0 |  0 ||
               |11111|  0 |  0 |  0 |  0 |  0 |  0 |    0 |  0 ||
               |11110| -1 | -1 |  d |  d |  0 |  0 |    0 |  0 ||
               |11101| -1 | -1 | -1 | -1 | -1 | -1 |   -1 | -1 ||
               |11100|  a |  b | -1 | -1 | -1 | -1 |   -1 | -1 ||
               |11011| -2 | -2 | -2 |  b | -1 | -1 |   -1 | -1 ||
               |11010| -2 | -2 | -2 | -2 | -2 | -2 |    b | -1 ||
               |11001| -2 | -2 | -2 | -2 | -2 | -2 |   -2 | -2 ||
               |11000|    |    | -2 | -2 | -2 | -2 |   -2 | -2 ||
               |10111|    |    |    | -2 | -2 | -2 |   -2 | -2 ||
               |10110|    |    |    |    |    | -2 |   -2 | -2 ||
               |10101|    |    |    |    |    |    |   -2 | -2 ||
               %-----------------------------------------------%
```

Mechanically Verifying a Family of Multiplier Circuits *

Deepak Kapur **M. Subramaniam**

Computer Science Department
State University of New York
Albany, NY 12222
kapur@cs.albany.edu, subu@cs.albany.edu

Abstract. A methodology for mechanically verifying a family of parameterized multiplier circuits, including many well-known multiplier circuits such as the linear array, the Wallace tree and the 7-3 multiplier is proposed. A top level specification for these multipliers is obtained by abstracting the commonality in their behavior. The behavioral correctness of any multiplier in the family can be mechanically verified by a uniform proof strategy. Proofs of properties are done by rewriting and induction using an automated theorem prover *RRL* (Rewrite Rule Laboratory). The behavioral correctness of the circuits is established with respect to addition and multiplication on numbers. The automated proofs involve minimal user intervention in terms of intermediate lemmas required. Generic hardware components are used to segregate the specification and the implementation aspects, enabling verification of circuits in terms of behavioral constraints that can be realized in different ways. The use of generic components aids reuse of proofs and helps modularize the correctness proofs, allowing verification to go hand in hand with the hardware design process in a hierarchical fashion.

1 Introduction

There has been a great deal of interest in verifying properties of hardware circuits at the input-output level. Many papers on this topic have appeared in conference proceedings and journals[10], to cite a few [3, 5, 8, 11, 6, 17]. Different approaches have been proposed in the literature, notably among them state-based approaches and the use of model checkers [5, 3], induction-based approaches adapted from software verification [8, 12] and finally approaches based on modeling hardware circuits using higher-order logics [6, 11].

Despite this widespread interest, verification efforts involving multiplier circuits have been few in comparison[14, 4, 13]. The state based approaches and model checking that employ binary decision diagrams (BDDs) or some variant of these, do not perform well on multiplier circuits due to the associated state explosion (see further discussion on this in the next section on related work). It is possible to verify the correctness of multipliers using theorem provers and proof checkers but such efforts have also been limited as they are ad hoc in nature and require considerable user ingenuity.

* Partially supported by the National Science Foundation Grant no. CCR-9308016.

The focus of this paper is on the use of an automated theorem prover for mechanically verifying parameterized multiplier circuits. A methodology for specifying and verifying a family of parameterized multipliers circuits is described. The behavioral correctness of many well-known multiplier circuits such as the linear array, the Wallace tree and the 7-3 multipliers, with respect to addition and multiplication over numbers, is mechanically verified using an automated theorem prover *RRL* (*Rewrite Rule Laboratory*)[15].

We first develop a common top level equational specification for a family of multiplier circuits by abstracting the commonality in behavior. A multiplier circuit is abstracted in terms of two components: a component that computes the partial sums, called the *partial sum computation* component and another that adds these partial sums to compute the final product, called the *partial sum addition* component. We then describe a uniform approach for mechanically verifying the correctness of any multiplier in the family using *RRL*. It is shown that the correctness of any multiplier circuit in the family can be mechanically established from the behavioral correctness of the partial sum computation component and that of the partial sum addition component. The correctness of these components follow from the correctness of the adder circuits used in them.

The proposed approach is highly generic – not only abstracting over the word size of multiplier circuits but also abstracting the common behavior of a variety of different multiplier circuits. The proofs of correctness are obtained for multiplier circuits of arbitrary word size. Secondly, seemingly different multiplier circuits share a common specification and proof of correctness using the same lemmas, with only a few different definitions for each multiplier circuit.

A major complaint against the use of theorem provers and proof checkers for hardware verification has been the semi-automatic nature of these systems. Verification efforts using these systems involve considerable user ingenuity. We believe that a common top level proof for a family of multiplier circuits with well-characterized intermediate lemmas that are independent of the underlying prover, is a step in addressing this issue. It is shown that the intermediate lemmas used in the proofs of correctness of multipliers reported here correspond to formulas that specify the input-output behavior expressed in terms of numbers, of different components of the circuits. Such lemmas, we speculate, can be generated systematically from the structure of the circuits.

In our specifications and proofs of various multiplier circuits, we abstract the adders in terms of generic hardware components with associated behavioral constraints. The correctness of the multiplier circuits is first established in terms of such generic components. It is shown later how a particular adder can realize a generic component by demonstrating that the adder satisfies the behavioral constraints of the generic component. Such a view provides a clear separation between specification and implementation aspects. The use of generic components aids reuse of proofs and modularize the correctness proofs, allowing verification to go hand in hand with the design process in a hierarchical fashion. Such modularization of proofs is crucial for any verification methodology to effectively scale up to larger and more complex hardware circuits.

Let us briefly review main aspects of different multiplier circuits considered in this paper. A linear array multiplier performs the multiplication of two n bit numbers in linear time by computing the partial sums corresponding to the given numbers and adding the partial sums together to obtain the required result. Addition of partial sums is done by considering one partial sum at a time. Wallace in [20] introduced a multiplication scheme, which has popularly come to be known as the Wallace tree multiplier, for multiplying two n bit numbers in logarithmic time. Improved performance is achieved in the Wallace tree multiplier by considering three partial sums for addition together. The multiplication scheme due to Wallace was generalized and improved upon by Dadda in [7] leading to a rich family of multipliers called the *Dadda multipliers*. In these multipliers, larger than three partial sums are taken up for addition at a particular time. Considering larger number of partial sums does not improve the asymptotic complexity but considerably reduces the number of stages required for multiplication resulting in reduced wiring delays. The 7-3 multiplier used in *IBM RS/6000* is based on this observation and has been attributed [9] as one of the important features that contributes to its good performance.

Most of these multiplier circuits are based on the grade school principle of multiplying any two given n bit numbers–computing the partial sums and adding the partial sums to obtain the required result. This basic underlying principle is often not evident in commonly found descriptions of these circuits. The computation of partial sums is done in the same manner in these circuits, and these circuits differ only in the number of partial sums that they consider for addition at any particular time. A common top level specification for the family of multiplier circuits based on this observation is developed in Section 3. In section 4, the behavioral correctness of different multiplier circuits with respect to addition and multiplication over numbers are presented. The use of generic hardware components in specifying and verifying different multiplier circuits using *RRL* is discussed in section 5.

2 Related Work

Among the various approaches employed for hardware verification, the state based approaches based on symbolic manipulation of boolean functions using binary decision diagrams *BDDs* [3] are perhaps the most popular for verifying hardware circuits of fixed word size (non-parametric circuits). A circuit is specified using a boolean function that can be succinctly represented using a *BDD*. Further *BDD*s provide a fast mechanism for comparing boolean functions. Even for linear circuits, in which the output is a linear function of the inputs, this approach has two major limitations: (i) it is unclear how circuits of arbitrary word size can be verified, and (ii) verification is limited to showing that a circuit implements a boolean function, and not a function on numbers.

It is well-known that for many important boolean functions, especially the ones for multiplication, that grow exponentially with the word size, the state-based approaches are less attractive for verification. Bryant and Chen recently introduced a new data structure *Multiplicative Binary Moment Diagram (BMD)* for modeling the functionality of circuits in terms of data at the word level [4].

Using this approach, a number of integer multiplier designs with word sizes up to 256 bits have been verified. However, such verifications are not fully automatic as Bryant and Chen in [4] state:

....the overall circuit is divided into components, each having a word level specification. Verification involves proving 1) that each component implements its word level specification and 2) that the composition of the word level component functions matches the specification.....

The approach advocated in this paper using a theorem prover *RRL* for verifying multiplier circuits is similar to the one suggested using BMDs. We decompose a multiplier circuit into two components, and establish the number-theoretic correctness of the individual components. The overall proof then follows by the composition of these two components. The automated proofs obtained using our theorem prover *RRL* do not entail any additional overheads. Due to the generality afforded by theorem provers like *RRL*, it was further possible to obtain common proof for a family of multiplier circuits of arbitrary sizes (parametric circuits) which would be infeasible otherwise.

Approaches based on theorem provers and proof checkers have been widely used to verify hardware circuits. Most of this effort has focussed on verification of different forms of processors [11, 17, 8], different forms of ALUs [19, 8] or has been used for the verification of adder circuits [19, 8, 16, 12]. In [14], a Braun Multiplier is formally specified using the Boyer-Moore logic and some properties about this specification are proven using *Nqthm* [1].

In [18], a framework for synthesizing a variety of hardware circuits including the carry save and Wallace tree multipliers is proposed. Higer order metafunctions with different circuit interconnection structures such as the carry save array and the Wallace tree as inputs are **manually** transformed to realize multipliers at the gate level. The correctness of the circuits is established by reasoning about the behavior of these metafunctions and the associated transformations using the automated theorem prover *HOL*. We are unaware of other mechanical verification efforts where the correctness of multipliers such as the Wallace tree multiplier or the 7-3 multiplier have been mechanically established with minimal user guidance using a uniform framework such as ours.

3 Specifying a family of Multiplier Circuits

A common, top level equational specification for a family of multiplier circuits is developed in this section. The Wallace tree multiplier is used as an example to illustrate the methodology. The overall structure of the Wallace tree multiplier can be described diagrammatically as in Fig. 1.

Given bit vectors x and y of equal length, a Wallace tree circuit first computes a list of partial sums ($P1, \cdots, P8$ in Fig. 1) using a function such as *psum-all*. Each partial sum in the list is a bit vector that corresponds to a single bit of x and is obtained by shifting y appropriately. The partial sums in the list are then added together by adding in parallel three partial sums at a time. Addition of any three partial sums is typically done using a carry save adder(CSA) that has three bit vectors as its inputs and produces a pair of bit vectors as its output.

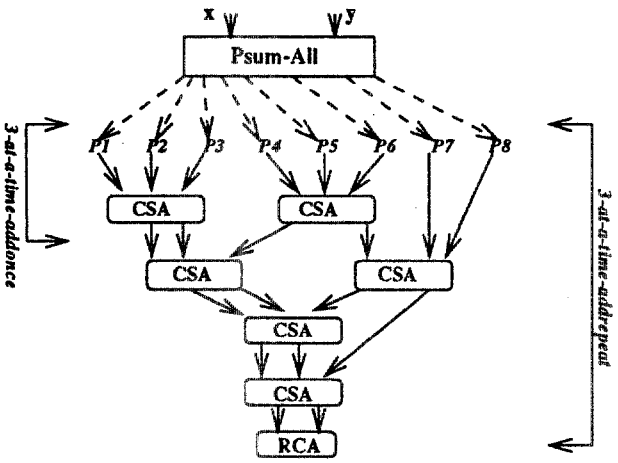


Fig. 1.

The outputs correspond to the bitwise sum and the bitwise carry of the inputs.[2] The parallel addition of three bit vectors at a time is repeated on the outputs of the carry save adder until we have only two bit vectors left. The final product is obtained by using a ripple carry (RCA in Fig. 1.) or a carry lookahead adder.

Specifying Partial Sums Computation in RRL: A bit vector is modeled in *RRL* as a list of bits (with 0 denoting bit zero and 1 denoting bit one) with *nl* and *cons*. A list of bit vectors is modeled as a list of lists with *lnl* denoting the empty list of lists and *consl* that adds a list to a list of lists.

Contrary to the usual convention, we assume that the bits increase in order from left to right i.e., the bit vector 01 stands for $0 * 2^0 + 1 * 2^1 = 2$.

The partial sum, *psum* corresponding to a single bit `x1` of `x` is the same as `y` if `x1` is 1; otherwise, it is the zero bit vector of the same length as `y`[3]:

```
psum(x1, y) := cond(x1 = 0, mkzero(y), y),
```

where `mkzero` generates a zero bit vector of the same length as its input.

The list of partial sums corresponding to all the bits of `x` is computed by applying the function `psum` pointwise to each bit of `x` and shifting `y` to the right by appending a *trailing* zero.

```
psum-all(nl, y) := lnl
psum-all(cons(x1, x), y)  := consl(psum(x1, y), psum-all(x, cons(0, y)))
```

Specifying Partial Sums Addition in RRL: In a Wallace tree circuit, each level in the tree in Fig. 1. contains a list of bit vectors that have to be added to produce the final result. The root contains the list of partial sums corresponding to each bit of the bit vector `x`. The successive levels of the tree are repeatedly constructed until there are less than three bit vectors at any given level(equations

[2] Further details on the specification of the carry save adder are given in section 5.

[3] We follow the convention of typesetting RRL specifications and italicizing other logical formulas.

1, 2 and 3 below). In the case of two bit vectors, addition using a ripple carry adder, `rca`, is performed(equation 3 below).

The Wallace tree multiplier is specified by `3-mult` below. The trace of a computation of `3-mult` on input vectors of a specific length corresponds to a specific circuit.

```
3-mult(cons(x1, nl), y)           := psum(x1, y)
3-mult(cons(x1, cons(x2, nl)), y) :=
  rca(0, pad(1, psum(x1, y)), psum(x2, cons(0, y)))
3-mult(cons(x1, cons(x2, cons(x3, x))), y)  :=
  3-repeat(psum-all(cons(x1, cons(x2, cons(x3, x))), y)).
```

The function `3-repeat` repeatedly takes 3 bit vectors and add them; it is specified:

```
3-repeat(lnl)                              := nl
3-repeat(consl(x1,lnl))                    := x1
3-repeat(consl(x1,consl(x2, lnl)))         := rca(0, pad(1, x1), x2) if
                                              (len(pad(1, x1)) = len(x2))
3-repeat(consl(x1, consl(x2, consl(x3, x)))) :=
  3-repeat(3-once(consl(x1, consl(x2, consl(x3, x)))),
```

where `len` denotes the length of a list. The function `pad(m, x)` produces a bit vector by appending `m` *leading* zeroes to the bit vector `x`. Bit vectors are typically padded by leading zeroes in these specifications so that the input bit vectors to the adders are of equal length. The last equation (equation 4) computes the bit vectors at the successive level by the function `3-once`.

The function `3-once` is defined on a list of bit vectors. If the input list contains less than three bit vectors(equations 1, 2 and 3 below), then the bit vectors in the input list are carried over to the output list. Otherwise, the bit vectors in the input list by considering bit vectors in groups of three and adding such groups in parallel using a carry save adder, `csa`, (equation 4 below). The outputs of the `csa`'s and the bit vectors in the input list that were not considered for addition together, constitute the bit vectors of the output list.

```
3-once(lnl)                                := lnl
3-once(consl(x1, lnl))                     := consl(x1, lnl)
3-once(consl(x1, consl(x2, lnl)))          := consl(x1, consl(x2, lnl))
3-once(consl(x1, consl(x2, consl(x3, x)))) := consl(fst(z1), consl(snd(z1)
                                              3-once(x))) if
  (z1 = carrysave-adder(pad(2, x1), pad(1, x2), x3)) and
  (len(pad(2, x1)) = len(x3)) and (len(pad(1, x2)) = len(x3))
```

3.1 A common top level specification for multipliers in RRL

Circuits that perform multiplication of two n bit numbers by first computing the partial sums and then adding these partial sums constitute a rich family of multipliers based on the number of partial sums that they consider for addition at a particular time. Any multiplier of this family can be specified in *RRL* using the same top level specification as that of the Wallace tree multiplier.

Consider a multiplier circuit defined by the function `k-mult` in which k ($k \geq 1$) partial sums are added together at any time. The multiplier can be abstracted in terms of two hardware components that are cascaded together. The first of these

components performs *partial sum computation* with two bit vectors as its inputs and produces a list of bit vectors as its output. The second of these components performs *partial sum addition* with a list of bit vectors as its input and produces a bit vector as its output.[4]

The partial sum computation component of the multiplier is specified by the functions `psum` and `psum-all`. The partial sum addition component is specified in terms of: `k-repeat` which adds k partial sums repeatedly, and `k-once` for adding partial sums at one level until there are fewer than k partial sums left.

The function `k-once` is defined in the same way as `3-once`. The function leaves the input list of bit vectors invariant if the list contains less than k (more precisely, maximum of $k-1$ and 1) bit vectors. Otherwise, a suitable adder is used to add the bit vectors in the list k at a time with the outputs of the adder constituting the bit vectors to be added in the next round. The definitions of the functions `k-mult` and `k-repeat` can be generalized from the definitions of `3-mult` and `3-repeat` respectively in a similar fashion. [5]

4 Establishing the correctness of multipliers in RRL

In this section we discuss how the behavioral correctness of multiplier circuits can be automatically established using *RRL*. *RRL* is a theorem prover based on rewriting techniques and induction. The main inference steps used in *RRL* are (i) contextual simplification using rewrite rules, (ii) case analysis, (iii) decision procedures for data types with free constructors, propositional calculus and quantifier-free Presburger arithmetic for reasoning about numbers, and (iv) proofs by well-founded induction. *RRL* implements many heuristics to select the order of application of these inferences. For more details on *RRL* the reader is referred to [15].

Consider a multiplier specified by `k-mult` that performs multiplication of its two input bit vectors x and y by considering k, $k \geq 1$, partial sums for addition at a time. To establish the correctness of this circuit with respect to multiplication over numbers, conversion functions from bit vectors and list of bit vectors to numbers are needed. The function `bton` converts a bit vector to the number it represents (recall that the first bit is the least significant bit).

```
bton(nl) := 0,
bton(cons(x1, x)) := cond(x1 = 0, 2 * bton(x), 1 + (2 * bton(x)).
```

Given a list of bit vectors as input, the function `btonlist` below defines a linear addition of numbers corresponding to each of the bit vectors.

```
btonlist(lnl)  :=  0,      btonlist(consl(x, y)) := bton(x) + btonlist(y).
```

[4] k itself can be treated as a parameter while adding the partial sums. Such a specification and the correctness proof can be found in *ftp.cs.albany.edu/pub/subu/Multipiers*. The specification uses generic adders (discussed in section 5). Instantiating such adders requires discharging assumptions on lengthes of the lists of bit vectors in terms of k and would be discussed in the expanded version of this paper.

[5] The complete specifications of the linear array, the Wallace tree and the 7-3 multiplier as done in *RRL* along with the *RRL* transcripts of their correctness proof are also available by anonymous ftp.

The correctness of a multiplier `k-mult` is stated in *RRL* as:

```
Kmult-thm: bton(k-mult(x, y)) == bton(x) * bton(y) if (len(x) = len(y)).
```

The basic strategy employed for proving the above theorem is simple. It involves characterizing the input-output behavior of the partial sum computation component and the partial sum addition component of the multiplier with respect to numbers, and then showing that cascading these two components leads to the desired overall behavior. It is shown that *i)* multiplying the numbers corresponding to the input bit vectors of the partial sum computation component is the same as number obtained by the linear addition of the list of partial sums output by this component. *ii)* And, the number corresponding to the bit vector output by the partial sum addition component is the same as the number corresponding to the linear addition of the list of partial sums input.

The same strategy can be used to prove the correctness of the correctness of any multiplier in the family of multipliers (for any fixed k). Linear addition of partial sums serves as a common denomination for any k and the addition of k partial sums together can always be reduced to linear addition.

Speculating the Intermediate Lemmas

Intermediate lemmas capturing the behavior of each of the component circuits are first established. For instance, lemma `L1` below states that the ripple carry adder correctly implements addition over numbers (needed for the final stage).

```
L1: bton(rca(0, y, z))  == bton(y) + bton(z) if (len(y) = len(z)).
```

There is a similar lemma for carry-save adders (lemma `L4` in section 5). Lemma `L2` captures the correctness of the behavior of the partial sum addition component; it states the number corresponding to the output bit vector is precisely the one obtained by adding numbers corresponding to the list of input bit vectors. Finally, `L3` relates the number corresponding to the list of bit vectors output by the partial sum computation component to the product of the numbers corresponding to its two input bit vectors.

```
L2:  bton(3-repeat(x)) == btonlist(x).
L3: btonlist(psum-all(x, y)) == bton(x) * bton(y)
```

Each of these lemmas can be verified completely automatically in *RRL* by the cover set induction method [15] and the associated heuristics.

We believe that each of the above intermediate lemmas can be speculated from the structure of the multiplier circuit. Lemmas relate the input-output behavior of components of a multiplier circuit with respect to numbers. For each component in the circuit, the number corresponding to its output bit vector (or a list of vectors) is related to the numbers corresponding to its input bit vectors. This important issue of generating intermediate lemmas from the circuit structure needs further investigation; the approach based on generating lemmas from the component behavior seems to be very promising.

For instance, the Wallace tree multiplier can be viewed as a linear composition of the ripple carry adder(`rca`), the partial sum addition (`3-repeat`), and the partial sum computation (`psum-all`) components. The main theorem `3mult-thm` can be expressed as:

```
bton(x) * bton(y)  = bton(rca(0, 3-repeat(psum-all(x, y)))),
```

by identifying the list of bit vectors output by `3-repeat` with the two input bit vectors of `rca`. The number theoretic correctness of the circuit `3mult-thm` can be reduced to the number theoretic correctness of each of these components relating their corresponding inputs and outputs. These correspond to the intermediate lemmas `L1-L3`. Lemma `small L4` can be speculated from the use of `3-once` in the iterative component `3-repeat`.

Establishing the correctness of the Wallace tree multiplier in RRL:

Below, we briefly review the proof of Wallace tree multiplier as obtained in *RRL* using the above-discussed strategy using lemmas `L1`, `L2`, `L3`, `L4`. Other proofs are similar.[6]

The correctness of the Wallace tree multiplier is stated in *RRL* as:

```
3mult-thm: bton(3-mult(x, y)) == bton(x) * bton(y) if (len(x) = len(y)).
```

The above theorem was proved in *RRL* by induction. Induction scheme based on the definition of the function `3-mult` is automatically chosen by the heuristics implemented in *RRL* without any user guidance. Here is the *RRL* transcript.

```
Let P(x): bton(3-mult(x, y)) == (bton(x) * bton(y)) if (len(x)  = len(y))
Induction will be done on x in 3-mult(x, y), with the scheme:
[1] P(cons(x1, nl))                              [2] P(cons(x1, cons(x2, nl)))
[3] P(cons(x1, cons(x2, cons(x3,  x))))
```

The subgoal corresponding to `[1]` is easily established by case analyses based on the definition of `psum`, using the definitions of `3-mult` and `bton` for simplification. The case analyses is automatically recognized by *RRL* based on the definition of `psum` given in terms of the ternary predicate *cond*.

The subgoal `[2]` follows from lemma `L1` (ensuring that the ripple carry adder correctly implements addition over numbers). The proof of the subgoal `[3]` is also direct from lemmas `L2`, `L3`, thus completing the proof of `3mult-thm` by induction.

The correctness of any other multiplier in the family of multipliers such as the linear array or the 7-3 multiplier can be performed in *RRL* using the same top level proof as that of the Wallace tree multiplier given above. For instance, the correctness of a linear array multiplier is proved in *RRL* using three lemmas which are exactly the same as `L1 - L3` with the lemma `L2` defined in terms of functions `1-repeat` instead of the function `3-repeat`. The correctness proof of the 7-3 multiplier also follows from the lemmas `L1 - L3` with the lemma `L2` defined in terms of the functions `7-repeat` instead of `3-repeat`.

5 The use of Generic Hardware Components

While proving the correctness of different multipliers , the specifications and the associated correctness proofs of the adders are duplicated. Such duplication can be avoided by noting that specifications of the input-output behavior of the adders is sufficient to reason about different multipliers; other details of adders

[6] Detailed proof transcripts are available via anonymous ftp from *ftp.cs.albany.edu: pub/subu/Multipliers.*

are irrelevant. So adder circuits are abstracted by generic hardware components with behavioral constraints. The correctness proof of multipliers is first performed in terms of these generic components. The generic components are then realized by specific adders that satisfy the associated behavioral constraints.

To specify and reason over generic hardware components, *RRL* has been extended along the lines of [2] to allow function instantiations and for handling theories. The behavioral constraints associated with a generic component are specified in *RRL* as equations(possibly conditional) using ? = to indicate that the equation is a behavioral constraint. For instance, a carry save adder can be specified in *RRL* in terms of the generic component g32-adder as:

```
bton(fst(g32-adder(x, y, z))) + bton(snd(g32-adder(x, y, z)))  ?=
bton(x) + bton(y) + bton(z) if (len(x) = len(y) = len(z)).
```

The behavioral constraints introduced on these generic components are oriented into rewrite rules by *RRL* and are subsequently available for simplification.

5.1 Realizing the generic components : Carry Save Adder

To complete correctness proofs of different multipliers, the generic components used are realized by specific adders that satisfy the associated behavioral constraints. In this section we use the correctness proof of a carry save adder as an example to realize the generic component g32-adder. The other generic components used in the proofs of the multiplier circuits have been realized similarly using *RRL*. For details refer to [12].

A carry save adder has three bit vectors of equal length as its inputs and outputs two bit vectors corresponding to the bitwise parity and the bitwise sum of its inputs. It is specified in *RRL* as:

```
csa(x, y, z) :=  pairl(paritylst(x, y, z), cons(0, majoritylst(x, y, z)))
                   if (len(x) = len(y) and (len(y) = len(z)),
```

where pairl given two bit vectors constructs a pair of bit vectors. The function paritylst, computes the bitwise parity of its three inputs, and the function majoritylst computes their bitwise majority. These functions can be easily defined by invoking parity and majority functions on bits.

The correctness of the carry save adder can be stated as:

```
L4: bton(x) + bton(y) + bton(z) == bton(paritylst(x, y, z)) + bton(cons(0,
    majoritylst(x, y, z))) if (len(x) = len(y)) and (len(y) = len(z)).
```

The above formula is proved directly in *RRL* by induction using the scheme based on the definition of paritylst.

The component g32-adder can be realized by the carrysave adder, csa in *RRL* using the *instantiate* directive with a set of *function replacements* such as ((g32-adder csa),...). Based on these function replacements the behavioral constraints are suitably instantiated by *RRL* and the instantiated formula is treated as a proof obligation which must be discharged from the properties of the realization.

6 Conclusion

A number of well-known multiplier circuits such as the linear array, the Wallace tree and the 7-3 multiplier employed in *IBM RS/6000* have been verified using the automated theorem prover *RRL*. It has been shown that by abstracting the commonality in behavior, a family of multiplier circuits can be specified using a common top level specification. Such a specification was used to illustrate a common top level correctness proof for the family of multiplier circuits. The basic strategy employed in performing these correctness proofs is simple, and it leads to concise proofs with a handful of meaningful lemmas that are independent of the underlying prover. It should be possible to duplicate these proofs using other provers which support capabilities similar to those implemented in *RRL*.

Circuit	*Comm. Defs*	*Comm. Lemm.*	*Spec. Defs*	*Spec. Lemm.*	*Time*
Linear Array	12	5	2	0	2.48
Wallace Tree			2	0	2.45
7-3			2	0	6.22

The intermediate lemmas used in these proofs correspond to the input-output behavior of the various components of the multiplier circuit. Speculation of such lemmas can be done by the user in a routine manner. The use of generic components to segregate the specification and implementation aspects was advocated. The use of such generic components lead to concise proofs and help reuse of proofs. It was also demonstrated that generic components lead to modular proof development in a hierarchical fashion analogous to the design process.

The specification and the correctness proofs of the Wallace tree multiplier were attempted first in *RRL* and it took less than a week. This time is inclusive of our attempts to familiarize ourselves with the multiplier itself. The subsequent multipliers were formalized and their correctness proof was proved in a couple of days. The statistics for the various correctness proofs obtained using *RRL* are given in the table. *RRL* is implemented in Common Lisp and the timings are on a Sun Sparc 5 station(64Mb memory). The proofs of the linear array and the Wallace tree multiplier can be performed in *RRL* within 5 secs. The time required for the 7-3 multiplier is larger due to extensive contextual rewriting required for establishing the appropriateness of the lengthes of seven bit vectors. There are no specific intermediate lemmas needed in the proofs. For each multiplier circuit, only two definitions specific to the circuit are needed.

The results of our initial experiments, in terms of adder circuits [12] and multiplier circuits performed in *RRL*, are encouraging, and they lead us to believe that *RRL* is well-suited for reasoning about the properties of hardware descriptions using recursive equations that can be oriented into rewrite rules. Particularly, *RRL* can be used for verifying properties of parameterized circuits, which cannot be handled by state based approaches, as well as for structuring proofs of larger circuits in terms of proofs of their component circuits. Further, circuit properties are verified in terms of the arithmetic functions they compute in contrast to boolean functions. The major stumbling block in the use of theorem provers is perhaps the need for intermediate lemmas. As shown for adder

and multiplier circuits, these lemmas correspond to capturing the arithmetic function of the component circuits; generation of such lemmas, we speculate, can be automated.

References

1. R.S. Boyer and J. Moore, *A Computational Logic Handbook.* New York: Academic Press, 1988.
2. R.S. Boyer, J. Moore and M. Kaufmann "Functional Instantiation in Nqthm", CLI Inc. Tech. Report.
3. Bryant R.E., "Graph-based Algorithms for boolean function manipulation", *IEEE trans. on Computers*, C-35(8), 1986.
4. R. E. Bryant, and Y.-A. Chen, "Verification of Arithmetic Functions with Binary Moment Diagrams", Tech. Rep. CMU-CS-94-160, June 1994.
5. J. R. Burch, E.M. Clarke, K. L. Mcmillan and D.L. Dill, "Sequential Circuit Verification using symbolic model checking", in proceedings of *Twenty seventh ACM/IEEE Design Automation Conference*, 1990.
6. A.J. Camilleri, M.J.C. Gordon and T.F.Melham, "Hardware verification using higher-order logic"", *HDL Descriptions to Guaranteed Correct Circuit Designs*, D. Borrione (editor) pp. 43-67, N.Holland, Amsterdam 1987.
7. L. Dadda "Some Schemes for parallel multipliers," *in Computer Arithmetic Vol. I*, E.E. Swartzlander Jr. (editor), IEEE Computer Society Press, 1990.
8. W.A. Hunt., "FM8501: A verified Microprocessor", Ph.D thesis, UT Austin, 1985.
9. R.K.Montoye, E. Hokenek and S.L.Runyon, "Design of the IBM RISC System/6000 floating-point execution unit," *IBM Journal*, Vol. 34, No. 1, 1990.
10. "Theorem Provers in circuit design",*IFIP Transactions*, V. Stavridou, T.F. Melham, R.T.Boute (eds.) N.Holland 1992.
11. J. Joyce, G. Birtwistle and M. Gordon, "Proving a computer correct in HOL", Tech. Report 100, Computer Lab. University of Cambridge 1986.
12. D. Kapur and M. Subramaniam, "Mechanical Verification of Adder Circuits Using Powerlists," CS.Tech. Report, Dept. of CS Suny Albany, November 1995.
13. R.P. Kurhshan, L. Lamport, "Verification of a Multiplier: 64 Bits and Beyond," *Fifth Intl. Conf. on CAV*, C. Courcoubetis (editor), LNCS 697, July 1993.
14. L. Pierre, "VHDL Description and Formal Verification of Systolic Multipliers," *in Proc. of CHDL*, D. Agnew and L. Claesen (eds.) N. Holland 1993.
15. D. Kapur, and H. Zhang, "An overview of Rewrite Rule Laboratory (RRL)," *J. of Computer and Mathematics with Applications*, 29, 2, 1995, 91-114.
16. D. Cyrluk and S. Rajan and N. Shankar and M. K. Srivas, "Effective Theorem Proving for Hardware Verification", Proc. 2^{nd} *conference on theorem provers in circuit design*, R. Kumar and T. Kropf (eds.), Sept. 1994.
17. M. Srivas and M. Bickford, "Formal Verification of a pipelined microprocessor.", *IEEE Software*, Sept. 1990.
18. Shui-Kai Chin, "Verified Functions for Generating Signed-Binary Arithmetic Hardware", *IEEE trans. on Computer Aided Design*, Vol. 11, No. 12, Dec. 1992.
19. D. Verkest, L. Claesen, and H. De Man, "Correctness Proofs of Parameterized Hardware Modules in the Cathedral-II Synthesis Environment", *EDAC'90*, Glasgow, Scotland, March 1990.
20. C.S. Wallace, "A Suggestion for a fast multiplier," *in IEEE Trans. Electron. Comput.*, EC-13:14-17, 1964.

Verifying Systems with Replicated Components in Murφ *

C. Norris Ip and David L. Dill

Computer Systems Laboratory, Stanford University
Email: {ip,dill}@cs.stanford.edu

Abstract. We present an extension to the Murφ verifier to verify systems with *replicated identical components*. Verification is by explicit state enumeration in an abstract state space where states do not record the exact numbers of components. Through a new datatype, called *RepetitiveID*, the user can suggest the use of such an abstraction to verify a system of fixed size. Murφ automatically checks the soundness of the abstract state graph, and automatically constructs the abstract state graph using the system description.
Using a simple run time check, Murφ can also determine if it can generalize the verification result of a system with fixed size to systems of larger sizes, including the system with infinite number of components.

1 Introduction

Finite-state systems such as cache coherence protocols, communication protocols or hardware controllers are often designed to be scalable, so that a description gives a family of different systems, each member of which has a different number of replicated identical components. It is therefore desirable to be able to verify the entire family of systems, independent of the exact number of replicated components.

The general problem of verifying systems with replicated components is known to be undecidable [AK86, GS92]. A number of approaches has been proposed for verifying particular classes of problems. Some of them use induction over the replicated components and require an invariant process or a network invariant [KMOS94, CG87, CGJ95, WL89]. Coming up with a proper invariant is not easy, and automatic generation of network invariants for certain classes of systems are restricted and expensive [RS93, BSV94, SG87, GS92].

There are also approaches that do not use induction. Shibata et al. [SHTO93] presented an algorithm to verify a simple telecommunication system with limited interaction between the processes. However, the class of problems they can verify is severely restricted. On the other hand, Graf [Gra94] has a more general method based on abstractions, which has been applied to a distributed cache memory, but it requires substantial manual effort to complete the proof.

The work described here is closely related to the methods by Lubachevsky [Lub84], Dijkstra [Dij85], and Pong et al. [PD95, PNAD95]. Lubachevsky verified a concurrent program by collapsing all reachable states into a fixed number of "metastates", in which the number of processes is represented by N with an unspecified value. Dijkstra

* This research was supported by Semiconductor Research Corporation under contract 95-DJ-389 and by the Advanced Research Projects Agency through NASA grant NAG-2-891.

used regular expressions to represent classes of similar states. Pong et al. used a set of repetition constructors to abstract away the exact number of components, for the verification of cache coherence protocols.

In this exposition, we consider systems with a collection of components, including fixed components and components that can be replicated from 1 to n times. Many of these systems, such as cache coherence protocols and mutual exclusion algorithms, can be proved correct without modelling the precise number of replicated components. For example, suppose a multiprocessor has identical caches numbered 1 to 8, and that a particular memory value is *invalid* in caches 1,2,3,5,6,7 and and *writable* in cache 4. The abstract state may record that more than zero caches are *invalid*, exactly one is *writable*, "forgetting" not only the number of processors in each state but also the ordering of the processors. Formally, the abstract state includes a mapping from component states to *repetition constructors* $\{0, 1, +\}$, representing zero, exactly one, or more than zero (respectively) components in that component state. This abstraction is an approximation of the original state graph, which can be used for verification of invariants and $\forall$CTL model checking. The approximation is conservative: it never fails to report an error, but may report an error when none exists.

This approach has been used for the verification of several applications, but most of the existing work requires a lot of expertise from the user. For example, Pong and Dubois' method requires the user to write an executable description of the abstract behavior. This description is different from the concrete description used for specification or synthesis, so their method requires more work, and raises the question of whether the concrete and abstract descriptions are consistent.

To reduce the amount of user effort, we incorporate this abstraction into our verification system, Murφ. We provide an extension to the existing high-level language in which the user can easily specify a protocol in its concrete form. The extension is a new datatype, called *RepetitiveID*, which can be used to represent the indices of the replicated components. The Murφ compiler can automatically detect whether the datatype is used in a way that admits the use of repetition constructors in verification. If so, it automatically verify the system using the abstract state space, instead of the concrete state space.

Furthermore, we also extend previous work so that the abstraction can be used to verify systems of fixed sizes, even when verification for unbounded sizes is infeasible, resulting substantial reductions in the state explosion. Through a simple run-time check, Murφ can determine automatically if it can generalize the verification result to systems of larger sizes.

A key problem in verifying in an abstract state space is how to generate the successors of an abstract state. We solve this problem by selecting up to two concrete states represented by an abstract state, and constructing the abstract successors from the concrete successors of these concrete states. Heurstics for an efficient construction of the abstract state space are also presented.

2 Modifications to Murφ

2.1 The Murφ Verification System

The basic Murφ Verification System [DDHY92] consists of the Murφ compiler and the Murφ description language. The Murφ description language is a high-level programming language for the description of finite-state asynchronous concurrent systems. The Murφ

compiler generates a C++ program for a Murφ program, which exhaustively generates the reachable states, checking for error conditions and deadlocks.

A Murφ program consists of four parts: declarations, transition rules, start state generation rules, and invariant descriptions. Examples of Murφ programs are shown in Figs. 1 and 2.

```
Type    – user defined type
    pid:        1..numProcessor;            – processor indices
    mid:        1..numMemory;               – memory modules-indices
    address:    1..numAddress;              – address space
    value:      1..valueCount;              – possible values in a memory word
    ...
Var     – state variables
    P:          Array [pid] of              – an array of records storing the status
                  Record                    – of each processor.
                    State: enum {Invalid, Shared, Master};
                    Value: value;
                    Outstanding_Request: Boolean;
                  End;
    M:          Array [mid] of MemoryState; – an array of memory modules.
    Net:        Multiset [ NetSize ] of Messages; – a multiset storing messages in the network.
```

Fig. 1. An Example of state variable declarations in Murφ

```
Ruleset n : pid Do              – parameterized rules for each processor
Ruleset h : mid Do              – parameterized rules for each memory module
Ruleset a : address Do          – parameterized rules for each address in a memory module
    Rule "Line Eviction"
        P[n].Cache[h][a].State = Shared          – if it is an shared copy
    ⟹
    Begin
        P[n].Cache[h][a].State := Invalid;       – the cache line is invalidated
        P[n].Cache[h][a].Value := Undefined;     – the data cannot be used
    End;
Endruleset;
Endruleset;
Endruleset;
```

Fig. 2. An Example of transition rules in Murφ

A system state is specified by the values of the global variables. The rules are conditional actions (guarded commands). As a Murφ description executes, a rule is chosen nondeterministically and executed, generating a new system state (since it assigns new values to the variables). Although a rule may consist of arbitrarily complex operations, it is executed *atomically*, without interference from other rules in the description. Hence, the use of Murφ leads to an asynchronous, interleaving model of concurrency in which different parts of the system interact via shared variables.

The types of variables are mostly conventional finite datatypes found in high level languages: arrays, records, integer subranges, Booleans and enumerations. Unlike conventional languages, there is also a special "undefined" value for each datatype. There are some non-traditional types as well, such as *scalarsets* [ID93a] and *multisets*. The scalarset type is a finite set of values, similar to a subrange except that their use is

restricted so that members of the scalarset can be interchanged in any state without affecting the future behavior. For example, in a multiprocessor, processors are symmetrical and we can model their indices as a scalarset. Multisets are also called "bags", of values of some other type. Multisets of messages are useful for modelling networks that do not preserve message order. A multiset is essentially an array indexed by an anonymous scalarset type. A **Choose** construct can be used to nondeterministically select an element from a multiset and bind a parameter to a reference to the selected element.

The rules contain control constructions, including sequences of statements, **if** and **for** statements. A set of rules that vary over a parameter can be abbreviated using the **RuleSet** construct, which has a parameter name, an index type, and a body containing one or more rules which refer to the parameter as a variable. The start state descriptions are special rules which initialize the state variables. The invariant descriptions are Boolean predicates; a conjunction of a set of Boolean predicates over a parameter can be abbreviated using the **forall** construct.

A Murφ program implicitly represents a state graph, defined as a set of states Q, a set of start states $Q_0 \subseteq Q$, a special error state **error** $\in Q$ which represents the occurrence of a run-time error in the system being verified, and a next-state relation $\Delta \subseteq Q \times Q$. A convenient representation for Δ is a set of *transition functions* $t_i : Q \rightarrow Q$. We can define Δ in terms of t_i: $\Delta(q, q')$ if there is some transition function t_i such that $q' = t_i(q)$.

In the state graph represented by a Murφ program, the set of states Q is the set of all legal assignments of values to the declared state variables plus a special state for **error**. The rules define transition functions by reading and writing the state variables. The error state is generated if there is a run-time error, such as referencing an undefined value, or if the resulting state from a transition function violates an invariant. The start states are generated by applying the transition functions of the start rules to the assignment that gives an undefined value to every state variable. A ruleset generates a set of transition functions, one for each parameter value of the ruleset index type. Nested rulesets generate transition functions by substituting all combinations of parameters.

When verifying by explicit state enumerations, it is important that errors be reported as quickly as possible, without unnecessarily generating states. This can be achieved using "on-the-fly" (online) algorithms, in which states are checked for error as they are generated. The basic algorithm for error and invariant checking is shown in Fig. 3.

```
SimpleAlgorithm()
    Reached = Unexpanded = Q0
    While Unexpanded ≠ φ Do
        Remove a state s from Unexpanded
        For each successor s' of s Do
            Examine(s')

Examine(state s)
    If s = error Then Report Error
    If s is not in Reached Then
        Put s in Reached
        Put s in Unexpanded
```

Fig. 3. A Simple On-the-fly Algorithm for Error and Invariant Checking

To describe a system of several components, such as a set of processes, the user must define state variables for the component state, and define a set of rules for the behavior of the component. The behavior of concurrent components is modelled by forming the union of the state variables and rules of the individual components. One of the components "takes a step" when its rule is chosen and executed.

Replicated identical components can be modelled in Murφ by defining a con-

stant for the number of components (say `CompCount`), and defining a subrange `CompID: 1 .. CompCount` for the indices of components. The local states of the components are stored in an array indexed by `CompID`. The rules describing the components are enclosed in a ruleset with a `CompID` parameter which represents the component to which the rule belongs. Using this convention, Murφ descriptions become *scalable*, meaning that the number of replicated components can be changed simply by modifying `CompCount`.

2.2 The RepetitiveID Type

We add a new datatype to Murφ, called *RepetitiveID*. The RepetitiveID is a restricted subclass of a conventional subrange (in fact, it is a scalarset), which should be used for the indices of replicated identical components, such as processors in a multiprocessor. For example, we can change the subrange `1..numProcessor` to `RepetitiveID(numProcessor)` to hint to the verifier to verify the system in the smaller abstract state space. Murφ automatically checks that certain restrictions are satisfied so that the verification is sound. Since a member i of the RepetitiveID type is used as the name of a component in the description, it is natural to identify i and the component, and refer to "component i" below.

A value of RepetitiveID type can be assigned to variables, tested for equality with other values, used as an array index, or bound in a **RuleSet** or **for** loop. There are six restrictions on the use of RepetitiveID. In spite of the restrictions, RepetitiveID can be used to model a wide range of systems, including bus-based multiprocessor cache coherence protocols [PD95], network-based cache coherence protocols with a central or distributed directory [PNAD95, DDHY92, LLG+90].

Intuitively, our goal is to isolate the parts of the state corresponding to the replicated components into a single array indexed by the RepetitiveID type. If two components i and j have identical component states, we would like the transition rule to produce a successor state where i and j have identical component states, also. There is one exception: we allow the rule to have one "special" component, whose component state is treated differently from other component states, even if they are otherwise identical. An example of where this is useful is mutual exclusion: many components may be in identical states, waiting for a resource, but only one (the special one) will obtain it.

The first two restrictions make it possible to separate the component states from other parts of the state.

1. *The Murφ program has at most one RepetitiveID type.*
2. *The elements of a symmetric array cannot contain another array with RepetitiveID index type.* A symmetric array is an array indexed by a scalarset or RepetitiveID type, or a multiset.

We illustrate the subsequent definitions with an example. Consider a central-directory-based cache coherence protocol whose state includes: an array of local processor control states; a multiset of messages representing a communications network; a memory where each memory line has an *owner* field pointing to a processor that has a writable copy of the line, along with the data in the memory line. The messages in the network have *to* and *from* fields, which can be processor indices or a value representing the memory itself.

With these restrictions, the following definition characterizes the parts of the state that "belong" to a replicated component.

Definition 1. *The* component state *of the component i includes all the state variables satisfying:*

- *for every array A indexed by the components, the element $A[i]$. (In the example, the local control state of processor i becomes part of the component state for i.)*
- *for every multiset M that is not assigned to a component state by the previous case, the elements of M containing i and no other components. (In the example, the messages between the memory and processor i become part of the component state for i.)*

In our example, the memory value (which does not contain a component index), the *owner* field (which is not in a multiset), and messages from one processor to another (which contain *two* component indices) are not included in any component states.

Definition 2. *A component, i, is* abstractable *if it contains all instances of i occuring in the global state, and contains no other component indices.*

In our example, a processor i would not be abstractable if the *owner* field had the value i, or if there were a message between i and another processor j.

In the rest of the paper, we regard the state as being a pair $(s, [r_1, \ldots, r_k])$: $[r_1, \ldots, r_k]$ is an array of component states indexed by the abstractable components, and s an assignment to the rest of the state variables. The component states for components i and j are considered to be the same if the only difference between corresponding variables is that variables in component i have the value i and variables in component j have the value j.

There are four restrictions on the use of RepetitiveIDs in Murφ to ensure the soundness of this verification method. Although we have made these as simple as we can, some are still quite technical.

3. *No "symmetry-breaking" operations [ID93a].* There are no literal constants in the type; arithmetic operations are not allowed; comparisons such as $<$ are not allowed.
4. *Bindings of the RepetitiveID type in* **RuleSet** *constructs may not be nested.*
5. *Bindings of the RepetitiveID type in* **for** *statements and* **forall** *expressions may not be nested. The variables written by each iteration of a* **for** *statement on the RepetitiveID must be disjoint.* In particular, a **forall** expression on the RepetitiveID is not allowed in the body of a **for** statement on the RepetitiveID.
6. *If a variable in the state has the RepetitiveID type, and its value is i for some abstractable components, the variable may not be used to index an array with RepetitiveID index type.*

Intuitively, the symmetry restriction makes sure that the components can be reordered arbitrarily without changing the behavior of the systems. The remaining restrictions ensure that transition rules treat identical component states identically, except for at most one special component.

3 Verification Using Repetition Constructors

3.1 Abstract States Using Repetition Constructors

Once we have isolated the abstractable components as in the previous section, it is possible to abstract away from the exact numbers of each component state by using the repetition constructors 0, 1, and +.

Definition 3 Abstract State. *An* abstract state *is similar to a concrete state except that the array of abstractable components is replaced with a mapping of each possible component state to one of the repetition constructors 0, 1, or +.*

Abstract states are written in the form $(s, \{q_1^{e_1}, \ldots, q_k^{e_k}\})$, where each e_i is 1 or + (when the constructor is 0, the component state is omitted).

A concrete state $a = (s, [r_1, ..., r_z])$ is represented by an abstract state $A = (s, \{q_1^{e_1}, ..., q_k^{e_k}\})$ if the following conditions are satisfied:

- $e_i = +$ if q_i occurs in $[r_1, \ldots, r_z]$ two or more times;
- $e_i = 1$ or $e_i = +$ if q_i occurs in $[r_1, \ldots, r_z]$ exactly once;
- a component state does not appear in $[q_1, \ldots, q_k]$ if it does not appear in $[r_1, \ldots, r_z]$.

The abstract states are partially ordered: $(s, \{q_1^{e_1}, ..., q_k^{e_k}\}) \leq (s, \{q_1^{e'_1}, ..., q_k^{e'_k}\})$ if and only if $e_i = +$ implies $e'_i = +$. In this case, $(s, \{q_1^{e'_1}, ..., q_k^{e'_k}\})$ is said to *cover* $(s, \{q_1^{e_1}, ..., q_k^{e_k}\})$. The notation $a \in A$ is used to indicate that A represents a. The set of abstract states representing a particular concrete state has a unique minimum element in this order; the abstracting function **abs** used in our verifier maps a concrete state to its minimum abstract representative.

In many cases, it is useful to maintain in the abstract state the total number of replicated components, while forgetting exactly how many components are in each component state.

Definition 4 Restricted Abstract State. *A* restricted abstract state *is an abstract state paired with a number representing the total number of replicated components.*

We write $(s, q_1^{e_1}, \ldots, q_k^{e_k})|_n$ to represent the restricted abstract state with n components.

3.2 The Basic On-The-Fly Algorithm

We can construct the abstract state graph for a Murφ program with a RepetitiveID type using an on-the-fly algorithm,

First of all, C++ code for the abstraction function **abs** is generated by the Murφ compiler. The start states of the abstract state graph are generated by using this function to abstract the concrete start states.

Given an abstract state, the verifier needs to generate all its successors in the abstract state graph. Because of the restrictions of the RepetitiveID, the verifier can always find a small number of concrete states that can be used to find the successors to the abstract state. The choice of concrete states depends on the abstract state, and on the nature of the concrete transition functions.

Given $(s, \{q_1^{e_1}, ..., q_k^{e_k}\})$, and a transition t, there are three possible situations:

1. There is no special component for the transition t, or the special component is not abstractable, i.e., it belongs to s.
2. The special component has repetition constructor 1.
3. The special component has repetition constructor +.

For brevity, we discuss only the third, most difficult, case in detail. Suppose i is the special component for t, and i has component state q_i. The restrictions of the RepetitiveID allow a transition to have different effects on the special component and the other components with the same component state. Therefore, we split our abstract state into two concrete states $(s, [q_1, ..., q_{i-1}, q_i, q_{i+1}, ..., q_k])$ and $(s, [q_1, ..., q_i, q_i, ..., q_k])$, as shown in Fig. 4.

If the successor of $(s, [q_1, ..., q_{i-1}, q_i, q_{i+1}, ..., q_k])$ is $(s', [r_1, ..., r_{i-1}, r, r_{i+1}, ..., r_k])$, we can convert it back to an abstract state by restoring the repetition constructors $(e_1, ..., e_{i-1}, 1, e_{i+1}, ..., e_k)$ to get $(s', \{r_1^{e_1}, ..., r_{i-1}^{e_{i-1}}, r^1, r_{i+1}^{e_{i+1}}, ..., r_k^{e_k}\})$. To make sure it is an legal abstract state, we may have to re-partition if the component with state r is no longer abstractable, and to combine any identical component states r_i and r_j. Similarly another abstract successor is generated from $(s, [q_1, ..., q_i, q_i, ..., q_k])$ with repetition constructors $(e_1, ..., 1, +, ..., e_k)$.

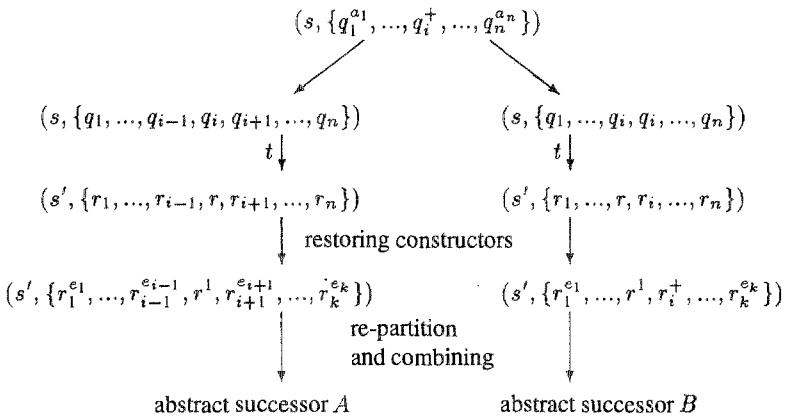


Fig. 4. Transition t With Special Component in State q_i and Constructor +.

Because of the restrictions of the RepetitiveID, the abstract state graph generated this way has the following property:

Property 1 *Given two abstract states A and B, if there exists concrete state $a \in A$ and $b \in B$ such that (a, b) is a transition in the original state graph, (A, B) is a transition in the abstract state graph.*

The abstract start states represent all concrete start states, and the abstracting function **abs** maps **error** to **error**. Therefore, it can easily be proved by induction that for every state reachable from a start state in the concrete state graph, there is an abstract

representative reachable from an abstract start state. This can be easily proved by induction.

It follows from this result and [BBG+93] that if the abstract state graph satisfies a $\forall$CTL formula f, the concrete state graph also satisfies f. Because the abstract state graph is an approximation, it may result in reports of non-existent errors and cannot be used for verification of deadlock-freedom (there is no reachable state with only itself as successor) or $\exists$ CTL model checking.

The algorithm implemented in Murφ uses the restricted abstract states. The Murφ program is required to specify the number of replicated components, n, and the Murφ verifier restricts all abstract states to size n. There are two situations when a restricted abstract state represents no concrete states, and therefore, is discarded. One is when it only represents concrete states with number of replicated components fewer than n, such as in the case of the restricted abstract state $(s, \{q_1^1, q_2^1\})|_3$. The other situation is when it only represents concrete states with number of replicated components more than n, such as in the case of the restricted abstract state $(s, \{q_1^+, q_2^+\})|_1$.

Furthermore, in some cases, the restricted abstract state represents the same set of concrete states as a similar restricted abstract state with the 1 constructor only. Murφ automatically converts such states to a restricted abstract state with the 1 constructor only. For example, the restricted abstract states $(s, [q_1^1, q_2^+])|_2$ and $(s, [q_1^+, q_2^1])|_2$ is the same as $(s, [q_1^1, q_2^1])|_2$, and Murφ automatically converts both of them to $(s, [q_1^1, q_2^1])|_2$.

During the verification process, if the discarded abstract state only represents concrete states with fewer components than n, and no restricted abstract state is converted to one with the 1 constructor only, the abstract state graph obtained is the same as that for systems of larger sizes. We call this state graph the *saturated state graph*. The saturated state graph represents the behavior of all systems with sizes n or larger. The verification result is therefore valid for all systems of size n or larger.

With the restricted abstract state graph, the verifier won't attempt to solve the problem for arbitrary sizes if it is unsolvable or if the abstract state graph for arbitrary system sizes is too large for us to verify.

3.3 The Efficient On-The-Fly Algorithm

The algorithm presented in the previous subsection is very inefficient, because it is wasteful to have two comparable states in the table of previously examined states. For example, $(s, \{q_1^1, q_2^+\})$ is redundant in the set of previously examined states when $(s, \{q_1^+, q_2^+\})$ is also in the set. A more efficient algorithm can be implemented as shown in Fig. 5. Two heuristics are presented in this section to reduce the time for checking whether a state is maximal in the set of previously generated states and to reduce the number of non-maximal states generated.

Checking if a State is Maximal For every abstract state generated, we check whether it is covered by a previously examined state, and remove any previously examined state that is covered by it. Pong et al. use a linear search on all previously examined states to do this.

If an abstract state p covers another abstract state p', p and p' can only differ in the 1 and + constructors on their component states. To store the abstract states, the hash function does not distinguish between 1 and + constructors. The states hashing to the same location are searched linearly to see if any cover or are covered by the current state.

```
EfficientAbstractAlgorithm()
    Reached = Unexpanded = { abs(q)  |  q ∈ Q0 }
    While Unexpanded ≠ φ Do
        Remove an abstract state s from Unexpanded
        For each abstract successor s' of s Do
            Examine(s')

Examine(state s)
    If s = error Then Report Error
    If s is not in Reached and s is not covered by a state in Reached Then
        Remove the states in Reached and Unexpanded that are covered by s
        Put s in Reached and Unexpanded
```

Fig. 5. The Efficient On-the-fly Algorithm Using the Abstract State Space

In practise, the lists are very short. In the industrial cache coherence protocol presented in Section 4, the maximum length is 6. When an abstract state is hashed into a location with a list of constructor arrays, the average number of states compared is fewer than 1.05. This method does not work well for the original scheme of Pong et al., because they have a fourth constructor, *, meaning "zero or more states".

Reducing the Number of Non-Maximal States Although an abstract model has very few states, many non-maximal states are temporarily stored and expanded. We have found that for simple depth first search (DFS) and breadth first search (BFS), more than 75% of the time is spent searching non-maximal states.

Instead of using simple DFS or BFS, we use a best-first strategy, where "best" is defined as the greatest number of + constructors. The abstract state that represents more concrete states are expanded to find its successors first, because its successors are more likely to be maximal states. For the industrial cache coherence protocol presented in Section 4, we are able to reduce the number of non-maximal states examined from 106,528 (3 times more than the number of maximal states) down to 3,527 (fewer than 10% of the maximal states) in a 9-processor system. No extra memory is required to store the non-maximal states, and the verification is more than three times faster.

Pong also has mentioned similar strategy in his thesis [Pon95], however, because of his choice of repetition constructors, it does not help much in reducing the number of non-maximal states.

4 Practical Results

The abstraction with the repetition constructors can be combined easily with the other two reduction strategies implemented in Murφ: symmetry reduction [ID93a, ID93b] and reduction by reversible rules [ID96].

We present in this section the verification results for an industrial cache coherence protocol (ICCP), using the Murφ verification system. This protocol is a typical central-directory-based cache coherence protocol, as described in [DDHY92]. Because of data forwarding, some replicated components in some states are not abstracted by the repetition constructors. However, since the extent of forwarding is limited, we are still able to verify it for arbitrary system sizes, as shown in Table 1.

As we increase the size of the system, the size of the abstract state graph increases accordingly until it becomes saturated. After the verification of a system of 14 processors,

Table 1. Results for the Verification of an Industrial Cache Coherence Protocol

# of processors (ICCP)	3	4	5	6	7	8	9
size (unordered network)	10,077	247,565	–	–	–	–	–
size (sym. only)	1,781	11,814	68,879	358,078	–	–	–
size (sym./rep.)	1,770	11,206	57,790	257,692	–	–	–
size (sym./rev.)	434	1,760	6,021	18,118	49,045	121,302	–
size (sym./rev./rep.)	427	1,590	4,542	10,587	19,485	28,927	35,515
time (unordered network)	5.1s	205s	–	–	–	–	–
time (sym. only)	2.4s	28s	349s	3,762s	–	–	–
time (sym./rep.)	4.4s	49s	497s	4,555s	–	–	–
time (sym./rev.)	2.1s	13s	98s	615s	3,283s	12,801s	–
time (sym./rev./rep.)	3.3s	27s	167s	811s	3,265s	7,593s	17,477s

# of processors (ICCP)	10	11	12	13	14 and up
size (sym./rev./rep.)	38,146	38,485	38,329	38,269	38,269
time (sym./rev./rep.)	29,871s	37,903s	43,352s	48,410s	49,932s

sym. : Symmetry Reduction
rev. : Reversible Rules Reduction
rep. : Repetition Constructor Reduction

Murφ was able to detect automatically that the abstract state graph is the same for systems with 15 processor or more. The saturated model has 38,269 states and is valid for 14 processors or more. This phenomenon is very similar to the data saturation phenomenon reported in [ID93a, ID93b].

Acknowledgement

We would like to thank Fong Pong for the discussion on the symbolic state model, Ganesh Gopalakrishnan, Seungjoon Park, Ulrich Stern, and Han Yang for their valuable feedback during the writing of this paper.

References

[AK86] Krzysztof R. Apt and Dexter C. Kozen. Limits for automatic verification of finite-state concurrent systems. *Information Processing Letters*, 22:307-309, 1986.

[BBG+93] A. Bouajjani, S. Bensalem, S. Graf, C. Loiseaux, and J. Sifakis. Property preserving abstractions for the verification of concurrent systems. *Formal Methods in System Design*, 1993.

[BSV94] F. Balarin and A.L. Sangiovanni-Vincentelli. On the automatic computation of network invariants. *6th Int'l Conf. on Computer-Aided Verification*, June 1994.

[CC92] Patrick Cousot and Radhia Cousot. Abstract interpretation and application to logic programs. Technical report, Ecole Polytechnique, Laboratoire d'Informatique, 1992.

[CG87] E.M. Clarke and O. Grumberg. Avoiding the state explosion problem in temporal logic model checking algorithms. *Proceedings of the 6th Annual ACM Symp. on Priniciple of Distributed Computing*, 1987.

[CGJ95] E.M. Clarke, O. Grumberg, and S. Jha. Verifying parameterized networks using abstraction and regular languages. *CONCUR'95*, 1995.

[DDHY92] David L. Dill, Andreas J. Drexler, Alan J. Hu, and C. Han Yang. Protocol verification as a hardware design aid. *Int'l Conf. on Computer Design: VLSI in Computers and Processors*, 1992.

[Dij85] E.J. Dijkstra. Invariance and nondeterminacy. In *Mathematical Logic and Programming Languages*. Prentice-Hall, 1985.

[GL93] S. Graf and C. Loiseaux. A tool for symbolic program verification and abstraction. *5th Int'l Conf. on Computer-Aided Verification*, April 1993.

[Gra94] Susanne Graf. Verification of a distributed cache memory by using abstractions. *6th Int'l Conf. on Computer-Aided Verification*, 1994.

[GS92] S.M. German and A.P. Sistla. Reasoning about systems with many processes. *Journal of Association for Computing Machinery*, 39(3):675–735, 1992.

[ID93a] C. Norris Ip and David L. Dill. Better verification through symmetry. *11th Int'l Symp. on Computer Hardware Description Languages and Their Applications*, 1993. Extended version with complete proofs and semantic analysis to appear in Formal Methods in System Design.

[ID93b] C. Norris Ip and David L. Dill. Efficient verification of symmetric concurrent systems. *Int'l Conf. on Computer Design: VLSI in Computers and Processors*, 1993.

[ID96] C. Norris Ip and David L. Dill. State reduction using reversible rules. *33rd Design Automation Conference*, June 1996.

[KMOS94] Robert P. Kurshan, Michael Merritt, Ariel Orda, and Sonia R. Sachs. A structural linearization principle for processes. *Formal Methods in System Design*, 5, 1994.

[LLG+90] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. The directory-based cache coherence protocol for the dash multiprocessor. *17th Int'l Symp. on Computer Architecture*, 1990.

[Lub84] Boris D. Lubachevsky. An approach to automating the verification of compact parallel coordination programs I. *Acta Informatica*, 21:125–169, 1984.

[PD93] F. Pong and M. Dubois. Correctness of a directory-based cache coherence protocol: Early experience. *5th Symp. on Parallel Distributed Processing*, 1993.

[PD95] F. Pong and M. Dubois. A new approach for the verification of cache coherence protocols. *IEEE Transactions on Parallel and Distributed Systems*, 6(2), 1995.

[PNAD95] F. Pong, A. Nowatzyk, G. Aybay, and M. Dubois. Verifying distributed directory-based cache coherence protocols: S3.mp, a case study. *EurPar'95*.

[Pon95] Fong Pong. *Symbolic State Model: A New Approach for the Verification of Cache Coherence Protocols*. PhD thesis, University of Southern California, 1995.

[RS93] June-Kyung Rho and Fabio Somenzi. Automatic generation of network invariants for the verification of iterative sequential systems. *5th Int'l Conf. on Computer-Aided Verification*, June 1993.

[SG87] A.P. Sistla and S.M. German. Reasoning with many processes. *Symp. on Logic in Computer Science*, 1987.

[SHTO93] Kenji Shibata, Yutaka Hirakawa, Akira Takura, and Tadashi Ohta. Reachability analysis for specified processes in a behavior description. *IEICE Transaction on Communication*, E76-B(11), November 1993.

[WL89] Pierre Wolper and Vinciane Lovinfosse. Verifying properties of large sets of processes with network invariants. In *Automatic Verification Methods for Finite State Systems*, volume 407 of *LNCS*, Springer-Verlag, 1989.

Verification of Arithmetic Circuits by Comparing Two Similar Circuits

Masahiro Fujita
Fujitsu Laboratories of America, Inc.
3350 Scott Blvd., Bldg. #34, Santa Clara, CA95054
fujita@fla.fujitsu.com

Abstract. Recently there have been a lot of progress in technologies for comparing two structurally similar large circuits [2, 14, 13]. Circuits having more than 10,000 gates, whose BDD cannot be built, have been verified in several minutes. However, arithmetic circuit verification with respect to specification is still a hard problem. As shown in [16] some arithmetic circuits, such as multipliers, square function. cube functions, etc., must satisfy some recurrence equations, such as $f(x+1,y) = f(x,y) + y$ *where* $f(x,y) = xy$, and those equations can be used for verification. In this paper, we use such recurrence equations in order to drive Boolean comparison problems of structurally similar circuits. That is, left hand sides and right hand sides of equations are realized as separated circuits and then compared. Using the recurrence equation properly, these circuits have many internal equivalent signals and many implications among signals, by which Boolean comparison programs, such as [2, 14, 13], can work very effectively. Using the proposed method, 16-bit multipliers, such as C6288 of ISCAS85 benchmark circuits, are verified within 12 minutes.

1 Introduction

Formal verification techniques have been paid much attention recently. There have been lots of works on formal hardware verification [11, 10], and among them, Binary Decision Diagram (BDD) [3] based verification techniques, such as [5, 8, 17, 6, 15], have given successful results for practical designs.

However, BDD may not work well for arithmetic circuits, such as multipliers. Therefore, several extensions are made on BDD, such as, BMD [4], HDD [7], OKFDD [9], etc. Although originally word-level verification is necessary in order to use BMD. by using the technique shown in [12] which compute BMD from outputs to inputs instead of inputs to outputs, we can now use BMD directly to verify arithmetic circuits, such as multipliers. However, if there are errors

(bugs) in the circuits, BMD can easily blow up and the verification program may not terminate, since those circuits represent different logic functions from multiplier which can have exponential sizes of BMD. Of course this depends on each error but we actually observed this BMD explosion by randomly inserting logical errors to the multiplier circuits and generating BMDs.

In this paper, we show another approach to attack the verification of arithmetic circuits. Instead of directly generating BDD (or its extensions) from given circuits, we create circuits based on the recurrence equations that must be satisfied by the circuits. This idea was originally proposed by Ochi [16]. For example a recurrence equation for multipliers is:

$$f(x+1,y) = f(x,y) + y \quad \textit{where } f(x,y) = xy \textit{ and } x,\ y \textit{ are inputs}$$

Any circuits which satisfy the above recurrence equation are multipliers[1]. He used recurrence equations to verify arithmetic circuits by first generating BDD from the circuits and then check if that BDD satisfy the required recurrence equations. Clearly this method has a drawback that we have to build BDD for the circuits first, which is often impossible for large arithmetic circuits.

On the other hand, recurrence equations, such as shown above (for multipliers), may indicate a comparison problem of two circuits (or Boolean functions). That is, checking recurrence equation means comparing the left hand side and right hand side of the equation. So, basically we can use Boolean comparison techniques for such equivalence checking.

Recently there have been much progress in technologies for Boolean comparison of similar circuits. Here "similar" means that we can find many logical relationships, such as equivalences or implications, among the internal signals in the two circuits to be compared. In a practical design environment, designers want to check the equivalence of the two circuits which are very structurally similar. For example, it is often the case to check the equivalence between unoptimized circuits and manually optimized circuits. For such cases, there are lots of relationships among the internal signals in the two circuits. By utilizing these relationships, methods like [2, 14, 13] can verify much larger circuits than the circuits which can be verified just by BDDs. 10,000 gates or larger circuits can be verified within practical time.

Basically recurrence equations suggest two structurally similar circuits (left hand side and right hand side). Here we propose a new verification method

[1] Circuits must also satisfy boundary conditions, such as, $f(0,y) = 0$, which can be checked rather easily.

for arithmetic circuits based on recurrence equations. We first generate two circuits which correspond to left hand side and right hand side of the recurrence equations. Then apply Boolean comparison program for similar circuits to those circuits. Please note that we need only one circuit which should be verified. The two circuits to be compared are generated from that circuit by adding appropriate extra circuits, such as adders, incrementors, etc. Also note that we do not need specification in Boolean functions. Specification is fully described in the recurrence equations that we are using to generate two circuits.

By using case splitting appropriately, we can verify 16-bit multipliers, such as C6288 of ISCAS benchmark circuits, in less than 12 minutes on Sparc20. Moreover, different from the method in [12], the proposed method can finish verification in similar time, even if the circuits are not correct as shown in section 3.

In the next section, we introduce our verification method. Then section 3 gives preliminary results. Section 4 is our concluding remarks. Although we discuss only about multipliers for simplicity, the proposed method can be applied many arithmetic functions which have proper recurrence equations, such as, square functions, cube functions, etc.

2 Verification algorithm

In this section we introduce our verification method. For simplicity, we use multipliers as examples all the time, although we can verify many other arithmetic circuits which have proper recurrence equations, such as, square functions, cube functions, etc. As long as there are proper recurrence equations, we can verify any circuits, including random circuits (assuming such recurrence equations are given)[2].

The basic idea for multipliers is illustrated in Figure 1. Since multipliers satisfy the following equation, two circuits which are derived from left hand side and right hand side of the equation respectively must realize the same Boolean function.

$$f(x+1,y) = f(x,y) + y \quad \textit{where } f(x,y) = xy \textit{ and } x,\ y \textit{ are inputs}$$

[2] In some sense, we can consider the proposed method is a kind of self-checking methods proposed in [1]. What we are doing in this paper can be described in the following way: by appropriatly using recurrence equations (self checking properties), we are reducing verification problems into Boolean comparison problem for similar circuits. Of course, if the reduced Boolean comparison problems are too large, we can use random simulation based checking just like in [1].

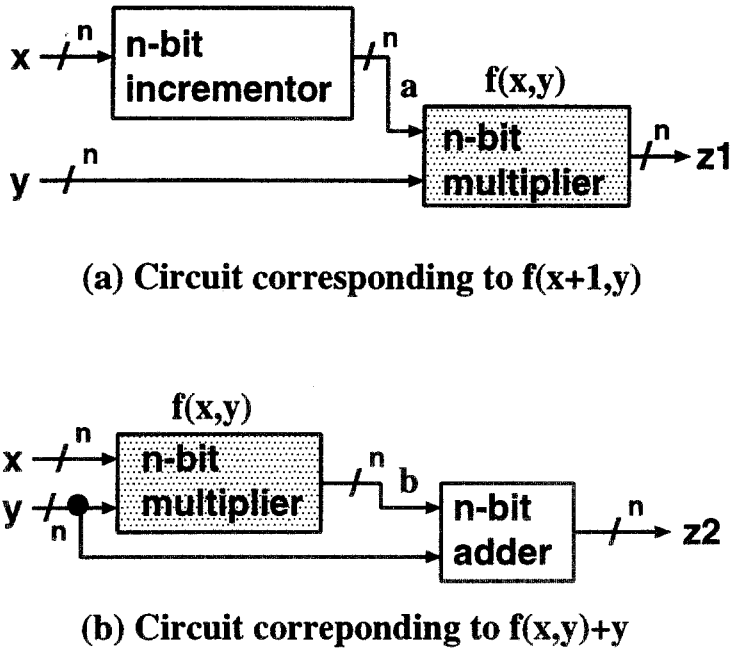


Fig. 1. Circuit realization of the recurrence equation for multipliers

Please note that the two multipliers in Figure 1 are the same circuit which we want to verify. We are assuming here that incrementor and adder are given and they are guaranteed to be correct.

Please also note that the above equation together with boundary condition for $x = 0$ completely specify the function and that must be multiplier.

So, by comparing the two circuits shown in Figure 1, we can formally verify multipliers[3]. But this is not an easy Boolean comparison problem. Clearly we cannot build BDD for each circuit, if that is a large bit width multipliers, such as C6288 of ISCAS85 benchmark circuits. Although large portion of the two circuits are the same sub-circuits (multiplier), they are not similar circuits in the sense that we cannot find many equivalent signals between the two circuits. So, we cannot directly apply the Boolean comparison methods like [2, 14, 13].

However, if we consider only the case where the least significant bit of x, x_0 is 0, then the incrementor becomes just an inverter as shown in Figure 2[4]. Thus the two circuits become like the ones shown in Figures 3. There are many equivalent

[3] In this paper, we assume that extra circuits, such as incrementor, adder, subtracter, etc., are guaranteed to be correct. Or those should be verified first

[4] If $x_0 = 0$, then increment does not affect the values of $x_1, x_2, ..., x_{n-1}$.

signals between the two circuits, since most inputs are common and large portion of the circuits are the same. In fact there are a lot of functional relationships among internal signals of the two circuits.

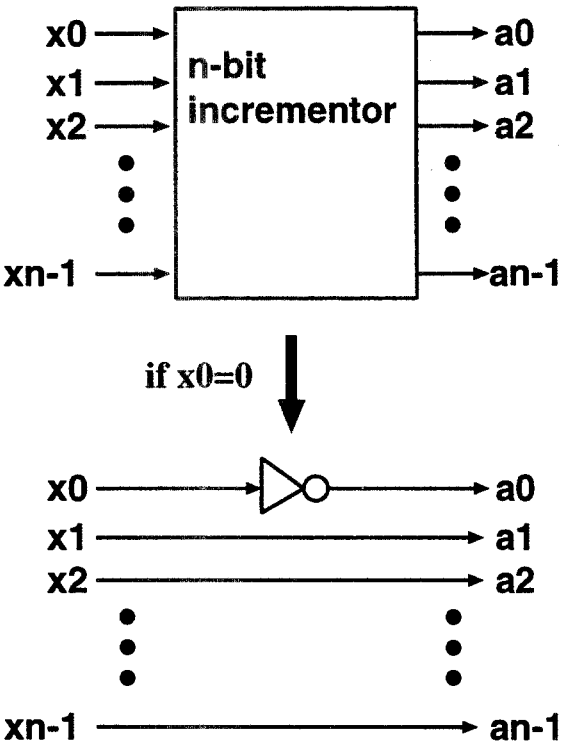


Fig. 2. If the least significant bit is 0, incrementor becomes just an inverter

By this case splitting, we can verify multipliers when $x_0 = 0$. Then how about the cases when $x_0 = 1$? We can proceed with the same idea: further case splitting with $x_1, x_2, ...$ For example, if $x_0 = 1$ but $x_1 = 0$, then the incrementor becomes just two inverters as shown in Figure 4. Again the two circuits generated are similar as shown in Figure 5.

The next splitting case is $x_0 = 1, x_1 = 1, x_2 = 0$ which needs three inverters for x_0, x_1, and x_2. This case splitting process can be continued until we reach the case where $x_0 = 1, x_1 = 1, x_2 = 1, ..., x_{n-2} = 1, x_{n-1} = 0$.

What we are doing here is just check the equation:

$$f(x+1, y) = f(x, y) + y \quad \textit{where } f(x, y) = xy \textit{ and } x,\ y \textit{ are inputs}$$

by case splitting the values of x_i.

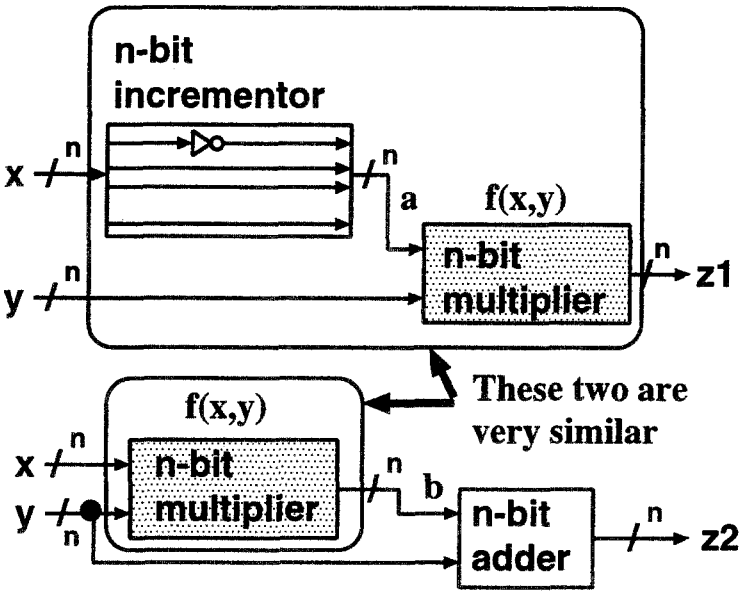


Fig. 3. By assuming $x_0 = 0$, the circuits become very similar

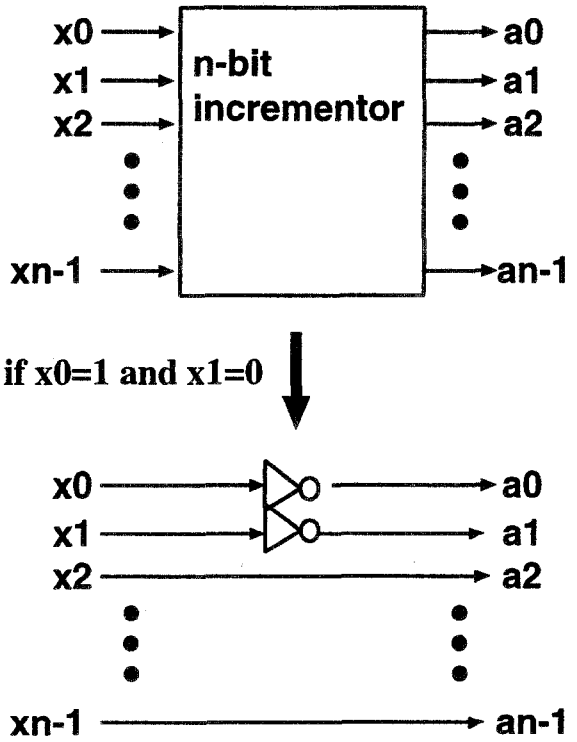


Fig. 4. The case where $x_0 = 1$ but $x_1 = 0$

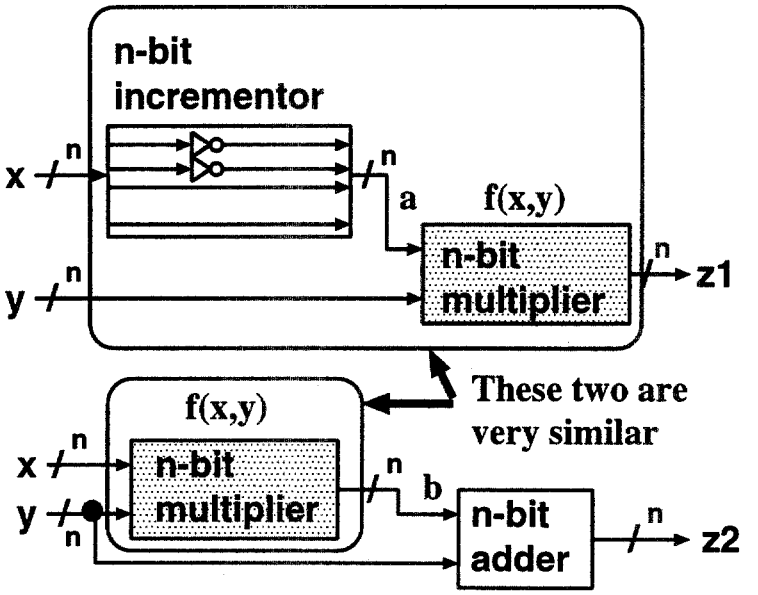


Fig. 5. By assuming $x_0 = 1, x_1 = 0$, again the circuits become very similar

As we have more number of inverters, the two circuits become less similar. However, as we have more number of inverters, we can fix the values of x_i more. That is, in the case of Figure 4, since this is the case where $x_0 = 1$ and $x_1 = 0$, we can fix the values of x_0 and x_1. So there are trade-offs in terms of difficulty and the most difficult case happens when there are two inverters as shown in Figure 4 according to our experiments for C6288 in the next section.

By using the above case splitting, we can keep the similarity of the two circuits. These circuits should be rather easy circuits for the Boolean comparison methods like [2, 14, 13]. In fact, as shown in the next section, we have found many equivalent signals which drastically reduce the verification time or complexity of the problem.

3 Preliminary experimental results

We did some preliminary experiments for multipliers. We plan to do more intensive experiments using other types of circuits, such as, square functions.

Our program first generates net-lists for the two circuits in Figure 2, 4, and others [5] from the given multipliers. Then apply our Boolean comparison programs to them.

[5] In the case of 16-bit multipliers, there are 16 cases in total. But some of them are trivial, since most of x_i are constants.

We verified C6288 of ISCAS85 benchmark circuits for its first 16 outputs, since as shown in Figure 1, all values should be the same bit-width (16bit in this case). The results are shown in Figure 6. We did two types of experiments. The first one is to just verify C6288 circuit, which is a correct multiplier. It took only less than 12 minutes in total to verify.

The program found 342 equivalent internal signals of the two circuits out of 360 internal signals for the case of Figure 2. So large portion of the two circuits are equivalent and that is why verification can finish so quickly. The most time consuming case is the one shown in Figure 4 which took 8 minutes to finish. This is the case where the two circuits are similar but not so much and their circuit sizes are still large (only small number of x_i have fixed value). All the other cases are less than one minute.

Multiplication Circuit C6288 (first 16 outputs)	**CPU time sec. on Sparc20**	
	Original (correct)	**Error inserted**
Case: x0=0	**14.0**	**2.0-60.0 depending on errors inserted**
Case: x0=1, x1=0	**496.0**	
All other cases	**Less than 60.0**	

Fig. 6. Results for 16-bit multipliers

Second experiment we did is to try to verify incorrect multipliers (verification fails) by intentionally inserting errors into C6288 (changing function of a gate,

etc.[6]). Depending on changes, it took less than one minute (sometime in a couple of seconds) to prove the circuit is not a multiplier. Depending errors, the cases where verification fails are different, but mostly verification fails in multiple cases. Again this is extremely fast. Please note that the method in [12] may not work well for incorrect circuits.

4 Conclusions

We have shown a verification method for arithmetic circuits. We also demonstrated that C6288 can be verified in less than 12 minutes. Even if the circuits are not correct (there is a bug in the circuits), verification time remain similar or less. Also, different from BMD or HDD based methods, we do not need another BDD package, such as, BMD package. We can use existing BDD packages or Boolean comparison programs to verify arithmetic circuits. We believe that the proposed method has a significance in its applications.

Although in this paper we only discussed about combinational circuits, the proposed techniques can be applied to sequential circuits by deriving appropriate recurrence equations. Surely this is one of our future research topics.

Also, the proposed method can be considered to be a kind of self-checking methods proposed in [1]. What we are doing here can be described in the following way: by appropriately using recurrence equations (self checking properties), we are reducing verification problems into Boolean comparison problem for similar circuits. Of course, if the reduced Boolean comparison problems are too large, we can use random simulation based checking just like in [1]. We are planning to explore this area and study on extensions of the proposed method.

References

1. M. Blum, M. Luby, and R. Rubinfeld. "self-testing/correctig with application to numerical problems". In *Proc. of 22nd ACM Theory of Computing*, pages 73–83, 1990.
2. D. Brand. "verification of large synthesized designs". In *Proc. of ICCAD*, pages 534–537, Nov. 1993.
3. R.E. Bryant. "graph-based algorithms for boolean function manipulation". *IEEE Trans. Computer*, C-35(8):667–691, Aug. 1986.

[6] Even if we make many changes in the circuit, situations are the same. The two circuits we generate according to Figure 1 are very similar.

4. R.E. Bryant and Y.-A. Chen. "verification of arithmetic functions with binary moment diagrams". In *Proc. of 32nd DAC*, Jun. 1995.
5. J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and J. Hwang. "symbolic model checking: 10^{20} states and beyond". In *Proc. of the Fifth Anual IEEE Symposium on Logic in Computer Science*, Jun. 1990.
6. H. Cho, G. Hachtel, S-W. Jeong, B. Plessier, E. Schwarz, and F. Somenzi. "atpg aspects of fsm verification". In *Proc. IEEE Int. Conf. on Computer-Aided Design (ICCAD-90)*, pages 134–137, Nov. 1990.
7. E.M. Clarke, M. Fujita, and Z. Zhao. "hybrid decision diagrams - overcoming the limitations pf mtbdds and bmds". In *Proc. of ICCAD*, pages 159–163, Nov. 1995.
8. O. Coudert and J.C. Madre. "a unified framework for the formal verification of sequential circuits". In *Proc. IEEE Int. Conf. on Computer-Aided Design (ICCAD-90)*, pages 126–129, Nov. 1990.
9. R. Drechsler, A. Sarabi, M. Theobald, B. Becker, and M. Perkowski. "efficient representation and manipulation of switching functions based on ordered kronecker functional decision diagrams". In *Proc. of 31st DAC*, Jun. 1994.
10. M. Fujita. "rtl design verification by making use of datapath information". In *Proc. of ICCD-92*, pages 592–597, Oct. 1992.
11. A. Gupta. "formal hardware verification methods: a survey". *Formal Methods in System Design*, Vol. 1(2/3), Oct. 1992.
12. K. Hamaguchi, A. Morita, and S. Yajima. "efficient construction of binary moment diagrams for verifying arithmetic circuits". In *Proc. of ICCAD*, pages 78–82, Nov. 1995.
13. J. Jain, R. Mukherjee, and M. Fujita. "advanced verification techniques based on learning". In *Proc. of 32nd DAC*, pages 420–426, Jun. 1995.
14. W. Kunz. "hannibal: An efficient tool for logic verification based on recursive learning". In *Proc. of ICCAD*, pages 538–543, Nov. 1993.
15. K.L. McMillan. "symbolic model checking: An approach to the state explosion problem". Technical Report CMU-CS-92-131, Carnegie Mellon University, May 1992.
16. H. Ochi and S. Yajima. "formal design verification of combinational circuits specified by recurrence equations". In *Proc. of SASIMI'95*, pages 101–105, Aug. 1995.
17. H. Touati, H. Savoj, B. Lin, R.K. Brayton, and A. Sangiovanni-Vincentelli. "implicit state enumeration of finite state machines using bdds". In *Proc. IEEE Int. Conf. on Computer-Aided Design (ICCAD-90)*, pages 130–133, Nov. 1990.

Automated Deduction and Formal Methods*

John Rushby

Computer Science Laboratory, SRI International,
Menlo Park, CA 94025, USA

Abstract. The automated deduction and model checking communities have developed techniques that are impressively effective when applied to suitable problems. However, these problems seldom coincide exactly with those that arise in formal methods. Using small but realistic examples for illustration, I will argue that effective deductive support for formal methods requires cooperation among different techniques and an integrated approach to language, deduction, and supporting capabilities such as simulation and the construction of invariants and abstractions. Successful application of automated deduction to formal methods will enrich both fields, providing new opportunities for research and use of automated deduction, and making formal methods a truly useful and practical tool.

1 Introduction

Formal methods are a natural application area for automated deduction—yet, with few exceptions, tools for mainstream formal methods provide little more than rudimentary support for deduction, and few theorem provers find application in formal methods. Model checking and related techniques are gaining acceptance in important specialized areas, but have yet to penetrate the larger field. This disconnect between formal methods and the very technologies that could help increase its utility and appeal is unfortunate, and deserves explanation and remedy.

My opinion is that many techniques for automated deduction (and for simplicity I include model checking under this heading) provide excellent solutions to individual problems, but that formal methods require more integrated approaches to provide solutions that are effective across a broad range of problems. In the following sections, I outline some prototypical applications of formal methods and suggest some of the capabilities required of automated deduction if it is to achieve more widespread use in this area. I discuss these topics under three headings: language, theories, and interaction in the sections that follow. Brief conclusions are presented in Section 5.

* This work was supported by the Air Force Office of Scientific Research under contract F49620-95-C0044 and by the National Science Foundation under contract CCR-9509931. The applications described were undertaken for NASA Langley Research Center under contracts NAS1-18969 and NAS1-20334 and for ARPA through NASA Ames Research Center under contract NASA-NAG-2-891.

2 Language

By *formal methods* I mean the use of techniques derived from mathematical logic for the specification and analysis of computational systems. There are two elements here: *specification*, by which I mean a descriptive activity in which logical notation is valued for its contributions to both the intellectual process of design and the communication of designs, and *analysis*, by which I mean systematic and repeatable methods for deducing properties of specifications and of the designs that they represent. Automated deduction has obvious relevance in the mechanization of analysis, but formal methods practitioners attach great importance to specification and are unwilling to compromise on the convenience of expression provided by a full specification language. To achieve acceptance, it therefore seems necessary that automated deduction should be harnessed to rather rich notations.

To suggest some of the capabilities desired, I outline a typical "requirements specification" for a function in the Space Shuttle's control system called "Jet-Select" [6]. This function is responsible for selecting which of the Shuttle's Reaction Control System (RCS) jets (or thrusters) should be fired in order to accomplish a given translational or rotational acceleration. I will concentrate on the "Vernier/Alt" component for rotation, which can operate in one of two modes: in Vernier mode, only the small "vernier" jets are considered for selection; in Alt (alternative) mode, only the larger "primary" jets are considered. The basic Jet-Select calculations are the same whether in Vernier or Alt mode, except that the six vernier jets are treated singly, while the 38 primary jets are treated in groups. (The primary jets are arranged in 14 groups, each consisting of two, three, or four jets located adjacent to each other and firing in the same direction; only 11 of the 14 groups are useful for rotational maneuvers.) In Vernier mode, Jet-Select chooses up to three individual vernier jets to fire, whereas in Alt mode it selects up to three groups of primary jets, and then selects exactly one jet from each of the chosen groups. (The jets within each group are ranked in a priority order and it is the available jet of highest priority that is fired when its group is selected in Alt mode.) Various vernier jets and groups of primary jets are excluded from consideration in certain submodes (e.g., jets whose plumes extend into the area above the cargo bay are excluded in "low +z" mode) and individual jets may be marked "unavailable" due to failure or by crew selection.

The selection of vernier jets or primary groups is performed by an algorithm known as "max dot-product" (this particular exercise in formalization was undertaken in preparation for introduction of a new algorithm called "min angle"). For each vernier jet and primary group, a table records the rotational velocity vector imparted by firing that jet (or a member of that group) for a standard period. (Actually, there are several tables, parameterized by whether there is a payload attached to the Shuttle's robotic arm, and where the arm is positioned.) The algorithm proceeds by first selecting the vernier jet or primary group whose acceleration has the largest scalar (dot) product with the rotational acceleration vector actually desired; the second and third jets (if required) are similarly selected as those with the second and third largest scalar products, *provided* the

dot-product of the second exceeds some fraction t_1 of the first, and that of the third exceeds some fraction t_2 of the second.

The major goal here is to use formal methods to specify the desired functionality as clearly as possible. The role of automated deduction in this example is to contribute to validation of the specification by examining putative "challenge" theorems such as "a failed jet will never be selected."

A good specification for this component of Jet-Select should make clear that the max dot-product algorithm is essentially the same in both Vernier and Alt modes, except that in the former it operates over individual vernier jets, while in the latter it operates over groups of primary jets. This argues for a specification notation that provides parameterized theories so that specification of the same algorithm can be instantiated over these different domains. Although not exemplified by Jet-Select, many applications of formal methods also require parameterized representations for standard computer science data structures such as lists, trees, and arrays.

Next, we can observe that the output of Jet-Select is most naturally considered as a set of jets, and the groups of primary jets are also naturally considered as sets. Thus, our specification notation should incorporate a representation for sets. Most practitioners of formal methods prefer their specification notation to be strongly typed, and this particular application seems to call for subtyping: surely the vernier and primary jets are naturally considered as subtypes of the type of all jets. But then the output of the algorithm will be either a set of vernier jets or a set of primary groups (the latter is then converted to a set of primary jets), whereas the output of Jet-Select as a whole must be a set of jets. Hence, our specification notation must somehow extend the subtyping relation between (for example) vernier jets and all jets to a compatible subtyping relation between the *sets* of such jets.

There are (at least) two ways to specify that the (intermediate) result of Jet-Select should be the set of vernier jets or primary groups satisfying the max dot-product criterion. One way would simply axiomatize the desired property, the other would attempt to represent the algorithm suggested by the informal description (i.e., the iterative selection of the three best jets or groups from among those available). The latter approach might require the specification language to incorporate a treatment of imperative programs. It would also require a way to identify the jet or group in a given set that has the maximum dot-product. For generality, we might like to provide a library axiom defining the maximum of a set to be its largest member with respect to some given ordering. This is most directly accomplished by quantification, but we must ensure that the ordering relation has the appropriate algebraic properties and must take proper care of the case where the set is empty, or risk unsoundness. A specification language should help ensure that these obligations are not overlooked.[2]

[2] For example, the PVS declaration

$$max(s : \text{setof}[T]) : \{t : T | t \in s \wedge \forall (x : T) : x \in s \supset (t > x \vee x = t)\}$$

generates a proof obligation (to show that the type assigned to the value of *max* is inhabited) that can be discharged only if the set s is nonempty and $>$ is a well-ordering.

Most theorem provers support raw logics that lack the notational conveniences mentioned above. In my experience, it quite hopeless to persuade users of formal methods (let alone those who are not yet users) to adopt such impoverished notations. To observe that it is perfectly *feasible* to provide a specification for Jet-Select in quite primitive logics (e.g., those without quantification) misses the point—this simply is not what users of formal methods want to do.

Left to their own devices, users of formal methods develop or adopt notations such as B, VDM, RAISE, or Z. These make few concessions to the needs of efficient automated deduction and the tools that have been developed for them provide little more than interactive proof checking unsupported by significant automation (e.g., [8, 10]). I have argued elsewhere [14] that choices made in the designs of these languages (e.g., in the case of Z, set theory with partial functions, and no notion of definition) are inimical to automated deduction, and that really efficient deductive support is therefore unlikely to be forthcoming for them.

One of the challenges to those who would provide automated deduction for formal methods is therefore to contribute to the design of specification languages that combine the felicity of expression desired for formal methods with the possibility of powerfully automated support. Rather than being a limitation on specification language design, I believe that closer integration of language and automated deduction can have a liberating effect—because it makes it possible to contemplate design choices that require theorem proving in typechecking. We have exploited this opportunity to some extent in PVS [12] (where subtyping, for example, can generate proof obligations) but many further opportunities remain.

It is not necessary that the logic supported by a theorem prover should *be* a full specification language, but there must be some translation from the latter to the former. Furthermore, the translation must be maintained during interaction with the prover: it is unlikely to be acceptable if proof of a conjecture expressed in the specification language must be conducted in terms of its translation into the primitives of the underlying logic.

3 Theories

Automated deduction must not only support the rich linguistic capabilities desired in formal methods, but must also provide very effective automation for theories that are commonly encountered.

For illustration, I will use a verification of the Interactive Convergence Algorithm for Byzantine fault-tolerant clock synchronization [9] that Friedrich von Henke and I performed some years ago [15]. The goal is to keep the clocks of distributed processors approximately synchronized, given that good clocks have some bounded drift rate, good processors can read the clocks of other good processors with some small error, and faulty processors and clocks are unconstrained (in particular, they can present conflicting information to different good processors). The clock of processor p is represented by an uninterpreted function $c_p(T)$

from "clock time" to "real time" (both interpreted as real numbers).[3] Clocks are adjusted every R clock time units (this duration is called a "frame" and the start time of the i'th frame is denoted $R^{(i)}$), during a "synchronization period" of duration S clock time units occurring at the end of the frame (the start of the i'th synchronization period is denoted $S^{(i)}$). The adjustment to clock p for period i is $C_p^{(i)}$ clock time units and the adjusted clock for that period is denoted $c_p^{(i)}(T)$, where $c_p^{(i)}(T) = c_p(T + C_p^{(i)})$.

In the i'th synchronizing period, each processor p obtains an estimate $\Delta_{qp}^{(i)}$ of the skew between its clock and that of processor q. A parameter ϵ bounds the error in this estimate as follows.

Assumption A2. *If conditions the clock synchronization conditions (defined below) hold for the i'th period, and processors p and q are nonfaulty through period i, then*

$$|\Delta_{qp}^{(i)}| \leq S$$

and

$$|c_p^{(i)}(T' + \Delta_{qp}^{(i)}) - c_q^{(i)}(T')| < \epsilon$$

for some time T' in $S^{(i)}$.

The algorithm is defined as follows.

Algorithm ICA. *For all processors p:*

$$C_p^{(i+1)} = C_p^{(i)} + \Delta_p^{(i)},$$

where

$$C_p^{(0)} \text{ is arbitrary,}$$

$$\Delta_p^{(i)} = \left(\frac{1}{n}\right) \sum_{r=1}^{n} \bar{\Delta}_{rp}^{(i)}, \quad \textbf{and}$$

$$\bar{\Delta}_{rp}^{(i)} = \textbf{if } |\Delta_{rp}^{(i)}| < \Delta \textbf{ then } \Delta_{rp}^{(i)} \textbf{ else } 0$$

and Δ is a clock time quantity that is a parameter to the algorithm.

The goal is to achieve the following *clock synchronization conditions*, provided that at most m processors (out of n) are faulty through period i, for real time constant δ and clock time constant Σ that are parameters to the algorithm.

Bounded skew: *If p and q are nonfaulty through period i, then*

$$|c_p^{(i)}(T) - c_q^{(i)}(T)| < \delta$$

for all T in $R^{(i)}$.

[3] A specification language with the ability to distinguish clock time and real time as different "dimensions" of the same type provides valuable additional error checking in these constructions.

Bounded adjustment: *If processor p is nonfaulty through period i, then*

$$|C_p^{(i+1)} - C_p^{(i)}| < \Sigma.$$

These conditions can be achieved, provided several assumptions (concerning, for example, the drift rate ρ of good clocks) are satisfied, together with several constraints on the parameters to the algorithm, such as the following.

Constraint C6. $\delta \geq 2(\epsilon + \rho S) + \dfrac{2m\Delta}{n-m} + \dfrac{n\rho R}{n-m} + \dfrac{n\rho\Sigma}{n-m} + \rho\Delta$

The proof depends on several lemmas, of which the following are among the most important.

Lemma 4. *If the clock synchronization conditions hold for i, processors p, q, and r are nonfaulty through period $i+1$, and $T \in S^{(i)}$, then*

$$|c_p^{(i)}(T) + \bar{\Delta}_{rp}^{(i)} - [c_q^{(i)}(T) + \bar{\Delta}_{rq}^{(i)}]| < 2(\epsilon + \rho S) + \rho\Delta.$$

Lemma 5. *If the bounded skew clock synchronization condition holds for i, processors p and q are nonfaulty through period $i+1$, and $T \in S^{(i)}$, then*

$$|c_p^{(i)}(T) + \bar{\Delta}_{rp}^{(i)} - [c_q^{(i)}(T) + \bar{\Delta}_{rq}^{(i)}]| < \delta + 2\Delta.$$

The items of interest here are the theories involved: we have arithmetic expressions and relations involving both real and natural numbers, and both interpreted and uninterpreted function symbols. The ubiquity and complexity of the arithmetic used here are such that it would be intolerable to attempt verification of this algorithm without efficient deductive support for arithmetic. A library of lemmas and rewrite rules will not be adequate to the task: decision procedures are needed. The question then is: decision procedures for which theories? The importance of integer arithmetic is such that some tools for formal methods include decision procedures for Presburger arithmetic—that is the quantified theory of integer linear arithmetic. Since we have real numbers as well, a decision procedure for real closed fields might also seem appropriate. The problem with these choices is that we also have uninterpreted function symbols, which takes us outside these decidable theories. Inspection of various formulas appearing in the presentation of the algorithm shows that only Assumption A2 involves a nested quantifier (for T'), everything else is (implicitly) universally quantified at the outermost level. We can conclude that the quantifier reasoning here is likely to be easy, and we may therefore be prepared to deal with it outside the arithmetic decision procedures (either heuristically, or with user guidance). This will allow us to restrict the arithmetic decision procedures to just the ground case—where the combination of linear arithmetic with uninterpreted function symbols is decidable [4].

My experience with formal methods applications is that this tradeoff in favor of deciding ground theories is always worthwhile, since it allows the different decision procedures to be combined. Some theories, such as arithmetic, equality with

uninterpreted function symbols, and arrays[4] are so ubiquitous that decision procedures for their ground cases are essential for all productive work. Decision procedures for additional theories may be highly advantageous for particular classes of applications. For example, our experience with processor verification [17] has shown that the (large) library of rewrite rules used for the theory of bitvectors is the main impediment to effective automation, and we conjecture that a decision procedure for bitvectors would have a dramatic benefit. The development of new decision procedures for theories arising in formal methods is a valuable topic for research.

Important requirements for such decision procedures are the following.

- They must work cooperatively to decide the *combination* of their theories.
- They must deal gracefully with terms outside the decided theory. For example, the theory decided by the *SUP-INF* [16] and similar procedures is ground *linear* arithmetic, but several of the formulas used in clock synchronization contain nonlinear terms (and division). Although the full nonlinear case cannot be decided, it is important to deal with special properties (e.g., commutativity, and "a minus times a minus is a plus") without losing those properties that follow simply by treating nonlinear multiplication as uninterpreted. A similarly effective extension to division is also required. (Notice also that some treatment for the partiality of division by zero is needed; this may require coordination between the specification language and its deductive support—in PVS, for example, division by zero is excluded through type rules that generate proof obligations to show the divisor is nonzero.)
- Their behavior must be predictable. One of the strengths of decision procedures over heuristics is that the user should not have to puzzle over whether the failure to prove a conjecture is due to its falsehood, or an inadequate heuristic. This benefit is lost if the decided theory is not clearly characterized. And although performance is hard to guarantee given the super-exponential complexity of most decision procedures, "black holes" (where a small and apparently simple problem takes an inordinate amount of time) are to be avoided. Because they will form the inner loop of larger procedures, even linear speedups in the performance of decision procedures can have a dramatic impact on overall efficiency; more needs to be known about the relative practical performance of various decision procedures for the same problem, which anecdotal evidence indicates can differ by an order of magnitude or more [4]. Conjectures in formal methods applications often give rise to very large formulas, so it is crucial that decision procedures should be implemented in ways that scale reasonably well (using, for example, structure-sharing techniques similar to those in BDDs[5]).

[4] That is (in PVS notation) $f[(x) := y](z) = \textbf{if } z = x \textbf{ then } y \textbf{ else } f(z)$. This is also known as function updating or overriding.

[5] It goes without saying that propositional reasoning must be implemented very efficiently. Ordered binary decision diagrams (OBDDs) are the natural choice, but the Davis-Putnam procedure and the patented algorithm of Stålmarck [18] may be superior in some applications.

- Expressions that cannot be decided should be simplified. Especially in an interactive environment, it is important that the information presented to the user should be as brief and as simple as possible. But it should also be familiar—that is to say, expressions should retain, to the extent possible, the form they were originally given by the user, and should not be arbitrarily normalized. Simplification should merely eliminate redundancy, so that, for example, $(a+1)-1$, **if** *true* **then** a **else** b, and **if** B **then** a **else** a all become a; it should generally refrain from transformations such as that from $x \times (a+b)$ to $x \times a + x \times b$. One of the great advantages of decision procedures over heuristics is that they are sensitive only to the content and not to the form of expressions, so that syntactic representations can be chosen for the convenience of the user rather than the prover.

With standard theories handled by ground decision procedures, the next candidate for automation is quantifier reasoning. Traditional methods for first-order reasoning, such as resolution, do not extend well to the presence of decided ground theories, and therefore find little application in formal methods. (Also, formal methods often use higher-order quantification.) Fortunately, as noted above, there is generally little nesting or alternation of quantifiers in these applications, so that a combination of specialized and heuristic methods work quite well for the majority of cases (difficult cases then require user guidance). Specialized methods include those for conditional rewriting in the presence of decided theories—the close integration of rewriting with linear arithmetic is the source for much of the effectiveness of Boyer and Moore's provers [3], and similar capabilities are required in any system intended to support formal methods. Matching techniques similar to those used in rewriting can also provide heuristic instantiation for general formulas. However, my experience with PVS is that while its conditional rewriter is almost completely effective (i.e., it rarely fails to find a match if one exists), its heuristic instantiation of lemmas and general quantifier reasoning fails (usually by finding an unproductive match) more often than I would like. More effective methods for quantifier reasoning in these contexts (and for restricted instances of the higher-order case) would be a good topic for research.

Inspection of the formulas for clock synchronization shown earlier suggests that, in addition to arithmetic, propositional, and quantifier reasoning, we will also need induction. Proof that the algorithm maintains the clock synchronization conditions is accomplished using simple induction on the frame index i. Several results on finite summations are also used (a key step in the proof is to split the summation in the definition of $\Delta_p^{(i)}$ into m terms constrained by Lemma 5, and $n-m$ constrained by Lemma 4), and these require bounded induction (i.e., induction over a subrange of the natural numbers) on the recursive function that is used to define summation. Given the need for induction, it might seem that powerful automation for inductive proofs, as provided in several systems, would be beneficial. Unfortunately, these methods have generally been developed for rather restricted (e.g., equational or unquantified) logics, and not for the richer context found here. In the absence of suitable automation, the user

may be expected to indicate when induction should be used, and to identify the induction variable or expression (PVS, for example, requires this). It is then relatively straightforward to automate selection and instantiation of the appropriate induction scheme; simple tactics can finish the proof of straightforward lemmas (e.g., those needed here for properties of summations), while more explicit user guidance is be needed in more complex cases (e.g., the main induction here). Many formal methods applications require only a couple of inductions and these simple methods are adequate in these cases. Nonetheless, more automated methods (including those for generalization) would be welcome, and the development of suitable techniques is a good research topic.

3.1 Model Checking

Compared to theorem proving methods, model checking and related techniques (such as state exploration and language inclusion) are becoming rather widely used in formal methods. However, I believe that these techniques currently tend to be used standalone in application domains (such as hardware and protocols) to which they are particularly well-suited, rather than being incorporated into traditional formal methods, or integrated with theorem proving.

For my next example, I describe an experiment undertaken by my colleagues Klaus Havelund and Shankar [7], who applied a combination of finite state exploration, theorem proving, and model checking approaches to a simple protocol. Many larger and more significant problems than this have been examined by finite state enumeration and model checking techniques; what is interesting in this exercise is that it points towards an integration of these techniques with theorem proving, and also highlights some of the areas where further research is needed.

Havelund and Shankar began by reducing the protocol to finite state (by manually assigning explicit small integers as the upper bound on the size of certain data structures) and checking certain safety properties with the Murϕ explicit state exploration system [5]. They next verified these properties for the full protocol by theorem proving in PVS using a traditional invariance argument, but found in the process that the desired invariant had to be strengthened by the addition of many additional conjuncts. These were discovered incrementally during the proof attempt; each new proposed conjunct was checked with Murϕ, added to the invariant, and the evolving proof attempted once more. The whole process was iterated until a sufficiently strong invariant was developed; this eventually comprised 57 conjuncts. Seeking a better approach, they developed a finite-state abstraction of the original protocol, verified (by theorem proving) that it was indeed an abstraction, and then verified properties of the abstraction by model checking.

First, notice that the initial "reduction" to finite state in preparation for examination with Murϕ was a manual and ad-hoc process. This seems typical of finite-state analyses: the original problem is transformed by hand into a form that is acceptable to the available tool. The transformation is usually an aggressive simplification that is adequate for refutation but not for verification—meaning

that bugs found in the transformed description are likely to correspond to bugs in the original, but the failure to detect bugs in the former cannot be interpreted as verification of the latter. In the case of the protocol studied in these experiments, the maximum number of messages in a file was arbitrarily set to three: bugs that are manifest only with larger file sizes will not be found by this method.

Next, the direct verification of the full protocol was extremely tedious, as the desired safety property had to be strengthened iteratively until it became an invariant. This process took many weeks, which is clearly unacceptable for general practice. Methods for the systematic—and preferably automated—development of invariants therefore constitute a very worthwhile research topic. Of course, one of the advantages of model checking is that it is largely automatic and does not require the development of such invariants. However, when model checking is used for verification rather than refutation, it is necessary to prove that the finite-state description is a true abstraction of the original specification, and this abstraction proof may itself require invariants. Havelund and Shankar in fact reused 45 of the 57 invariants developed for their protocol in their abstraction proof, so the overall saving in effort was not great in this case. This experience highlights another very fruitful area for research: systematic and automated methods for developing finite-state abstractions. Good results are already known for some special cases [2] and I speculate that integration of these methods with model checking will eventually provide an efficient way to verify properties of infinite-state systems.

There were interesting differences between the "reduced" finite-state description checked with Murϕ and the "abstracted" version that was model checked. In the reduced Murϕ description, a file could comprise 1, 2, or 3 messages; in the abstracted description, the size of the untransmitted portion of the file is chosen from the uninterpreted enumeration `NONE`, `ONE`, and `MANY`. The relation between these different approaches—fixing the size vs. introducing abstraction (and additional nondeterminism)—is worthy of investigation.

Although these experiments indicate several areas where additional research is needed, they also demonstrate some promising directions. First, use of Murϕ to check the plausibility of proposed new invariants is representative of a useful general technique: testing conjectures using some lightweight technique before undertaking a full proof. In formal methods applications, many conjectures are false when first proposed and it is best to discover these falsehoods as early and as cheaply as possible, reserving the investment in a full proof until some confidence has been developed that it is likely to be successful. Lightweight methods generally apply to specific, or reduced, cases of the full specification, and automated assistance for creating these reduced cases a useful addition to any support environment for formal methods. Apart from finite state enumeration, other lightweight techniques include direct evaluation (for executable specifications), and interactive simulation (for specifications that are not directly executable). The latter methods are usually based on specialized and optimized techniques for automated deduction (e.g., rewriting and enumeration over finite quantifiers).

Second, the combination of theorem proving and model checking in the last of the exercises reported above is representative of a promising direction for integrating powerful, but narrow, techniques into a larger system. For example, model checking in PVS is accomplished using an external decision procedure for Park's μ-calculus. This is extended to a decision procedure for μ-calculus on the hereditarily finite fragment of PVS's type system[6] by encoding their values in propositional variables. The branching time temporal logic CTL is then defined in PVS and its model checking problem is cast as a decision problem in μ-calculus. This allows CTL model checking to be smoothly integrated as a proof procedure in PVS. A benefit of this integration is that model checking is available for any conjecture that has the appropriate semantic attributes, independently of its linguistic representation. For example, a tabular specification construct was recently added to PVS; this was then used to formalize a requirements methodology known as SCR, and model checking was then immediately available for SCR specifications [11].

Interesting challenges for the future are to integrate other highly efficient but narrow procedures into a general purpose framework. Examples include model checking methods for hybrid systems and binary moment diagrams.

4 Interaction with the User

I believe that formal methods can deliver most value when applied to problems where traditional methods are inadequate. All the evidence points two principal sources of failure in complex systems: inadequate understanding of potential interactions, and the intrinsically hard parts of a design. Examples of the former often arise in requirements specification, where it is particularly difficult to anticipate all the interactions among the components of a system and between a system and its environment, particularly when operating in the presence of faults. In the case of Jet-Select, for example, our formalization revealed that certain interactions between error reporting and optimization allowed the possibility of firing a failed jet [6]. Examples of the latter often concern algorithms for concurrent, real time, or fault-tolerant behavior (e.g., cache-coherence or clock-synchronization)—where, again, it is difficult to anticipate all possible interactions—or highly optimized calculations whose correctness rests on a long or complex argument (e.g., SRT division and other efficient floating point algorithms).

A consequence of this observation is that automated deduction in support of formal methods will often be applied to very hard problems. It is, in my view, quite unrealistic to expect that such difficult problems can be solved automatically. The issue, then, is how should the user guide and interact with the process of automated deduction? This raises a dual issue: what information and services can automated deduction provide to the user that will assist in the analysis of very difficult problems?

[6] That is, types built recursively from the Booleans, enumerations, explicit finite subranges of the integers, and records, tuples, predicates, and functions of these.

All interaction between the user and tools for automated deduction can be considered an iteration of the following basic steps. What differs from tool to tool is the relative effort devoted to each step, and the rate of iteration.

1. Decide the procedure to be used at the next step. This can range from coarse decisions of overall strategy ("I'll use SMV") to fine issues of tactics ("instantiate the third variable of formula 3 with the following expression").
2. Transform the current representation of the problem into one that is appropriate for the procedure chosen in the previous step. This may be a major undertaking with pencil and paper (e.g., to reduce an infinite-state protocol specification to a finite-state description in the language of SMV), or it may involve mechanized transformations (including recursive application of this whole activity).
3. Set appropriate switches and dials to tune the selected procedure (e.g., choose a variable ordering for BDDs, a weighting strategy for resolution, or an ordering and orientation of lemmas for Nqthm).
4. Invoke the chosen procedure, contemplate the result returned, and iterate the whole process (sometimes, iterate locally over step 3).

My opinion is that the ability to direct this activity in an efficient and productive manner is largely determined by the predictability of the consequences selected by steps 1 and 3, the quality of information returned in step 4, and the efficiency and repeatability of step 2. The user should be able to select a procedure in step 1 on the basis of a description of what it does, not how it works. Deterministic proof procedures (e.g., elementary transformations such as a case split, or quantifier instantiation) and decision procedures are attractive from this point of view, whereas heuristic procedures are not. By the same token, the switches and dials of step 3 should be minimized, since they generally concern how a proof procedure works, rather than the substance of the conjecture under examination. Few users whose interest is formal methods are willing to learn enough about the workings of a proof procedure that they can master many choices here.

The information returned in step 4 should include the result of applying the proof procedure if it was successful (e.g., "proved," or a list of transformed or new subgoals), and an explanation if it was unsuccessful. Decision procedures and model checkers have a special value in the latter case, because they can often return a counterexample that pinpoints the source of difficulty. The ability to return useful information from failure is particularly important in applications of automated deduction to formal methods because it is to be expected that many conjectures will be false—indeed, the efficient discovery and correction of errors is one of the primary reasons for undertaking formal analysis. For this reason, techniques for automated deduction used in formal methods should not be biased towards successful outcomes—for example, they should not be set up to terminate quickly on success at the expense of taking inordinate time to discover failure.

The whole process of formal analysis will be repeated several times as errors are discovered and the design or its specification are adjusted. But the process

is not over once we successfully get to "proved" for the first time. Mechanization allows formal methods to be used to explore and refine designs—just as computational fluid dynamics is used to refine aerofoils. Our verification of clock synchronization, for example, has been modified many times: to improve the proof, to eliminate assumptions, to change the specification so that it connects better with the formalization of another part of the overall fault tolerant architecture, to tighten the bound on synchronization achieved, and to change from a Byzantine fault model to a more complex "hybrid" model [13].

The fact that formal analysis will be repeated many times as a specification is first debugged and then refined has consequences for automated deduction. First, it makes it essential, in my view, that step 2 of the interaction loop described above be automated: as the design and its specification evolve, we should recalculate the "reduced" form required for a particular proof procedure, rather than tinker with the existing one. In particular, for reliability as well as efficiency, I believe that reductions and abstractions from infinite-state to finite-state models should be formalized and mechanized, rather than left as an ad-hoc manual process. Second, the "script" of a proof needs to be recorded in manner that is reasonably robust to small changes in the specification. This argues against conducting and recording proofs in low-level and highly specific terms (e.g., "instantiate formula 3 with `x!1`" where `x!1` is the name of a Skolem constant), since the details may change with the specification. It will be more robust to indicate a procedure (e.g., "use unification to find an instantiation"), or to invoke truly automated deduction (e.g., "finish off the proof using resolution"). Finally, it is important to record dependencies among proofs and specifications, so that the user can speedily answer questions such as "what assumptions does this proof depend on?" and "what proofs may be affected if I change this lemma?"

5 Conclusion: The Need for Integration

The field of automated deduction has developed many powerful techniques that could be applied to formal methods. However, the special character of formal methods applications means that some techniques may need to be adapted to the needs of those applications, (e.g., to return more useful information on failure) and that priorities may be different than in other areas (e.g., decision procedures become more important and first order methods such as resolution may become less so). More importantly, most techniques in automated deduction, and also those related to model checking, tend to be rather brittle "point solutions" that are effective against specific classes of problems, whereas formal methods requires an integrated capability that is effective across a wide range of applications. The research challenge in this area is therefore broadly that of integration: different techniques must work together, different theories must be decided in combination, theorem proving and model checking must cooperate, and the needs and capabilities of efficient automated deduction must influence, and be influenced by, the design of expressive specification languages. Success in this endeavor will enrich both fields, providing a new and exciting application for

automated deduction, and making formal methods a truly useful and practical tool for the analysis of interesting real systems.

Acknowledgments

My opinions have formed through many stimulating discussions with my colleagues Judy Crow, David Cyrluk, Klaus Havelund, Friedrich von Henke, Patrick Lincoln, Sam Owre, N. Shankar, and M.K. Srivas, and by experiences using PVS (primarily built by Sam Owre and Shankar) and its predecessors.

References

Papers by SRI authors can generally be retrieved from `http://www.csl.sri.com/fm.html`.

[1] R. Alur and T.A. Henzinger, editors. *Computer-Aided Verification, CAV '96*, Lecture Notes in Computer Science, New Brunswick, NJ, July 1996. Springer-Verlag.

[2] S. Bensalem, Y. Lakhnech, and H. Saidi. Powerful techniques for the automatic generation of invariants. In Alur and Henzinger [1].

[3] R. S. Boyer and J S. Moore. Integrating decision procedures into heuristic theorem provers: A case study with linear arithmetic. In *Machine Intelligence*, volume 11. Oxford University Press, 1986.

[4] David Cyrluk, Patrick Lincoln, and N. Shankar. On Shostak's decision procedure for combinations of theories. In *International Conference on Automated Deduction (CADE)96*, Lecture Notes in Computer Science, New Brunswick, NJ, July 1996. Springer-Verlag.

[5] D.L. Dill, A.J. Hu, C.H. Yang, A. Drexler, R. Melton, S. Park, C.N. Ip, and U. Stern. The Murphi verification system. In Alur and Henzinger [1].

[6] David Hamilton, Rick Covington, and John Kelly. Experiences in applying formal methods to the analysis of software and system requirements. In *WIFT '95: Workshop on Industrial-Strength Formal Specification Techniques*, pages 30–43, Boca Raton, FL, 1995. IEEE Computer Society.

[7] Klaus Havelund and N. Shankar. Experiments in theorem proving and model checking for protocol verification. In *Formal Methods Europe FME '96*, number 1051 in Lecture Notes in Computer Science, pages 662–681, Oxford, UK, March 1996. Springer-Verlag.

[8] C. B. Jones, K. D. Jones, P. A. Lindsay, and R. Moore. *mural: A formal Development Support System.* Springer-Verlag, London, UK, 1991.

[9] L. Lamport and P. M. Melliar-Smith. Synchronizing clocks in the presence of faults. *Journal of the ACM*, 32(1):52–78, January 1985.

[10] Mogens Nielsen, Klaus Havelund, Kim Ritter Wagner, and Chris George. The RAISE language, method and tools. *Formal Aspects of Computing*, 1(1):85–114, January–March 1989.

[11] Sam Owre, John Rushby, and Natarajan Shankar. Analyzing tabular and state-transition specifications in PVS. Technical Report SRI-CSL-95-12, Computer Science Laboratory, SRI International, Menlo Park, CA, July 1995. Available, with specification files, from `http://www.csl.sri.com/csl-95-12.html`.

[12] Sam Owre, John Rushby, Natarajan Shankar, and Friedrich von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, February 1995.

[13] John Rushby. A formally verified algorithm for clock synchronization under a hybrid fault model. In *Thirteenth ACM Symposium on Principles of Distributed Computing*, pages 304–313, Los Angeles, CA, August 1994. Association for Computing Machinery.

[14] John Rushby. Mechanizing formal methods: Opportunities and challenges. In Jonathan P. Bowen and Michael G. Hinchey, editors, *ZUM '95: The Z Formal Specification Notation; 9th International Conference of Z Users*, volume 967 of *Lecture Notes in Computer Science*, pages 105–113, Limerick, Ireland, September 1995. Springer-Verlag.

[15] John Rushby and Friedrich von Henke. Formal verification of algorithms for critical systems. *IEEE Transactions on Software Engineering*, 19(1):13–23, January 1993.

[16] Robert E. Shostak. On the SUP-INF method for proving Presburger formulas. *Journal of the ACM*, 24(4):529–543, October 1977.

[17] Mandayam K. Srivas and Steven P. Miller. Formal verification of the AAMP5 microprocessor. In Michael G. Hinchey and Jonathan P. Bowen, editors, *Applications of Formal Methods*, Prentice Hall International Series in Computer Science, chapter 7, pages 125–180. Prentice Hall, Hemel Hempstead, UK, 1995.

[18] Gunnar M. N. Stålmarck. System for determining propositional logic theorems by applying values and rules to triplets that are generated from Boolean formulae. United States Patent 5,276,897, January 4 1994.

A Platform for Combining Deductive with Algorithmic Verification*

Amir Pnueli† Elad Shahar†

Abstract. We describe a computer-aided verification system which combines deductive with algorithmic (model-checking) verification methods. The system, called TLV (for temporal verification system), is constructed as an additional layer superimposed on top of the CMU SMV system, and can verify finite-state systems relative to *linear temporal logic* (LTL) as well as CTL specifications. The systems to be verified can be either hardware circuits written in the SMV design language or finite-state reactive programs written in a simple programming language (SPL).

The paper presents a common computational model which can support these two types of applications and a high-level interactive language TLV-BASIC, in which temporal verification rules, proofs, and complex assertions can be written. We illustrate the efficiency and generality gained by combining deductive with algorithmic techniques on several examples, culminating in verification of fragments of the Futurebus+ system. In the analysis of the Futurebus+ system, we even managed to detect a bug that was not discovered in a previous model-checking analysis of this system.

1 Introduction

As part of the general program for combining deductive with algorithmic methods for the verification of reactive systems (see [Man94] for a declaration of this manifest, and [RSS95] for an important contribution in this direction), we constructed a computer-aided verification system, called TLV (a *Temporal Logic Verifier*), for experimenting with some of these ideas.

Compared to algorithmic verification (model checking), deductive verification is handicapped by the requirement of user interaction, which necessitates a good understanding of the program and a certain degree of creative ability and high skills. Therefore, any proposal for replacing or even combining algorithmic methods with deductive methods must be accompanied by analysis of the expected gains from such a combination.

The main conceived advantages of combining deduction with model checking are:

1. Generality : In the finite-state world (which is the main concern of the work reported here), deductive verification can provide a *uniform* proof which establishes the correctness of a system of N processes for any $N > 0$ in a single

* This research was supported in part by a basic research grant from the Israeli Academy of Sciences, and by the European Community ESPRIT Basic Research Action Project 6021 (REACT).

† Department of Computer Science, Weizmann Institute, Rehovot, Israel, e-mail: amir@wisdom.weizmann.ac.il

proof. In comparison, model checking can only examine the systems for particular values of N.

2. Efficiency of Deduction: Most of the model-checking algorithms are based on computation of the closure of the transition relation, which is applied either to the initial state or to some target states. This is an iterative process that may take a large number of steps to converge. In comparison, in the deductive verification of the same property, we only have to check the two implications

$$\Theta \rightarrow p \qquad \text{and} \qquad \rho \wedge p \rightarrow p',$$

where Θ is an assertion characterizing the initial condition and ρ is the transition relation. It stands to reason that checking these implications takes less time and requires smaller BDDs than the iterative computation of the closure.

3. Constrained model checking : A possible way of combining deduction with model checking is to use deduction to establish the invariance of an assertion φ. Then, we can carry out regular model checking but use φ to restrict the range of considered states. This amounts to model checking with the transition relation $\varphi \wedge \rho$ instead of the original ρ.

The (TLV) system described here has been constructed on top of the CMU SMV system, which supports verification of CTL specifications of finite-state systems ([BCM+92], [McM93]). TLV uses the BDD library and the SMV input language parser from SMV. The model checking algorithms were replaced by a layer which consists of a high-level interactive language, to which we refer as TLV-BASIC. The main data structure of TLV-BASIC is a quantifier-free assertion, obeying the SMV syntax for state-formulas, and represented internally by a BDD.

The TLV-BASIC language is used for three purposes:

- Temporal verification rules, such as the basic invariance rule BINV and the single-step response rule RESP, as well as algorithms for model-checking invariance and response properties, are written as TLV-BASIC procedures.
- For each particular system to be verified, the user usually prepares a *proof script* file which contains definitions of the assertions used in the property to be verified.
- The interactive dialog with the user is carried out in a restricted subset of TLV-BASIC.

The main running example and one of the motivating drives for our system is the Futurebus+ system considered in [CGH+93]. That paper presented an SMV model for the Futurebus+ system and established several properties of the model, using the model-checking techniques of SMV. We considered it an interesting challenge to see whether the same properties can be verified using deductive techniques, and compare the efficiency and effectiveness of the two methods.

At its current state of implementation, the TLV system cannot yet consider variable-size systems where the system size is not fixed at analysis time. Therefore, we cannot yet demonstrate uniform proofs of such parameterized systems, and all the examples presented in this paper relate to specific values of the size parameter. To compensate for this temporary deficiency, we developed methods

by which the deductive proof of a parametric system can be parameterized itself, so that running a deduction for different values of the size parameter n only requires modifying a line in the proof script file from "$n = 20$" to, say, "$n = 40$." In particular, we developed a special format by which one can specify an arbitrary configuration of Futurebus+ and generate automatically the proof appropriate for this configuration. Details about these instantiation mechanisms are given in [PS96].

Many approaches to the deductive verification of reactive systems and hardware circuits were proposed over the years, accompanied by systems supporting their automation. Examples of applications for hardware verification are the methods described in [Gor86] and [ORSS94]. An effective system for the deductive verification of linear temporal logic properties of reactive programs is reported in [MAB+94].

There have been also several approaches which combine deductive and algorithmic verification methods. The work in [JS93] combines the HOL theorem prover with the Voss system. Another combination of methodologies is reported in [KL93], where TLP, the proof checker for TLA, the temporal logic of actions, is combined with the COSPAN verifier. Perhaps closest to our work is [RSS95] which embeds symbolic model-checking into the PVS high-order prover.

The unique feature of our approach is that it is built as the minimal extension of an existing symbolic model checking system (SMV) needed in order to handle parametric systems. The specification language and associated deductive verification approach are based on linear temporal logic [MP95]. At present, the only deductive machinery we employ is provided by the BDD capabilities of the underlying SMV system.

The rest of the paper is organized as follows. In Section 2 we present the underlying computational model and its relation to the FTS model of [MP95]. In Section 3, we describe the languages that can serve as inputs to the TLV system. These include the TLV-BASIC language in which verification rules, model-checking procedures, and proof scripts are written; the SMV input languages used to specify systems; and the SPL language used to describe simple reactive programs [MP95]. In Section 4, we present some of the verification rules supported by the system. Section 5 presents several simple examples of deductive and combined verification, comparing their efficiency with standard model-checking verification of the same properties. In Section 6, we present our main case study, the Futurebus+ verification, and identify the bug that has escaped previous model-checking analysis.

2 The Computational Model

As an underlying computational model, we adopt the notion of an *always-enabled fair transition system* (ETS). The ETS model can be viewed as a variant of the *fair transition system* (FTS) model, introduced in [MP91] for the specification and verification of reactive systems. An ETS consists of the following components:

- $\mathcal{V}$ — A finite set of *state variables*. We define a state to be an interpretation of $\mathcal{V}$. The set of all states is denoted by Σ.

- Θ — Initial condition.
- $\mathcal{T}$ — A finite set of *transitions*. Each transition $\tau \in \mathcal{T}$ is a function
  $\tau : \Sigma \mapsto 2^{\Sigma} - \emptyset$,
  mapping a state s to $\tau(s) \subseteq \Sigma$, a non-empty set of τ-*successors* of s.
- $\mathcal{J} \subseteq \mathcal{T}$ — A *justice* (weak fairness) set of transitions.
- $\mathcal{C} = \{(\tau_1, \varphi_1), \ldots, (\tau_k, \varphi_k)\}$ — A *compassion* (strong fairness) set of pairs (τ_i, φ_i), $i = 1, \ldots, k$, each consisting of a transition τ_i and an assertion φ_i.

The requirement that every state has a non-empty set of successors implies that every transition is enabled on every state.

A *model* is an infinite sequence of states. Given an ETS Φ, we define a *computation of* Φ to be a model

$$\sigma : s_0, s_1, s_2, \ldots,$$

satisfying the following requirements:

- *Initiation*: s_0 is an initial state (i.e it satisfies Θ).
- *Consecution*: For each pair of consecutive states s_i, s_{i+1} in σ, there exists a transition τ in $\mathcal{T}$ such that $s_{i+1} \in \tau(s_i)$. That is, s_{i+1} is a τ-successor of s_i.
- *Justice*: Every transition $\tau \in \mathcal{J}$ is taken infinitely many times.
- *Compassion*: For every $(\tau_i, \varphi_i) \in \mathcal{C}$, if φ_i holds at infinitely many positions in σ then τ_i is taken at φ_i-positions infinitely many times.

The main differences between the FTS and ETS models are the ETS requirement that transitions be always enabled, and the implications this requirement has on the requirements of justice and compassion.

The reason for this difference is that the natural SMV representation of transition relations, in particular those which result from SPL programs, is such that the transition can always be taken. Under the circumstances in which the corresponding FTS transition would be disabled, the ETS transition is still enabled but has no effect on the system variables, i.e., it changes the value of no system variable.

An FTS Φ is called a *leisurely fair transition system* (LFTS), if the idling transition τ_I is contained in the justice set of Φ. Thus, every computation of an LFTS contains infinitely many idling steps, i.e. steps which preserve the values of all system variables. Obviously, every FTS Φ has a corresponding LFTS Ψ, such that Φ and Ψ are equivalent up to stuttering.

The following claim shows that no expressive power is lost in moving from the FTS model into the ETS model.

Claim 1 *A set of models S is the set of computations of an* ETS Φ *iff it is the set of computations of some* LFTS Ψ.

In [PS96], we provide a proof of this claim.

3 The Languages of TLV

3.1 The SMV Input Language

Systems to be verified by TLV are described using the SMV input language [McM93], which has been slightly extended to allow for the richer set of fairness

requirements associated with the ETS model. In Fig. 1, we present file **sem.smv**, which contains the SMV description of a mutual exclusion algorithm MUX-SEM, which implements mutual exclusion by semaphores. Note that, standardly in our

```
MODULE main
VAR
  y : boolean; -- the semaphore variable. It is assigned by both processes.
  proc[1] : process user(y); -- The two processes have interleaved execution.
  proc[2] : process user(y);
ASSIGN
  init(y) := 1;
MODULE user(y)
VAR
  loc : {0,1,2,3,4};
ASSIGN
  init(loc) := 0;
  next(loc) :=
    case
      loc in {0,3}     : loc+1;
      loc = 1          : {1,2};
      loc = 2 & y = 1  : 3;
      loc = 4          : 0;
      1 : loc;
    esac;
  next(y) :=                          -- changes to the semaphore variable.
    case
      loc = 2 & next(loc) = 3 : 0; -- turned off when moving from l_2 to l_3
      loc = 4 & next(loc) = 0 : 1; -- turned on when moving from l_4 to l_0
      1 : y;
    esac;
JUSTICE
  proc[1], proc[2];
COMPASSION
  (proc[1],proc[1].loc = 2 & y > 0), (proc[2],proc[2].loc = 2 & y > 0)
```

Fig. 1. File **mux-sem.smv**: an SMV description of Algorithm MUX-SEM for $n = 2$ processes.

applications, we do not use the **FAIR** or **SPEC** declarations but introduce instead **JUSTICE** or **COMPASSION** declarations, wherever necessary.

Such an SMV specification is input into the TLV system which creates internally the ETS corresponding to the specification. In general, there will be one ETS transition for each process. Thus, in the **mux-sem.smv** example, the system will generate an ETS with two transitions, one corresponding to each process. The justice requirement requests that each of the two processes will be activated infinitely many times in every computation of this ETS.

3.2 The SPL Input Language

While direct coding of hardware circuits in the SMV input language is a practice to which experienced users of the SMV system have resigned themselves, we can offer a higher description level for applications to reactive programming. To

represent reactive programs, we adopted the *simple programming language* (SPL) introduced in [MP91]. We refer the reader to [MP91] or [MP95] for details of this language. In Fig. 2, we present an SPL program for the MUX-SEM algorithm.

Here, we consider the instance $n = 2$ of this generic program. On reading the SPL file with the additional definition $n := 2$, the system translates it first into the SMV representation, presented in Fig. 1.

$$\begin{array}{l} \textbf{in} \quad n : \textbf{integer where } n > 0 \\ \textbf{local } y : \textbf{integer where } y = 1 \end{array}$$

$$\Big\|_{i=1}^{n} C[i] :: \left[\begin{array}{l} \ell_0 : \textbf{loop forever do} \\ \left[\begin{array}{l} \ell_1 : \textbf{noncritical} \\ \ell_2 : \textbf{request } y \\ \ell_3 : \textbf{critical} \\ \ell_4 : \textbf{release } y \end{array}\right] \end{array}\right]$$

Fig. 2. Program MUX-SEM (mutual exclusion by semaphores - general case).

3.3 TLV-BASIC

The TLV-BASIC language is easy to learn and simple to program with. It is used to program rules, model-checking algorithms, and compute assertions. The main (and only) data structure is a function with boolean arguments and integer range. As such, it can represent integers, booleans (a function with range $\{0, 1\}$, and assertions, which are represented as boolean functions. The underlying implementation is a BDD, which is manipulated using the SMV BDD library. Expressions in the language are constructed out of integer constants and variables to which we apply integer operations, integer comparisons, and all the boolean and quantifying operators available in the SMV language.

There are no variable declarations. Like BASIC, variables are created dynamically, whenever they are assigned values, or mentioned as parameters of a procedure. In addition, all the variables defined in an SMV input file which is loaded into the system can be referenced within TLV-BASIC expressions.

Following are some of the statements available in TLV-BASIC:

- `Let` *var* `:=` *exp* — Assign the value of expression *exp* to variable *var*.
- `Proc` *proc-name* $(par_1, \ldots, par_n)$; S `End` — Define a procedure *proc-name* with parameters $par_1, \ldots, par_n$ and body S.
- `While (`*exp*`)` S `End` — Repeatedly execute statement S until *exp* is 0.
- `If (`*exp*`)` S_1 [`else` S_2] `End` — If *exp* evaluates to a non-zero value, execute statement S_1. Otherwise, execute statement S_2.
- `Load "`*file-name*`"` — Load file *file-name* into the system. The loaded file can be a rules file or a proof script file.
- `Run` *proc-name* $par_1, \ldots, par_n$ — Invoke procedure *proc-name* with the given actual parameters.

The last two statements are the main commands that are used in interactive mode.

In Fig. 3 we present a TLV-BASIC proof script which computes the assertion

$$\texttt{mux:}\quad \bigwedge_{i=1}^{n}\bigwedge_{j=1}^{i-1}\neg(\texttt{proc[i].loc} = 3\ \&\ \texttt{proc[j].loc} = 3).$$

for $n = 10$. This assertion specifies mutual exclusion for program MUX-SEM. When we consider the same program for a different number of processes, say 11,

```
Let n := 10;
Proc prepare;
  Let mux := TRUE;
  Let i := n;
  While (i)
    Let j := i - 1;
    While (j)
      Let mux := mux & !(proc[i].loc = 3 & proc[j].loc = 3);
      Let j := j - 1;
    End -- end loop on j
    Let i := i - 1;
  End -- end loop on i
End -- end procedure

Run prepare
```

Fig. 3. File `mux-sem.pf`: Proof script for program MUX-SEM for $n = 10$.

it is only necessary to change the first statement in this file to `Let n := 11`.

4 Verification Rules

The TLV system comes equipped with a set of deductive verification rules as well as various model-checking algorithms. As previously explained, these rules are implemented using the TLV-BASIC language. This means that a sophisticated user can easily modify any of the existing rules, as well as write new ones.

In Fig. 4, we present the two verification rules that have been used for verifying the examples presented in this paper.

$$\frac{\begin{array}{ll}\text{B1}: \Theta \rightarrow p & \\ \text{B2}: \rho_\tau \wedge p \rightarrow p' & \forall \tau \in \mathcal{T}\end{array}}{\Box p}$$

Rule BINV

$$\frac{\begin{array}{l}\text{A1}: \varphi \wedge \delta = 0 \rightarrow q \\ \text{A2}: (\varphi \wedge \neg q) \rightarrow \exists \tau \in \mathcal{T}\, \exists V' (\rho_\tau \rightarrow \delta \succ \delta')\end{array}}{\textbf{AG EF}\ q}$$

Rule AGEF

Fig. 4. Verification rules.

5 Simple Verification Examples

In this section we illustrate the use of the TLV system for the verification of several simple examples taken from [MP95].

Program MUX-SEM

In Fig. 2, we presented the general MUX-SEM program parameterized by n. Fig. 1 illustrated its SMV translation for the case $n = 2$. The main safety property of this program can be specified by the invariance of assertion **mux** presented above.

Direct application of rule BINV failed (and produced a counter-example). According to the terminology of [MP95], this means that assertion **mux** is invariant but not *inductive*, i.e., it does not carry sufficient information to rule out inaccessible states. The standard remedy is to *strengthen* assertion **mux** by additional invariants, which will exclude such states.

Indeed, our next step in the verification process, was to formulate the auxiliary invariant assertion

$$\texttt{phi:}\quad \texttt{y} \quad <-> \quad \bigwedge_{i=1}^{n} \neg(\texttt{proc[i].loc in } \{3,4\})$$

Application of rule BINV to the conjunction **mux & phi** succeeded which established the invariance of both **mux** and **phi** over program MUX-SEM.

This experimentation was carried out for the low value of $n = 2$. However, once the strategy was established we prepared a proof script for computing the conjunction **mux & phi** and can now run the verification for various values of n, changing only the value of the parameter between successive runs.

To compare the time and space complexity of conventional model checking and the deductive approach, we plot in Fig. 5 the time and space complexity of verifying the invariance of assertion **mux** by the two approaches for increasing number of processes in program MUX-SEM. The line labeled SMV represents the conventional model-checking approach, while the line labeled TLV represents the deductive approach.

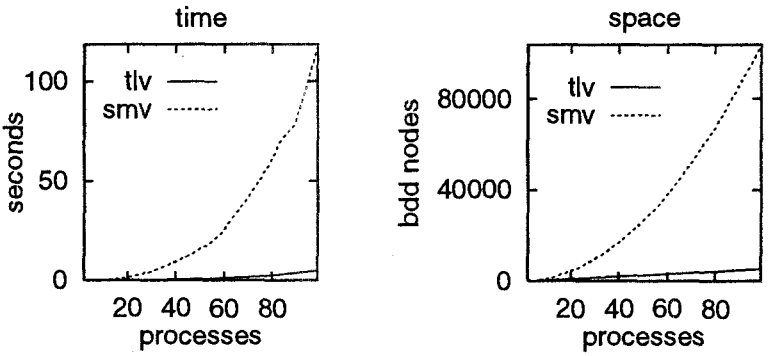


Fig. 5. Comparison of SMV and TLV for MUX-SEM

Program RES-SV

As the next example, we considered program RES-SV, presented in Fig. 6. Program RES-SV consists of an alloator process A and customer processes $C[i]$, $i = 1, \ldots, n$. The allocator provides a centralized control which is exptected to guarantee mutual exclusion between the customers. We refer readers to [MP95] for

$$\begin{array}{l}\textbf{in} \quad n \ : \textbf{integer where } n > 0\\ \textbf{local } g, r : \textbf{array } [1..n] \textbf{ of boolean where } g = \mathbf{F},\ r = \mathbf{F}\end{array}$$

$$A :: \left[\begin{array}{l}\textbf{local } t : \textbf{integer where } t = 1\\ m_0 : \textbf{loop forever do}\\ \left[\begin{array}{l} m_1 : \textbf{if } r[t] \textbf{ then}\\ \left[\begin{array}{l} m_2 : g[t] := \mathbf{T}\\ m_3 : \textbf{await } \neg r[t]\\ m_4 : g[t] := \mathbf{F}\end{array}\right]\\ m_5 : t := t \oplus_n 1\end{array}\right]\end{array}\right] \quad \| \quad \|_{i=1}^{n} C[i] :: \left[\begin{array}{l}\ell_0 : \textbf{loop forever do}\\ \left[\begin{array}{l}\ell_1 : \textbf{noncritical}\\ \ell_2 : r[i] := \mathbf{T}\\ \ell_3 : \textbf{await } g[i]\\ \ell_4 : \textbf{critical}\\ \ell_5 : r[i] := \mathbf{F}\\ \ell_6 : \textbf{await } \neg g[i]\end{array}\right]\end{array}\right]$$

Fig. 6. Program RES-SV (resource allocator).

details of this algorithm and its verification. Here we set to ourselves the more modest goal of verifying mutual exclusion between customers $C[1]$ and $C[2]$ in a system of $n \geq 2$ customers.

This property can be specified as the invariance of the assertion

$$\texttt{mux}: \quad \neg(at\_\ell_4[1] \wedge at\_\ell_4[2]),$$

where, for any i and j, $at\_\ell_i[j]$ stands for `C[j].loc = i`.

As in the previous case, assertion `mux` is an invariant of the program but is not inductive. To complete the proof, we used six strengthening assertions for $i \in \{1, 2\}$. The first two assertions of this set are:

$$\begin{array}{ll}\varphi_1[i] : & at\_m_{3,4} \wedge t = i \leftrightarrow g[i]\\ \varphi_2[i] : & at\_\ell_{3..5} \quad \leftrightarrow r[i]\end{array}$$

Using these strengthening invariants, assertion `mux` has been proven an invariant of program RES-SV.

In Fig. 7, we plot the time and space complexity of verifying the invariance of assertion `mux` over program RES-SV as a function of the number of processes. Again, the conventional model checking and deductive approaches are compared.

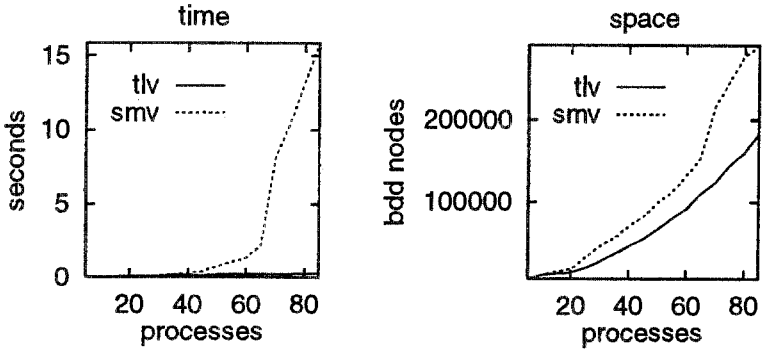


Fig. 7. Comparison of SMV and TLV for RES-SV

Constrained Model Checking

In addition to the purely deductive approach, we also implemented and tested a mixed (or combined) approach, in which we use deductively derived invariants to

restrict the range of the transition function in computing the backwards closure, usually employed in model checking for invariance properties.

We considered again program RES-SV but used the deductive approach to verify only the two first invariants in the list: $\varphi_1[i]$ and $\varphi_2[i]$. These are very simple invariants, which can be discovered automatically by various heuristics (as explained in [MP95]). At this point we ceased using deductive methods, and invoked a special model-checking procedure CMCINV, written in TLV-BASIC, with a constraint parameter, which is the conjunction of $\varphi_1[i]$ and $\varphi_2[i]$. This procedure performs regular backwards closure computation, but eliminates all states which do not satisfy the given constraint.

In Fig. 8, we present plots of time and space complexity which compare regular model checking with constrained model checking for program RES-SV. The line representing constrained model checking is labeled by CMC, as compared to regular model checking which is labeled by MC. Both were performed by appropriate TLV-BASIC procedures.

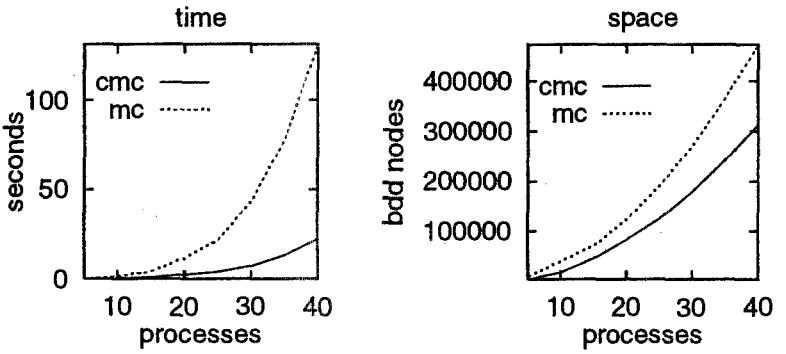


Fig. 8. Comparison of Model Checking and Constrained Model Checking for RES-SV

6 Verification of the Futurebus+

The IEEE Futurebus+ protocol specification is a technology-independent protocol for single-bus and multiple-bus multiprocessor systems. Part of this standard is the cache coherence protocol designed to work in a hierarchically structured multiple-bus system. Coherence is maintained by having the caches observe all bus transactions. Coherence across buses is maintained using bus bridges. A bus bridge is a memory agent/cache agent pair, each of them on a different bus, which can communicate. The memory agent represents the memory on its bus. The cache agent represents all the *remote caches*, caches on the bus of the corresponding memory agent, which may need to get access to the cache line via the bus bridge.

The protocol defines various transactions which let caches on a bus obtain readable and writable copies of *cache lines*. A cache line is a series of consecutive memory locations that is treated as a unit for coherence purposes.

We refer the reader to [CGH+93] for additional explanations and details about the SMV coding of the Futurebus+.

6.1 Specifying and verifying Cache Coherence

The following specifications are the ones which were proved in [CGH+93]. We repeated their verification, using deductive methods. There are four classes of safety properties and one for liveness.

The first class of safety properties is used to check that no device ever observes an illegal combination of bus signals or an unexpected transaction. Thus, we have the following formulas for every device d:

$$\mathbf{AG}\ \neg d.bus\text{-}error \qquad\qquad \mathbf{AG}\ \neg d.error$$

If these formulas are true then we say that the model is *error free.*

The *exclusive write* property states that if a cache has an exclusive modified copy of some cache line, then no other cache has a copy of that line. The specification includes the formula

$$\mathbf{AG}\ (p1.writable \rightarrow \neg p2.readable).$$

for each pair of caches $p1$ and $p2$. $p1.writable$ is true when $p1$ is in the exclusive-modified state. Similarly, $p2.readable$ is true when $p2$ is not in the invalid state.

The *consistency* property requires that if two caches have copies of a cache line, then they agree on the data in that line:

$$\mathbf{AG}\ (p1.readable \wedge p2.readable \rightarrow p1.data = p2.data)$$

The *memory consistency* property is similar to the consistency property. It specifies that any cache line that has a readable copy must agree with the memory device on the data.

$$\mathbf{AG}\ (p1.readable \wedge \neg m.memory\text{-}line\text{-}modified \rightarrow p1.data = m.data)$$

There is only one class of liveness specifications. It is used to check that it is always possible for a cache to get read and write accesses to a line. In a sense, it says that the model does not get stuck.

$$\mathbf{AG}\ \mathbf{EF}\ p.readable \qquad\qquad \mathbf{AG}\ \mathbf{EF}\ p.writable$$

All these properties were verified for small configurations, using deductive methods. We refer the reader to [PS96] for details of the inductive assertions that were used.

6.2 A Bug was Found

During our verification process, we came across a bug which seems to have escaped the attention of the previous verifiers of this design. In all probability, this is due to the fact that they have not considered the particular configuration in which this particular bug was lurking. We managed to prove the specifications for this configuration after fixing this bug.

The bug is manifested under the following circumstances. Consider a bus with a memory agent and three processors. We start from a reachable state where all processors have a shared copy of the cache line and the memory agent is in the `remote-shared-unmodified-invalid` state which indicates that the current bus has shared copies on it but the memory agent itself does not have a copy. Suppose that process p1 wants an exclusive copy of the cache line. It issues an invalidate transaction on the bus, which tells all other caches to release

their copies of the cache line. However, the other two processors, p2 and p3, choose to split the request so they continue to hold a shared copy but they each owe a response. Eventually, p2 responds and enters an invalid state. The memory agent observes this and enters the **remote-exclusive-modified** state. This means that the memory agent thinks that p1 already has an exclusive-modified copy but, in fact, p1 and p3 still hold shared copies. When p3 issues a response the memory agent sets on the error flag since, if only one process has a copy of the cache line, no other process should owe a response indicating a release of its hold on a shared copy.

References

[BCM+92] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, 1992.

[CGH+93] E.M Clarke, O. Grumberg, H. Hiraishi, S. Jha, D.E. Long, K.L. McMillan, and L.A. Ness. Verification of the Futurebus+ cache coherence protocol. In L. Claesen, editor, *Proc. of the 11th Int. Symp. on Computer Hardware Description Languages and their Applications*. North-Holland, April 1993.

[Gor86] M. Gordon. Why higher-order logic is a good formalism for specifying and verifying hardware. In *Formal Aspects of VLSI Design*, pages 153–177. Elsevier Science Publishers (North Holland), 1986.

[JS93] J.J Joyce and C.-J.H. Seger. Linking BDD-based symbolic evaluation to interactive theorem proving. In *Proc. of the 30th Design Automation Conf.*. ACM, 1993.

[KL93] R. Kurshan and L. Lamport. Verification of a multiplier: 64 bits and beyond. In C. Courcoubetis, editor, *Proc. of 5th CAV*, volume 697 of *Lect. Notes in Comp. Sci.*, pages 166–179. Springer-Verlag, 1993.

[MAB+94] Z. Manna, A. Anuchitanukul, N. Bjørner, A. Browne, E. Chang, M. Colón, L. De Alfaro, H. Devarajan, H. Sipma, and T. Uribe. STeP: The Stanford Temporal Prover. Technical Report STAN-CS-TR-94-1518, Dept. of Comp. Sci., Stanford University, Stanford, California, 1994.

[Man94] Z. Manna. Beyond model checking. In D. L. Dill, editor, *Proc. of 6th CAV*, volume 818 of *Lect. Notes in Comp. Sci.*, pages 220–221. Springer-Verlag, 1994. Invited talk.

[McM93] K.L McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, Boston, 1993.

[MP91] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer-Verlag, New York, 1991.

[MP95] Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems: Safety*. Springer-Verlag, New York, 1995.

[ORSS94] S. Owre, J.M. Rushby, N. Shankar, and M.K. Srivas. A tutorial on using PVS for hardware verification. In R. Kumar and T. Kropf, editors, *Proc. of the 2nd Conf. on Theorem Provers in Circuit Design*, pages 167–188. FZI Publication, Universität Karlsruhe, 1994. Preiminary Version.

[PS96] A. Pnueli and E. Shahar. The TLV system and its applications. Technical report, The Weizmann Institute, 1996.

[RSS95] S. Rajan, N. Shankar, and M.K. Srivas. An integration of model checking with automated proof checking. In P. Wolper, editor, *Proc. of 7th CAV*, volume 939 of *LNCS*, pages 84–97. Springer-Verlag, 1995.

Verifying Invariants Using Theorem Proving

Susanne Graf and Hassen Saïdi

Verimag *
{graf,saidi}@imag.fr

Abstract. Our goal is to use a theorem prover in order to verify invariance properties of distributed systems in a "model checking like" manner. A system S is described by a set of sequential components, each one given by a transition relation and a predicate *Init* defining the set of initial states. In order to verify that P is an invariant of S, we try to compute, in a model checking like manner, the weakest predicate P' stronger than P and weaker than *Init* which is an *inductive* invariant, that is, whenever P' is true in some state, then P' remains true after the execution of any possible transition. The fact that P is an invariant can be expressed by a set of predicates (having no more quantifiers than P) on the set of program variables, one for every possible transition of the system. In order to prove these predicates, we use either automatic or assisted theorem proving depending on their nature.
We show in this paper how this can be done in an efficient way using the Prototype Verification System PVS. A tool implementing this verification method is presented.

1 Introduction

Using a theorem prover to do model checking is not a new idea[2]. Theorem proving has been used successfully for the verification of temporal logic formulas on programs, specially systems like [BM88], [OSR93a][3], [GM93] and [CCF+95].

In most of these approaches, it is mainly emphasized how to define the syntax of a specification formalism and its semantics (in terms of sets of computations) as well as the satisfaction of temporal logic formulas on computations. Then, a system S satisfies a property f if every computation of S satisfies f. In general, not much is told about *how to verify* the obtained formulas.

[RSS95] explains how model checking (for *finite* state systems) is implemented in PVS as a tactic (which consists in transforming the model checking problem into a decidable μ-calculus formula and to run a decision procedure on this formula). In [RSS95], [DF95] and [HS96] model checking is used to prove abstract descriptions of systems, while "ordinary" theorem proving is used to show the

* Verimag is a joint laboratory of CNRS, Institut National Polytechnique de Grenoble, Université J. Fourier and Verilog SA associated with IMAG.

[2] See [RSS95], where a set of combination attempts are mentioned.

[3] see [CLN+95] for many examples of the use of PVS.

correctness of this abstract description with respect to a more concrete (in general infinite state) description. In [Hun93] it is proposed to verify the correctness of each component using model checking, and then to deduce the correctness of the composed system by means of compositional rules embedded as inference rules in a theorem prover. [BGMW94] describes an integration of the PVS theorem prover in an environment for the verification of hardware specification. It is used for discharging verification conditions expressing the fact that a specification simulates another.

1.1 Our approach

Our intention is not to verify arbitrary temporal logic formulas, but particular formula schematas corresponding to useful property classes. In order to prove that a system S satisfies a property expressed by a temporal formula f, we do not use its semantics, but a proof rule generating a set of first order logic formulas (without temporal modalities and without new quantifiers) such that their validity is sufficient to prove that S satisfies f. Here, we mention only safety properties expressible by formulas of the form "$\Box P$" (invariants) or "$\Box(P \Rightarrow P_1 \mathcal{W} P_2)$", where P, P_1, P_2 are predicates[4].

For example, in order to prove that P is an invariant of S ($S \models \Box P$) — where S is defined by a set T of transitions and a predicate $Init$ defining the set of initial states — it is necessary and sufficient to find a predicate P' weaker than $Init$ and stronger than P which is an inductive invariant, that is P' is preserved by any computational step of S, i.e $P' \Rightarrow \widetilde{pre}[\tau](P')$[5] is valid for each transition τ of T. Model checking consists in computing iteratively the weakest predicate satisfying the implication $Q \Rightarrow \widetilde{pre}[T](Q)$ starting with $Q_0 = P$ and taking $Q_{i+1} = Q_i \wedge \widetilde{pre}[T](Q_i)$ that is by strengthening the proposed solution at each step. This method can be completely automatized under the condition that the above predicates are decidable. However, in the case of infinite state systems convergence is not guaranteed, and in real life systems with this very simple tactic, convergence is too slow, anyway. Convergence can be accelerated by replacing the predicate transformers $\widetilde{pre}[\tau]$ by some (lower) approximation or by using structural invariants (see Section 4.3) extracted from the program obtained by constant propagation, variable domain information, etc. Theorem proving (or an appropriate decision procedure) is used for establishing $Q_i \Rightarrow Q_{i+1}$ that is for verifying that a fixed point has been reached.

1.2 Related work

Tools like STeP [MAB+94], TPVS [BLUP94] and CAVEAT [GR95] use this technique. In CAVEAT systematic strengthening of invariants is not foreseen. STeP

[4] in [MP95] many such schemata and corresponding verification rules are presented for which we will implement strategies in the future

[5] The state predicate $\widetilde{pre}[\tau](P)$ defines the smallest set of states that via the transition τ have only successors satisfying P.

provides a lot of automatization and implements most of the rules presented in [MP95].

In [HS96], a new strengthening method has been proposed, in order to avoid the fast growth of the formulas due to the systematic strengthening: suppose that Q_i is not inductive for some transition τ, that is, the proof of the goal $Q_i \Rightarrow \widetilde{pre}[\tau](Q_i)$ does not reduce to *true* but to some formula R. Then, instead of checking in the next step the formula $Q_{i+1} = Q_i \wedge \widetilde{pre}[\tau](Q_i)$, it is proposed to check R (which is often simpler) for invariance. However, this method does not accelerate convergence.

This paper is organized as follows: in Section 2, we recall some general ideas concerning theorem proving and give a small overview on PVS. In Section 3, it is explained how to define our method completely within PVS and also, why we have abandoned this approach. Finally, in Section 4, we give a short presentation of our tool which acts like an interface with PVS. In Section 5, we demonstrate our method and tool on two examples: a finite state program implementing a mutual exclusion algorithm, and an infinite state program implementing a simple buffer using lists as data type.

2 The theorem proving paradigm

Theorem proving is the paradigm of developing and verifying mechanically mathematical proofs. The specification languages used (higher order logic) allow to define usual mathematical objects such as sets, functions, propositions and even proofs[6], and can be generally understood as a mixture of predicate calculus, recursive definitions à la ML and inductively defined types. These languages are strong enough to model systems and express properties on them. Theorem provers provide an interactive environment for developing mathematical proofs using a set of tactics (elementary proof steps) and tacticals (combination of tactics). Possible tactics are implementations of either a deduction rule, rewriting rule, induction scheme or a decision procedure.

PVS

PVS is an environment for writing specifications and developing proofs. It consists of a specification language integrated with a powerful and highly interactive theorem prover. PVS uses higher order logic as a specification language, the type system of PVS includes uninterpreted types, sub-typing and recursively defined data-types. Four "sorts" characterize this language: **Theory**, **Type**, **Expression** (*term*), **Formula** (*proposition*). Any PVS specification is structured into parameterized theories. A **Theory** is a set of **Type**, variable, constant, function and **Formula** declarations. The PVS theorem prover implements a set of powerful tactics with a mechanism for composing them into proof tacticals. The tactics available are combinations of deduction rules and decision procedures. Some of

[6] See [CCF+95] for this purpose.

these tactics such as *assert* and *bddsimp* invoke efficient decision procedures for arithmetic and boolean expressions. PVS has emacs as user interface.

3 Specification and verification within PVS

One of the drawbacks when using theorem provers is the tedious encoding of semantics and writing of specifications. In Coq [CCF+95], grammar extension is allowed which makes specifications easier to write and to read.

In PVS, this technique can be generalized to allow user-defined specification syntax (e.g. [Saï95]). The defined specification syntax can be a combination of the PVS specification syntax and user specification syntax since it can be constructed using non-terminals of the PVS grammar.

To prove that a predicate is an invariant of a system is usually done by embedding the semantics of transition systems and the notion of invariance of a property in the specification language of a theorem prover. In PVS, this can be done by means of the following definitions:

```
Program [State : TYPE] : THEORY
BEGIN
  Action : TYPE = [guard:bool, assignments:State]
  System : TYPE = [vars:State, acts:list[Action], init:bool]

  is-inductive? (S:System, P:Pred[State]) : bool =
    (init(S)  =>  P(vars(S)))          AND
    (P(vars(S)) => WPC-System(acts(S),P))

  WPC-System(L:list[Action], P:Pred[State]) : RECURSIVE =
    CASES L of
      null          : TRUE
      cons(act,rest): WPC-Action(act,S) AND WPC_System(rest,P)
    END CASES

  WPC-Action(act:Action, P:Pred[State]) : bool =
    guard(act) => P(assignments(act))
END Program
```

The PVS theory named **Program** is parameterized by the type **State** defining the tuple type of the state vector, that means, its i^{th} component defines the type of the i^{th} state variable. **System** is given as list of actions, where **Action** is defined as a record type with two fields, a guard and an assignment. **guard** is the condition under which the given action is activated. **assignments** is a tuple of type **State** representing the new value of the state vector after the execution of the given action. The predicate **is-inductive?** taking as arguments a system S and a predicate P, yields the result **true** if P is an inductive invariant of S.

In order to show that P is an invariant of S, we have to prove the following obligation:

```
prove-invariant : OBLIGATION
 EXISTS (P':PRED[State]):
  (FORALL (t: State) : (P'(t) => P(t)) AND is-inductive?(S,P'))
```

This proof obligation does not tell us how to find a satisfactory predicate `P'`. This is the reason why we use the iterative computation described in Section 1.1 which replaces the above (second order) obligation by an (infinite) suite of first order obligations such that the proof of any obligation of this suites validates the initial obligation.

But we found that such an embedding of the semantics of transition systems directly in PVS is still not satisfactory for the verification of large systems. Writing programs is tedious, proofs are very slow since much time is lost in expanding the definitions of `is-inductive?`, `WPC-System` and `WPC-Action`. We also found that we cannot perform static analysis on programs written in this way.

Therefore, we prefered to describe programs in a more natural way and not to translate them into a PVS theory, but just to generate automatically proof obligations equivalent to `is-inductive?(S,P')` and to submit them to the PVS proof checker.

4 A verification tool

Figure 1 shows the architecture of our tool for computer-aided verification. We first present how systems are described in this tool and how the verification process works. We also show how both specification and verification are connected with the PVS system.

4.1 A specification formalism

In our tool, systems are discribed in a formalism close to Dijkstra's language of guarded commands. In fact, a system is defined as a set of components where each component is given by a set of transitions defining conditional data transformations, where program variables are of any data type definable in PVS and allowed value expressions are any expressions definable in PVS. The grammar defining this specification formalism is the following[7]:

system ⇒ *id_system* [**PARAMETER** *id*] : **SYSTEM**
BEGIN
⟨ *pvs_declarations* ⟩
BEGIN
⟨ *sys_components* ⟩
END
INITIALLY : ⟨ *pvs_boolean_formula* ⟩
END *id_system*

[7] This grammar is presented using the conventions of [OSR93b]

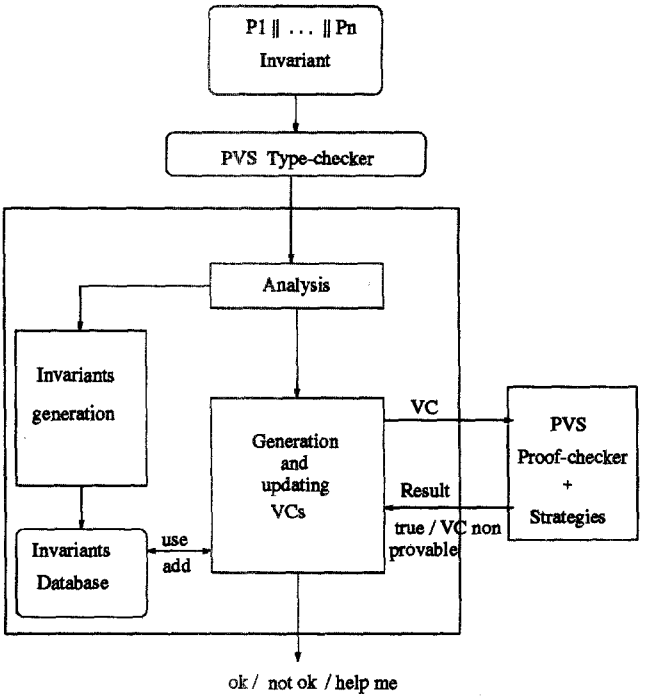


Fig. 1. Tool architecture

sys_components	⇒	⟨ *program* ⟩ | ⟨ *program* ⟩ || ⟨ *sys_components* ⟩
program	⇒	⟨ *action* ⟩$^+$ | ⟨ *named_program* ⟩
named_program	⇒	*id_program* : **PROGRAM**
		BEGIN
		⟨ *pvs_declarations* ⟩
		BEGIN
		⟨ *action* ⟩$^+$
		END
		END *id_program*
action	⇒	⟨ *pvs_boolean_formula* ⟩ ---> ⟨ *assignment* ⟩$^+$
assignment	⇒	*id* := ⟨ *pvs_expression* ⟩

where all declarations are global, but the variables declared within a component of the form *named_program* are only used locally.

This grammar uses some non-terminals of the grammar of the PVS specification language[8]. This allows to type check easily all PVS declarations and expressions by invoking the PVS parser and type checker. There are additional type correctness conditions for actions which have the form of invariants. For example, an action of the form

```
guard ---> x:= x-1,
```

[8] The non-terminals of the form ⟨ *pvs_*$\cdots$ ⟩

where **x** is declared as natural number, is type correct if **guard** $\Rightarrow$ **x** > 0 is a valid formula; but it is sufficient that **guard** $\Rightarrow$ **x** > 0 is an invariant of the whole system under consideration.

4.2 A proof methodology

We implemented some of the verification rules presented in [MP95] such as the **Inv** rule and the **Waiting** rule corresponding respectively to the proof of properties of the form $\Box P$ and $\Box(P \Rightarrow P_1 \mathcal{W} P_2)$, where P, P_1 and P_2 are predicates. Verification conditions are extracted automatically from the considered specification S and the property we want to verify by a proof obligation or verification condition (VC) generator. The VCs generated for the **Inv** rule are respectively $Init \Rightarrow Q_i$ and $\{Q_i \Rightarrow \widetilde{pre}[\tau](Q_i) \mid \tau \in T\}$ where $Init$ is the predicate defining the set of initial states, T the set of transitions of S and Q_i defined as in Section 1.1. We start with $i = 0$ and increase it until a provable set of verification conditions is obtained or $Init \Rightarrow Q_i$ is not provable anymore (a counter example for this obligation proves that P is *not* an invariant of S).

The VC generator generates only VCs which are not "trivially true". For example, if an action τ does not affect the variables occurring in Q_i, then the VC "$Q_i \Rightarrow \widetilde{pre}[\tau](Q_i)$" is not generated. If Q_i is of the form "$(pc = i) \Rightarrow Q$", where pc a control variable and i a possible value, it is only necessary to prove that Q_i is preserved by every action leading to control point i. In fact, it is often the case that predicates of the form $\widetilde{pre}[\tau](Q_i)$ are of the form $(pc = i) \Rightarrow Q$. Also, the auxiliary invariants (see Section 4.3) are of this form.

The generated obligations are submitted to the PVS proof checker, which tries to prove their validity by means of a set of tacticals we have defined. First an efficient but incomplete tactical for first order predicates is used. It combines rewriting with boolean simplification using Bdds[9] and an arithmetic decision procedure: after rewriting all definitions, the Bdd procedure breaks formulas into elementary ones, where other decision procedures such as arithmetic ones can be applied. If the proof fails, another tactical combining automatic induction and decision procedures is applied. If the proof fails again, a set of non-reducible goals is returned and one iteration step is performed. The user can always suspend this process and try to prove the unproved obligation in an interactive manner using the PVS proof checker.

4.3 Use of auxiliary invariants

It is in general essential to use already proved invariants or systematically generated structural invariants obtained by static analysis ([MAB+94], [BBM95], [MP95] and [BLS96]). Let $\mathcal{I}$ stand for the conjunction of all these invariants. In order to prove that P is inductive, it is sufficient to prove

$$\mathcal{I} \wedge P \Rightarrow \widetilde{pre}[\tau](P) \qquad (*)$$

[9] A Bdd simplifier is available in PVS as a tactic.

instead of $P \Rightarrow \widetilde{pre}[\tau](P)$. As $\mathcal{I}$ is usually a huge formula, we have to use it in an efficient way, that is only its "relevant conjuncts". Invariants of the particular form $(pc = i) \Rightarrow Q$, providing information about values of variable at some control point i, are only relevant for (*) when τ starts at control point i. In [Gri96], a more refined strategy is defined which selects in a formula of the form $h_1 \wedge h_1 \cdots \wedge h_n \Rightarrow c$, formulas h_i which are relevant for establishing the validity of c.

4.4 An efficient implementation

The implementation language of PVS is Lisp. Theories, expressions and formulas are defined as Lisp classes. In our tool, programs are also defined as Lisp classes. Type checking a program creates a class containing the corresponding declarations and actions. A current list of type checked programs is maintained. Static analysis described in Section 4.3 is performed using the internal representation of programs. The fact that our internal structures are very close to the internal PVS representation, allows to use many PVS features.

5 Examples

We present two examples. The first one, which is finite state, is a mutual exclusion algorithm studied in [Sif79].

```
mutex : SYSTEM
BEGIN
  ina, inb, PAB : VAR bool
  p1, p2        : VAR nat
  BEGIN
    p1=1                    --->  p1 := 2  ; ina := true        (t11)
    p1=2 AND inb            --->  p1 := 3  ;                    (t12)
    p1=3 AND NOT(PAB)       --->  p1 := 4  ; ina := false       (t13)
    p1=4 AND PAB            --->  p1 := 2  ; ina := true        (t14)
    p1=3 AND PAB            --->  p1 := 2  ;                    (t15)
    p1=2 AND NOT(inb)       --->  p1 := 5  ;                    (t16)
    p1=5                    --->  p1 := 6  ; ina := false       (t17)
    p1=6                    --->  p1 := 1  ; PAB := false       (t18)
     ||
    p2=1                    --->  p2 := 2  ; inb := true        (t21)
    p2=2 AND ina            --->  p2 := 3  ;                    (t22)
    p2=3 AND PAB            --->  p2 := 4  ; inb := false       (t23)
    p2=4 AND NOT(PAB)       --->  p2 := 2  ; inb := true        (t24)
    p2=3 AND NOT(PAB)       --->  p2 := 2  ;                    (t25)
    p2=2 AND NOT(ina)       --->  p2 := 5  ;                    (t26)
    p2=5                    --->  p2 := 6  ; inb := false       (t27)
    p2=6                    --->  p2 := 1  ; PAB := true        (t28)
  END
  INITIALLY : p1=1 AND p2=1
END mutex
```

We want to verify that the predicate

$$P = (p1 = 2) \Rightarrow ((p2 = 2) \Rightarrow (ina \vee inb))$$

expressing the impossibility that both processes may enter the critical section (pi =5) at the same moment, is an invariant for this program[10]. Since $Q_0 = P$ is not inductive for the transitions `t15` leading to $p1 = 2$ and `t25` leading to $p2 = 2$, the predicate $Q_1 = P \wedge \widetilde{pre}[t15](P) \wedge \widetilde{pre}[t25](P)$ is calculated:

$$\begin{aligned} Q_1 = & (p1 = 2 \Rightarrow (p2 = 2 \Rightarrow (ina \vee inb))) \\ & \wedge (p1 = 3 \wedge PAB \Rightarrow (p2 = 2 \Rightarrow (ina \vee inb))) \\ & \wedge (p2 = 3 \wedge \neg PAB \Rightarrow (p1 = 2 \Rightarrow (ina \vee inb))) \end{aligned}$$

$Q_1 \Rightarrow \widetilde{pre}[\tau](Q_1)$ is a valid formula for all transitions τ leading to $pi = 2$ or $pi = 3$ and the proof of this fact succeeds using our tactical. In this example, iteration is not necessary when using the following structural invariant obtained by an extension of the method described in [BLS96]:

$$I = (p1 = 3 \Rightarrow ina) \wedge (p2 = 3 \Rightarrow inb)$$

The proof of $Q_0 \wedge I \Rightarrow \widetilde{pre}[\tau](Q_0)$ succeeds also for the transitions $\tau =$ `t15` and $\tau =$ `t25`. This example was treated automatically by our tool.

The second example, which is infinite state, describes a simple buffer with two actions "input" and "output".

```
simple_buffer :  SYSTEM
BEGIN
  elem : TYPE
  outp, e, x, y : var elem
  IMPORTING Buffer[elem]
  B : var Buffer[elem]

  BEGIN
    TRUE             ---> B := cons(e,B)
    NOT(null?(B))    ---> outp := first(B) ; B := tail(B)
  END
  INITIALLY : B = null
END simple_buffer
```

The variable `e` represents the input of the the buffer. The imported PVS theory `Buffer` that contains the definition of buffers and some basic functions operating on them, is defined as follows:

```
Buffer [elem:TYPE]   : THEORY
BEGIN
  IMPORTING list[elem]
```

[10] Using the predicate $\neg(p1 = 5) \wedge (P2 = 5)$ to express the mutual exclusion property, leads to exactly one more iteration step

```
Buffer : TYPE = list[elem]

isin(B1:Buffer, e1:elem) : RECURSIVE bool =
  CASES B1 OF
    null : FALSE,
    cons(e2,B2) : IF (e1=e2) THEN TRUE ELSE isin(B2,e1) ENDIF
  ENDCASES

first(B:(cons?)) : RECURSIVE elem =
  IF null?(cdr(B)) THEN car(B) ELSE first(cdr(B)) ENDIF

tail(B:(cons?)) : RECURSIVE Buffer =
  IF null?(cdr(B)) THEN null ELSE cons(car(B),tail(cdr(B))) ENDIF

isbefore(x,y:elem, B1:Buffer) : RECURSIVE bool =
  CASES B1 OF
    null : FALSE,
    cons(e1,B2) :
      IF null?(B2) THEN (e1=x)  ELSE
         IF (e1=x) THEN NOT(isin(B2,y)) OR isbefore(x,y,B2)
                   ELSE isbefore(x,y,B2)
         ENDIF
      ENDIF
  ENDCASES

Buffer-lemma : OBLIGATION
  FORALL (B: Buffer, x: elem, y: elem):
    NOT(null?(B)) AND NOT(isin(B,y))   =>   NOT(isin(tail(B),y))

END Buffer
```

We want to verify that

```
BOX ( NOT(null?(B)) AND (x=car(B)) AND NOT(isin(B,y))
        =>
      isbefore(x,y,B) WEAK-UNTIL (outp=x)                    )
```

is an invariant. It expresses the fact that elements leave the buffer in the same order they have entered it, that is, the FIFO property. The following VCs are generated by our tool using the `Waiting` rule:

```
VC-1 : OBLIGATION
         isbefore(x,y,f)
           =>
         isbefore(x,y,cons(e,f)) OR (outp=x)

VC-2 : OBLIGATION
         isbefore(x,y,f) AND  NOT(null?(f))
           =>
         isbefore(x,y,tail(f)) OR (first(f)=x)
```

```
VC-3 : OBLIGATION
        NOT(null?(f))  AND  (x=car(f))  AND
        NOT(isin(f,y)) AND  NOT(x=y)
           =>
        isbefore(x,y,f) OR (outp=x)
```

The obligations VC-1 and VC-3 are proved automatically in one single step proof using our tactical. VC-2 is proved automatically with the same tactical using **Buffer-lemma**, which expresses a trivial property of buffers. That means the property can be verified without iteration.

6 Conclusions and future work

In this paper, we have presented a method and a tool allowing to do model checking using a theorem prover. Our approach takes advantage of the automatizability of algorithmic model checking and of the power of axiomatic methods which allows to deal with infinite state programs. It is clearly only a partial method as the fixed point may never be reached by the algorithmic method. Sometimes, the user will be able to guess a solution (which often can be checked easily).

In this paper we have hardly mentioned compositionality; however, for example for the verification of the mutual exclusion program (consisting of the parallel composition of two components) no product is built; also the method deriving structural invariants [BLS96] is compositional. In the future, more compositionality will by added by means of well-known rules.

Another interesting direction is the use of abstraction in the manner proposed for example in [Gra94]. The present framework is appropriate for this approach as in the above mentioned paper, the most difficult part was to argue that the considered abstract operations are in fact abstractions of the concrete operations. Here, all the necessary proofs can be done with PVS. Similar proposals have been made in [DF95] or in [HS96].

References

[BGMW94] H. Barringer, G. Gough, B. Monahan, and A. Williams. *The ELLA Verification Environment: A Tutorial Introduction.* Technical Report UMCS CS-94-12-2. University of Manchester.

[BLS96] S. Bensalem, Y. Lakhnech, H. Saïdi. *Powerful Techniques for the Automatic Generation of Invariants.* In this volume.

[BBM95] N. Bjørner, A. Browne and Z. Manna. *Automatic Generation of Invariants and Intermediate Assertions.* In U. Montanari, editor, First International Conference on Principles and Practice of Constraint Programming, LNCS, Cassis, September 1995.

[BLUP94] A. Blinchevsky, B. Liberman, I. Usvyatsky, A. Pnueli. *TPVS: Documentation and Progress Report.* Weizmann Institute Of Science, Department of Applied Mathematics and Computer Science, 1994.

[BM88] R. S. Boyer and J. S. Moore. *A Computational Logic Handbook.* Academic Press, 1988.

[CCF+95] C. Cornes, J. Courant, J. C. Filliâtre, G. Huet, P. Manoury, C. Paulin-Mohring , C. Muñoz, C. Murthy, C. Parent, A. Saïbi, and B. Werner. *The Coq Proof Assistant Reference Manual. Version 5.10.* Technical Report, I.N.R.I.A, February 1995.

[CLN+95] D. Cyrluk, P. Lincoln, P. Narendran, S. Owre, S. Ragan, J. Rushby, N. Shankar, J. U. Skekkebæk, M. Srivas, and F. von Henke. *Seven Papers on Mechanized Formal Verification.* Technical Report SRI-CSL-95-3, Computer Science Laboratory, SRI International, 1995.

[DF95] J. Dingel and Th. Filkorn. *Model checking for infinite state systems using data abstraction, assumption-commitment style reasoning and theorem proving.* Computer-Aided Verification, CAV'95, LNCS 939, Liège, June 1995.

[GM93] M. J. C. Gordon and T. F. Melham. *Introduction to HOL.* Cambridge University Press, 1993.

[Gra94] S. Graf. *Characterization of a sequentially consistent memory and verification of a cache memory by abstraction,* CAV'94. To appear in Journal of Distributed Computing.

[GR95] E. P. Gribomont and D. Rossetto. *CAVEAT: technique and tool for Computer Aided VErification And Transformation.* Computer-Aided Verification, CAV'95, LNCS 939, Liège, June 1995.

[Gri96] E. P. Gribomont. *Preprocessing for invariant validation.* AMAST'96.

[HS96] K. Havelund, and N. Shankar. *Experiments In Theorem Proving and Model Checking for Protocol Verification.* Proceedings of Formal Methods Europe. 1996.

[Hun93] H. Hungar. *Combining Model checking and Theorem proving to Verify Parallel Processes.* Computer-Aided Verification, CAV'93, LNCS 697, Elounda, June 1993.

[MAB+94] Z. Manna and al. *STeP: The Stanford Temporal Prover.* Department of Computer Science, Stanford University, 1994.

[MP91] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification.* Springer-Verlag, New York, 1991.

[MP95] Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems: Safety.* Springer-Verlag, New York, 1995.

[OSR93a] S. Owre, N. Shankar, and J. M. Rushby. *A Tutorial on Specification and Verification Using PVS.* Computer Science Laboratory, SRI International, February 1993.

[OSR93b] S. Owre, N. Shankar, and J. M. Rushby. *The PVS Specification Language.* Computer Science Laboratory, SRI International, February 1993.

[RSS95] S. Rajan, N. Shankar, and M. K. Srivas. *An integration of model checking with automated proof checking.* Computer-Aided Verification, CAV'95, LNCS 939, Liège, June 1995.

[Rus95] J. M. Rushby. *Model Checking and Other Ways of Automating Formal Methods.* Position paper for panel on Model Checking for Concurrent Programs Software Quality Week, San Francisco, May/June 1995.

[Saï95] H. Saïdi. *Syntax extentions in PVS, some suggestions.* Unpublished notes. September 1995.

[Sif79] J. Sifakis. *Le Contrôle des Systèmes Asynchrones: Concepts, Propriétés, Analyse Statique.* Phd Thesis. INPG, Grenoble, 1979.

Deductive Model Checking*

Henny B. Sipma, Tomás E. Uribe, Zohar Manna

Computer Science Department
Stanford University
Stanford, CA. 94305
sipma|uribe|manna@CS.Stanford.EDU

Abstract. We present an extension of classical tableau-based model checking procedures to the case of infinite-state systems, using deductive methods in an incremental construction of the behavior graph. Logical formulas are used to represent infinite sets of states in an abstraction of this graph, which is repeatedly refined in the search for a counterexample computation, ruling out large portions of the graph before they are expanded to the state-level. This can lead to large savings, even in the case of finite-state systems. Only local conditions need to be checked at each step, and previously proven properties can be used to further constrain the search. Although the resulting method is not always automatic, it provides a flexible and general framework that can be used to integrate a diverse number of other verification tools.

1 Introduction

We present a model checking procedure for verifying temporal logic properties of general infinite-state systems. It extends the classical tableau-based model checking procedure for verifying *linear-time temporal logic* specifications of reactive systems described by *fair transition systems*. To verify that a system $\mathcal{S}$ satisfies a specification φ, the classical procedure checks whether the $(\mathcal{S}, \neg\varphi)$ *behavior graph* admits any counterexample computations. This behavior graph is the product of the state transition graph for $\mathcal{S}$ and the temporal tableau for $\neg\varphi$, which makes the procedure essentially applicable to finite-state systems only.

Our procedure starts with the temporal tableau for $\neg\varphi$ and repeatedly refines and transforms this graph until a counterexample computation is found or it is demonstrated that such a computation cannot exist. Even for finite-state systems, this can lead to significant savings, since portions of the product graph can be eliminated long before they are fully expanded to the state level. For instance, in the verification of accessibility for the Peterson mutual-exclusion algorithm, expansion to 12 nodes suffices to demonstrate that no counterexample exists, whereas the full behavior graph contains 76 nodes.

* This research was supported in part by the National Science Foundation under grant CCR-92-23226, the Advanced Research Projects Agency under NASA grant NAG2-892, the United States Air Force Office of Scientific Research under grant F49620-93-1-0139, the Department of the Army under grant DAAH04-95-1-0317, and a gift from Intel Corporation.

For infinite-state systems, the procedure will terminate in many cases. In Section 5 we illustrate the procedure by model checking an accessibility property for the Bakery algorithm. Expansion to 16 nodes suffices to verify this property over this infinite-state system. Even when the procedure does not terminate, partial results can still be valuable, giving a representation of all potential counter-example computations that can be used for further verification or testing.

We present our procedure in the framework of [14], where deductive methods are used to verify linear-time temporal logic specifications for reactive systems described by fair transition systems. However, the main ideas can be easily adapted to other temporal logics and system specification languages as well.

Related work

Model Checking: Most approaches to temporal logic model checking [10, 16] have used explicit state enumeration, or specialized data structures to represent the transition relation and compute fixpoints over it, as in BDD-based "symbolic" model checking [8, 15]. While automatic, and particularly successful for hardware systems, these approaches require that the system, or a suitable abstraction of it, conform to the particular data structure used. Most often, the system must be finite-state. Furthermore, even in the finite-state case these techniques are limited by the size of the specialized representation, which is still ultimately limited by the number of reachable states.

The "on-the-fly model checking" for CTL* presented in [1] constructs only a portion of the state-space as required by the given formula, but is still restricted to finite-state systems. Our procedure is similarly "need-driven," but expands the state-space in a "top-down" manner as well, moving from an abstract representation to a more detailed one as necessary.

A method for generating an abstract representation of a possibly infinite state-space is presented in [4], using partitioning operations similar to the ones we describe below. However, in [4] this is done independently of any particular formula to be verified. Finally, the local model checking algorithm for real-time systems in [18] can be seen as a specialized variant of our procedure; it too refines a finite representation of an infinite product graph, consecutively splitting nodes to satisfy constraints arising from the formula and system being checked.

Deductive Methods: A complete deductive system for temporal verification of branching-time properties is presented in [11], while [5] presents a proof system for the modal mu-calculus. Manna and Pnueli [14] present a deductive framework for the verification of fair transition systems based on *verification rules*, which reduce temporal properties of systems to first-order premises. *Verification diagrams* [13, 6] provide a graphical representation of the verification conditions needed to establish a particular temporal formula.

All of these methods apply to infinite-state systems and enjoy relative completeness, but can require substantial user guidance to succeed. These methods yield a direct proof of the system-validity of a property, but do not produce counterexample computations when the property fails.

Like standard model checking, our procedure does not require user-provided auxiliary formulas, and allows the construction of counterexamples; the process is guided by the search for such computations. Like deductive methods, it only needs to check local conditions, and allows the verification of infinite-state systems through the use of powerful representations to describe sets of states (e.g. first-order formulas). We also accommodate the use of previously established invariants and simple temporal properties.

The procedure presented in [3] for automatically establishing temporal *safety* properties is based on an *assertion graph* similar to the *S-refined tableau* we use, and can also produce counterexamples. Our approach is a dual one: instead of checking that all computations satisfy the temporal tableau of the formula φ being proved, we check that no computations satisfy the tableau for $\neg\varphi$.

2 Preliminaries

Fair Transition Systems: The computational model, following [14], is a *fair transition system* (FTS). An FTS $\mathcal{S}$ is a triple $\langle \mathcal{V}, \Theta, \mathcal{T} \rangle$, where $\mathcal{V}$ is a set of variables, Θ is the *initial condition*, and $\mathcal{T}$ is a finite set of *transitions*. A finite set of *system variables* $V \subset \mathcal{V}$ determines the possible states of the system. The *state-space*, Σ, is the set of all possible valuations of the system variables.

We use a first-order[2] assertion language $\mathcal{A}$ to describe Θ and the transitions in $\mathcal{T}$. Θ is an assertion over the system variables V. A transition τ is described by a *transition relation* $\rho_\tau(\mathbf{x}, \mathbf{x}')$, an assertion over the set of system variables $\mathbf{x}$ and a set of *primed variables* $\mathbf{x}'$ indicating their values at the next state. $\mathcal{T}$ includes an *idling transition*, *Idle*, whose transition relation is $\mathbf{x} = \mathbf{x}'$.

A *run* is an infinite sequence of states $s_0, s_1, \ldots$ such that s_0 satisfies Θ, and for each $i \geq 0$, there is some transition $\tau \in \mathcal{T}$ such that $\rho_\tau(s_i, s_{i+1})$ evaluates to true. We then say that τ is *taken* at s_i, and that state s_{i+1} is a *τ-successor* of s. A transition is *enabled* if it can be taken at a given state. Such states are characterized with the formula

$$enabled(\tau) \stackrel{\text{def}}{=} \exists \mathbf{x}'.\rho_\tau(\mathbf{x}, \mathbf{x}') \ .$$

As usual, we define the *strongest postcondition* $post(\tau, \varphi)$ and the *weakest precondition* $pre(\tau, \varphi)$ of a formula φ relative to a transition τ as follows:

$$post(\tau, \varphi) \stackrel{\text{def}}{=} \exists \mathbf{x}_0.\,(\rho_\tau(\mathbf{x}_0, \mathbf{x}) \wedge \varphi(\mathbf{x}_0))$$
$$pre(\tau, \varphi) \stackrel{\text{def}}{=} \forall \mathbf{x}'.\,(\rho_\tau(\mathbf{x}, \mathbf{x}') \rightarrow \varphi(\mathbf{x}'))$$

We also use the notation $\{\varphi\}\,\tau\,\{\psi\} \stackrel{\text{def}}{=} (\varphi(\mathbf{x}) \wedge \rho_\tau(\mathbf{x}, \mathbf{x}')) \rightarrow \psi(\mathbf{x}')$.

Fairness: The transitions in $\mathcal{T}$ can be optionally marked as *just* or *compassionate*. A just (or *weakly fair*) transition cannot be continually enabled without

[2] Although it can be augmented with features such as interpreted symbols and constraints, or specialized to the finite-state case, e.g. using BDDs.

ever being taken; a compassionate (or *strongly fair*) transition cannot be enabled infinitely often but taken only finitely many times. A *computation* is a run that satisfies these fairness requirements.

Linear-time Temporal Logic: As specification language we use linear-time temporal logic (LTL) over the assertion language $\mathcal{A}$, where no temporal operator is allowed to appear within the scope of a quantifier. We use the usual future and past temporal operators, such as $\Box$, $\Diamond$, $\bigcirc$, $\mathcal{U}$, $\mathcal{W}$ (future) and $\boxminus$, $\diamondminus$, $\ominus$, $\mathcal{B}$, $\mathcal{S}$ (past). A formula with no temporal operators is called a *state-formula* or an *assertion*. For details on LTL and tableau constructions, we refer the reader to [14], and define only the basic concepts we need.

The Formula Tableau: Given an LTL formula φ, we can construct its *tableau* Φ_φ, a finite graph that describes all of its models [14]. Briefly, each node in the tableau is identified with an *atom*, which is a set of state- and temporal formulas expected to hold whenever a model resides at this node. Two nodes A_1 and A_2 are connected with a directed edge $\langle A_1, A_2 \rangle$ if the formulas in A_2 can hold at a state following one that satisfies the formulas in A_1.

An atom is called *initial* if its formulas can hold at the initial state of a model. φ is satisfiable only if there is a *strongly connected subgraph* (SCS) in Φ_φ that is reachable from an initial atom. Furthermore, if a given model satisfies, e.g., $\Diamond p$ at some point, it must in fact satisfy p at this or another point later on. A *fulfilling* SCS is one where all such eventualities are satisfied.

Proposition 1. *φ is satisfiable iff there is a fulfilling, reachable SCS in Φ_φ.*

3 Deductive Model Checking

The classical approach to model checking [10, 16] verifies a property φ by constructing the *product graph* between the system's reachable-state graph and the temporal tableau for $\neg\varphi$. Any infinite path through the product graph that satisfies the fairness constraints on the transitions and is fulfilling with respect to its tableau atoms is a counterexample to φ.

The explicit construction of the state-graph restricts the method to finite-state systems. The procedure we present works in a top-down fashion, starting with a general skeleton of the product graph and refining it until a counterexample is found, or the impossibility of such a counterexample is demonstrated.

Definition 2 ($\mathcal{S}$-refined tableau). Given an FTS $\mathcal{S}$ and a temporal property φ, an *$\mathcal{S}$-refined tableau* is a directed graph $\mathcal{G}$ whose nodes are labeled with pairs (A, f), where A is an atom for the temporal tableau for $\neg\varphi$ and f is a state-formula, and whose edges are labeled with subsets of $\mathcal{T}$. For nodes M, N, we write $\tau \in \langle M, N \rangle$ if transition τ is in the label of the edge from M to N, or simply say that τ *labels* $\langle M, N \rangle$. A subset of the nodes in $\mathcal{G}$ is marked as *initial*.

The $\mathcal{S}$-refined tableau can be viewed as a finite abstraction of the product graph. The state-formula f in a node (A, f) describes a superset of the states reachable

at that node; similarly, the transitions labeling an edge $\langle (A_1, f_1), (A_2, f_2) \rangle$ are a superset of those that can be taken from an f_1-state to reach an f_2-state. We will see that any path through an $\mathcal{S}$-refined tableau corresponds to a path through the corresponding temporal tableau. That is, for any path $(A_0, f_0), (A_1, f_1), \ldots$ through $\mathcal{G}$, the *underlying path* $A_0, A_1, \ldots$ will be a path in $\Phi_{\neg\varphi}$.

3.1 The DMC Procedure

We begin with the tableau graph $\Phi_{\neg\varphi}$, from which we construct an initial $\mathcal{S}$-refined tableau $\mathcal{G}_0$ as follows:

1. Replace each node label A by (A, f_A), where f_A is the conjunction of the state-formulas in A.
2. For each node $N = (A, f)$ such that A is initial in the tableau $\Phi_{\neg\varphi}$, add a new node $N_0 = (A, f \wedge \Theta)$, which has no incoming edges, and whose outgoing edges go to exactly the same nodes as those of N. A self-loop $\langle N, N \rangle$ becomes an edge $\langle N_0, N \rangle$ in the new graph. These new nodes are the *initial nodes* in the $\mathcal{S}$-refined tableau.
3. Label each edge in $\mathcal{G}_0$ with the entire set of transitions $\mathcal{T}$.

Figure 2 in Section 5 presents an example of an initial $\mathcal{S}$-refined tableau.

The main data structure maintained by the procedure is an $\mathcal{S}$-refined tableau graph $\mathcal{G}_i$ and a list of strongly connected subgraphs of this graph. We present the deductive model checking (DMC) procedure as a set of transformations on this pair. Initially, the SCS list contains all the *maximal strongly connected subgraphs* (MSCS's) of $\mathcal{G}_0$. Deductive model checking proceeds by repeatedly applying one of transformations 1–11 described below. The process stops if we find a *reachable*, *fulfilling* and *adequate* SCS (see Section 3.3) or obtain an empty SCS list.

Basic Transformations:

- **1 (remove label).** If an edge $\langle (A_1, f_1), (A_2, f_2) \rangle$ is labeled with a transition τ and $f_1(\mathbf{x}) \wedge f_2(\mathbf{x}') \wedge \rho_\tau(\mathbf{x}, \mathbf{x}')$ is unsatisfiable, remove τ from the edge label.
- **2 (empty edge).** If an edge is labeled with the empty set, remove the edge.
- **3 (unsatisfiable node).** If f is unsatisfiable for a node (A, f), or if a node has no successors, remove the node.
- **4 (unreachable node).** Remove a node unreachable from an initial node.
- **5 (unfulfilling SCS).** If an SCS is not fulfilling, remove it from the SCS list. (An SCS is fulfilling if its underlying tableau SCS is fulfilling.)
- **6 (SCS split).** If an SCS becomes disconnected (because a node or an edge is removed from the graph), replace it by its constituent MSCS's.

These basic transformations should be applied whenever possible.

Node Splitting: In the following, we will have the opportunity to replace a node N by new nodes N_1 and N_2. Any incoming edge $\langle M, N \rangle$ is replaced by edges $\langle M, N_1 \rangle$ and $\langle M, N_2 \rangle$ with the same label, for $M \neq N$. Similarly, any outgoing edge $\langle N, M \rangle$ is replaced by edges $\langle N_1, M \rangle$ and $\langle N_2, M \rangle$ with the same label as the original edge. If a self-loop $\langle N, N \rangle$ was present, we add edges $\langle N_1, N_1 \rangle$, $\langle N_2, N_2 \rangle$,

$\langle N_1, N_2 \rangle$ and $\langle N_2, N_1 \rangle$, all with the same label as $\langle N, N \rangle$. If an initial node is split, the two new nodes are also labeled as initial. If the split node was part of an SCS in the SCS list, this SCS is updated accordingly.

Basic Refinement Transformations:

- **7 (postcondition split).** Consider an edge from node N_1 to N_2, $\langle N_1, N_2 \rangle = \langle (A_1, f_1), (A_2, f_2) \rangle$, whose label includes transition τ. If $f_2 \wedge \neg post(\tau, f_1)$ is satisfiable (that is, f_2 does not imply $post(\tau, f_1)$), then replace (A_2, f_2) by the two nodes

$$N_{2,1} = (A_2, f_2 \wedge post(\tau, f_1))$$
$$N_{2,2} = (A_2, f_2 \wedge \neg post(\tau, f_1))$$

  Note that we can immediately apply the **remove label** transformation to the edge between N_1 and $N_{2,2}$, removing transition τ from its label.
  Nodes N_1 and N_2 need not be distinct. If $N_1 = N_2$ then we split the node into two new nodes as above, only now the self-loop for $N_{2,2}$ as well as the edge from $N_{2,1}$ to $N_{2,2}$ do not contain the transition τ.
- **8 (precondition split).** Consider an edge $\langle N_1, N_2 \rangle = \langle (A_1, f_1), (A_2, f_2) \rangle$, labeled with transition τ. If $f_1 \wedge \neg(enabled(\tau) \wedge pre(\tau, f_2))$ is satisfiable, then replace (A_1, f_1) by the two nodes

$$N_{1,1} = (A_1, f_1 \wedge enabled(\tau) \wedge pre(\tau, f_2))$$
$$N_{1,2} = (A_1, f_1 \wedge \neg(enabled(\tau) \wedge pre(\tau, f_2))) \quad .$$

  Here, transition τ can be removed from the $\langle N_{1,2}, N_2 \rangle$ edge.

The conditions for applying these transformations can be weakened if the required satisfiability checks are too expensive (see Section 4). Variants of these transformations, such as n-ary splits according to possible control locations, are convenient in practice. In general, arbitrary conditions can be used to split nodes. However, our refinement transformations account for the structure of the system and property being checked, and can be automated as well.

3.2 Fairness Transformations

Together, transformations 1-8 are sufficient for the analysis of transition systems with no fairness requirements. If an *adequate* SCS is found (see Section 3.3), a counterexample is produced. If the set of SCS's (all of which are actually MSCS's, in this case) becomes empty, then we know there is no counterexample computation. However, to account for just and compassionate transitions we need the following extra transformations:

- **9 (enabled split).** Consider a just or compassionate transition τ and an SCS containing a node $N = (A, f)$ such that $f \wedge \neg enabled(\tau)$ is satisfiable. Then replace N by the two nodes

$$N_1 = (A, f \wedge enabled(\tau))$$
$$N_2 = (A, f \wedge \neg enabled(\tau)) \quad .$$

Definition 3. A transition τ is *fully enabled* at a node (A, f) if $f \rightarrow \mathit{enabled}(\tau)$ is valid; τ is *fully disabled* at a node (A, f) if $f \rightarrow (\neg \mathit{enabled}(\tau))$ is valid. A transition is *taken* on an SCS S if it is included in an edge-label in S. An SCS S is *just* (resp. *compassionate*) if every just (resp. compassionate) transition is either taken in S or not fully enabled at some node (resp. all nodes) in S.

That is, an SCS S is unjust (resp. uncompassionate) if some just (resp. compassionate) transition is never taken in S and fully enabled at all nodes (resp. some node) in S.

We now present the last two transformations, which, like the basic ones, should be applied whenever possible:

- **10 (uncompassionate SCS).** If an SCS S is not compassionate, then let $\tau_1, \ldots, \tau_n$ be all the compassionate transitions that are not taken in S. Replace S by all the MSCS's of the subgraph resulting by removing all the nodes in S where one of these transitions is fully enabled.
- **11 (unjust SCS).** If an SCS is not just, remove it from the SCS list.

Note that these transformations do not change the underlying graph $\mathcal{G}$, but only the SCS's under consideration. (However, unjust or unfulfilling SCS can be fully removed from the graph if they have no outgoing edges.)

3.3 Reachability and Termination

The process of transforming the $\mathcal{S}$-refined tableau can continue until there are no SCS's under consideration, in which case the original property φ is guaranteed to hold for the system $\mathcal{S}$.

Finding a counterexample computation in the case that φ fails, however, requires some additional work. Whereas the above transformations remove SCS's from consideration that are known to be unreachable because they are disconnected from an initial node, no provisions ensure that a node is indeed reachable in an actual computation, or that a computation can in fact reside indefinitely within an SCS.

To identify portions of the product graph known to be reachable, we do some additional book-keeping:

- **(executable transition).** Given an edge $\langle (A_1, f_1), (A_2, f_2) \rangle$ labeled with transition τ, mark τ as *executable* if the following formula is valid:

$$(f_1 \rightarrow \mathit{enabled}(\tau)) \wedge (\{f_1\}\, \tau\, \{f_2\}) \quad .$$

That is, τ is labeled as executable if it can be taken at all states satisfying f_1 and always reaches a state that satisfies f_2. For example, the idling transition can be marked as executable on all self-loops.

Definition 4 (fully just and compassionate). A transition is *fully taken* at an SCS if it is marked as executable for an edge in the SCS. An SCS S is *fully just* (resp. *fully compassionate*) if every just (resp. compassionate) transition is either fully taken in S or fully disabled at some node (resp. all nodes) in S.

Definition5 (adequate SCS). An SCS S is *adequate* if after removing all edges not marked with executable transitions we obtain a subgraph S' where:

1. S' is still strongly connected;
2. S' is fully just and fully compassionate;
3. there is a path of executable transitions from a satisfiable initial node to a node in S';
4. the state-formulas in S' and the path that leads to S' are satisfiable.

An adequate SCS guarantees the existence of a computation of $\mathcal{S}$ that satisfies $\neg\varphi$ (but the reverse does not hold).

Using Previously Proven Properties: Known invariants can be used to strengthen all (or only some) of the assertions in the $\mathcal{S}$-refined tableau; if $\Box p$ is a known invariant for a state-formula p, then we can replace any node (A, f) by the node $(A, f \wedge p)$.

Similarly, simple temporal properties of the system can be used to rule out paths in the tableau. For example, if we know that $\Box(p \rightarrow \Diamond q)$ is $\mathcal{S}$-valid, then we can require that any candidate SCS featuring a state-formula which implies p also contain a state-formula compatible with q.[3]

4 Analysis

The soundness of the procedure is based on the fact that each transformation preserves the set of computations through the $\mathcal{S}$-refined tableau. Since this is equal to the $\neg\varphi$ computations of $\mathcal{S}$ for the initial graph $\mathcal{G}_0$, the procedure reports success only if there are no such computations. On the other hand, a computation that is obtained by reaching and then residing in an adequate SCS must indeed be a model of $\neg\varphi$ and a computation of $\mathcal{S}$, and thus a counterexample.

The tableau Φ_φ can be exponential in the size of φ; however, properties to be model checked are usually simple, so the tableau is small when compared with the system's state-space (even for finite-state systems). *Incremental* and *particle tableau* constructions [14] reduce the expense of building Φ_φ.[4]

Proposition6. *For a finite-state system $\mathcal{S}$, the exhaustive application of transformations 1-11 terminates, deciding the $\mathcal{S}$-validity of φ.*

If the system $\mathcal{S}$ is finite-state, we can use a finite-state assertion language $\mathcal{A}$. Note that the satisfiability tests required at the splitting steps are now decidable, and there will only be a finite number of distinct nodes. Since every transformation reduces the size of the graphs under consideration or replaces a node with more specific ones (that is, nodes covering strictly fewer states), the process must terminate. If the SCS list is empty, the original property φ is $\mathcal{S}$-valid; otherwise, any remaining SCS must be adequate, and thus provide a counterexample.

[3] $\neg\varphi$ could always be conjoined with all other known temporal properties of $\mathcal{S}$, but at the risk of further increasing the size of the temporal tableau.

[4] If necessary, this construction can be interleaved with the state-space refinement.

The node formulas may well be encoded using binary decision diagrams (BDDs) [7] or, in general, any finite-domain constraint language. The efficient tests for implication between BDDs can be used to maintain encapsulation conventions. Hybrid representations (including first-order constructs) can be used if the BDD size becomes problematic.

In the general case of infinite-state systems, the model checking problem is undecidable. However, we point to several features of our approach:

• The test for satisfiability used in the splitting rules need not be complete; we can change the condition "if X is satisfiable then..." to be "if X is not *known* to be unsatisfiable then..." without compromising soundness. Thus, the available theorem-proving and simplification techniques are not required to give a definite answer at any given time. When the validity of a formula is hard to decide, additional splits can make subsequent satisfiability questions easier.

This lazy evaluation of satisfiability makes specialized constraint languages such as those used in Constraint Logic Programming [12] well-suited to the task. Reactive programs based on such constraint languages, such as concurrent constraint programs [17], may be specially amenable to such a verification framework. We expect constraint-solving and propagation techniques, as in [3], to play a central role in the deductive model checking of large systems.

• Even when the model checking effort is not completed, the resulting $\mathcal{S}$-refined tableau can be used to restrict the search for a counterexample, since all such computations must follow the $\mathcal{S}$-refined tableau. Backward propagation (possibly approximated) [3] can be used to find sets of initial states that can generate a counterexample computation. A similar approach is used in [9] to generate test cases for processor designs.

• The DMC procedure can benefit from user guidance in two forms: first, the choice of refinement transformation to perform next determines how the state-space is explored. Second, the process can be speeded up considerably by refinement steps based on auxiliary formulas provided by the user.

Inductive and well-foundedness arguments can also be used: for example, if a transition decreases a well-founded relation that is known to hold across an SCS, then we can remove it from all the edges in the SCS (but still account for it for reachability). Adding support for well-founded relations and ranking functions similar to those used in Verification Diagrams [13, 6] could make the method relatively complete and further the combination of theorem-proving and model checking we propose.

5 Example

We illustrate deductive model checking by proving accessibility for the BAKERY program, an infinite-state program implementing a mutual exclusion protocol, shown in Figure 1. Each of the statements in the program corresponds to a transition, denoted by its label; thus, $\mathcal{T} = \{Idle, \ell_0..\ell_4, m_0..m_4\}$. All transitions are just, except for m_0 and ℓ_0, which have no fairness requirements. Accessibility can be expressed in LTL by the formula $\varphi : \Box(\ell_1 \rightarrow \Diamond \ell_3)$, i.e., always if control is

at ℓ_1 it will eventually reach ℓ_3. The following describes the output of our DMC implementation based on the STeP system [2]. The splits are chosen by the user, but the underlying simplification and pruning are performed automatically.

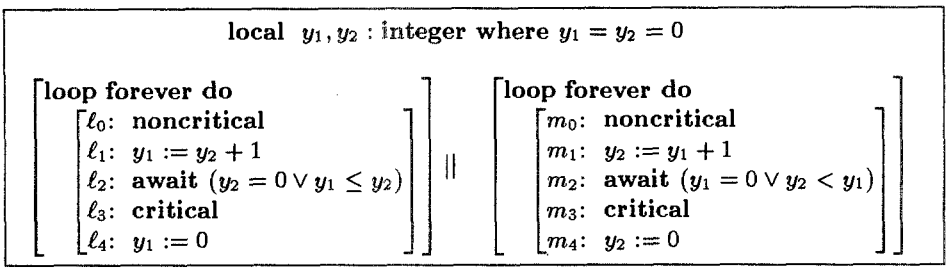


Fig. 1. Program BAKERY

The initial $\mathcal{S}$-refined tableau for $\neg\varphi : \Diamond(\ell_1 \wedge \Box\neg\ell_3)$, based on its *particle tableau*, is shown in the left of Figure 2. Nodes 3 and 4 correspond to the initial nodes in the $\neg\varphi$ tableau. Node 1 results from adding the initial condition to node 4; the initial node from node 3 is pruned since $\ell_1 \wedge \Theta$ is unsatisfiable. The SCS $\{4\}$ is not fulfilling, but $\{2\}$ is. We now perform a precondition split on edge $\langle 2, 2\rangle$ and transition ℓ_0, replacing node 2 by nodes 6 and 5. An ℓ_4-precondition split on $\langle 6, 5\rangle$ yields nodes 8 and 7. At this point, nodes 5 and 7 are unreachable from the initial state and can be removed. The only candidate SCS is $\{8\}$.

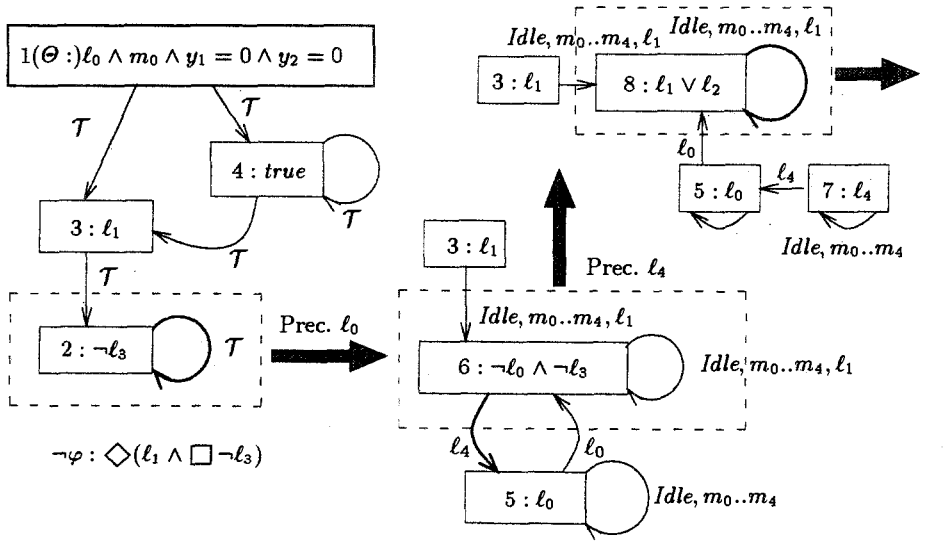


Fig. 2. Initial $\mathcal{S}$-refined tableau and first 2 refinement steps

An ℓ_1-postcondition split for $\langle 8, 8\rangle$ yields nodes 9 and 10 in Figure 3. The only fulfilling SCS is $\{10\}$, since $\{9\}$ is unjust for ℓ_1. An enabled split for node 10 and transition ℓ_2 produces nodes 11 and 12. The SCS $\{11\}$ is unjust for ℓ_2. Node 12 is now strengthened with the invariant $(y_2 \neq 0) \rightarrow (m_2 \vee m_3 \vee m_4)$. (Such invariants are generated automatically by STeP based on the program text.) An m_3-precondition split on $\langle 12, 12\rangle$ produces nodes 13 and 14. SCS $\{13\}$ is unjust for m_3. Finally, an m_2-precondition split on $\langle 14, 13\rangle$ results in 15 and 16. Now, SCS $\{16\}$ is unjust for m_4, while $\{15\}$ is unjust for m_2. Since there are no candidate SCS left, we have established that φ is $\mathcal{S}$-valid.

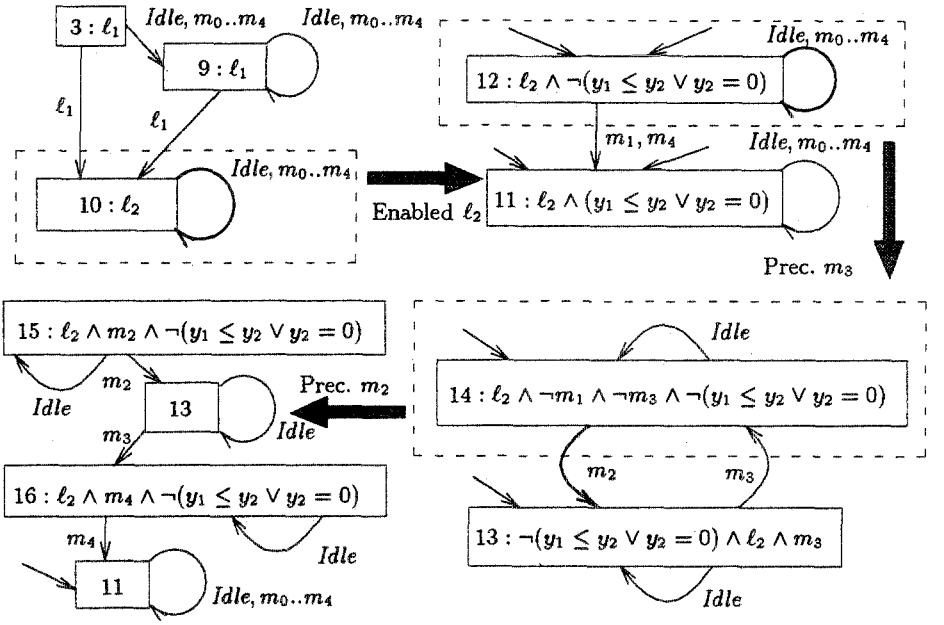


Fig. 3. Final 3 refinement steps to model check BAKERY

Note that when model checking progress properties it may be profitable to concentrate on splitting and eliminating candidate SCS's, as done in this example. However, in general it may be necessary to show that certain parts of the state-space are unreachable through forward propagation from the initial nodes or backward propagation from the unreachable ones. We model checked mutual exclusion for BAKERY ($\Box \neg(\ell_3 \wedge m_3)$) using 3 splits (including a user-provided one) and automatically generated invariants.

We also model checked accessibility for the infinite-state 3-process version of BAKERY, expanding to 27 nodes. This included one user-provided case split according to the priority between processes (4 cases), together with 5 trivial location splits and one enabled split.

Acknowledgements: We thank Nikolaj Bjørner, Anca Browne and Arjun Kapur for their comments.

References

1. BHAT, G., CLEAVELAND, R., AND GRUMBERG, O. Efficient on-the-fly model checking for CTL*. In *Proc. 10th IEEE Symp. Logic in Comp. Sci.* (1995), pp. 388–397.
2. BJØRNER, N., BROWNE, A., CHANG, E., COLÓN, M., KAPUR, A., MANNA, Z., SIPMA, H., AND URIBE, T. STeP: Deductive-algorithmic verification of reactive and real-time systems. In *Proc. 8^{th} Intl. Conference on Computer Aided Verification* (July 1996), Springer-Verlag.
3. BJØRNER, N., BROWNE, A., AND MANNA, Z. Automatic generation of invariants and intermediate assertions. In 1^{st} *Intl. Conf. on Principles and Practice of Constraint Programming* (Sept. 1995), vol. 976 of *LNCS*, Springer-Verlag, pp. 589–623.
4. BOUAJJANI, A., FERNANDEZ, J.-C., AND HALBWACHS, N. Minimal model generation. In *Proc. 2^{nd} Intl. Conference on Computer Aided Verification* (1990), vol. 531 of *LNCS*, pp. 197–203.
5. BRADFIELD, J. C. *Verifying Temporal Properties of Systems.* Birkhäuser, 1992.
6. BROWNE, A., MANNA, Z., AND SIPMA, H. Generalized verification diagrams. In *15th Conference on the Foundations of Software Technology and Theoretical Computer Science* (Dec. 1995), vol. 1026 of *LNCS*, pp. 484–498.
7. BRYANT, R. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers C-35*, 8 (Aug. 1986), 677–691.
8. BURCH, J., CLARKE, E., MCMILLAN, K., DILL, D., AND HWANG, L. Symbolic model checking: 10^{20} states and beyond. In *Proc. 5th IEEE Symp. Logic in Comp. Sci.* (1990), IEEE Computer Society Press, pp. 428–439.
9. CHANDRA, A., IYENGAR, V., JAWALEKAR, R., MULLEN, M., NAIR, I., AND ROSEN, B. Architectural verification of processors using symbolic instruction graphs. In *International Conference on Computer Design: VLSI in Computers and Processors* (1994), IEEE Press, pp. 454–459.
10. CLARKE, E., AND EMERSON, E. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Proc. IBM Workshop on Logics of Programs* (1981), vol. 131 of *LNCS*, Springer-Verlag, pp. 52–71.
11. FIX, L., AND GRUMBERG, O. Verification of temporal properties. *J. Logic and Computation 6*, 3 (1996), 343–362.
12. JAFFAR, J., AND LASSEZ, J.-L. Constraint logic programming. In *Proc. 14th ACM Symp. Princ. of Prog. Lang.* (Jan. 1987), pp. 111–119.
13. MANNA, Z., AND PNUELI, A. Temporal verification diagrams. In *Proc. Int. Symp. on Theoretical Aspects of Computer Software* (1994), vol. 789 of *LNCS*, Springer-Verlag, pp. 726–765.
14. MANNA, Z., AND PNUELI, A. *Temporal Verification of Reactive Systems: Safety.* Springer-Verlag, New York, 1995.
15. MCMILLAN, K. L. *Symbolic Model Checking.* Kluwer Academic Pub., 1993.
16. QUEILLE, J., AND SIFAKIS, J. Specification and verification of concurrent systems in CESAR. In *Intl. Symposium on Programming* (1982), M. Dezani-Ciancaglini and U. Montanari, Eds., vol. 137 of *LNCS*, Springer-Verlag, pp. 337–351.
17. SARASWAT, V. A. *Concurrent Constraint Programming.* MIT Press, 1993.
18. SOKOLSKY, O. V., AND SMOLKA, S. A. Local model checking for real-time systems. In *Proc. 7^{th} Intl. Conference on Computer Aided Verification* (1995), vol. 939 of *LNCS*, pp. 211–224.

Automated Verification by Induction with Associative-Commutative Operators

Narjes Berregeb[§] Adel Bouhoula[§‡] Michaël Rusinowitch[§]

[§] INRIA Lorraine & CRIN
Campus Scientifique, 615, rue du Jardin Botanique - B.P. 101
54602 Villers-lès-Nancy Cedex, France
E-mail:{berregeb, bouhoula, rusi}@loria.fr

[‡] Computer Science Laboratory, SRI International
333 Ravenswood Avenue, Menlo Park, California 94025, USA
E-mail: bouhoula@csl.sri.com

Abstract. Theories with associative and commutative (AC) operators, such as arithmetic, process algebras, boolean algebras, sets, ... are ubiquitous in software and hardware verification. These AC operators are difficult to handle by automatic deduction since they generate complex proofs. In this paper, we present new techniques for combining induction and AC reasoning, in a rewrite-based theorem prover. The resulting system has proved to be quite successful for verification tasks. Thanks to its careful rewriting strategy, it needs less interaction on typical verification problems than well known tools like NQTHM, LP or PVS . We also believe that our approach can easily be integrated as an efficient tactic in other proof systems.

1 Introduction

Powerful tools based on model checking have been developed for the verification of finite-state systems [6]. Their extensions to some classes of infinite-state systems has only produced moderate success. Therefore deductive methods offer a promising complementary approach especially for verifying parameterized components or systems involving infinite data-types. Besides, when a program or a circuit is not correct, more high-level information about how to correct it can be derived with deductive methods.

Effective verification with deductive techniques requires efficient primitive inference procedures in order to free the user from tedious low-level proof construction details. Rewriting is now widely recognized as an important technique for efficiency and is part of many systems. In this framework, the induction prover SPIKE [1] [3, 2], has been developed. It relies on implicit induction whose principle is to simulate induction by term rewriting. Given a theory presented by conditional equations, the prover instanciates some particular variables of a conjecture to be proved, called *induction variables*, by terms from a *test set*

[1] Spike is available by ftp from ftp.loria.fr in /pub/loria/protheo/softwares/Spike

which is a finite description of the model, then simplifies them by axioms, other conjectures or induction hypotheses. Every iteration generates new subgoals that are processed in the same way as the initial conjectures.

However, many theories of interest include AC operators, which are hard to handle since they cause divergence or generate complex proofs. To overcome this problem, we propose to have the AC axioms built in the inference mechanism of SPIKE. The advantage of our approach over other implicit induction techniques [5, 9], is that it does not use AC unification (which is doubly exponential) during the proof process, but only AC matching.

In this paper, we propose methods for automatically selecting the induction variables of a conjecture to be proved, and for constructing test sets in the case of an AC conditional theory. We present our proof procedure as an inference system based on new simplification techniques. This inference system is correct, refutationally complete (when the procedure stops with failure we can ensure that the given conjecture is wrong) under some reasonable restrictions on the initial AC conditional theory. These results have been implemented in the system SPIKE-AC, and computer experiments have shown the gain we obtain when handling AC operators by these techniques. In particular, the procedure has allowed us to prove directly theorems (for example, the correctness of a ripple carry adder) that require more interaction with other systems.

Overview on an example

To illustrate our approach, let us describe the correctness proof of a simple digital circuit. We consider a ripple carry adder (see figure 1), whose inputs are two bit-vectors $A = (A_0, A_1, \ldots, A_{n-1})$ and $B = (B_0, B_1, \ldots, B_{n-1})$, and a carry C_0. This circuit performs addition of A and B and the result is a bit vector $S = (S_0, S_1, \ldots, S_{n-1})$, and a carry C_n. This problem is easily specified with conditional rules, and the specification obtained reflects clearly the circuit description. The circuit function computing the sum of two bit-vectors A and B given a carry C_0, is $add(A, B, C_0)$. We define a mapping function *bvtonat* which transforms a bit vector into an integer. The constructor *bitv(x,y)* builds a new bit-vector by concatening x as the least significant bit of the vector y, and the constant *Btm* is the empty vector. The correctness theorem states that when given two bit-vectors of the same size as inputs, the resulting output is, up to conversion, the arithmetic sum of the inputs. The conjecture to be proved is:

$$size(x_1) = size(x_2) \Rightarrow bvtonat(add(x_1, x_2, False)) = bvtonat(x_1) + bvtonat(x_2)$$

Using our techniques described in section 5, the test set computed for the specification is: $\{Btm; bitv(True, x_1); bitv(False, x_1); 0; s(x_1); True; False\}$, where: *Btm*, $bitv(True, x_1)$ and $bitv(False, x_1)$ are of type *vect*, 0 and $s(x_1)$ are of type *nat*, *True* and *False* are of type *bool*. The next step consists in applying an induction on the *induction variables* (see section 4). Here, the variables x_1 and x_2 are replaced by elements of the test set (whose variables are renamed), and the instances obtained are simplified. We thus obtain 9 subgoals to be proved.

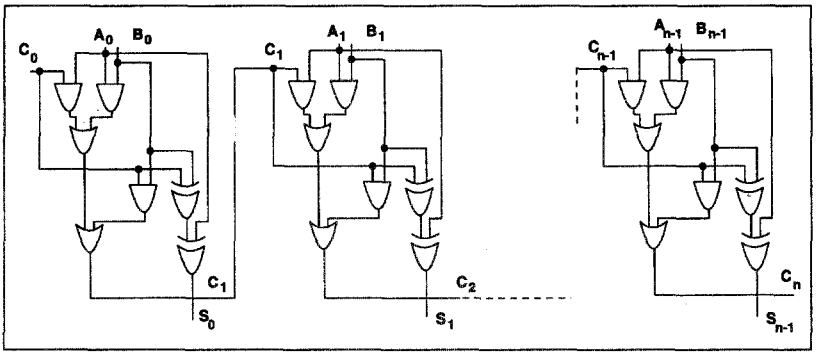


Fig. 1. Ripple carry adder

The simplification strategy may use axioms, other conjectures (even when they are not proved) and inductive hypotheses, provided they are smaller (w.r.t a well founded ordering on clauses). For example, the subgoal:

$$size(bitv(True, x_1)) = size(bitv(False, x_2)) \Rightarrow$$
$$s(bvtonat(add(x_1, x_2, False)) + bvtonat(add(x_1, x_2, False))) =$$
$$bvtonat(bitv(True, x_1)) + bvtonat(bitv(False, x_2))$$

is simplified using the axioms to:

$$size(x_1) = size(x_2) \Rightarrow$$
$$bvtonat(add(x_1, x_2, False)) + bvtonat(add(x_1, x_2, False)) =$$
$$bvtonat(x_1) + bvtonat(x_1) + bvtonat(x_2) + bvtonat(x_2).$$

This simplification is not possible if we simply use the commutativity and associativity of + as lemmas, since then it would not be possible to derive a clause which is smaller than the starting one.
After simplifying and deleting the tautologies, only one subgoal remains to be proved:

$$size(x_1) = size(x_2) \Rightarrow$$
$$bvtonat(add(x_1, x_2, True)) + bvtonat(add(x_1, x_2, True)) =$$
$$s(s(bvtonat(x_1) + bvtonat(x_1) + bvtonat(x_2) + bvtonat(x_2))).$$

This is the case for the addition of 2 bit-vectors when the carry is set to True. An induction on x_1 and x_2 must be applied. We obtain 9 subgoals to prove, and after simplification, 2 conjectures remain. The first one is:

$$size(x_1) = size(x_2) \Rightarrow$$
$$bvtonat(add(x_1, x_2, False)) + bvtonat(add(x_1, x_2, False)) +$$
$$bvtonat(add(x_1, x_2, False)) + bvtonat(add(x_1, x_2, False)) =$$
$$bvtonat(x_1) + bvtonat(x_1) + bvtonat(x_1) + bvtonat(x_1) + bvtonat(x_2) +$$
$$bvtonat(x_2) + bvtonat(x_2) + bvtonat(x_2)$$

It is reduced to a trivial identity using *inductive contextual rewriting* (see section 6) with the induction hypothesis:

$$size(x_1) = size(x_2) \Rightarrow bvtonat(add(x_1, x_2, False)) = bvtonat(x_1) + bvtonat(x_2)$$

The other remaining conjecture is simplified in the same way. Hence, all the subgoals are proved, and the initial goal is valid. The proof is completely automatic. Besides, it is easy to understand and very close to a mathematical proof. Note also that it does not use any specialized tactic nor any heuristic. The same conjecture has been proved with the NQTHM system [14], but then requires a non trivial generalisation of the input theorem. A proof was also done with PVS [7], but it uses a high-level user-defined proof strategy.

2 Basic concepts and notations

We assume that the reader is familiar with the basic concepts of term algebra, term rewriting, equational reasoning and mathematical logic. A many sorted signature σ is a pair $(\mathcal{S}, \mathcal{F})$ where $\mathcal{S}$ is a set of *sorts* and $\mathcal{F} = F \cup F_{AC}$, where F and F_{AC} denote sets of function symbols. For short, a many sorted signature σ will simply be denoted by $\mathcal{F}$. The *variadic term algebra* is a generalisation of the term algebra, where AC functions symbols have a non fixed arity [12]. It allows us to express associative and commutative axioms by means of *flattening*. The variadic term algebra $TV(F, F_{AC}, \mathcal{X})$ over the signature $\mathcal{F}$ and the set of variables $\mathcal{X}$ is defined as the smallest set TV containing $\mathcal{X}$ such that:

- if $f \in F, arity(f) = n \geq 0$, and $t_1, \ldots, t_n \in TV$ then $f(t_1, \ldots, t_n) \in TV$.
- if $f \in F_{AC}, n \geq 2$, and $t_1, \ldots, t_n \in TV$ then $f(t_1, \ldots, t_n) \in TV$

In the following, $+$ will denote an AC symbol. *Flattening* a term consists of rewriting it to normal form w.r.t. the set of *flattening* rules: $f(x_1, \ldots, f(y_1, \ldots, y_r), z_1, \ldots, z_n) \rightarrow f(x_1, \ldots, y_1, \ldots, y_r, \ldots, z_1, \ldots, z_n)$ for all $f \in F_{AC}$. We denote by $flat(t)$, the term obtained by flattening t. A term s is *flattened* if $s = flat(s)$. We assume that we have a partition of $\mathcal{F}$ in two subsets, the first one, C, contains the *constructor symbols* and the second, D, is the set of *defined symbols*. We denote by $Var(t)$, the set of all variables appearing in t. A term is *linear* if all its variables occur only once in it. If $Var(t)$ is empty then t is a *ground* term. The set of all ground terms is $T(\mathcal{F})$. A *substitution* assigns terms of appropriate sorts to variables. Let t be a term, and η be a substitution, $t\eta$ is the flattened term obtained by applying η to t. The domain of η is defined by: $Dom(\eta) = \{x \mid x\eta \neq x\}$. If η applies every variable to a ground term, then η is a ground substitution. We denote by $\equiv$ the syntactic equivalence between objects. The smallest congruence generated by the equations $f(f(x, y), z) = f(x, f(y, z))$ and $f(x, y) = f(y, x)$ for all $f \in F_{AC}$ is denoted by $=_{AC}$. *Positions* in a term are defined in the same way than in [12]. The *replacement* of a term s by t at a position p is denoted by $s[p \leftarrow t]$. We assume that the term obtained is flattened. The term t/p is the subterm of t at position p. The notation $t[s]_p$ means that the term t contains a

subterm s at position p. We also denote by $t(p)$ the symbol of t at position p. For example, if $t = a+b+c$, then $t/\{2,3\} = b+c, t(\{2,3\}) = +, t/3 = c$. A position u in a term t such that $t(u) = x$ and $x \in \mathcal{X}$, is a *linear variable position* if x occurs only once in t, otherwise, u is a *non linear variable position.* A position u is a *strict position* of a term t if $t(u) \notin \mathcal{X}$, and $u = \varepsilon$ or $u = u'.i$ $(i \in \mathbb{N})$. The depth of a term t is defined as follows: $depth(t) = 0$ if t is a constant or a variable, otherwise, $depth(f(t_1, \ldots, t_n)) = 1 + max(depth(t_i))$. The strict depth of a term t, denoted by $sdepth(t)$, is the maximum of length of function positions in t.

A term s *matches* a term t if there exists a substitution σ such that $t =_{AC} s\sigma$; the term t is called an *AC instance* of s. A term t is *AC unifiable* with a term s, if there exists a substitution σ such that $t\sigma =_{AC} s\sigma$.

An ordering $\succ$ is *AC compatible* if $s' =_{AC} s$, $s \succ t$ and $t =_{AC} t'$ implies $s' \succ t'$. In the following, we suppose that $\succ$ is a transitive irreflexive relation on the set of terms, that is noetherian, monotonic ($s \succ t$ implies $w[s]_u \succ w[t]_u$), stable ($s \succ t$ implies $s\sigma \succ t\sigma$), AC compatible and satisfy the subterm property ($f(\cdots, t, \cdots) \succ t$). The multiset extension of $\succ$ will be denoted by $\gg$.

A *conditional* equation a formula of the following form: $a_1 = b_1 \wedge \cdots \wedge a_n = b_n \Rightarrow l = r$. It will be written $a_1 = b_1 \wedge \cdots \wedge a_n = b_n \Rightarrow l \to r$ and called a *conditional rule* if $\{l\sigma\} \gg \{r\sigma, a_1\sigma, b_1\sigma, \cdots, a_n\sigma, b_n\sigma\}$ for each substitution σ and every variable of the conditional equation occurs in l. The term l is the *left-hand side* of the rule. A rewrite rule $c \Rightarrow l \to r$ is *left-linear* if l is linear. A set of conditional rules is called a *rewrite system.* A constructor is *free* if it is not the root of a left-hand side of a rule. We denote by $lhss(R)$, the set of subterms of all left-hand sides of R. The number of elements of a set T is $card(T)$. A rewrite system R is *left-linear* if every rule in R is left-linear. We say that R is flattened if all its left-hand sides are flattened. Let f be an AC symbol, we denote by b_f the maximal arity of f in the left-hand sides of R. The depth (resp. strict depth) of a rewrite system R, denoted by $depth(R)$ (resp. $sdepth(R)$), is the maximum of the depths (resp. strict depth) of its flattened left-hand sides. We define $D(R)$ as $depth(R) - 1$ if $sdepth(R) < depth(R)$ and R is left-linear, otherwise $depth(R)$.

Let t be a flattened term, we write $t \to_R t'$ if there exists a conditional rule $\wedge_{i=1}^{n} a_i = b_i \Rightarrow l \to r$ in R, a position p and a substitution σ such that:

- $t/p =_{AC} l\sigma$, $t' =_{AC} t[p \leftarrow r\sigma]$.
- *for all* $i \in [1 \cdots n]$ there exists c_i, c_i' such that $a_i\sigma \to_R^* c_i$, $b_i\sigma \to_R^* c_i'$ and $c_i =_{AC} c_i'$.

where the reflexive-transitive closure of $\to$ is denoted by $\to^*$. In this case we say that the term t is *reducible*, otherwise, it is *irreducible.* From now on, we assume that there exists at least one irreducible ground term of each sort. We say that two terms s and t are *joinable*, if $s \to_R^* v$, $t \to_R^* v'$ and $v =_{AC} v'$. A term t is *inductively reducible* iff all its ground instances are reducible. A symbol $f \in \mathcal{F}$ is *completely defined* if all ground terms with root f are reducible. We say that R is *sufficiently complete* if all symbols in D are completely defined.

A *clause* C is an expression of the form: $\neg(s_1 = t_1) \vee \neg(s_2 = t_2) \vee \cdots \vee \neg(s_n = t_n) \vee (s_1' = t_1') \vee \cdots \vee (s_m' = t_m')$. We naturally extend the notion of *flat-*

tening, substitution, positions to clauses. Let H be a set of conditional equations and F_{AC} a set of AC symbols. The clause C is a *logical consequence* of H if C is valid in any model of $H \cup \{f(f(x,y),z) = f(x,f(y,z)),\ f(x,y) = f(y,x)$ for all $f \in F_{AC}\}$. This will be denoted by $H \models C$. We say that C is *inductively valid* in H and denote it by $H \models_{ind} C$, if for any ground substitution σ, *(for all* i $H \models s_i\sigma = t_i\sigma$ *) implies (there exists* j *such that* $H \models s'_j\sigma = t'_j\sigma$*)*. The rewrite system R is *ground convergent* if the terms u and v are joinable whenever $u, v \in T(F)$ and $R \models u = v$. In this paper, we suppose that all clauses and terms are flattened, and we denote by R a flattened rewrite system.

3 Induction schemes

To prove a conjecture by induction, the prover computes automatically an induction scheme, which consists of a set of variables on which induction is applied and a set of terms covering the possibly infinite set of irreducible ground terms.

Definition 3.1 *Given a term* t*, a set* $V \subseteq Var(t)$ *and a set of terms* T*, a* (V,T)*-substitution is a substitution of domain* V*, such that for all* $x \in V$*,* $x\sigma$ *is an element of* T *whose variables have been given new names.*

Definition 3.2 *An induction scheme* $\mathcal{I}$ *for a term* t *is a couple* (V,T)*, with* $V \subseteq Var(t)$ *and* $T \subseteq T(F, F_{AC}, \mathcal{X})$*, such that: for every ground irreducible term* s*, there exists a term* t *in* T *and a ground substitution* σ *such that* $t\sigma =_{AC} s$*.*

These induction schemes allow us to prove theorems by induction, by reasoning on the domain of irreducible terms rather than on the whole set of terms. However, they cannot be used to refute false conjectures. In the following, we refine induction schemes so that to be able, not only to prove conjectures, but also to refute the false ones.

Definition 3.3 *A term* t *is strongly irreducible if none of its subterms is an instance of a left-hand side of a rule in* R*.*

Definition 3.4 *A strong induction scheme* $\mathcal{I}$ *for a term* t *is an induction scheme* (V,T)*, where* V *is called the set of induction variables, and* T *is called the test set, such that: for each term* t *and* $\mathcal{I}$*-substitution* σ*, if* $t\sigma$ *is strongly irreducible, then there exists a ground substitution* τ *such that* $t\tau$ *is irreducible.*
An $\mathcal{I}$*-substitution is called* test substitution*.*

The next definition provides us with a criteria to reject false conjectures. Then, we show that strong induction schemes are fundamental for this purpose (see theorem 3.1)

Definition 3.5 *A clause* $\neg(s_1 = t_1) \vee \cdots \vee \neg(s_m = t_m) \vee (g_1 = d_1) \vee \cdots \vee (g_n = d_n)$ *is provably inconsistent with respect to* R *if there exists a test substitution* σ *such that:*

1. $\forall i \in [1 \cdots m] : s_i\sigma = t_i\sigma$ *is an inductive theorem w.r.t.* R*.*

2. $\forall j \in [1 \cdots n] : g_j\sigma \neq_{AC} d_j\sigma$, *and the maximal elements of* $\{g_j\sigma, d_j\sigma\}$ *w.r.t.*$\succ$ *are strongly irreducible.*

The next result shows that a provably inconsistent clause cannot be inductively valid w.r.t. R. This is proved by building a well-chosen ground instance of the clause which gives us a counterexample.

Theorem 3.1 *Suppose R is a ground convergent rewriting system. If a clause C is provably inconsistent, then C is not inductively valid w.r.t R.*

In the following sections, we propose methods for automatically computing each component of a strong induction scheme, that are, the induction variables and the test set.

4 Selecting induction variables

To prove a conjecture by induction, the prover selects automatically the induction variables of the conjecture where induction must be applied, then, instanciates them with terms of the test set. It is clear that the less induction variables we have, the more efficient the induction procedure will be.

To determine induction variables, the prover computes first the induction positions of the functions. These positions enable to decide whether a variable position of a term t is an induction variable or not. The induction positions computation is done only once and before the proof process. It is independent from the conjectures to be proved since it is based only on the given conditional theory.

Definition 4.1 *Let t be a term such that $t(\varepsilon) = f$ and $f \in \mathcal{F}$. A position $i \in \mathbb{N}$ is an inductive position of f in t if i is either a strict position in t, or a non linear variable position. We define $pos\_ind(f,t)$ as the set of inductive positions of f in t and $pos\_ind(f) = \bigcup_{t \in lhss(R)} pos\_ind(f,t)$.*

The idea is that a variable in a term t will be selected as an induction variable if it occurs below an inductive position. Hence, instantiating these variables may trigger a rewriting step. A problem happens with an AC symbol f, since the inductive positions of f can be permuted. For example, let $R = \{x + 0 + 0 \rightarrow 0\ ,\ x + 1 + 1 \rightarrow 0\}$, with $F_{AC} = \{+\}$ and $F = \{0, 1\}$. We have $pos\_ind(+) = \{2, 3\}$. Considering only y and z as induction variables, the proof of the conjecture $x + y + z = 0$ fails. However, it is an inductive theorem since all its ground instances are logical consequences of R.

This leads us to take all variables occuring under an AC symbol, as induction variables, so that to ensure the refutational completeness of our procedure, that is, whenever the proof of a clause finitely fails, we can ensure that it is not an inductive consequence of R. However, in order to make the proof process efficient, we have identified some cases where the number of induction variables to consider can be reduced while preserving refutational completeness.

For this purpose, for each $f \in F_{AC}$, we define the number $nb\_pos\_ind(f) = max_{t \in lhss(R)} card(pos\_ind(f,t))$. We denote by $var\_ind(t)$ the set of induction variables of t. The procedure computing induction variables is given in figure 2. We assume that the three predicates P_1, P_2, P_3 are defined as follows:

$P_1(f,R) \Leftrightarrow f$ is completely defined and $nb\_pos\_ind(f) = 1$
$P_2(f,R) \Leftrightarrow f$ is completely defined and $nb\_pos\_ind(f) > 1$
$P_3(f,R) \Leftrightarrow R$ is left-linear and for each $f(t_1,\ldots,t_n) \in lhss(R)$ there does not exist two non variable terms t_i, t_j which are AC unifiable

input: t **output:** $var\_ind(t)$ **init:** $Vind := \emptyset$
if t is a variable
then $Vind := \{t\}$
else for each position u in t such that $t(u) = f$ and $f \in F$ **do:**
$Vind := Vind \cup \{x \mid x$ appears at position $u.i$, and $i \in pos\_ind(f)\}$
endfor
for each $f \in F_{AC}$ in t **do:**
case 1: $P_1(f,R)$ and there is a variable x which is an argument of each occurrence of f in t: $Vind := Vind \cup \{x\}$
endcase 1
case 2: $P_2(f,R)$:
for each position u in t such that $t(u) = f$ **do:**
let $X_u = \{x \in \mathcal{X} \mid x$ appears at a position $u.i (i \in \mathbb{N})\}$
if $(X_u \cap Vind = \{x\})$ and $(\exists y \in X_u)$
then $Vind := Vind \cup \{y\}$
else if $(X_u \cap Vind = \emptyset)$ and $(X_u = \{x\})$
then $Vind := Vind \cup \{x\}$
else if $(X_u \cap Vind = \emptyset)$ and $(\{x,y\} \subseteq X_u)$
then $Vind := Vind \cup \{x,y\}$
endif
endif
endif
endcase 2
case 3: $P_3(f,R)$ and there is a variable x which is an argument of each occurrence of f in t: $Vind := Vind \cup \{x\}$
endcase 3
case 4: otherwise:
for each position u such that $t(u) = f$ **do:**
let $X_u = \{x \in \mathcal{X} \mid x$ appears at a position $u.i (i \in \mathbb{N})\}$
$Vind := Vind \cup X_u$
endfor
endcase 4
endfor
endif
return$(Vind)$

Fig. 2. Induction variables computation

Example 4.1 *Consider the following rewriting system R, with $F_{AC} = \{+, *\}$.*

$$R = \begin{cases} x + 0 \to x & x + s(y) \to s(x + y) \\ x * 0 \to 0 & x * s(y) \to x + x * y \\ exp(z, 0) \to s(0) & exp(z, s(n)) \to z * exp(z, n) \end{cases}$$

We have: $nb\_pos\_ind(+) = 1$, $nb\_pos\_ind(*) = 1$, $pos\_ind(exp) = 2$. A test set for R is $\{0, s(x)\}$. The conjecture to prove is $(x+y+z)*w = x*w+y*w+z*w$. If we take x, y, z, w as induction variables, we would obtain 16 lemmas to prove. Now, since $+$ and $*$ are completely defined AC operators, $nb\_pos\_ind(+) = 1$ and $nb\_pos\_ind(*) = 1$, we can choose $\{x, w\}$ or $\{y, w\}$ or $\{z, w\}$ as induction variables. Thus, we obtain only 4 lemmas to prove.

5 Computing test sets

The computation of test sets according to definition 3.4 relies on the inductive reducibility property [9], which is unfortunately undecidable in AC theories [10]. However we can use a semi-decision procedure that has proved to be quite useful for practical applications [11]. For the more restricted case where the rewrite system is left-linear and sufficiently complete, and the relations between constructors are equational we propose algorithms basically extending the equational case [9, 5] (see theorem 5.1). For the case where the rewrite system is not left linear but it is sufficiently complete over free constructors there is an easy algorithm to produce test-sets (see theorem 5.2).

We denote by $extension(t)$ the term obtained by replacing each subterm of t of the form $f(t_1, \ldots, t_{b_f+1})$ by $f(t_1, \ldots, t_{b_f+1}, x)$, where $f \in F_{AC}$ and x is a new variable. Given a set of terms T, $extension(T) = \bigcup_{t \in T} extension(t)$.

Theorem 5.1 *Let R be a left-linear conditional rewriting system. Let $T = \{t \mid t$ is a term of depth $\leq D(R)$ such that all variables occur at depth $D(R)$, and each AC operator has a number of arguments $\leq b_f + 1\}$. Let $T' = \{t \in T \mid t$ is not inductively reducible $\}$. Then $extension(T')$ is a test set for R.*

Example 5.1 *Let $F = \{0, 1, s\}$, $F_{AC} = \{+\}$ and*

$$R = \begin{cases} 0 + x \to x, \\ 1 + s(x) \to s(s(x)), \\ s(1) \to s(s(0)) \end{cases}$$

We have: $D(R) = 1$. By applying theorem 5.1, we obtain: $T' = \{0,\ 1,\ s(x),\ x + y,\ x + y + z\}$, $extension(T') = \{0,\ 1,\ s(x),\ x + y,\ x + y + z,\ x + y + z + t\}$, which can be simplified by deleting the subsumed terms and give the test set: $\{0,\ 1,\ s(x),\ x + y\}$.

A sort $s \in \mathcal{S}$ is said *infinitary* if there exists an infinite set of ground irreducible terms of sort $\mathcal{S}$. The next theorem provides a method for constructing test sets for non left-linear rewriting system with free constructors.

Theorem 5.2 *Let R be a conditional rewriting system. Suppose that R is sufficiently complete over free constructors. Then, the set T of all constructors terms of depth $\leq D(R)$ such that all variables are infinitary and occur at a depth $D(R)$, is a test set for R.*

6 Inference system

Our inference system rules (see figure 3) is based on a set of transition rules applied to (E, H), where E is the set of conjectures to prove and H is the set of induction hypotheses. The initial set of conditional rules R is oriented with a well founded and AC compatible ordering.

The inference system is constituted of two rules: *generation* and *simplification*. An *I-derivation* is a sequence of states: $(E_0, \emptyset) \vdash_I (E_1, H_1) \vdash_I \ldots (E_n, H_n) \vdash_I \ldots$. We say that an I-derivation is *fair* if the set of persistent clauses $(\cup_i \cap_{j \geq i} E_j)$ is empty. An I-derivation fails when it is not possible to extend it by one more step and there remains conjectures to prove. We denote by $\prec_c$ a noetherian ordering on clauses, stable modulo AC, that extends $\prec$. In the following, W denotes a set of conditional equations which can be induction hypotheses or conjectures not yet proved. Let us now present briefly the rewriting techniques used by the prover. Inductive contextual rewriting is a generalization of both inductive rewriting [3] and contextual rewriting [15].

Definition 6.1 (Inductive contextual rewriting) *Given a clause C, we write:*

$$C \equiv \Delta \Rightarrow A \longmapsto_{R<W>} C' \equiv \Delta \Rightarrow A[u \leftarrow t\sigma]$$

if there exists $\delta \equiv \Gamma \Rightarrow s = t \in R \cup W$ and a position u in A such that:

- $A/u =_{AC} s\sigma$
- $C' \prec_c C$
- *if* $\delta \in W$ *then* $\delta\sigma \prec_c C$
- $R \cup W^{\prec_c} \models_{ind} \Delta \Rightarrow \Gamma\sigma$

where $W^{\prec_c} = \{\Phi \mid \Phi \in W \text{ and } \Phi \prec_c C\}$.

Example 6.1 *Let $F_{AC} = \{+\}$ and $C \equiv (odd(1+1) = True \vee equal(1,1) = False \vee even(1+1) = True)$. Suppose that we have an induction hypothesis: $H \equiv (equal(x,y) = False \vee even(x+y) = True)$. The inductive contextual rewriting of C by H gives: $C' \equiv (odd(1+1) = True \vee equal(1,1) = False \vee True = True)$.*

Inductive case rewriting provides us with a possibility to perform a case-based reasoning; it simplifies a conjecture with an axiom, a conjecture or an inductive hypothesis, provided it is smaller than the initial conjecture and the disjunction of all conditions is inductively valid.

Definition 6.2 (Inductive case rewriting) *Let G be the set $\{< C[u \leftarrow d\sigma], P\sigma > |$ there exists $\mathcal{R} \equiv P \Rightarrow g \rightarrow d$ in $R \cup W$, and a position u in C such that $C/u =_{AC} g\sigma$, and if $\mathcal{R} \in W$, then $\mathcal{R} \prec_c C\}$. If $R \models_{ind} (\bigvee_{<C',P> \in G} P)$, then* Inductive_case_rewriting(C,W) $= \{P \Rightarrow C' \mid < C', P > \in G\}$.

Generation: $(E \cup \{C\}, H) \vdash_I (E \cup E', H \cup \{C\})$
if $E' = \bigcup_\sigma simplify(C\sigma, H \cup E \cup \{C\})$, σ ranging over test substitutions of C

Simplification: $(E \cup \{C\}, H) \vdash_I (E \cup E', H)$
if $E' = simplify(C, H \cup E)$

Fig. 3. Inference system I

Definition 6.3 (simplify) *The procedure* simplify *is defined in the following way:*

simplify(C, W) =
  if *C is a tautology or subsumed by a clause of* $R \cup W$
  then $\emptyset$
  else if $C \longmapsto_{R<W>} C'$
    then $\{C'\}$
    else Inductive_case_rewriting(C,W)

The correctness of the inference system I is expressed by the following theorem:

Theorem 6.1 (Correctness) *Let* $(E_0, \emptyset) \vdash_I (E_1, H_1) \vdash_I \ldots$ *be a fair I-derivation. If it does not fail then* $R \models_{ind} E_0$.

Now, consider boolean specifications. To be more specific, we assume there exists a sort *bool* with two free constructors $\{true, false\}$. Every rule in R is of type: $\wedge_{i=1}^{n} p_i = p_i' \Rightarrow s \rightarrow t$ where for all i in $[1 \cdots n]$, $p_i' \in \{true, false\}$. Conjectures will be *boolean clauses*, i.e. clauses whose negative literals are of type $\neg(p = p')$ where $p' \in \{true, false\}$. If for all rules of form $p_i \Rightarrow f(t_1, \ldots, t_n) \rightarrow r_i$ whose left hand sides are identical up to a renaming μ_i, we have $R \models_{ind} \vee_i p_i\mu_i$, then f is weakly complete w.r.t R. We say that R is weakly complete if any function in $\mathcal{F}$ is weakly complete [1]. We can show that refutational completeness is also preserved in the AC case.

Theorem 6.2 (Refutational completeness) *Let R be a weakly complete and ground convergent rewrite system. Let* E_0 *be a set of boolean clauses. Then* $R \not\models_{ind} E_0$ *iff all fair derivations issued from* $(E_0, \emptyset)$ *fail.*

7 Conclusion

We have presented a new induction procedure for the associative and commutative theories. An advantage of this approach is that inference steps are performed in a homogeneous well-defined framework. Another important point is that our

procedure does not need AC unification like completion methods, but only AC matching. Our inference system is based on two rules: the generation rule which performs induction, and the simplification rule which simplifies conjectures by elaborated rewriting techniques. This system is correct and refutationally complete for boolean ground convergent rewrite systems under reasonable restrictions. In experiments, refutational completeness is particularly useful for debugging specifications. These results have been integrated in the prover SPIKE-AC, and interesting examples such as circuits verification have demonstrated the advantages of the approach.

References

1. A. Bouhoula. Using induction and rewriting to verify and complete parameterized specifications. *Theoretical Computer Science*, 170, December 1996.
2. A. Bouhoula, E. Kounalis, M. Rusinowitch. Automated mathematical induction. *Journal of Logic and Computation*, 5(5):631-668, 1995.
3. A. Bouhoula, M. Rusinowitch. Implicit induction in conditional theories. *Journal of Automated Reasoning*, 14(2):189–235, 1995.
4. R. S. Boyer, J. S. Moore. *A Computational Logic Handbook*. 1988.
5. R. Bündgen, W. Küchlin. Computing ground reducibility and inductively complete positions. In N. Dershowitz, editor, *Rewriting Techniques and Applications*, *LNCS* 355, pages 59–75, 1989.
6. J.R. Burch, E. M. Clarke, K.L. McMillan, D.L. Dill. Symbolic Model Checking: 10^{20} states and beyond. *5th Annual IEEE Symposium on Logic in Computer Science*, pages 428-439, 1990.
7. D. Cyrluk, S. Rajan, N. Shankar, M. K. Srivas. Effective Theorem Proving for Hardware Verification. In K. Ramayya and K. Thomas, editors, *Theorem Provers in Circuit Design LNCS* 901, pages 203–222, 1994.
8. S. J. Garland, John V. Guttag. An overview of LP, the Larch Prover. In N. Dershowitz, editor, *Rewriting Techniques and Applications*, *LNCS* 355, pages 137–151, 1989.
9. J.-P. Jouannaud, E. Kounalis. Automatic proofs by induction in theories without constructors. *Information and Computation*, 82:1–33, 1989.
10. D. Kapur, P. Narendran, D. J. Rosenkrantz, H. Zhang. Sufficient completeness, ground-reducibility and their complexity. *Acta Informatica*, 28:311–350, 1991.
11. E. Kounalis M. Rusinowitch. Reasoning with conditional axioms. *Annals of Mathematics and Artificial Intelligence*, (15):125–149, 1995.
12. C. Marché. *Réécriture modulo une théorie présentée par un système convergent et décidabilité du problème du mot dans certaines classes de théories équationnelles.* Th. univ., Université de Paris-Sud (France), 1993.
13. S. Owre, J.M. Rushby, N. Shankar. A prototype verification system. In D. Kapur, editor, *International Conference on Automated Deduction*, *LNAI* 607, pages 748–752, 1992.
14. L. Pierre. An automatic generalisation method for the inductive proof of replicated and parallel architetures. In K. Ramayya and K. Thomas, editors, *Theorem Provers in Circuit Design*, *LNCS* 901, pages 72–91, 1994.
15. H Zhang. Contextual rewriting in automated reasoning. *Fundamenta Informaticae*, (24):107–123, 1995.

Analysis of Timed Systems Based on Time-Abstracting Bisimulations

S. Tripakis* and S. Yovine*

VERIMAG, France

Abstract. We adapt a generic minimal model generation algorithm to compute the coarsest finite model of the underlying infinite transition system of a timed automaton. This model is minimal modulo a time-abstracting bisimulation. Our algorithm uses a refinement method that avoids set complementation, and is considerably more efficient than previous ones. We use the constructed minimal model for verification purposes by defining abstraction criteria that allow to further reduce the model and to compare it to a specification.

1 Introduction

Behavioral equivalences based on bisimulation relations have proven useful for verifying the correctness of concurrent systems. They allow comparing an implementation to a usually more abstract specification both represented as labeled transition systems. This approach also allows reducing the size of the system by identifying equivalent states which is crucial to avoid the explosion of the state-space. Since the introduction of strong bisimulation in [Mil80], many equivalences have been defined. Moreover, practice followed theory and several algorithms and tools have been developed.

Despite this fact, behavioral equivalences have not been thoroughly studied in the framework of timed systems. In particular, there is a lack of tools based on this approach. The transition system modeling the behavior of a timed system comprises two kinds of transitions, namely *timeless actions* representing the discrete evolutions of the system, and *time lapses* corresponding to the passage of time. Due to density of time, there are infinitely many time transitions. A finite model can be obtained by defining an appropriate equivalence relation inducing a finite number of equivalence classes. Examples of such relations are the *region-graph* equivalence [AD94] and the *ta-bisimulation* [LY93]. The main idea behind these relations is that they abstract away from the exact amount of time elapsed and they are therefore refer to as *time-abstracting* equivalences.

An important problem consists in constructing the quotient of a labeled transition system w.r.t. an equivalence relation. Many generic algorithms exist to solve this problem, e.g. [BFH+92, LY92]. For timed systems represented by timed automata [AD94], these algorithms have been adapted for computing the

* E-mail: {Stavros.Tripakis,Sergio.Yovine}@imag.fr. Tel: +33 76 90 96 30. Fax: +33 76 41 36 20. Miniparc-Zirst, Rue Lavoisier, 38330 Montbonnot St. Martin.

minimal region graph in [ACD+92b, ACD+92a]. Based on the results reported in [ACD+92a] it comes out that straightforward implementations of those algorithms result in poor performances. In fact, one main obstacle towards efficiency is the cost of computing set complementation.

In this paper, we adapt the generic minimal model generation algorithm of [BFH+92] in order to avoid set complementation, in the spirit of [YL93]. Experimental results carried out on several benchmarks show that this algorithm is more efficient than the ones implemented in [ACD+92a]. Furthermore, we use the constructed minimal model for verification purposes by defining an appropriate abstraction criterion that allows using the tool ALDEBARAN [FGM+92] for further reducing the transition system or comparing it to a specification.

2 Background

2.1 Bisimulations, models, and minimal models

A *model* (or LTS) is a triple $\langle Q, Q^0, \rightarrow \rangle$. Q is a set of states, $Q^0 \subseteq Q$ is the set of initial states, and $\rightarrow \subseteq Q \times L \times Q$ is a set of labeled transitions, for some label set L. We write $q \xrightarrow{l} q'$ instead of $(q, l, q') \in \rightarrow$. A relation $r \subseteq Q \times Q$ is a *bisimulation* iff : $\forall (q_1, q_2) \in r$, $\forall l \in L$,

(1) $\forall q_1' \in Q$ s.t. $q_1 \xrightarrow{l} q_1'$, $\exists q_2'$ s.t. $q_2 \xrightarrow{l} q_2'$ and $(q_1', q_2') \in r$, and

(2) $\forall q_2' \in Q$ s.t. $q_2 \xrightarrow{l} q_2'$, $\exists q_1'$ s.t. $q_1 \xrightarrow{l} q_1'$ and $(q_1', q_2) \in r$.

From now on, $\approx$ denotes the greatest bisimulation. Two models G_1, G_2, $G_i = \langle Q_i, Q_i^0, \rightarrow_i \rangle$, $i = 1, 2$, are bisimilar, denoted $G_1 \approx G_2$, if $\forall q_1 \in Q_1^0, q_2 \in Q_2^0$, $q_1 \approx q_2$.

Let $G = \langle Q, Q^0, \rightarrow \rangle$. A *partition* Π of Q is a set of disjoint *classes* $B \subseteq Q$, the union of which yields Q. The quotient of G w.r.t. Π is $\langle \Pi, \pi, \rightarrow \rangle$, where $\pi = \{B \in \Pi \mid B \cap Q^0 \neq \emptyset\}$, and $B \xrightarrow{l} C$ iff $pre_l(B, C) \neq \emptyset$, where $pre_l(B, C) = \{q \in B \mid \exists q' \in C.\ q \xrightarrow{l} q'\}$. We write $B \rightarrow C$ if $\exists l \in L.\ B \xrightarrow{l} C$. We define $Succs_\pi(B) = \bigcup_{l \in L} Succs_\Pi^l(B)$ where $Succs_\Pi^l(B) = \{C \in \Pi \mid B \xrightarrow{l} C\}$ is the set of successors of B by l, and $Preds_\pi(B) = \bigcup_{l \in L} Preds_\Pi^l(B)$ where $Preds_\Pi^l(B) = \{C \in \Pi \mid C \xrightarrow{l} B\}$ is the set of predecessors of B by l.

B is *stable* w.r.t. C if $\forall l \in L.\ pre_l(B, C) \in \{B, \emptyset\}$. B is stable w.r.t. Π if it is stable w.r.t. all classes $C \in \Pi$. Π is stable if all its classes are stable w.r.t. Π. Let $\Pi_\approx$ be the partition induced by $\approx$. Clearly, $\Pi_\approx$ is stable. The *minimal model* of G modulo bisimulation, is the quotient of G w.r.t. $\Pi_\approx$, denoted $G_\approx$. Notice that $\forall l \in L, B, C \in \Pi_\approx$, $B \xrightarrow{l} C$ iff $pre_l(B, C) = B$.

2.2 A general minimal model generation algorithm

We recall here the generic algorithm developed in [BFH+92] (referred to as MMGA) for computing the reachable part of the minimal model $G_\approx$.

$\Pi := \Pi_0$; $\alpha := \{B \in \Pi_0 \mid B \cap Q^0 \neq \emptyset\}$; $\sigma := \emptyset$;
while $(\exists B \in \alpha \setminus \sigma)$ **do** { (0)
$C_B := Split(B, \Pi)$; (1)
if $(C_B = \{B\})$ **then** { (2)
$\sigma := \sigma \cup \{B\}$; $\alpha := \alpha \cup Succs^l_\Pi(B)$; (3)
} **else** { (4)
$\alpha := \alpha \setminus \{B\}$; $\Pi := (\Pi \setminus \{B\}) \cup C_B$; $\sigma := \sigma \setminus Preds_\pi(B)$; (5)
if $B \cap Q^0 \neq \emptyset$ **then** $\alpha := \alpha \cup \{C \in C_B, \mid C \cap Q^0 \neq \emptyset\}$;
} }

Π denotes the current partition, α the set of accessible classes (i.e., containing at least one accessible state), and $\sigma \subseteq \alpha$ the set of stable accessible classes. $Split(B, \Pi)$ refines the class B by choosing a class C w.r.t. which B is potentially unstable, then computing $B_1 = pre_l(B, C)$, $B_2 = B \cap \overline{pre_l(B, C)}$. If indeed $B_i \neq \emptyset, i = 1, 2$, B is *effectively* split (4), and its predecessors become unstable (5). Otherwise (2), B is both accessible (i.e., it contains a reachable state, say q) and stable, meaning that each one of its successors C has a state q' such that $q \rightarrow q'$. Thus, C contains at least one reachable state, and can be inserted to α (3). Termination depends on whether $\approx$ induces a finite partition of the initial model.

2.3 Avoiding complementation

In the context of timed systems set complementation is very costly and should be avoided. This can be done following the idea presented in [YL93]. Let us first illustrate it with an example. Assume that $B \in \alpha$ is found stable, so that one of its successors, C, becomes accessible, and is split into C_1, C_2, C_3, thus B is no longer stable. Now, instead of splitting B w.r.t. only one of the C_i's, which would yield $\{pre_l(B, C_i), B \cap \overline{pre_l(B, C_i)}\}$, B can be split directly into B_1, B_2, B_3, where $B_i = pre_l(B, C_i)$ (possibly, some B_i's are empty). Now, let

$$Ref^l_\Pi(B) \stackrel{\text{def}}{=} \{B' \mid \exists C \in Succs^l_\Pi(B).\ B' = pre_l(B, C) \wedge B' \neq \emptyset\}.$$

Assuming that whenever $Ref^l_\Pi(B) \notin \{\emptyset, \{B\}\}$, the classes in $Ref^l_\Pi(B)$ satisfy :
(1) *coverness*: $\bigcup Ref^l_\Pi(B) = B$, and
(2) *disjointness*: $\forall B', B'' \in Ref^l_\Pi(B)$, if $B' \neq B''$ then $B' \cap B'' = \emptyset$,
the function *Split* can be redefined as follows:

$$Split(B, \Pi) = \begin{cases} Ref^l_\Pi(B) & if\ \ \exists l \in L.\ Ref^l_\Pi(B) \notin \{\emptyset, \{B\}\} \\ \{B\} & otherwise \end{cases}$$

which does not require using set complementation.

3 Timed systems

3.1 Timed automata

Let $\Omega = \{x_1, ..., x_n\}$ be a finite set of *clocks*. All clocks advance at the same rate. A *valuation* is an n-tuple $v \in \mathbb{R}^n_+$. $v(x_i)$ is the value of clock x_i in v, and

$v+t$, $t \in \mathbb{R}_+$ stands for the valuation v', such that $\forall x \in \Omega.\ v'(x) = v(x) + t$, and $v[\mathcal{X} := 0]$, $\mathcal{X} \subseteq \Omega$ is the valuation v'', such that $v''(x) = 0$ if $x \in \mathcal{X}$, $v''(x) = v(x)$ otherwise. A *clock constraint* ψ is a conjunction of atoms of the form $x \# c$, where $x \in \Omega$, $c \in \mathbb{Z}$, $\# \in \{<, \leq, =, \geq, >\}$.

A *timed automaton* is a quadruple $\langle S, s_0, E, I, \Omega \rangle$. S is a finite set of *control states*, $s_0 \in S$ being the *initial* one. E is a finite set of *arcs*, where an arc $(s, a, s', \psi, \mathcal{X})$ from s to s', is annotated with a label $a \in L$, a clock constraint ψ, and a set of clocks $\mathcal{X} \subseteq \Omega$ to reset. I is a function associating with each control state s an *invariant*. The semantics of a TA is a LTS $G = \langle Q, Q^0, \rightarrow \rangle$, where: $Q = \{\langle s, v \rangle \mid s \in S, v \in I_s\}$; $Q^0 = \{\langle s_0, v \rangle \mid v \in I_{s_0}\}$; and $\rightarrow \subseteq Q \times (E \cup \mathbb{R}_+) \times Q$ is defined by the following rules :

1. (time passage) $$\frac{v, (v+t) \in I_s\ ,\quad t \in \mathbb{R}_+}{\langle s, v \rangle \xrightarrow{t} \langle s, v+t \rangle}$$
2. (action) $$\frac{e = (s, a, s', \psi, \mathcal{X}) \in E,\quad v \in \psi,\quad v' = v[\mathcal{X} := 0]}{\langle s, v \rangle \xrightarrow{e} \langle s', v' \rangle}$$

For $q = \langle s, v \rangle$, $q[\mathcal{X} := 0]$ denotes $\langle s, v[\mathcal{X} := 0] \rangle$, and $q + t$ stands for $\langle s, v + t \rangle$.

3.2 Tai-bisimulation

Given $G = \langle Q, Q^0, \rightarrow \rangle$ we define $G_{tai} = \langle Q, Q^0, \Rightarrow_{tai} \rangle$ by abstracting away the exact amount of time elapsed in a time transition. This is done by replacing all labels $t \in \mathbb{R}^+$ by the label $\varepsilon \notin (E \cup \mathbb{R}^+)$ as follows:

$$\frac{q \xrightarrow{e} q'}{q \overset{e}{\Rightarrow}_{tai} q'} \qquad \frac{q \xrightarrow{t} q'}{q \overset{\varepsilon}{\Rightarrow}_{tai} q'}$$

The tai-bisimulation,[2] denoted $\approx_{tai}$, is the greatest bisimulation defined on G_{tai}, that is, $G \approx_{tai} G'$ iff $G_{tai} \approx G'_{tai}$.

It can be easily shown that $\approx_{ta}$ is coarser than the *region graph* equivalence [AD94] which induces a finite partition. Thus, we can state the following.

Proposition 1. *The partition induced by the tai-bisimulation is finite.*

4 Minimization with respect to the tai-bisimulation

The set of valuations Z satisfying a clock constraint is a simple convex polyhedron, called a *convex zone*. A *(non-convex) zone* is a union of convex zones. The class of zones is closed under complementation and set difference, whereas the class of convex zones is not. We write $\langle s, Z \rangle$, for the class $\{\langle s, v \rangle \mid v \in Z\}$, and say that $\langle s, Z \rangle$ is convex if Z is a convex zone. A partition Π is convex iff all its classes are convex. Finally, we say that Π satisfies the *enabledness* condition iff for each class $\langle s, Z \rangle \in \Pi$ and each arc $e = (s, a, s', \psi, \mathcal{X}) \in E$, it holds : $Z \cap \psi \in \{Z, \emptyset\}$. From now on, we only consider initial partitions respecting convexity and enabledness.

[2] The name comes from *time-abstracting, action-immediate*.

4.1 Refinement

There are two types of preconditions, corresponding to time and action transitions of the timed model. For $e \in E$ we define:

$$pre_e(\langle s, Z\rangle, \langle s', Z'\rangle) \stackrel{\text{def}}{=} \begin{cases} \langle s, Z \cap \psi \cap (Z'[\mathcal{X} := 0])\rangle & \text{if } e = (s, a, s', \psi, \mathcal{X}), \\ \emptyset & \text{otherwise} \end{cases}$$

Proposition 2. *1. $q \in pre_e(B, C)$ iff $q \in B \wedge \exists q' \in C. q \stackrel{\varepsilon}{\Rightarrow}_{tai} q'$.*
2. If B, C are convex, then $pre_e(B, C)$ is also convex.

The time precondition is nonempty only for pairs of classes having the same control-state component, since the latter does not change with time transitions:

$$pre_\varepsilon(\langle s, Z\rangle, \langle s, Z'\rangle) \stackrel{\text{def}}{=}$$
$$\langle s, \{v \in Z \mid \exists t \in \mathbb{R}_+. (v + t) \in Z' \ \wedge \forall 0 < t' < t. (v + t') \in Z \cup Z'\}\rangle$$

Proposition 3. *1. If $q \in pre_\varepsilon(B, C)$, then $q \in B \wedge (\exists q' \in C,\ q \stackrel{\varepsilon}{\Rightarrow}_{tai} q')$.*
2. If B, C are convex, then $pre_\varepsilon(B, C)$ is also convex.

Note that the inverse of case 1 above does not hold, contrary to proposition 2. For example, if $B = \langle s, \{x < 1\}\rangle$, $C = \langle s, \{x > 2\}\rangle$, then $\langle s, x = 0\rangle \stackrel{\varepsilon}{\Rightarrow}_{tai} \langle s, x = 3\rangle$, but $pre_\varepsilon(B, C) = \emptyset$. Indeed, $pre_\varepsilon(B, C)$ is nonempty only if B can lead to C by letting time pass while the system continuously stays in $B \cup C$ during the passage from B to C. Nevertheless, no information is lost regarding time stability in the sense of tai-bisimulation, as the following lemma shows. (See also section 4.2 for more.)

Lemma 4. *Let B, C be two classes of a partition Π such that $q \stackrel{\varepsilon}{\Rightarrow}_{tai} q'$ for some $q \in B$, $q' \in C$. Then, there exist classes $B = D_0, D_1, ..., D_m = C$ in Π such that $q \in pre_\varepsilon(D_0, pre_\varepsilon(D_1, ...pre_\varepsilon(D_{m-1}, D_m)...))$.*

In the previous example, we have $D_0 = B, D_2 = C$, and $D_1 = \langle s, \{1 \leq x \leq 2\}\rangle$.

The definition of $Succs^l_\Pi(B)$ for $l \in E$ is identical to the one given in section 2.1. Care must be taken in the case $l = \varepsilon$, where we remove the (trivial) time successor of every class, that is, the class itself. The definition of $Ref^l_\Pi(B)$, for $l \in E \cup \{\varepsilon\}$, is identical to the one given in section 2.3.

It remains to prove that coverness and disjointness are preserved during the refinement of B. This is true if the partition is *complete*, i.e., $\forall s \in S$, $I_s = true$. In section 4.3 we discuss the alternatives in the case this condition does not hold.

Proposition 5. *Let Π be a complete partition, and $B \in \Pi$. Also let $Ref^l_\Pi(B)$ be the set $\{B_1, ..., B_m\}$. Then, $\forall i \neq j.\ B_i \cap B_j = \emptyset$, and $\bigcup B_i = B$.*

4.2 The minimal model

In this section we make explicit the relation between $G_{\approx_{tai}}$, the quotient graph w.r.t. $\approx_{tai}$, and G_{min}, the actual model computed by the MMGA adapted as above. Although not identical to G_{min}, $G_{\approx_{tai}}$ can be easily computed from the former by a simple saturation of its ε-transitions.

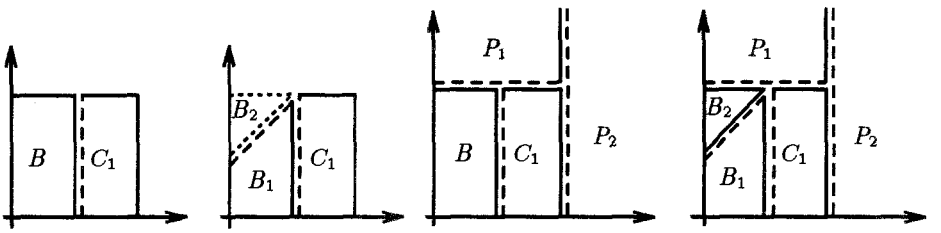
Formally, let $G_{\approx_{tai}} = \langle \Pi_{\approx_{tai}}, \pi_{\approx_{tai}}, \Rightarrow_{tai} \rangle$, and $G_{min} = \langle \Pi, \pi, \Rightarrow \rangle$. Let $\stackrel{\varepsilon}{\Rightarrow}^*$ be the reflexive, transitive closure of $\stackrel{\varepsilon}{\Rightarrow}$.

Proposition 6. *$\Pi = \Pi_{\approx_{tai}}$, $\pi = \pi_{\approx_{tai}}$, and for all $B, C \in \Pi$, (1) $B \stackrel{\varepsilon}{\Rightarrow}_{tai} C$ iff $B \stackrel{\varepsilon}{\Rightarrow} C$, and (2) $B \stackrel{\varepsilon}{\Rightarrow}_{tai} C$ iff $B \stackrel{\varepsilon}{\Rightarrow}^* C$.*

In other words, the partitions of the two graphs are identical, as well as their action transitions, while $\stackrel{\varepsilon}{\Rightarrow}_{tai}$ is the reflexive, transitive closure of $\stackrel{\varepsilon}{\Rightarrow}$.

4.3 Correctness in the presence of strict control-state invariants

If $I_s \subset \mathbb{R}_+$ then the timed model does not contain states $\langle s, v \rangle$ such that $v \in \mathbb{R}_+ \setminus I_s$. In this case coverness is not ensured, as shows the example of figure 1(a), where $B \cup C_1$ is the invariant, $\Pi = \{B, C_1\}$, and $Succs^{\varepsilon}_{\Pi}(B)$ is $\{C_1\}$. Then, $Ref^{\varepsilon}_{\Pi}(B) = \{B_1\}$, which does not cover B. There are several ways to solve this problem:



$Split(B, \{B, C_1\}) = \{B_1\}$
$B_1 = pre_{\varepsilon}(B, C_1)$
$B_2 = B \setminus B_1$

(a)

$Split(B, \{B, C_1, P_1, P_2\}) = \{B_1, B_2\}$
$B_1 = pre_{\varepsilon}(B, C_1)$
$B_2 = pre_{\varepsilon}(B, P_1)$

(b)

Fig. 1. Incomplete refinement (a) ; Adding pseudo-classes to a partial partition (b)

1. A class $\langle s, Z \rangle$ is called a *border* one, if $\exists v \in Z, t \in \mathbb{R}_+$. $(v + t) \in \overline{I_s}$ and $\forall t' \leq t$, $(v + t') \in (Z \cup \overline{I_s})$. In figure 1(a), B is a border class, while B_1 is not. Assume that a border class B is refined w.r.t. $\{C_1, ..., C_m\}$, which yields $\{B_1, ..., B_l\}$ ($l \leq m$, since some B_i may be empty). Let $B' = B \setminus \bigcup B_i$,

which is not convex in general. If $B' \neq \emptyset$, we take an arbitrary (but minimal in number) partition of B' into convex classes $\{B'_1, ..., B'_k\}$, and define $\mathit{Split}(B, \Pi) = \{B_1, ..., B_l, B'_1, ..., B'_k\}$. This solution makes complementation inevitable. What is more, the number of times where complementation will be employed cannot be determined a priori. Indeed, it is always the case that after splitting a border class, at least one of its subclasses is border. The latter may in turn become accessible, be split, and so on.

2. A second solution is to start with a complete initial partition respecting the invariants : $\forall \langle s, Z \rangle \in \Pi_0.\ Z \cap I_s \in \{Z, \emptyset\}$. A class $\langle s, Z \rangle$ is called a *pseudo*-class if $Z \cap I_s = \emptyset$, otherwise it is normal. Pseudo-classes are never split (it suffices to make sure that they are never inserted into the set α of accessible classes). Normal classes can be split w.r.t. pseudo-classes. If all successors of a normal class B are pseudo-classes, then B need not be split. Figure 1(b) shows how the situation of figure 1(a) changes after applying this solution. P_1, P_2 are pseudo-classes, and we now have $\mathit{Succs}^{\varepsilon}_{\Pi}(B) = \{C_1, P_1\}$, and $\mathit{Ref}^{\varepsilon}_{\Pi}(B) = \{B_1, B_2\}$, which covers B. On the other hand, C does not have to be split, since $\mathit{Succs}^{\varepsilon}_{\Pi}(C) = \{P_1, P_2\}$, that is, all its successors are pseudo-classes.

5 Applications

We have implemented the algorithm and applied it to generate the minimal models for a number of case studies. Further, we have used the tool ALDEBARAN to compare the constructed minimal models against labeled transition systems modeling untimed requirements. The main idea consists in considering ε-transitions to be τ-transitions, that is, non-observable or silent ones. Other labels can also be hidden (i.e. replaced by τ) according to the property to be verified. The resulting transition system is then reduced or compared to another model. In particular, we have used the τ^*a-bisimulation equivalence, denoted $\approx_{\tau^*a}$, as well as the τ^*a-simulation preorder [FM91][3].

Due to space limitations, here we illustrate this methodology in detail for only one application, namely the Philips audio control protocol [BPV94]. Experimental results obtained for other well-known examples (e.g. CSMA-CD and FDDI [DOTY95] and Tick-Tock [DOY94] communication protocols) are shown in table 1. The TA column presents the size of the input TA. The M column displays the size of the minimal model, while C_{tot} is the total number of classes created (including classes which were finally found non-accessible). The "splittings" column presents the total number of *Split* operations, the effective time ones (ε subcolumn) and the effective action ones (e subcolumn). N is the number of processes, stations, etc, depending on the protocol. We have used a Sparc 10 with 128 Mbytes of main memory.

[3] Recall that a simulation preorder is a relation satisfying only (1), in the definition of bisimulation given in section 2.

Example	N	TA		M		C_{tot}	splittings			time
		states	arcs	states	trans		total	ϵ	e	(secs)
CSMA-CD	2	9	21	26	52	62	112	18	15	0.4
	3	26	90	340	1,055	559	1,264	150	173	3.8
	4	72	312	3,828	16,066	4,855	13,592	1,070	1,797	90.9
FDDI	3	19	25	525	933	1,873	3,202	377	637	8.5
	4	25	33	1,606	2,859	7,760	10,980	1,341	2,264	57.4
	5	31	41	4,621	8,801	26,900	32,385	3,878	6,755	315
Tick-Tock	1	24	64	78	121	202	223	31	15	1
	2	72	240	585	976	1,663	1,658	243	163	8.7

Table 1. Minimization results of various examples

5.1 Philips audio-control protocol

The protocol deals with the transmission of a bit stream through a wire, using a Manchester encoding. The receiver can only detect low-to-high voltage changes, which imposes that a bit stream either has an odd length or ends with two 0-bits (all streams start by "1"). Also, the protocol permits a small drift in the clock rates of the sender and the receiver. This is modeled in [DY95] using *multirate* TA, a subclass of hybrid automata which can be transformed into TA [OSY94]. Here, we follow directly the TA model obtained after the transformation, using the automata Sender, Receiver, and Stream(the last one models correct bit streams), which are omitted here (see [DY95] for a full description).

model	TA		M		C_{tot}	splittings			time
	states	arcs	states	trans.		total	ϵ	e	(secs)
TA_1	146	351	50	61	815	289	114	36	0.9
TA_1 †			283	402	1,557	1,326	445	151	2.3
TA_2	77	207	51	62	674	300	126	39	1
TA_2 †			62	86	672	312	126	39	1.1

Table 2. Philips protocol : minimization results

The main correctness property we want to prove is that the stream received is identical to the one sent. In fact, this can be done only if we make sure that the sender does not start transmitting (action *IN*) a new stream before the last one has been completely received (action *OUT*), that is, no two consecutive *IN* actions take place without on intermediate *OUT*. In order to ensure this property, we have two options :

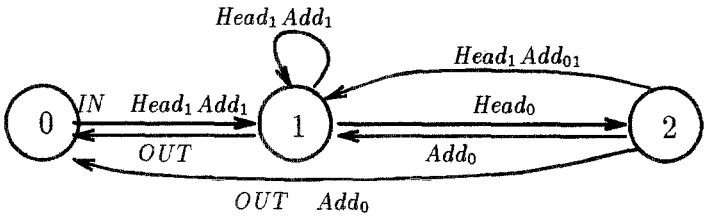
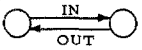


Fig. 2. Good

1. Either to compose the system with the following automaton (called InOut) which prevents the above bad behaviors : 

. Let TA_1 be Sender||Receiver||Stream||InOut.
2. Or to modify Sender by adding a clock which controls the delay between the end of a transmission and the beginning of the next one. This delay should be greater than the time elapsed between the last bit sent by the sender and the action OUT of the receiver. Let TA_2 be Sender'||Receiver||Stream.

For each $TA_i, i = 1, 2$, we obtain two minimal models, one for a correct case (where the maximum drift is $\frac{1}{20}$) and one for an erroneous case (max. drift: $\frac{1}{17}$). Table 2 shows performance results. (The erroneous cases are marked with †.)

Then, we model the correctness requirement by the LTS Good, shown in figure 2 [4]. Let M_i be the minimal model of TA_i for $i = 1, 2$. As expected, in the correct case, we find that $M_i \sqsubseteq_{\tau^* a}$ Good. This does not hold in the erroneous case, and as a diagnostic, we find sequences where the receiver terminates before the sender does.

However, Good $\not\sqsubseteq_{\tau^* a} M_i$, since Good also models bit streams that not satisfy the requirement imposed by Stream. In order to explain this further, consider the LTS depicted in figure 3 obtained by reducing the correct M_1 w.r.t. the $\tau^* a$-bisimulation [5]. This is almost the automaton modeling correct bit streams, except that it contains an additional state 5, grouping all states of the timed model where the sender has sent a "0", but still has bits to transmit. Therefore, the receiver does not have time to perform OUT, since it will first see the next bit transmission taking place. Although trivial, this example shows that often the actual behavior of the system is not exactly the one expected.

[4] $Head_i$ (Add_i) means that bit i is sent (resp. received).
[5] The reduction of the correct M_2 gives exactly the same LTS.

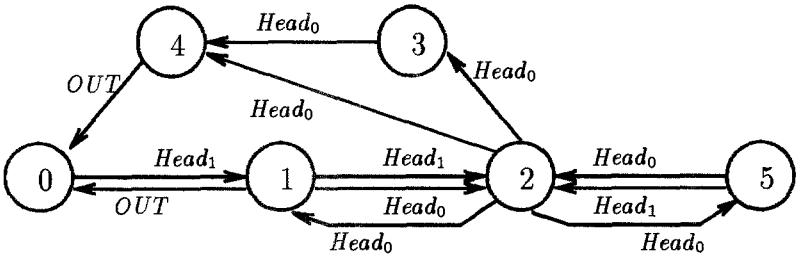


Fig. 3. Minimization with respect to $\approx_{\tau^* a}$

6 Related work

6.1 The ta-bisimulation

In [LY93] another time-abstracting bisimulation has been studied. Given $G = \langle Q, Q^0, \rightarrow \rangle$, we define $G_{ta} = \langle Q, Q^0, \Rightarrow_{ta} \rangle$, as follows:

$$\frac{q \xrightarrow{t} q'' \xrightarrow{e} q'}{q \overset{e}{\Rightarrow}_{ta} q'} \qquad \frac{q \xrightarrow{t} q'}{q \overset{e}{\Rightarrow}_{ta} q'}$$

The ta-bisimulation, denoted $\approx_{ta}$, is the greatest bisimulation defined on G_{ta}, that is, $G \approx_{ta} G'$ iff $G_{ta} \approx G'_{ta}$.

G_{ta} is more abstract than G_{tai}, in the sense that $\Rightarrow_{tai} \subseteq \Rightarrow_{ta}$. Since greater abstractions yield weaker bisimulations [FM91], $\approx_{tai}$ is stronger than $\approx_{ta}$. In fact, we shall prove a stronger property. Let $G_d = \langle \Pi_{\approx_{tai}}, \pi_{\approx_{tai}}, \Rightarrow_d \rangle$, where $B \overset{l}{\Rightarrow}_d C$ iff $\exists D. B \overset{e^*}{\Rightarrow}_{tai} D \overset{l}{\Rightarrow}_{tai} C\}$, and let $\approx_d$ denote the greatest bisimulation on G_d [6].

Proposition 7. *$G \approx_{ta} G'$ iff $G_{tai} \approx_d G'_{tai}$.*

This result, combined with the one of proposition 6, shows how the ta-minimal model can be computed in two steps: first, one computes the model G_{min} using our adapted algorithm, next, G_{min} is further minimized w.r.t. $\approx_d$.

6.2 Other algorithms

In [ACD+92a] the generic minimization algorithms of [BFH+92] and [LY92], referred to as MAI and MAII respectively, have been adapted for timed systems. Table 3 shows the results obtained with our algorithm and compares its running times ($\star$) to the ones reported in [ACD+92a] for two well known examples, namely the Train-Gate Controller (TGC) and the Fischer's Mutual Exclusion protocol (FMX). The authors of [ACD+92a] used a DEC-5100 with 40 Mbytes

[6] This is essentially the *delay-bisimulation*[FM91].

of main memory. It should be mentioned that our algorithm also required much less memory that the others. $\perp$ denotes nontermination due to memory shortage, and "—" is used for cases that do not appear in [ACD+92a].

Let us note that the idea of avoiding set complementation has been suggested in [YL93]. However, the algorithm presented there is an adaptation of [LY92], whereas our algorithm is based on the [BFH+92] one.

Example	N	TA		M		C_{tot}	splittings			time (secs)		
		states	arcs	states	trans		total	ε	e	$\star$	MAI	MAII
TGC		24	69	25	50	125	113	30	17	0.2	6	12
	†			62	138	159	201	36	27	0.5	57	155
FMX	2	24	34	22	26	34	34	2	0	0	1	2
	2†			47	85	63	118	7	13	0	3	6
	3	119	213	77	108	182	133	15	0	0	8	146
	3†			402	1,117	708	1,379	157	172	1.5	893	$\perp$
	4	548	1,164	252	420	872	493	76	0	2.1	496	$\perp$
	4†			4,437	17,902	7,850	16,144	1,931	2,022	40.4	$\perp$	$\perp$
	5	2,402	5,850	807	1,590	3,887	1,785	325	0	16.3	—	—
	5†									$\perp$	—	—

Table 3. TGC and FMX: minimization results and comparison.

7 Conclusions

We have implemented the algorithm on top of the tool KRONOS [DY95] and have performed experiments with different options. As a result, we have found that among the strategies described in section 4.3 concerning the invariant conditions, the pseudo-classes solution gave in general the worst performances. One the other hand, it turned out that giving priority to splitting w.r.t. timed instead of untimed transitions does not make an important difference. Our implementation includes these options, as well as other ones, that allow, for instance, to specify a set of initial states and/or an initial partition. Experimental results obtained on several case studies are presented in table 1. Based on these results, we claim that using a refinement technique which avoids costly complementations leads to considerable gains in efficiency (both in running times and memory usage) that make minimization possible for larger systems.

We have used the tool ALDEBARAN to further reduce the model generated by our algorithm and compare it to a requirement modeled as an untimed transition system. The requirement does not specify quantitative timing constraints, however its verification strongly depends on the timing conditions embedded in the timed automaton which are indeed preserved by the tai-bisimulation. As we

have found out by the examples, the real behavior of a system is often more complex than expected. Discovering unexpected behaviors helps to gain insight of a system, often revealing intrinsic design problems, and at the same time offering diagnostic traces which are valuable for debugging.

It is worth noting that model checking of TCTL formulas on the minimal model is possible, in the manner of [ACD+92b]. We intend to exploit this possibility as part of our future work. We are also currently studying in more depth the combinations of time-abstracting bisimulations with untimed bisimulation and simulation equivalences and preorders.

References

[ACD+92a] R. Alur, C. Courcoubetis, D. Dill, N. Halbwachs, and H. Wong-Toi. An implementation of three algorithms for timing verification based on automata emptiness. In *Proc. IEEE RTSS'92*, 1992.

[ACD+92b] R. Alur, C. Courcoubetis, D. Dill, N. Halbwachs, and H. Wong-Toi. Minimization of timed transition systems. In *Proc. CONCUR 1992*. LNCS 630, 1992.

[AD94] R. Alur and D.L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.

[BFH+92] A. Bouajjani, J.C. Fernandez, N. Halbwachs, P. Raymond, and C. Ratel. Minimal state graph generation. *Science of Computer Programming*, 18:247–269, 1992.

[BPV94] D. Bosscher, I. Polak and F. Vaandrager. Verification of an audio control protocol. In *Proc. FTRTFT'94*, LNCS 863, 1994.

[DOTY95] C. Daws, A. Olivero, S. Tripakis and S. Yovine. The tool KRONOS. Workshop on Hybrid Systems and Autonomous Control, DIMACS, 1995. To appear in LNCS.

[DOY94] C. Daws, A. Olivero and S. Yovine. Verifying ET-LOTOS programs with KRONOS. In *Proc. FORTE'94*, 1994.

[DY95] C. Daws and S. Yovine. Two examples of verification of multirate timed automata with KRONOS. In *Proc. IEEE RTSS'95*, 1995.

[FGM+92] J.Cl. Fernandez, H. Garavel, L. Mounier, A. Rasse, C. Rodriguez, and J. Sifakis. A tool box for the verification of LOTOS programs. In *14th Int. Conf. on Software Engineering*, 1992.

[FM91] J.C. Fernandez and L. Mounier. On the fly verification of behavioural equivalences and preorders. In *Proc. CAV'91*, LNCS 757, 1991.

[LY92] D. Lee and M. Yannakakis. On-line minimization of transition systems. In *Proc. ACM Symposium on Theory of Computing*, 1992.

[LY93] K. G. Larsen and W. Yi. Timed abstracted bisimulation: implicit specification and decidability. In *Proc. MFPS'93*, 1993.

[Mil80] R. Milner. *A Calculus of Communicating Systems*, LNCS 92, 1980.

[OSY94] A. Olivero, J. Sifakis, and S. Yovine. Using abstractions for the verification of linear hybrid systems. In *CAV'94*, LNCS 818, 1994.

[YL93] M. Yannakakis and D. Lee. An efficient algorithm for minimizing real-time transition systems. In *CAV'93*, LNCS 697, 1993.

Verification of an Audio Protocol with Bus Collision Using UPPAAL*

Johan Bengtsson[2] *W.O.David Griffioen*[3,4] *Kåre J. Kristoffersen*[1]
Kim G. Larsen[1] *Fredrik Larsson*[2] *Paul Pettersson*[2] *Wang Yi*[2]

[1] BRICS‡, Aalborg University, Denmark. E-mail: {jelling,kgl}@iesd.auc.dk
[2] Department of Computer Systems, Uppsala University, Sweden.
E-mail: {johanb,fredrikl,paupet,yi}@docs.uu.se
[3] CWI, Amsterdam, The Netherlands. E-mail: griffioe@cwi.nl
[4] Computing Science Institute, University of Nijmegen, The Netherlands.

Abstract. In this paper we apply the tool UPPAAL[1] to an automatic analysis of a version of the Philips Audio Control Protocol with *two* senders and bus collision handling. This case study is significantly larger than the real-time/hybrid systems previously analysed by automatic tools. During the case study the tool UPPAAL was extended with a new feature, *committed locations*, allowing efficient modelling of broadcast communication.

1 Introduction

During the last few years a number of tools for automatic verification of hybrid and real-time systems have emerged [DY95, HHWT95, BLL+95, HRP94]. These tools have by now reached a state, where they are mature enough for application on realistic case–studies; a claim we hope to substantiate in this paper.

We present an application of our tool UPPAAL to an automatic analysis of a version of the Philips Audio Control Protocol with *two* senders and the consequently caused problem of bus collision. The case study is comprehensive compared with previous verification efforts of real-time and hybrid systems, e.g. the node-space is 10^3 times larger than the case with only *one* sender [BPV94, HWT95, DY95, LPY95]. Also, the number of clocks, variables and channels has increased considerably. The bus collision version studied in this paper has previously been verified in [Gri94] without tool support.

UPPAAL is a tool for automatic verification of safety and bounded liveness properties of networks of timed automata and certain hybrid automata. UPPAAL

* This work has been supported by the European Communities (under CONCUR2 and REACT), NUTEK (Swedish Board for Technical Development) TFR (Swedish Technical Research Council) and Netherlands Organization for Scientific Research (NWO) under contract SION 612-316-125.

‡ **B**asic **R**esearch in **C**omputer **S**cience, Centre of the Danish National Research Foundation.

[1] The current version of UPPAAL is available on the World Wide Web via the UPPAAL home page `http://www.docs.uu.se/docs/rtmv/uppaal`.

contains a number of features including a graphical interface and automatic generation of diagnostic traces, and applies a combination of on-the-fly state-space examination together with efficient constraint solving techniques [YPD94, BLL+95].

In modelling the Audio Protocol with bus collision it turned out to be convenient in certain situations to apply broadcast communication. An extension of UPPAAL with so-called *committed locations* allows broadcasts to be modelled as atomic sequences of two-process synchronizations, and yields in addition performance improvements.

The verification of Philips Audio Protocol with Bus Collision was carried out using the extended version of UPPAAL installed on a SGI ONYX machine. As results we have verified the correctness of the protocol for an error tolerance of 5% on the timing, demonstrated that correctness fails if the error tolerance is increased to 6%, and analysed an incorrect version of the protocol which is actually implemented by Philips.

2 The Committed UPPAAL model

The basis of the UPPAAL model for real-time systems is networks of timed automata [AD90] with data variables [YPD94]. However, to meet requirements arising from various case studies, the UPPAAL model has been extended with various new features such as urgent transitions [BLL+95] etc. The present case study indicates that we need to further extend the UPPAAL model with *committed locations* to model behaviours such as atomic broadcasting in real-time systems. Our experiences with UPPAAL show that the notion of committed locations introduced in UPPAAL is not only useful in modelling but also yields significant improvements in performance.

We assume that a real-time system consists of a fixed number of sequential processes communicating with each other via channels. We further assume that each communication synchronizes two processes as in CCS. Broadcasting communication can be implemented in such systems by repeatedly sending the same message to all the receivers. To ensure atomicity of such 'broadcast' sequences, we mark the intermediate locations of the sender as so–called *committed locations* which are to be left immediately.

An Example. To introduce the notion of committed locations in timed automata, consider the scenario shown in Figure 1: A sender S is to broadcast a message m to two receivers R_1 and R_2. As this requires synchronization between *three* processes this can not directly be expressed in UPPAAL where synchronization, as in CCS, is between two processes based on complementarity of actions. However, as an initial attempt we may model the broadcast as a sequence of two two-process synchronizations, where first S synchronizes with R_1 on m_1 and then with R_2 on m_2. However, this is not an accurate modelling as the intended atomicity of the broadcast is not preserved (i.e. other processes may interfere during the 'broadcast' sequence). To ensure atomicity, we mark the intermediate location S_2 of the sender S as a so-called *committed location* (indicated by the

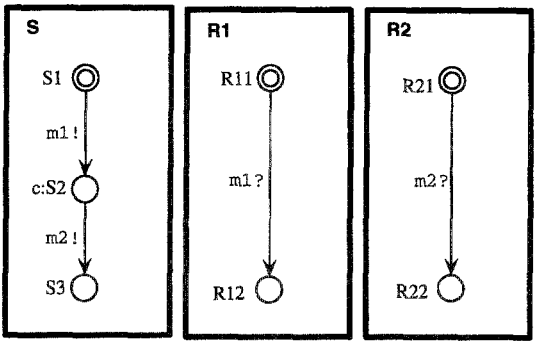


Fig. 1. Broadcasting Communication and Committed Locations.

c:-prefix). The atomicity of the action sequence $m_1!m_2!$ is now achieved by insisting that a committed location must be left immediately! This behaviour is quite similar to what has been called "urgent transitions" [HHWT95, DY95, BLL+95] which insists that the next transition taken must be an action (and not a delay). The precise semantics of committed locations will be formalized in the transition rules for networks of timed automata with data variables in the following.

Preliminaries. We assume a finite set of clock variables C ranged over by x, y, z and a finite set of data variables V ranged over by i, j, k. We use $G(C, V)$ to stand for the set of formulas ranged over by g, generated by the following syntax: $g ::= a \mid g \wedge g$, where a is a constraint of the form: $x \sim n$ or $i \sim n$ for $x \in C$, $i \in V$, $\sim \in \{\leq, \geq, =\}$ and n being a natural number. We shall call elements of $G(C, V)$ *guards*. To manipulate clock and data variables, we use reset-set of the form: $\overline{w} := \overline{e}$ which is a set of assignment-operations in the form $w := e$ where w is a clock or data variable and e is an expression. A reset-set is a *proper* reset-set when the variables are assinged a value at most once, we use R to denote the set of all proper reset-sets. A reset-operation on a clock variable should be in the form $x := n$ where n is a natural number and a reset-operation on an integer variable should be in the form: $i := c * i + c'$ where c, c' are integer constants. We assume that processes synchronize with each other via channels. Let A be a set of channel names with a subset U of urgent channels on which processes should synchronize whenever possible. We use $\mathcal{A} = \{\alpha? | \alpha \in A\} \cup \{\alpha! | \alpha \in A\} \cup \{\tau\}$ to denote the set of actions that processes can perform to synchronize with each other, where τ is a distinct symbol representing internal actions. We use $\text{name}(a)$ to denote the channel name of a, defined by $\text{name}(\alpha?) = \text{name}(\alpha!) = \alpha$.

The UPPAAL Model with Committed Locations. An automaton A over actions $\mathcal{A}$, clock variables C and data variables V is a tuple $\langle N, l_0, E, N_C \rangle$ where N is a finite set of locations (control-locations) with a subset $N_C \subseteq N$ being the set of committed locations, l_0 is the initial location, and $E \subseteq N \times G(C, V) \times \mathcal{A} \times R \times N$ corresponds to the set of edges. To model urgency, we require that the guard of an edge with an urgent action should always be $\mathtt{tt}$, i.e. if $\text{name}(a) \in U$ and $\langle l, g, a, r, l' \rangle \in E$ then $g \equiv \mathtt{tt}$.

In the case, $\langle l, g, a, r, l' \rangle \in E$ we shall write, $l \xrightarrow{g,a,r} l'$ which represents a transition from the location l to the location l' with guard g (also called the enabling condition of the edge), action a to be performed and a set of reset-operations r to update the variables. Also, we shall write $\mathcal{C}(l)$ whenever $l \in N_C$.

To model networks of processes, we introduce a CCS-like parallel composition operator for automata. Assume that $A_1 ... A_n$ are automata. We use $\overline{A}$ to denote their parallel composition. The intuitive meaning of $\overline{A}$ is similar to the CCS parallel composition of $A_1 ... A_n$ with *all* actions being restricted, that is, $\overline{A} = (A_1 | ... | A_n) \backslash \mathcal{A}$. Thus only synchronization between the components A_i is possible. We shall call $\overline{A}$ a network of automata. We simply view $\overline{A}$ as a vector and use A_i to denote its ith component.

Informally, a process modelled by an automaton starts at location l_0 with all its variables initialized to 0. The values of the clocks increase synchronously with time at location l. At any time, the process can change location by following an edge $l \xrightarrow{g,a,r} l'$ provided the current values of the variables satisfy the enabling condition g. With this transition, the variables are updated by r.

A *variable assignment* is a mapping which maps clock variables C to the non-negative reals and data variables V to integers. For a variable assignment v and a delay d, $v \oplus d$ denotes the variable assignment such that $(v \oplus d)(x) = v(x) + d$ for any clock variable x and $(v \oplus d)(i) = v(i)$ for any integer variable i. This definition of $\oplus$ reflects that all clocks operate with the same speed and that data variables are time-insensitive. For a reset-operation r (a set of assignment-operations), we use $r(v)$ to denote the variable assignment v' with $v'(w) = \mathrm{val}(e, v)$ whenever $w := e \in r$ and $v'(w') = v(w')$ otherwise, where $\mathrm{val}(e, v)$ denotes the value of e in v. Given a guard $g \in G(C, V)$ and a variable assignment v, $g(v)$ is a boolean value describing whether g is satisfied by v or not.

A *control vector* $\bar{l}$ of a network $\overline{A}$ is a vector of locations where l_i is a location of A_i. We shall write $\bar{l}[l_i'/l_i]$ to denote the vector where the ith element l_i of $\bar{l}$ is replaced by l_i'.

A *state* of a network $\overline{A}$ is a configuration $\langle \bar{l}, v \rangle$ where $\bar{l}$ is a control vector of $\overline{A}$ and v is a variable assignment. The initial state of $\overline{A}$ is $\langle \bar{l}_0, v_0 \rangle$ where $\bar{l}_0$ is the initial control vector whose elements are the initial locations of A_i's and v_0 is the initial variable assignment that maps all variables to 0.

To model progress properties, we use the following notion of maximal delay:

$$\mathrm{MD}(l, v) = \begin{cases} 0 & \text{if } \mathcal{C}(l) \\ \max\{d \mid l \xrightarrow{g,a,r} l' \text{ and } g(v \oplus d)\} & \text{otherwise} \end{cases}$$

So if l is a committed location, there will be no delay at l. We extend the notion of maximal delay to networks of automata such that synchronization on urgent channels happens immediately:

$$\mathrm{MD}(\bar{l}, v) = \begin{cases} 0 & \text{if } \exists \alpha \in U, i \neq j, l_i, l_j \in \bar{l} : l_i \xrightarrow{\alpha?, r_i} \ \& \ l_j \xrightarrow{\alpha!, r_j} \\ \min\{\mathrm{MD}(l, v) \mid l \in \bar{l}\} & \text{otherwise} \end{cases}$$

The *semantics of a network* of automata $\overline{A}$ is given in terms of a transition

system with the set of states being the set of configurations and the transition relation defined as follows:

- $\langle \bar{l}, v \rangle \rightsquigarrow \langle \bar{l}[l_i'/l_i], r_i(v) \rangle$ if there exist $l_i \in \bar{l}, g_i, r_i$ such that $l_i \xrightarrow{g_i,\tau,r_i} l_i'$, $g_i(v)$, and for all k if $C(l_k)$ then $k = i$.
- $\langle \bar{l}, v \rangle \rightsquigarrow \langle \bar{l}[l_i'/l_i, l_j'/l_j], (r_i \cup r_j)(v) \rangle$ if there exist $l_i, l_j \in \bar{l}, g_i, g_j, \alpha, r_i$ and r_j such that $i \neq j, l_i \xrightarrow{g_i,\alpha!,r_i} l_i'$, $l_j \xrightarrow{g_j,\alpha?,r_j} l_j'$, $g_i(v)$, $g_j(v)$, $r_i \cup r_j \in R$ and for all k if $C(l_k)$ then $k = i$ or $k = j$.
- $\langle \bar{l}, v \rangle \rightsquigarrow \langle \bar{l}, v \oplus d \rangle$ if $d \leq \mathrm{MD}(\bar{l}, v)$

Thus, if a state $\langle \bar{l}, v \rangle$ contains a committed location no delays can take place. Moreover, any component with committed location must participate in the next (action-) transition.

3 The Committed UPPAAL Implementation

In the following, we present the notion of committed locations in terms of the UPPAAL model and its implementation in UPPAAL. In the current version [BLL+95], UPPAAL is able to check for invariance properties, $\forall\Box\beta$, and reachability properties, $\exists\Diamond\beta$, with respect to constraints, β, on the admissible locations of the various components and the values of the clock and data variables.

The model-checking is performed using backwards reachability analysis together with an efficient constraint-solving technique. Also, UPPAAL adopts on-the-fly generation of the state space in order to avoid explicit construction of the product automaton and the immediately caused memory problems.

The model-checking is based on a partitioning of the (otherwise infinite) state-space into finitely many symbolic states of the form $[\bar{l}, U]$, where U is a simple constraint system (i.e. a conjunction of atomic clock and data constraints [2]). The backwards reachability algorithm checks if a symbolic state $[\bar{l}_f, U_f]$ is reachable from the initial state $[\bar{l}_0, U_0]$, where U_0 expresses that all clocks and data variables are initialized to 0.

The algorithm essentially performs a backwards, breadth-first search of the symbolic states. The search is guided and pruned by two buffers: Wait, holding the symbolic states waiting to be explored and Passed holding the symbolic states under exploration and already explored. Initially Passed is empty and Wait holds the single symbolic state $[\bar{l}_f, U_f]$. The algorithm then repeats the following:

1. Pick a state $[\overline{m}, U']$ from the Wait buffer.
2. Check if $\overline{m} = \bar{l}_0$ and $U_0 \subseteq U'$. If this is the case, return the answer *yes*.
3. If $\overline{m} = \overline{n}$ and $U' \subseteq U''$, for some $[\overline{n}, U'']$ in the Passed buffer, drop $[\overline{m}, U']$ and go to step 1. Otherwise save $[\overline{m}, U']$ in the Passed buffer.
4. Find all symbolic states $[\bar{o}, Z]$ that lead to $[\overline{m}, U']$ in one step and store them in the Wait buffer.
5. If the Wait buffer is not empty go to step 1, otherwise return the answer *no*.

[2] Simple constraint systems are also know under the term zone.

We will not treat the algorithm in more detail here, but refer the reader to to [YPD94, BL96].

Despite its on-the-fly examination of the symbolic state space the above algorithm is bound to run into space problems for sufficiently large systems witnessed by an explosion in the size of the Passed buffer, which is used to record the states already visited in order to enable pruning of redundant examinations (in 3) and eventually ensure termination. The key question is how to limit the growth of this buffer? When using committed locations to ensure atomicity of finite transition sequences of one component (as in modelling broadcast) it obviously suffices to save the symbolic state at the beginning of the sequence. Hence, our proposed solution is simply *not* to save symbolic states in the Passed buffer which involves *committed* locations. We therefore modify step 3 of the algorithm in the following way:

3'. a. If *committed*$(\overline{m})$ go directly to step 4.
   b. If $\overline{m} = \overline{n}$ and $U' \subseteq U''$, for some $[\overline{n}, U'']$ in the Passed buffer, drop $[\overline{m}, U']$ and go to step 1.
   c. If neither of the above steps are applicable, save $[\overline{m}, U']$ in the Passed buffer.

4 The Audio Control Protocol with Bus Collision

In this section an informal introduction to the audio protocol with bus collision is given. The audio control protocol is a bus protocol, all messages are received by all components on the bus. If a component receives a message not addressed to it, the message is just ignored. Philips allows up to 10 components.

Messages are transmitted using Manchester encoding. Time is divided into bit-slots of equal length, a bit "1" is transmitted by an up-going edge halfway a bit-slot, a bit "0" by a down-going edge halfway a bit-slot. If the same bit is transmitted twice in a row the voltage changes at the end of the first bit-slot. Note that only a single wire is used to connect the components, no extra clock wire is needed. This is one of the properties that makes it a nice (read cheap) protocol.

The protocol has to cope with some problems: (a) The sender and the receiver must agree on the beginning of the first bit-slot, (b) the length of the message is not known in advance by the receiver, (c) the down-going edges are not detected by the receiver. To resolve these problems the following is required: Messages must start with a bit "1" and messages must end with a down-going edge. This ensures that the voltage on the wire is low between messages. Furthermore the senders must respect a 'radio silence' between the end of a message and the beginning of the next one. This radio silence marks the end of a message and the receiver knows that the next up-going edge is the first edge of a new message. It is (almost) possible to decode a Manchester encoded message by only looking to the up-going messages (problem c) only the last zero bit of a message can not be detected (consider messages "10" and "1"). To resolve this it is required that all messages are of odd length.

It is possible that two or more components start transmitting at the same time. The behavior of the electric circuit is such that the voltage on the wire will be high as long as one of the senders pulls it high. In other words: The wire implements the or-function. This makes it possible for a sender to notice that someone else is also transmitting. If the wire is high while it is transmitting a low, a sender can detect a bus collision. This collision detection happens at certain points in time. Just before each up-going transition, and at one and three quarters of a bit-slot after a down going edge (if it is still transmitting a low). When a sender detects a collision it will stop transmitting and will try to retransmit its message later.

If two messages are transmitted at the same time and one is a prefix of the other, the receiver will not notice the prefix message. To ensure collision detection it is not allowed that a message is a prefix of an other message in transit. In the Philips environment this restriction is met by embedding the source address in each message (and assigning each component a unique source address).

In Figure 2 an example is depicted. Two senders start transmitting at exactly the same time. Because two lines on top of each other is hard to distinguish from one line, they are shifted slightly. The thick sender starts transmitting "11..." and the other "101...". At the end of the first bit-slot the thick sender does a down, to prepare for the next up-going edge. But one quarter after this down it detects a collision and stops transmitting. The thin sender did not notice the other and continues transmitting. Note that the receiver will decode the message of the thin sender correctly.

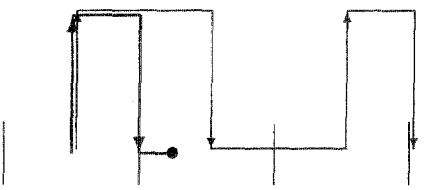


Fig. 2. an example

The protocol has to cope with one more thing: timing uncertainty. Because perfect clocks do not exist in the physical world and because the protocol is implemented on a processor that also has to execute a number of other time critical tasks, a quite large timing uncertainty is allowed. A bit-slot is 888 microseconds, so the ideal time between two edges is 888 or 444 microseconds. On the generation of edges a timing uncertainty of $\pm 5\%$ is allowed. That is: between 844 and 932 for one bit-slot and between 422 and 466 for half a bit-slot. The collision detection just before an up-going edge and the actual generation of this up-going edge must be at most 20 microseconds. The timing uncertainty on the collision detection on one and three quarters after the generation of a down-going edge is ± 22 microseconds. Also the receiver has a timing uncertainty of $\pm 5\%$. And, to complete the timing information, the distance between the end of one

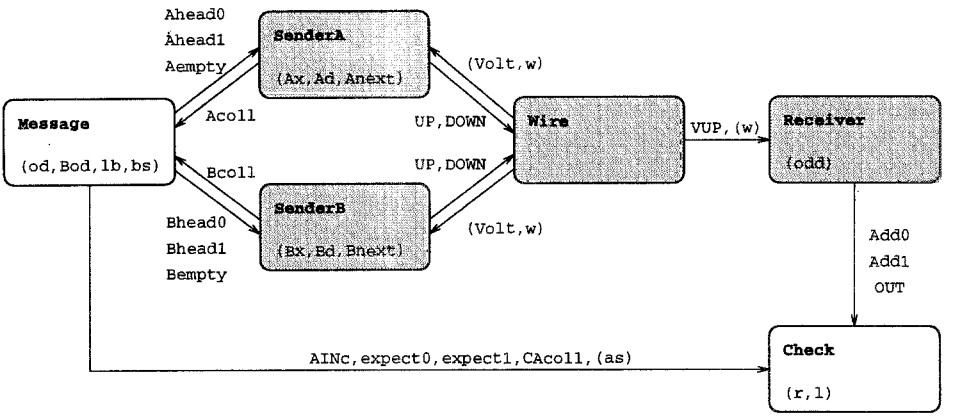


Fig. 3. Philips Audio-Control Protocol with Bus Collision.

message and the beginning of the next must be at least 8000 microseconds (8 milliseconds).

5 A Formal Model of the Protocol

To analyze the behavior of the protocol we model the system as a network of six timed automata. The network consists of two parts: a *core part* and a *testing environment*. The core part models the components of the protocol to be implemented: two senders, a wire and a receiver. The testing environment, consisting of a message generator and an output checker, is used to model assumptions about the environment of the protocol and for testing the behavior of the core part. Figure 3 shows a flow-graph of the network where nodes represent timed automata and edges represent synchronization channels or shared variables (enclosed within parenthesis).

The general idea of the specification is as follows. The automaton Message generates messages for both senders, and also informs the Check automaton on the bits it generated for SenderA. The senders transmit the messages via the wire to the receiver. The receiver communicates the bits it decoded to the checker. Thus the Check automaton is able to compare the bits generated by Message and the bits received by Receiver. If this matches the protocol is correct.

The senders A and B are, modulo renaming (all A's in identifiers to B's), exactly the same. Because of this symmetry, it is enough to check that the messages transmitted by sender A are received correctly. We will proceed with a short description of each automaton. The definition of these uses a number of constants that are declared in Figure 4.

The Senders. SenderA is depicted in Figure 5. It takes input actions Ahead0?, Ahead1? and Aempty?. The output actions UP! and DOWN! will be the Manchester encoding of the message. The clock Ax is used to measure the time between

UP! and DOWN! actions. The idea behind the specification (taken from [DY95]) is that the sender changes location each half of a bit-slot. The locations HS (wire is high in second half of bit-slot) and HF (high in first half of bit-slot) refer to this idea. Extra locations are needed because of the collision detection.

The clock Ad is used to measure the time elapsed between the detection just before UP! action and the corresponding UP! action. Furthermore the time elapsed since the last DOWN! action is measured. The system is in the locations ar_Qfirst and ar_Qlast when the next thing to do is the collision test at one or three quarters of a bit-slot. When Volt is greater than zero, at that moment, the sender detects a collision, stops transmitting and returns to the idle location. The clock w is used to ensure the 'radio silence' between messages. This variable is checked on the transition from idle to ar_first_up.

The Wire. This small automaton keeps track of the voltage on the wire and generates VUP! actions when appropriate, that is when a UP? action is received when the voltage is low.

The Receiver. Receiver (Figure 6) decodes the bit sequence using the up-going (modeled as VUP?) changes of the wire. Decoded bits are signaled to the environment using output actions Add0!, Add1! and OUT! (OUT! is used for signaling the end of a decoded message). The decoding algorithm of the receiver is a direct translation of the algorithm in the Philips documentation of the protocol. In the automaton each VUP? transition is followed by a transition modeling the decoding. This decoding happens 'at once' therefore these intermediate locations are modeled as committed locations. The automaton has two important locations, L1 and L0. When the last received bit is a bit "1" the receiver is in location L1, after receiving a bit "0" it will be in location L0. The error location is entered when a VUP? is received much to early. In the complete specification the error location is not reachable, see Section 6. The receiver keeps track of the parity of the received message using the integer variable odd. When the last received bit is a bit "1" and the message is even, a bit "0" is added to make the complete message of odd length.

The Message Generator. The message generator generates messages of odd length for both sender A and B. Furthermore, the messages generated for sender A, are communicated to the checker. When a collision is detected by sender A this is communicated to the message generator via Acoll?. The message generator will communicate this on his turn to the checker via CAcoll!. Generating messages of odd length is quite simple. The only problem is that it is not allowed that a message for one sender is a prefix of the message for the other sender. To be more precise: If only one sender is transmitting there is no prefix restriction. Only when the two senders start transmitting at the same time, it is not allowed that one sender transmits a prefix of the message transmitted by the other. As mentioned before the reason for this restriction is that the prefix message is not received by the receiver and it is possible that the senders do not notice the collision. In other words: The prefix message can be lost.

The Checker. This automaton keeps track of the bits 'in transit', that is the

bits that are generated by the message generator but not yet decoded by the receiver. Whenever a bit is decoded or the end of the message is detected not conform the generated message the checker enters an error location. Furthermore when sender A detects a collision the checker returns to its initial location.

6 Verification in UPPAAL

In this section we verify correctness of the protocol described in previous sections. Recall, that the system is modelled as a network of the six timed automata: Message SenderA, SenderB, Wire, Receiver and Check, and that properties are specified as logical formulas.

The Correctness Criteria. The correct behaviour of the protocol is ensured whenever the control of the automaton Check is in location a or start. If an incorrect behaviour is detected the Check-automaton enters the error-location, consequently property (1) requires that the Check-automaton is always in location start or a:

$$\forall\Box\ (\text{Check.start} \vee \text{Check.a}) \tag{1}$$

For the property to be satisfied it is required that the bit sequence received by the Receiver matches the bit sequence sent by SenderA. Furthermore, it is also required that the *entire* bit sequence is received by Receiver (and communicated to the Check-automaton). This is ensured since the error-location of the Check-automaton is reachable if the end of a bit sequence is signalled by Receiver (i.e. OUT!) when unmatched bits exists in the Check-automaton.

If the Receiver-automaton observes changes of the wire too early in location L1 or L0 control is changed to location error. It is imaginable that error recovery can be implemented from this location. However, if the other components of the protocol conform to the specification this location should not be reachable, thus property (2) requires that the error-location in Receiver is never reachable.

$$\forall\Box\neg\ \text{Receiver.error} \tag{2}$$

Incorrectness. Unfortunately the protocol described in this paper is not the protocol that Philips has implemented. The original sender checked less often for bus collisions. The 'just before the up going edge' collision detection was only performed before the *first* up. (In our modelling this corresponds to modifying SenderA and SenderB in the following way: delete the outgoing transitions of location ar_Qlast_ok and use the outgoing transitions of location ar_up_ok instead.) This version is incorrect. In general the problem is that if both senders are transmitting and one is slow and the other fast, the distance can cumulate to a high value and this can confuse the receiver. UPPAAL generated a counter example trace.

Although this problem was known by Philips is it interesting to see how powerful the diagnostic traces can be. It enables us not only to find mistakes in the *model* of a protocol, but also to find design mistakes in real life protocols.

The Verification Results. UPPAAL successfully verifies the correctness properties (1) and (2) for an error tolerance of 5% on the timing. Recall that SenderA and SenderB are, modulo renaming, exactly the same, implying that the verified properties for SenderA also applies to the symmetric case for SenderB. Property (1) was verified in 7.5 hrs using 527.4 MB of memory, property (2) in 1.32 hrs using 227.9 MB of memory.

The analysis of the incorrect version of the protocol with less collision detection (discussed above) uses UPPAAL's ability to generate diagnostic traces whenever a certain property is not satisfied by the system. The trace, consisting of 46 transitions, was generated in 13.0 min using 290.4 MB of memory. Also, attempts to verify Property (1) for the full protocol with an error tolerance of 6% on the timing failed. The scenario is similar to the one found by Bosscher et al. in [BPV94] for the one sender protocol.

The properties (above) were verified using the verification algorithm for handling committed locations, described in Section 3, implemented in a new prototype version of UPPAAL, installed on a SGI ONYX.

7 Conclusion

In this paper it is shown to be possible to verify properties of a realistic case study using UPPAAL. The tool is able to verify the correctness properties of the Philips Audio Protocol, that is: the receiver only receives messages that are transmitted. Furthermore the ability of UPPAAL to generate diagnostic traces proved very useful. When writing formal specifications (some) humans tend to make mistakes. These mistakes are much easier to locate using a tool that can generate scenarios. This in contrast with using a tool that only provides Yes/No answers to queries.

We proposed the use of committed locations in UPPAAL specifications. Using these provides a significant efficiency improvement. Furthermore the memory consumption decreases when using committed locations.

Even more important than the efficiency improvement is that committed locations sometimes allow a more natural specification. If a system does a broadcast or multi-way synchronization, this can be modelled much nicer using committed locations. Without committed locations it is not possible in UPPAAL to prohibit other components to perform actions during the broadcast. With committed locations these multi communications can be modelled as a single atomic action.

Another option to model broadcast synchronization is to use another synchronization mechanism than handshake as used in UPPAAL. We prefer the use of committed locations because it is easier to embed in the model and easier to implement. We also think that committed locations and handshake synchronization provide a flexible and expressive model for specifying protocols.

References

[AD90] R. Alur and D. Dill. Automata for Modelling Real-Time Systems. In *Proc. of ICALP'90*, LNCS 443, 1990.

[BL96] Johan Bengtsson and Fredrik Larsson. UPPAAL a Tool for Automatic Verification of Real-time Systems. Master's thesis, Uppsala University, 1996.

[BLL+95] Johan Bengtsson, Kim G. Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. UPPAAL— a Tool Suite for Automatic Verification of Real–Time Systems. In *Proc. of the 4th DIMACS Workshop on Verification and Control of Hybrid Systems*, 1995. To appear in LNCS, 1996.

[BPV94] D.J.B. Bosscher, I. Polak, and F.W. Vaandrager. Verification of an Audio-Control Protocol. In *Proc. of FTRTFT'94*, LNCS 863, pages 170–192, 1994.

[DY95] C. Daws and S. Yovine. Two examples of verification of multirate timed automata with KRONOS. In *Proc. of the 16th IEEE Real-Time Systems Symposium*, pages 66–75, December 1995.

[Gri94] W.O.D. Griffioen. Analysis of an Audio Control Protocol with Bus Collision. Master's thesis, University of Amsterdam, Programming Research Group, 1994.

[HHWT95] Thomas A. Henzinger, Pei-Hsin Ho, and Howard Wong-Toi. HYTECH: The Next Generation. In *Proc. of the 16th IEEE Real-Time Systems Symposium*, pages 56–65, December 1995.

[HRP94] N. Halbwachs, P. Raymond, and Y.-E. Proy. Verification of linear hybrid systems by means of convex approximations. In *Static Analysis Symposium*, LNCS 864, pages 223–237, 1994.

[HWT95] Pei-Hsin Ho and Howard Wong-Toi. Automated Analysis of an Audio Control Protocol. In *Proc. of CAV'95*, LNCS 939, 1995.

[LPY95] Kim G. Larsen, Paul Pettersson, and Wang Yi. Diagnostic Model-Checking for Real-Time Systems. In *Proc. of the 4th DIMACS Workshop on Verification and Control of Hybrid Systems*, 1995. To appear in LNCS, 1996.

[YPD94] Wang Yi, Paul Pettersson, and Mats Daniels. Automatic Verification of Real-Time Communicating Systems By Constraint-Solving. In *Proc. of the 7th International Conference on Formal Description Techniques*, 1994.

The constants used in the formulas		
q	2220	One quarter of a bitslot: 222 micro sec
d	200	Detection 'just before' the UP: 20 micro sec
g	220	'Around' 25% and 75% of the bitslot: 22 micro sec
w	80000	The radio silence: 8 milli sec
t	0.05	The timing uncertainty: 5%

The constants in the automata		
W	w	80000
D	d	200
A1min	q-g	2000
A1max	q+g	2440
A2min	3*q-g	6440
A2max	3*q+g	6880
Q2	2*q	4440
Q2minD	2*q*(1-t)-d	4018
Q2min	2*q*(1-t)	4218

The constants continued		
Q2max	2*q*(1+t)	4662
Q3min	3*q*(1-t)	6327
Q3max	3*q*(1+t)	6993
Q5min	5*q*(1-t)	10545
Q5max	5*q*(1+t)	11655
Q7min	7*q*(1-t)	14763
Q7max	7*q*(1+t)	16317
Q9min	9*q*(1-t)	18981
Q9max	9*q*(1+t)	20979

Fig. 4. Declaration of Constants.

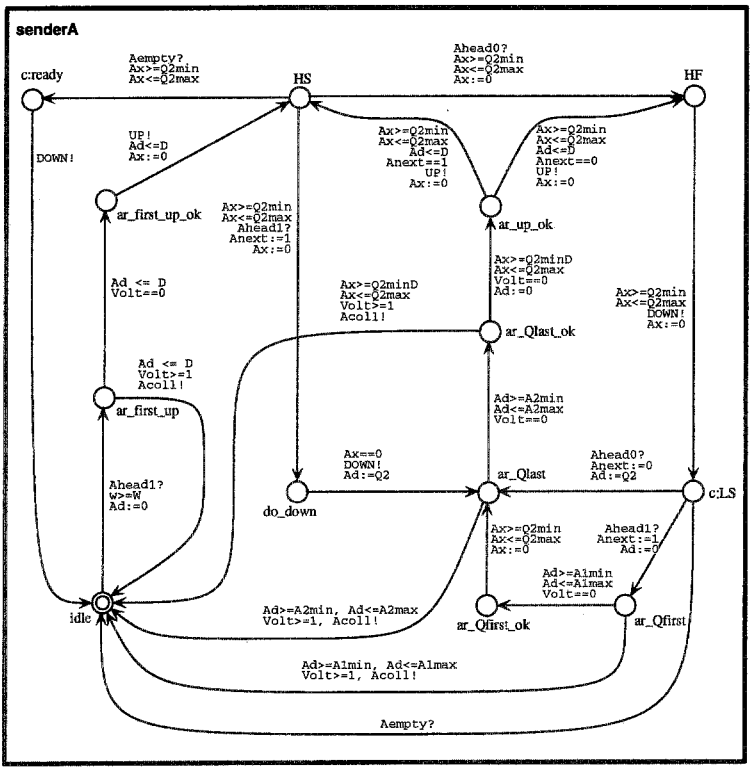


Fig. 5. The SenderA Automaton.

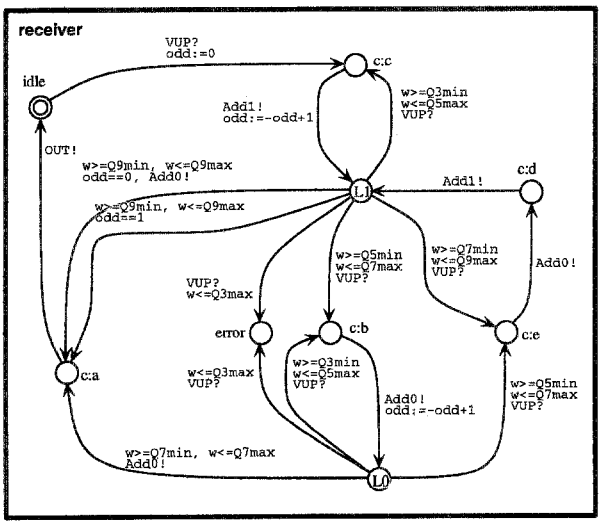


Fig. 6. The Receiver Automaton.

Selective Quantitative Analysis and Interval Model Checking: Verifying Different Facets of a System*

Sérgio Campos[1] and Orna Grumberg[2]

[1] Carnegie Mellon University School of Computer Science, Pittsburgh, PA 15213, USA.
[2] The Technion Department of Computer Science, Haifa 32000, Israel.

Abstract. In this work we propose a verification methodology consisting of *selective quantitative timing analysis* and *interval model checking*. Our methods can aid not only in determining if a system works correctly, but also in understanding how *well* the system works. The selective quantitative algorithms compute minimum and maximum delays over a selected subset of system executions. A linear-time temporal logic (LTL) formula is used to select either infinite paths or finite intervals over which the computation is performed. We show how tableaux for LTL formulas can be used for selecting either paths or intervals and also for model checking formulas interpreted over paths or intervals.
To demonstrate the usefulness of our methods we have verified a complex and realistic distributed real-time system: Our tool has been able to analyze the system and to compute the response time of the various components. Moreover, we have been able to identify inefficiencies that caused the response time to increase significantly (about 50%). After changing the design we not only verified that the response time was lower, but were also able to determine the causes for the poor performance of the original model using interval model checking.

1 Introduction

This work presents a verification methodology that can provide both quantitative and qualitative analysis of systems. The analysis can aid not only in determining system correctness, but also in understanding how *well* the system works. The method consists of *selective quantitative timing analysis* and *interval model checking* and is based on two concepts: *quantitative timing analysis*, and *tableaux* for linear-time temporal logic.

In [8] we have shown how quantitative symbolic algorithms can be used to analyze the behavior of a system. Our method computes minimum and maximum delays between the occurrence of two events, as well as the number of times a specified condition occurs in such an interval. The timing correctness of a system can be evaluated by this method. Reaction time to important events can be computed as well as analyzing how the system behaves during the interval between event and response. In general, performance parameters can be analyzed using this technique.

* This research was sponsored in part by the National Science Foundation under grant no. CCR-8722633, by the Semiconductor Research Corporation under contract 92-DJ-294, and by The Defense Advanced Research Projects Agency, Information Science and Technology Office, under the title "Research on Parallel Computing", ARPA Order No. 7330, issued by DARPA/CMO under Contract MDA972-90-C-0035.

Typically, the quantitative analysis investigates *all* intervals between a set of initial states *start* and a set of final states *final*. In many cases, however, it is desirable to restrict the consideration to only execution paths that satisfy a certain condition. This can help in understanding how the system reacts to different conditions. For example, one common technique for achieving good performance is to optimize a design for the most common cases, while maintaining correctness for the uncommon ones. The designer can optimize response time by restricting system behavior to the most frequent cases. Correctness can then be checked by removing the restrictions.

In this work we use a formula of the linear-time temporal logic LTL to specify a set of paths selected to be verified. Quantitative analysis is then applied only to those paths along which the formula holds. We also extend the technique for cases in which a more precise analysis is needed, by requiring that the selecting formula be true exactly on the investigated interval and not just anywhere on the path.

To strengthen our verification methodology, we combine selective quantitative analysis with model checking. Traditionally, LTL model checking procedures [11, 19, 27] accept a structure that models the system, a set of initial states, and an LTL formula. The procedures determine whether the formula holds on all infinite paths of the structure starting on some initial state. In this work we extend the construction of [11] also for *interval model checking*, that is, checking a formula with respect to finite intervals.

Main Characteristics: Both interval model checking and selective quantitative analysis can be used to extract information related to specific "parts" of a system *without* changing the model. Similar information sometimes can be obtained by restricting the model to disable uninteresting behaviors, or by marking the interesting ones using observer modules. However, these techniques frequently modify system behavior, and consequently properties are checked on a model different than the original one, possibly hiding important errors, or introducing false ones.

Moreover, the fact that properties are verified over finite intervals, allows very different types of properties to be expressed. It is possible to check for "traditional" properties such as safety and liveness, but also to investigate system behavior in more detail. In the real-world not all possible execution sequences are equally interesting. Nor are all possible time intervals within a path. Understanding how the system reacts in different situations allows for a detailed analysis that can aid not only in determining if the system works, but also in understanding how *well* the system works.

Related Methods: There are several other approaches to the verification of timed systems. For example, dense time is modeled by [1, 2, 17, 23, 28]. Those methods model time very accurately. However, the state space of dense time models is infinite, and these tools rely on the construction of a finite quotient structure called region graph. This construction is extremely expensive, limiting the size of problems that can be handled. Dense time models seem to be better suited for smaller problems in which time accuracy must be very high. On the other hand, models such as the one proposed are well suited to model large complex systems in which the accuracy can be easily handled by choosing an appropriate time quantum, as can be seen by the example in this paper.

Discrete time is used by other tools such as [16, 29]. These tools, however, do not allow the quantitative analysis of systems as the proposed method. In [14] quantitative

analysis is implemented, but with a more limited scope. Dense time models allowing restricted quantitative analysis can be found in [17, 28].

Linear-time temporal logics interpreted over both infinite paths and finite intervals have been introduced in [20, 21]. However, they use tableau only for satisfiability and do not handle either quantitative analysis or interval model checking. Interval logics are also used in [25], but in a theorem proving context.

More important when comparing these methods, however, is the fact that these tools do not allow a selective verification of properties as the proposed method. They provide no natural way in which a subset of behaviors can be analyzed in isolation, not allowing as rich an analysis as the proposed method. The closest method to our selection of paths or intervals is the use of fairness constraints in model checking [13, 15, 22]. However, there a fairly restricted types of properties were used for selection, while we can handle any LTL formula. Moreover, only infinite paths can be selected in these works.

A Distributed Real-Time System: To demonstrate the usefulness of our method, we have applied it to a distributed real-time system of realistic complexity, derived from the example described in [26]. Real-time systems are used in many critical applications such as aircraft control or medical monitoring systems. Because of the consequences of failures in such systems, determining their correctness is a vital task.

Several features of this example make it an interesting target for our techniques. It is a system of realistic complexity, its components are existing systems and protocols executing a mixture of multimedia, traditional real-time and non-real time tasks. Also, the distributed nature of the system makes the interaction among its various components much richer. This also makes its analysis more difficult.

Our tools have been able to analyze the system and verify that the deadlines are met by the design. Moreover, we have been able to identify inefficiencies that caused the response time to increase significantly (about 50%). After changing the design we not only verified that the response time was lower, but were also able to determine the causes for the poor performance of the original model using interval model checking.

2 A tableau for LTL

Our specification language is a *linear-time temporal logic* called LTL [24]. The logic is used for two different purposes. One is to specify a property of the system that needs to be verified. The other is to specify a set of selected paths that will be verified. In both cases we use a *tableau* [19, 27, 11] for the formula.

We first give the syntax of LTL. Given a set of atomic propositions AP, LTL is defined inductively as follows. Every atomic proposition is an LTL formula. If f and g are LTL formulas then $\neg f$, $f \vee g$, $\mathbf{X} f$ and $f \mathbf{U} g$ are also LTL formulas.

The semantics of LTL is defined with respect to a labeled state transition graph called *Kripke Structure*. A Kripke structure $M = (S, R, L)$ has a finite set of states S, a transition relation $R \subseteq S \times S$, and a labeling function $L : S \to \mathcal{P}(AP)$, associating with each state the set of atomic propositions true in that state.

An infinite sequence $s_0, s_1, \ldots$ of states in S is a *path* in the structure M from a state s iff $s = s_0$ and for every $j \geq 0$, $(s_j, s_{j+1}) \in R$. A finite sequence $[s_0, \ldots, s_n]$

is an *interval* in a structure M from a state s iff $s = s_0$ and for every $0 \leq j < n$, $(s_j, s_{j+1}) \in R$. An interval may be a prefix of either a finite or an infinite path. Thus, s_n may or may not have successors in M. The size of interval $\sigma = [s_0, \ldots, s_n]$, denoted $|\sigma|$, is n. σ^j is defined iff $0 \leq j \leq n$ and it denotes the suffix of σ, starting at s_j.

For a formula f, a path π, and an interval σ, $M, \pi \models_{path} f$ means that f holds along path π in the Kripke structure M. $M, \sigma \models_{int} f$ means that f holds along interval σ in M. Given a set of initial states S_0, we say that $M, S_0 \models_{path} f$ iff for every path π from every state in S_0, $M, \pi \models_{path} f$. Given two sets of states $start$ and $final$, we say that $M, [start, final] \models_{int} f$ iff for every interval σ from some state in $start$ to some state in $final$, $M, \sigma \models_{int} f$. In this work whenever we refer to a path (an interval) that satisfies a formula, satisfaction is with respect to $\models_{path}$ ($\models_{int}$). The relation $\models_{path}$ is the standard satisfaction relation for LTL (see, for example, [11]). The relation $\models_{int}$ is identical to $\models_{path}$ for atomic propositions and boolean connectives. For temporal operators it is defined by (M is omitted if clear from the context):

4. $\sigma \models_{int} \mathbf{X}\, g_1 \quad \Leftrightarrow \quad |\sigma| > 0$ and $\sigma^1 \models_{int} g_1$.
5. $\sigma \models_{int} g_1 \,\mathbf{U}\, g_2 \Leftrightarrow \quad \exists k[0 \leq k \leq n \wedge \sigma^k \models_{int} g_2 \wedge \forall j[0 \leq j < k \rightarrow \sigma^j \models_{int} g_1]]$.

When writing LTL formulas, we use the abbreviations $\mathbf{F}\, f = true\ \mathbf{U}\ f$ and $\mathbf{G}\, f = \neg\, \mathbf{F} \neg f$. Note that, in the definition of $[s_0, \ldots s_n] \models f$ we do not consider successors of s_n (whether exist or not). This definition is meant to capture the notion of an interval satisfying a formula independently of its suffix (satisfaction is always defined independently of the prefix).

Let f be an LTL formula. We construct a Kripke structure $T(f)$, called the *tableau* for f, that contains all paths and intervals satisfying f. To identify paths in the tableau that satisfy f we will use *fairness constraints*. A *fairness constraint* for a structure M can be an arbitrary set of states in M, usually described by a formula of the logic. A path in M is said to be *fair* with respect to a set of fairness constraints if each constraint holds *infinitely often* along the path.

Let AP_f be the set of atomic propositions in f. $T(f) = (S_T, R_T, L_T)$ has AP_f as its set of atomic propositions. The set of tableau states is $S_T = \mathcal{P}(el(f))$, where $el(f)$ is the set of *elementary formulas* defined by:

- $el(p) = \{p\}$ if $p \in AP_f$
- $el(\neg g) = el(g)$
- $el(\mathbf{X}\, g) = \{\mathbf{X}\, g\} \cup el(g)$
- $el(g \vee h) = el(g) \cup el(h)$
- $el(g\ \mathbf{U}\ h) = \{\mathbf{X}(g\ \mathbf{U}\ h)\} \cup el(g) \cup el(h)$

Let $sat(f)$ be the set of states in the tableau that should satisfy f. It is defined by:

- $sat(g) = \{s \mid g \in s\}, g \in el(f)$
- $sat(\neg g) = \{s \mid s \notin sat(g)\}$
- $sat(g \vee h) = sat(g) \cup sat(h)$
- $sat(g\ \mathbf{U}\ h) = sat(h) \cup (sat(g) \cap sat(\mathbf{X}(g\ \mathbf{U}\ h)))$

The transition relation R_T is $R_T(s, s') = \bigwedge_{\mathbf{X} g \in el(f)} (s \in sat(\mathbf{X}\, g) \Leftrightarrow s' \in sat(g))$. Finally, the labeling function, $L_T(s) = s \cap AP_f$ and the set of fairness constraints for f, $Fair(f) = \{sat(\neg(g\ \mathbf{U}\ h) \vee h) \mid g\ \mathbf{U}\ h$ occurs in $f\}$.

The constructed tableau $T(f)$ includes every path and every interval which satisfies f. The following theorem characterizes those paths and intervals.

Theorem 2.1 *For every path π in $T(f)$, if π starts from a state $s \in sat(f)$ and π is fair for $Fair(f)$ then $T(f), \pi \models_{path} f$. Moreover, For every interval $\sigma = [t_0, \ldots, t_n]$ in $T(f)$, if $t_0 \in sat(f)$ and $t_n \in \mathcal{P}(AP_f)$ then $T(f), \sigma \models_{int} f$.*

In the algorithms presented later we will use the *product* $P = (S_P, R_P, L_P)$ of $T(f) = (S_T, R_T, L_T)$ with the verified structure $M = (S_M, R_M, L_M)$. We restrict the atomic propositions of f, AP_f to be a subset of AP:

- $S_P = \{(s,t) \mid s \in S_M, t \in S_T$ and $L_M(s) \cap AP_f = L_T(t)\}$.
- $R_P((s,t),(s',t'))$ iff $R_M(s,s')$ and $R_T(t,t')$.
- $L_P((s,t)) = L_T(s)$.

3 CTL Model Checking

CTL [4, 12] is a *branching-time temporal logic* that is similar to LTL except that each temporal operator is preceded by a *path quantifier* – either **E** standing for "there exists a path" or **A** standing for "for all paths". CTL is interpreted over a state in a Kripke structure. The path quantifiers are interpreted over the *infinite paths* starting at that state.

CTL *model checking* is the problem of finding the set of states in a Kripke structure where a given CTL formula is true. One approach for solving this problem is *symbolic* model checking using a representation called *binary decision diagram* (BDD) [5] for the transition relation of the structure. This representation is often very concise. We use the SMV model checking system [22] that takes a CTL formula f, and the BDD that represents the transition relation. SMV computes exactly those states of the system that satisfy the formula f. SMV can also handle model checking of a CTL formula with respect to a structure with fairness constraints. The path quantifiers in the CTL formula are then restricted to fair paths. The CTL model checking under given fairness constraints can also be performed using BDDs.

4 Quantitative Timing Analysis

Several methods have been proposed to verify timed systems, as has been discussed in the introduction. Typically, verifiers assume that timing constraints are given explicitly in some notation like temporal logic and determine if the system satisfies the constraint. In [8] we have described how to verify timing properties using algorithms that explicitly compute timing information as opposed to simply checking a formula. This section briefly describes that approach, which is later used in this work.

A Kripke structure is the model of the system in our method. Currently the system is specified in the SMV language [22]. The structure is represented symbolically using BDDs. It is then traversed using algorithms based on symbolic model checking techniques [6]. All computations are performed on states reachable from a predefined set of initial states. We also assume that the transition relation is total.

We consider first the algorithm that computes the minimum delay between two given events (figure 1). Let *start* and *final* be two nonempty sets of states, often given as formulas in the logic. The *minimum* algorithm returns the length of (i.e. number of edges

in) a shortest interval from a state in *start* to a state in *final*. If no such interval exists, the algorithm returns infinity. The function $T(S)$ gives the set of states that are successors of some state in S. The function T, the state sets I and I', and the operations of intersection and union can all be easily implemented using BDDs [6, 22]. The *minimum* algorithm is relatively straightforward. Intuitively, the loop in the algorithm computes the set of states that are reachable from *start*. If at any point, we encounter a state satisfying *final*, we return the number of steps taken to reach that state.

```
proc minimum (start, final)
i = 0;
R =start;
R' = T(R) ∪ R;
while (R' ≠ R ∧ R ∩ final = ∅) do
      i = i + 1;
      R = R';
      R' = T(R') ∪ R';
if (R ∩ final ≠ ∅)
      then return i;
      else return ∞;
```

```
proc maximum (start, final, not_final)
if (start ∩ (final ∪ not_final) = ∅)
      then return ∞;
i = 0;
R =TRUE;
R' =not_final;
while (R' ≠ R ∧ R' ∩ start ≠ ∅) do
      i = i + 1;
      R = R';
      R' = T^{-1}(R') ∩ not_final;
if (R = R')
      then return ∞;
      else return i;
```

Fig. 1. Minimum and Maximum Delay Algorithms

The second algorithm returns the length of a longest interval from a state in *start* to a state in *final*. If there exists an infinite path beginning in a state in *start* that never reaches a state in *final*, the algorithm returns infinity. The function $T^{-1}(S)$ gives the set of states that are predecessors of some state in S. *not_final* represents the states that do not satisfy *final* (except in the interval selective case, see below).

The initial conditional is only used when computing properties over intervals. As will be seen later, in this case *not_final* correspond to states not in *final*, but which eventually lead to *final*. Therefore, if no starting state is in *final*, or leads to *final*, the algorithm returns infinity.

Informally, the algorithm computes at stage i the set R' of all states at the beginning of an interval of size i, all contained in *not_final*. The algorithm stops in one of two cases. Either R' does not contain states from *start* at stage i. Since it contained states from *start* at stage $i-1$, the size of the longest interval in *not_final* from a state in *start* is $i-1$. Since the transition relation is total, this interval has a continuation to a state outside *not_final*, i.e. to a state in *final*. Thus, there is an interval of length i from *start* to *final* and the algorithm returns i. In the other case, a fixpoint is reached meaning that there is an infinite path within *not_final* from a state in *start*. The algorithm in this case returns infinity. Both minimum and maximum algorithms are proven correct in [8].

5 The Proposed Method

5.1 Selective Quantitative Analysis — Over Paths

Given two sets of states *start* and *final* in M and an LTL formula f, we compute the lengths of a shortest interval and a longest interval from a state in *start* to a state in

$final$ along paths from $start$ that satisfy f. The formula f is interpreted over infinite paths and is used to select the paths over which the computation is performed. The *minimum* and *maximum* algorithms with path selection are:

1. Construct the tableau for f, $T(f)$.
2. Construct the product P of $T(f)$ and M.
3. Use the model checking system SMV on P to identify the set of states *fair* in P, where a state $(s,t) \in S_P$ ($s \in M$, $t \in T(f)$) is in *fair* iff t is the beginning of a path which is fair with respect to *Fair(f)*.
4. Construct P', the restriction of P to the state set *fair*. $P' = (S'_P, R'_P, L'_P)$ is defined by: $S'_P = fair$, $R'_P = R_P \cap (S'_P \times S'_P)$ and for every $s \in fair$, $L'_P(s) = L_P(s)$.
5. Apply $minimum(st, fn)$ and $maximum(st, fn, not\_fn)$ to P', with $st = (start \times sat(f)) \cap fair$, $fn = (final \times S_T) \cap fair$, and $not\_fn = fair - fn$.

The algorithms work correctly because P contains all paths of M that are also paths of $T(f)$ (proof in the full paper). P' is restricted to the fair paths of $T(f)$. Thus, every path in P' from $(start \times sat(f)) \cap S'_P$ satisfies f. Consequently, applying the algorithms to P' from $(start \times sat(f)) \cap S'_P$ to $(final \times S_T) \cap S'_P$ over states in $fair$ gives the desired results.

As mentioned before, in order to work correctly, the algorithm *maximum* must work on a structure with a total transition relation. The transition relation of P is not necessarily total. However, the transition relation of P' is total since every state in $fair$ is the beginning of some infinite (fair) path.

We have applied the method in the analysis of the PCI Local Bus [10], where it has been used to limit the number of transaction aborts being considered.

5.2 Selective Quantitative Analysis — Over Intervals

Given two sets of states $start$ and $final$ and an LTL formula f, we compute the lengths of a shortest and a longest intervals from a state in $start$ to a state in $final$ such that f holds along the interval. Here the formula f is interpreted over intervals and we consider only the intervals between $start$ and $final$ that satisfy f. We will use a special formula $prop$ to identify the set of tableau states that contain only atomic propositions.

$$prop = \{s \in S_T | s \in \mathcal{P}(AP_f)\}.$$

The formula $prop$ is a set of states in $T(f)$. We extend $prop$ to $prop_p$, which is the corresponding set of states in P. The formula $final_p$ is the similar extension of $final$:
- $prop_p = \{(s,t) \in S_P \mid s \in S_M, t \in prop\}$
- $final_p = \{(s,t) \in S_P \mid s \in final, t \in S_T\}$

We will also use a CTL formula $\mathcal{C}$ to identify the set of states over which the *maximum* algorithm is computed.

$$\mathcal{C} = \neg final_p \wedge \mathbf{E}[\neg final_p \, \mathbf{U} \, (prop_p \wedge \mathbf{EF} \, final_p)].$$

States in $\mathcal{C}$ lead to states that are endpoints of intervals satisfying f (states in $prop_p$, see theorem 2.1), and then lead to states in $final_p$. The requirement that $final_p$ does not hold until $prop_p$ is needed because an interval ending in $final_p$ without going through $prop_p$ does not satisfy f.

The *minimum* and *maximum* algorithms with interval selection are:

1. Construct the tableau for f, $T(f)$.
2. Construct the product P of $T(f)$ and M.
3. Use the model checking system SMV on P to identify the set of states that satisfy the CTL formula $\mathcal{C}$.
4. Let $st = (start \times sat(f)) \cap S_P$ and let $fn = (final \times prop) \cap S_P$. The algorithms $minimum(st, fn)$ and $maximum(st, fn, \mathcal{C})$ when applied to P will return the length of the shortest and longest intervals, respectively, between $start$ and $final$ that satisfy f.

The correctness of the algorithm relies on the fact that P contains all intervals that are both in $T(f)$ and M. Moreover, intervals of $T(f)$ from $sat(f)$ to $prop$ satisfy f. Thus, the algorithms compute shortest and longest lengths over intervals from $start$ to $final$ that satisfy f. The proof can be found in the full paper.

When the *maximum* algorithm is computed over the set *not_final* of states not in *final*, it is necessary to require that the transition relation of the structure is total in order to guarantee that the computed intervals terminate at a state in *final*. Here the *maximum* algorithm is computed over the set of states satisfying the formula $\mathcal{C}$. This guarantees that the computed intervals terminate at *final* without the need to require that the transition relation is total.

5.3 Interval Model Checking

Given a structure M and two set of states $start$ and $final$, we say that an interval $\sigma = [s_0, \ldots, s_n]$ from a state in $start$ to a state in $final$ is *pure* iff for all $0 < i < n$, s_i is neither in $start$ nor in $final$.

Given a structure M, two sets of states $start$ and $final$, and an LTL formula f, the *interval model checking* is the problem of checking whether the formula f, interpreted over intervals, is true of all pure intervals between $start$ and $final$ in M.

Interval model checking is useful in verifying *periodic* behavior of a system. A typical example is a behavior occurs in a transaction on a bus. If we want to verify that a certain sequence of events, described by an LTL formula f, occurs in a transaction we can define $start$ to be the event that starts the transaction and $final$ to be the event that terminates the transaction. Interval model checking will verify that f holds on all intervals between $start$ and $final$.

Let M, $start$, $final$, and f be as above. The algorithm given below determines the interval model checking problem using the algorithm *minimum* of figure 1.

1. Construct the tableau for $\neg f$, $T(\neg f)$.
2. Compute the product P of $T(\neg f)$ and M.
3. Apply the algorithm $minimum(st, fn)$ to P with $st = (start \times sat(\neg f)) \cap S_P$ and $fn = (final \times prop) \cap S_P$.
4. If the *minimum* is ∞ then there is no pure interval from $start$ to $final$ that satisfies $\neg f$. Thus, every such interval satisfies f. If *minimum* returns some value k, then the interval found by *minimum* can serve as a counterexample to the checked property.

6 A Distributed Real-Time System

In this section we analyze a distributed real-time system using the techniques presented in this paper. This is a complex and realistic application, its components are existing systems and protocols that are actually used in many real situations. The example consists of three main components, a FDDI network, a multiprocessor connected to this network and one of the processors in the multiprocessor, the control processor [26].

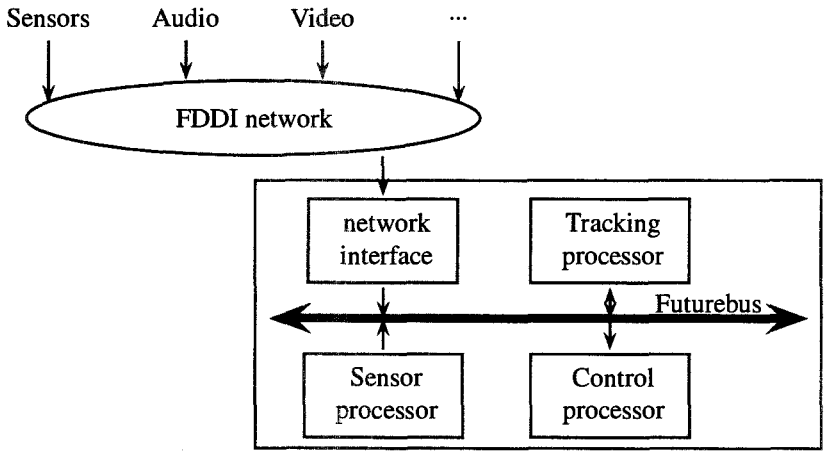


Fig. 2. System Architecture

The FDDI network is a 100Mb/s local/metropolitan area network that uses a token ring topology [3]. There are several stations connected to the network in the system. They generate multimedia and sensor data sent to the control processor, as well as additional traffic inside the network. The traffic in the network has been modeled as proposed in [26]. At every 16 time units the stations utilize the network as follows: *video* station, 6 units; *audio* station, 1 unit; and remainder network traffic, 8 units.

In the multiprocessor, four active processors are connected through a Futurebus+ [18]. The first is the *network interface*, it receives data from the network and sends it to the *control* processor. The network interface uses the bus for 7ms at each time. A *sensor* processor reads data from sensors every 40ms. It buffers this data and sends it once every four readings to the tracking processor. The *tracking* processor processes this data and sends it to the control processor. Both sensor and tracking data use the bus for 3ms each. The deadline for sensor data to be processed is 785ms. Access to the bus is granted using priority scheduling. Priorities are assigned according to the rule: processors with shorter periods have higher priority.

In the control processor there are several periodic tasks. The timing requirements for these tasks can be seen in figure 3. Priority scheduling is also used in the control processor, using the same priority assignment rule. Two tasks in the control processor have special functions, τ_3 processes sensor data, and τ_5 processes multimedia data.

Each of the components of the system (FDDI, network and control processor) has been implemented separately. No data is actually exchanged between the components in the model. Data has been abstracted out of the model, because data dependencies would

Process	τ_1	τ_2	τ_3	τ_4	τ_5
Period	100	150	160	300	100
Exec. time	5	78	30	10	3

Fig. 3. Timing requirements for tasks in the control processor (times in ms)

significantly increase the size of the model and the complexity of verification. However, while simplifying verification, abstractions can also introduce invalid execution sequences. The constraints imposed by data dependencies significantly reduce the number of reachable execution sequences. In an abstract model such dependencies do not exist. We have used selective quantitative analysis to ensure that only execution sequences that are valid (and all such sequences) have been considered during verification.

Using this model we have checked the deadline between a sensor reading in the sensor processor and the processing of this data by τ_3 in the control processor. This deadline is 785ms. Ideally, we would like to compute these time bounds using `MIN{MAX}[sensor_observation, t3.finish]`. However, since in our model there is no synchronization between tasks, this would consider intervals in which `t3` finishes executing just after `sensor`, without going through `track`. To identify the valid intervals in the model, we must consider only intervals that satisfy the constraint:

```
F(sensor.finish & F(track.start & F(track.finish & F t3.start)))
```

This formula guarantees that the correct ordering of events is maintained during verification. We have computed the time between sensor observation and τ_3 processing to be in the interval [197, 563], well within the deadline. However, by looking into the design we noticed a potential source for inefficiencies in the Futurebus. Using standard model checking techniques we then printed a counterexample for the longest response time. It confirmed our speculations.

In this system both sensor and tracking processors access the bus periodically, sending data every 160ms. In the counterexample, however, data required two periods of 160ms to reach the control processor. It was sent by the sensor processor to the tracking processor, but this processor would only send it to the control processor in the next period. Further investigation of the model showed that this was caused by the priority order in which processors accessed the bus. The tracking processor had a higher priority than the sensor processor. This means that when the sensor processor sends data to the tracking processor, it had already used the bus for this period, and would only request access again in 160ms. We modified the design by changing the priorities, and the response time became [37, 403], an improvement of almost 50%.

We have been also able to compare the performance of both designs using interval model checking. We have analyzed the behavior of the system between the time the sensor produces data until the time the tracking processor processes it. Bus utilization is inefficient in this interval if the bus is idle or a lower priority process is executing.

Using interval model checking we have been able to check the LTL formula `G !(bus_idle | bus_granted = lower_priority)` on the intervals between sensor finishing sending data and tracking sending its data to the control processor. The original design showed the existence of priority inversion, as expected. In the modified design, on the other hand, the formula above is true in all intervals under consideration. Notice that the formula is clearly false outside these intervals. This shows that the modified design is optimal with respect to the prioritized utilization of the bus.

The modified design has a better response time, and is clearly preferred in this application. But in other applications this might not be true. There might be cases, for example, in which the tracking processor sends data to the sensor processor. In those cases the modified design is worse than the original one. This again shows how selective quantitative analysis and interval model checking can be used to analyze the different facets of a system. The designer can choose to optimize the behavior of a critical application, even if at the expense of a less critical one.

7 Conclusion

In this paper we have described a method that can produce both quantitative and qualitative information about the behavior of a system. Quantitative analysis and model checking can be performed on state-transition graphs representing the system to be verified. Moreover, the user can specify a subset of execution sequences satisfying a given property using an LTL formula, and verification is performed only on those paths (or intervals) that satisfy that property. The results produced can be used not only to determine the correctness of the system, but also to analyze (and optimize) its performance.

We have used this method to analyze a real-time distributed system. This example shows how the proposed method can assist in understanding the behavior of complex systems. We have been able not only to check properties of the whole system, but also to analyze specific execution sequences of interest. This allowed us to uncover subtleties about the application that might have been very difficult to discover otherwise. We believe that this method can be of great use in analyzing and understanding other complex systems, as it has been in analyzing this one.

Acknowledgment

The authors would like to thank Edmund Clarke for the original idea of combining LTL model checking and quantitative analysis.

References

1. R. Alur, C. Courcourbetis, and D. Dill. Model-checking for real-time systems. In *Symposium on Logic in Computer Science*, pages 414–425, 1990.
2. R. Alur and D. Dill. Automata for modeling real-time systems. In *Lecture Notes in Computer Science, 17th ICALP*. Springer-Verlag, 1990.
3. ANSI Std. *FDDI Token Ring Media Access Control*, s3t95/83-16 edition, 1986.
4. M. Ben-Ari, Z. Manna, and A. Pnueli. The temporal logic of branching time. *Acta Informatica*, 20:207–226, 1983.
5. R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8), 1986.
6. J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *Symposium on Logic in Computer Science*, 1990.
7. S. Campos, E. Clarke, W. Marrero, and M. Minea. Verus: a tool for quantitative analysis of finite-state real-time systems. In *ACM Workshop on Languages Compilers and Tools for Real-Time Systems*, 1995.

8. S. Campos, E. Clarke, W. Marrero, M. Minea, and H. Hiraishi. Computing quantitative characteristics of finite-state real-time systems. In *Real-Time Systems Symposium*, 1994.
9. S. V. Campos, E. M. Clarke, W. Marrero, and M. Minea. Timing analysis of industrial real-time systems. In *Workshop on Industrial-strength Formal specification Techniques*, 1995.
10. S. Campos, E. Clarke, W. Marrero, and M. Minea. Verifying the performance of the pci local bus using symbolic techniques. In *International Conference on Computer Design*, 1995.
11. E. Clarke, O. Grumberg, and H. Hamaguchi. Another look at ltl model checking. In D. Dill, editor, *proceedings of the Sixth Conference on Computer-Aided Verification*, Lecture Notes in Computer Science 818, pages 415–427. Springer-Verlag, 1994.
12. E. M. Clarke and E. A. Emerson. Synthesis of synchronization skeletons for branching time temporal logic. In *Logic of Programs: Workshop, Yorktown Heights, NY, May 1981*. Springer-Verlag, 1981. *Lecture Notes in Computer Science*, volume 131.
13. E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, 1986.
14. P. C. Clements, C. L. Heitmeyer, B. G. Labaw, and A. T. Rose. MT: a toolset for specifying and analyzing real-time systems. In *IEEE Real-Time Systems Symposium*, 1993.
15. E.A. Emerson and Chin Laung Lei. Modalities for model checking: Branching time strikes back. *Twelfth Symposium on Principles of Programming Languages*, January 1985.
16. A. N. Fredette and R. Cleaveland. RTSL: a language for real-time schedulability analysis. In *IEEE Real-Time Systems Symposium*, 1993.
17. T. A. Henzinger, P. H. Ho, and H. Wong-Toi. HyTech: the next generation. In *IEEE Real-Time Systems Symposium*, 1995.
18. IEEE Standard Board and American National Standards Institute. *Standard Backplane Bus Specification for Multiprocessor Architectures: Futurebus+*, ansi/ieee std 896.1 edition, 1990.
19. O. Lichtenstein and A. Pnueli. Checking that finite state concurrent programs satisfy their linear specification. In *Proc. 12^{th} Conference on Principle of Programming languages*, 1985.
20. O. Lichtenstein, A. Pnueli, and L. Zuck. The glory of the past. In *Proc. Conf. Logics of Programs*, Lecture Notes in Computer Science 193, pages 196–218. Springer-Verlag, 1985.
21. Z. Manna and A. Pnueli. The anchored version of the temporal framework. In *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency*, Lecture Notes in Computer Science 354, pages 201–284. Springer-Verlag, 1989.
22. K. L. McMillan. *Symbolic model checking — an approach to the state explosion problem*. PhD thesis, SCS, Carnegie Mellon University, 1992.
23. X. Nicollin, J. Sifakis, and S. Yovine. From atp to timed graphs and hybrid systems. In *Lecture Notes in Computer Science, Real-Time: Theory in Practice*. Springer-Verlag, 1992.
24. A. Pnueli. The temporal semantics of concurrent programs. In *Proceedings of the eighteenth conference on Foundation of Computer Science*, 1977.
25. Y. Ramakrishna, P. Melliar-Smith, L. Moser, L. Dillon, and G. Kutty. Really visual temporal reasoning. In *IEEE Real-Time Systems Symposium*, 1993.
26. L. Sha, R. Rajkumar, and S. Sathaye. Generalized rate-monotonic scheduling theory: a framework for developing real-time systems. In *Proceedings of the IEEE*, Jan 1994.
27. M. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proceedings of the First Symposium on Logic in Computer Science*, 1986.
28. F. Wang. Timing behavior analysis for real-time systems. In *Proceedings of the Tenth Symposium on Logic in Computer Science*, 1995.
29. J. Yang, A. Mok, and F. Wang. Symbolic model checking for event-driven real-time systems. In *IEEE Real-Time Systems Symposium*, 1993.

Verifying Continuous Time Markov Chains

Adnan Aziz	Kumud Sanwal	Vigyan Singhal	Robert Brayton
ECE	Bell Labs	CBL	EECS
UT Austin	AT&T	Cadence	UC Berkeley

Abstract. We present a logical formalism for expressing properties of continuous time Markov chains. The semantics for such properties arise as a natural extension of previous work on discrete time Markov chains to continuous time. The major result is that the verification problem is decidable; this is shown using results in algebraic and transcendental number theory.

Introduction

Recent work on formal verification has addressed systems with stochastic dynamics. Certain models for discrete time Markov chains have been investigated in [6, 3]. However, a large class of stochastic systems operate in continuous time. In a generalized decision and control framework, continuous time Markov chains form a useful extension [9]. In this paper we propose a logic for specifying properties of such systems, and describe a decision procedure for the model checking problem. Our result differs from past work in this area [2] in that quantitative bounds on the probability of events can be expressed in the logic.

1 Continuous Markov Chains

We will consider models of the form $M = (S, \Lambda, A, \theta)$, where $S = \{s_1, s_2, \ldots, s_n\}$ is a finite set of *states*, Λ is the *transition rate matrix*, A is a finite set of *outputs*, and $\theta : S \rightarrow A$ is the *output function*. A *path* through M is a map from $\mathbb{R}^+$to S (here $\mathbb{R}^+$denotes the set of non-negative reals); $S^{\mathbb{R}^+}$ is the set of all paths.

The transition rate matrix Λ is an $\mid S \mid \times \mid S \mid$ matrix. The off diagonal entries are non-negative rationals; the diagonal element a_{jj} is constrained to be $-(\sum_{i \neq j} a_{ji})$.

At state s_j, the probability of making a transition to state s_k (where $k \neq j$) in time dt is given by $a_{jk}\, dt$. This is the basis for formulating a stochastic differential equation for the evolution of the probability distribution whose solution is

$$D(t) = e^{\Lambda t} \cdot D_0$$

Here D_0 is a column vector of dimension $|S|$, with the constraint that $\sum_i [D_0]_i = 1$.

Technically, with any state s in M we associate a natural *probability space* $\mathcal{P}^s_M = (U^s, \mathcal{C}^s, \mu^s)$, where the set of all paths starting at s is the *universe* U^s, and the Borel sigma field on U^s gives the associated space of *events* $\mathcal{C}^s$, i.e.

the class of subsets of U^s to which probabilities can be assigned. The transition rate matrix Δ yields the probability measure $\mu^s : \mathcal{C}^s \rightarrow [0,1]$; by the measure extension theorem [10], μ^s is well defined. Given a set β of functions from $\mathbb{R}^+$to A, we will abuse notation and refer to the probability of β when we mean the probability of the set of all state sequences starting at s which map under θ to elements in β.

We will not dwell on the technicalities of measure theory; all the sets of paths defined later in this paper will be readily seen to be events, i.e. elements of $\mathcal{C}^s$.

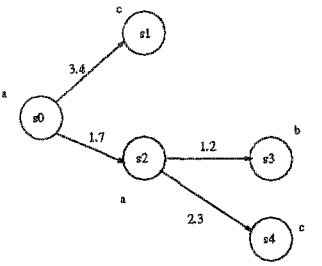


Fig. 1. A continuous time Markov chain: $S = \{s_0, s_1, s_2, s_3\}$; only edges with positive weights are shown.

2 CSL syntax and semantics

Let $M = (S, A, A, \theta)$ be a continuous Markov chain. In this section, we develop formal syntax and semantics for CSL (Continuous Stochastic Logic). This logic is inspired by the logic CTL [4], and its extensions to discrete time stochastic systems (pCTL [6]), and continuous time non-stochastic systems (tCTL [1]).

There are two types of formulae in CSL: state formulae (which are true or false in a specific state), and path formulae (which are true or false along a specific path). A state formula is given by the following syntax:

1. a for $a \in A$
2. If f_1 and f_2 are state formula, then so are $\neg f_1, f_1 \vee f_2$
3. If g is a path formula, then $Pr_{\sim c}(g)$ is a state formula. (c is a rational between 0 and 1 expressed as the ratio of two binary coded integers).

Path formulas are formulas of the form

- $f_1 U_{[a_1,b_1]} f_2 U_{[a_2,b_2]} \cdots f_n$, where $f_1, f_2, \ldots f_n$ are state formulas, and $a_1, b_1, \ldots, a_{n-1}, b_{n-1}$ are non-negative rationals expressed as the ratio of two binary coded integers.

CSL is the set of state formulae that are generated by the above rules.

Let f be a state formula, and g be a path formula. We now define the satisfaction relation ($\models_M$) using induction on the length of the formula. For a state formula f we use $[\![f]\!]_M$ to denote the set of states satisfying f.

1. f is of the form **a**: $s \models_M f$ iff $\theta(s) = a$.
2. f is of the form $(\neg f_1)$: $s \models_M f$ iff $s \not\models_M f$.
3. f is of the form $(f_1 \vee f_2)$: $s \models_M f$ iff $s \models_M f_1$ or $s \models_M f_2$.
4. f is of the form $Pr_{\geq c}(g)$: $s \models_M f$ iff $\mu^s(\{\pi \in S^{\mathbb{R}^+} \mid \pi \models_M g\}) > c$.
5. g is a path formula of the form $f_1 U_{[a_1,b_1]} f_2 U_{[a_2,b_2]} \cdots f_n$: $\pi \models_M g$ iff there exist $t_1, \ldots, t_{n-1}$ such that $(\forall i)\ [(a_i \leq t_i \leq b_i) \wedge (\forall t' \in [t_{i-1}, t))\ (\pi(t) \in [\![f_i]\!]_M)]$ (where t_{-1} is defined to be 0 for convenience).

Example 1. The formula $\phi = Pr_{>0.03}(aU_{[0.0,4.0]}b)$ is a state formula for the example in figure 1. It formally expresses that with probability greater that 0.03, the system will remain in a state where the output is a before making a transition before 4.0 seconds have elapsed to a state where the output is b.

The probability of the set of paths starting at s_0 on which the output is a before becoming b before time t_1 is given by the following integral:

$$\int_0^{t_1} \int_0^{t_1 - \tau_1} e^{-(r_1+r_2)\cdot\tau_1} \cdot e^{-(r_3+r_4)\cdot\tau_2} \cdot (r_2/(r_1+r_2)) \cdot (r_3/(r_3+r_4)) \cdot d\tau_1 \cdot d\tau_2$$

Setting t_1 equal to 4.0, and taking the rates $r_1 = 1.0, r_2 = 2.0, r_3 = 3.0$, and $r_4 = 3.0$, this simplifies to $(1/45) - (e^{-12}/18) + (e^{-20}/30)$. Observe that $e^{-12}/18 > e^{-20}/30$; hence the probability is bounded above by $(1/45)$ which is less than 0.03, and so ϕ is false at s_0.

3 CSL model checking

The CSL model checking problem is as follows: given a continuous Markov chain M, a state s in the chain, and a CSL formula f, does $s \models_M f$? In this section we establish that the model checking problem for CSL is decidable.

Theorem 1. CSL model checking is decidable.

Proof. The non-trivial step in model checking is to model check formula of the form $Pr_{>c}(g)$. In order to do this we need to be able to effectively reason about the measure of the set $\{\pi \in S^{\mathbb{R}^+} \mid \pi(0) = s_0 \wedge \pi \models_M g\}$ under μ^{s_0}.

First, we review some elementary algebra. An *algebraic complex number* is any complex number which is the root of a polynomial with rational coefficients. Properties of the algebraic numbers are derived in [8]; of particular interest to us is the fact that they constitute a field, and that the real and imaginary parts of an algebraic number are also algebraic.

We will denote the set of complex numbers which are of the form $\sum_j \eta_j e^{\delta_j}$ where the η_j and δ_j are algebraic by $E_{\mathcal{A}}$. This set is a ring, and is referred to as the *transcendental extension* of A by e [8].

Tarksi [11] proved that the theory of the field of complex numbers (i.e. the theory of the structure $<\mathbb{C}, +, \times, 0, 1>$) was decidable; an effective (in the recursion theoretic sense) procedure for converting formulas to a logically equivalent quantifier free form was given. Consequences of this result include the existence

of effective procedures for determining the number of distinct roots of a polynomial, and testing the equality of algebraic numbers defined by formulas.

We now demonstrate how to measure the set of paths which start at a designated state and satisfy a specified path formula. Consider a path formula of the form $\psi_0 U_{[a_1,b_1]} \psi_1 U_{[a_2,b_2]} \psi_2 \cdots$

First, consider the case where the time intervals $[a_1, b_1]$, $[a_2, b_2], \ldots$ are non overlapping.

We define the following matrices.

- a transition matrix $Q_{i,i}$ obtained from Λ, that treats $[\![\psi_i]\!]^c_M$ as an absorbing set of states. This is obtained by using

$$\begin{aligned} q(j,k) &= \lambda_{j,k} \text{ if } j \in [\![\psi_i]\!]_M \\ &= 0 \quad \text{if } j \in [\![\psi_i]\!]^c_M \end{aligned}$$

  this enables us to model the transitions where the Markov chain remains in $[\![\psi_i]\!]_M$.
- a transition matrix $Q_{i-1,i}$ obtained from Λ, that treats $[\![\psi_{i-1}]\!]^c_M \cap [\![\psi_i]\!]^c_M$ as an absorbing set of states. For this we use

$$\begin{aligned} q(j,k) &= \lambda_{j,k} \text{ if } j \in [\![\psi_i]\!]_M \cup [\![\psi_{i-1}]\!]_M \\ &= 0 \quad \text{if } j \in [\![\psi_i]\!]^c_M \cap [\![\psi_{i-1}]\!]^c_M \end{aligned}$$

  this allows us to model the transitions from $[\![\psi_{i-1}]\!]_M$ to $[\![\psi_i]\!]_M$.
- An indicator matrix I_i for $[\![\psi_i]\!]_M$, such that

$$\begin{aligned} I_i(j,k) &= 1 \text{ if } j = k \in [\![\psi_i]\!]_M \\ &= 0 \text{ otherwise} \end{aligned}$$

Hence, the probability of a formula of the form

$$f_1 = \psi_0 U_{[a_1,b_1]} \psi_1 U_{[a_2,b_2]} \psi_2 \cdots U_{[a_n,b_n]} \psi_n \tag{1}$$

is given by

$$\begin{aligned} \mu^s(f_1) = \pi_s \cdot P_{0,0}(a_1) \cdot I_0 \cdot P_{0,1}(b_1 - a_1) \cdot I_1 \cdot P_{1,1}(a_2 - b_1) \cdot \\ I_1 \cdot P_{1,2}(b_2 - a_2) \cdot I_2 \cdots P_{n-1,n}(b_n - a_n) \cdot I_n \cdot \underline{1} \end{aligned} \tag{2}$$

where $P_{l,m}(t), t \geq 0$ is the one step transition matrix for time t corresponding to the rate matrix $Q_{l,m}$, π_s is the starting probability distribution, which in our case has unity for state s and zeros otherwise, and $\underline{1}$ is the column vector whose elements are all 1. For a finite state Markov chain with a transition rate matrix Q, this matrix is given by

$$P(t) = e^{Qt}$$

Note that Q is composed of rational entries, and the arguments of $P_{i-1,i}$ are rationals (since a_i, b_i are rational). This observation leads to the following lemma:

Lemma 2. Each element of the $P_{l,m}(t)$ matrices may be expressed as $\sum_j \eta_j e^{\delta_j}$ where η_j and δ_j are algebraic complex numbers.

Proof. Any square matrix B can always be expressed in Jordan canonical form [7], i.e. in the form $C \cdot J \cdot C^{-1}$. Here J is an upper block diagonal matrix as shown below:

$$\begin{bmatrix} J_1 & 0 & \cdots & 0 \\ 0 & J_2 & 0 \cdots & 0 \\ 0 & \cdots & J_3 \cdots & 0 \\ & & \ddots & \\ 0 & & \cdots & J_n \end{bmatrix}$$

The diagonal entries of each J_i are the eigenvalues of B, and the remaining entries of J_i are unity, as shown below:

$$\begin{bmatrix} \lambda_i & 1 & 0 & \ldots & 0 \\ 0 & \lambda_i & 1 & \ldots & 0 \\ & & & \ddots & \\ 0 & & \cdots & 0 & \lambda_i \end{bmatrix}$$

The size of J_i is equal to the multiplicity of λ_i. Since the eigenvalues are the solutions of the characteristic equation of B and the entries of B are rationals, the eigenvalues are, by definition, algebraic complex numbers. Similarly, the entries of C, C^{-1} are also algebraic complex numbers.

The matrix e^{Bt} is equal to $C \cdot e^{Jt} \cdot C^{-1}$ and e^{Jt} is of the following form:

$$\begin{bmatrix} e^{J_1 t} & 0 & \cdots & 0 \\ 0 & e^{J_2 t} & 0 & \cdots & 0 \\ 0 & \cdots & e^{J_3 t} & \cdots & 0 \\ & & & \ddots & \\ 0 & & \cdots & & e^{J_n t} \end{bmatrix}$$

The sub-matrix $e^{J_i t}$ is of the form

$$\begin{bmatrix} e^{\lambda_i t} & te^{\lambda_i t} & (t^2 e^{\lambda_i t})/2! & \cdots & (t^{m_i} e^{\lambda_i t})/(m_i)! \\ 0 & e^{\lambda_i t} & te^{\lambda_i t} & \cdots & (t^{m_i - 1} e^{\lambda_i t})/(m_i - 1)! \\ & & & \ddots & \\ & & & & e^{\lambda_i t} \end{bmatrix}$$

By inspection, the elements of $e^{J_i t}$ are members of $E_{\mathcal{A}}$. Since $E_{\mathcal{A}}$ is a ring, it is closed under products and sums. Hence the lemma follows. It also follows that $\mu^s(f_1)$ is a member of $E_{\mathcal{A}}$ i.e. equal to an expression of the form $\sum_k \eta_k e^{\delta_k}$ where the η_k, δ_k are algebraic. ∎

Consider again the expression for $\mu^s(f_1) = \sum_k \eta_k e^{\delta_k}$. The δ_k's are algebraic; since are effective procedures for checking the equality of algebraic numbers, $\mu^s(f_1)$ can be effectively simplified to an expression of the form $\sum_{k'} \eta_{k'} e^{\delta_{k'}}$ where the $\eta_{k'}$'s are non zero, and the $\delta_{k'}$'s are distinct.

In order to decide if $\mu^s(f_1) > c$, we exploit a celebrated theorem of transcendental number theory [8].

Theorem 3 (Lindemann-Weirstrass:). *Let $c_1, \ldots c_n$ be pairwise distinct algebraic numbers belonging to $\mathbb{C}$. Then there exists no equation $a_1 e^{c_1} + \cdots + a_n e^{c_n} = 0$ in which $a_1, \ldots, a_n$ are algebraic numbers and are not all zero.*
Historical note:This result implies the transcendence of π (take $n = 2$, $c_1 = 2, c_2 = i\pi$); it was the first proof of the non-algebraic nature of π. For a highly readable account of the development of this theorem, refer to [5].

Suppose the expression $\sum_{k'} \eta_{k'} e^{\delta_{k'}}$ is degenerate, i.e. it consists of a single term of the form η_0. Then the expression denotes an algebraic number, and it can be effectively checked if it is greater than c.

If it is not degenerate, invoking the Lindemann-Weirstrass theorem and noting that c is rational, we see that $\mu^s(f_1)$ can not be equal to c and so $|\mu^s(f_1) - c| > 0$.

Decidability of model checking follows from the following lemma.

Lemma 4. Given a transcendental real r of the form $\sum_j \eta_j e^{\delta_j}$ where the η_j and δ_j are algebraic complex numbers, and the δ_j's are pairwise distinct, there is an effective procedure to test if $r > c$ for rational c.

Proof. Suppose a sequence of algebraic numbers $S_1, S_2, \ldots$ such that $|r - S_k| < 2^{-k}$ can be effectively constructed. Let $|r - c| = a > 0$. By the triangle inequality, $|r - c| \leq |r - \mathtt{Re}(S_k)| + |\mathtt{Re}(S_k) - c|$. Hence $|r - \mathtt{Re}(S_k)| + |\mathtt{Re}(S_k) - c|$ is bounded away from 0 by a. Since r is real, $|r - \mathtt{Re}(S_k)| \leq |r - (S_k)| < 2^{-k}$, and $|r - \mathtt{Re}(S_k) + |\mathtt{Re}(S_k) - c|$ is bounded away from 0 by a, for sufficiently large k, it must be that $|\mathtt{Re}(S_k) - c| > 2^{-k}$. The sign of of $\mathtt{Re}(S_k) - c$ is the sign of $r - c$.

In order to construct the sequence $S_1, S_2, \ldots$, we use the fact that e^z can be approximated with an error of less than ϵ (when $\epsilon < 1$) by taking the first $\lceil (3 \cdot |z|^2 / \epsilon) \rceil + 1$ terms of the Maclaurin expansion for e^z. This can be extended to obtain an upper bound on the number of terms to sum for an expression of the form $\sum_j \eta_j e^{\delta_j}$ (which being the finite sum of algebraic numbers is algebraic) in order to achieve an error of less than ϵ. ∎

Now consider the case where the successive intervals where the transitions are desired ($[a_i, b_i]$, $i = 1, 2, \ldots$ are allowed to overlap. Since a formula is finite, we can have a finite number of overlapping intervals. A key observation is that the finite number of overlaps allows us to partition the time in a finite number of non-overlapping intervals and write the probability of the specification (set of acceptable paths) as a sum of the probabilities of disjoint events. This enables us to write $\mu^s(f_1)$ as the sum of exponentials of algebraic complex numbers,

weighted by algebraic coefficients. To illustrate this, consider the formula

$$f_2 = \psi_0 U_{[a_1,b_1]} \psi_1 U_{[a_2,b_2]} \psi_2$$

where $0 < a_1 < a_2 < b_1 < b_2$. In this case, we may realize f_2 as one of four disjoint cases and hence we can write

$$\begin{aligned}\mu^s(f_2) = \mu^s(\psi_0 U_{[a_1,a_2]} \psi_1 U_{[a_2,b_1]} \psi_2) + \mu^s(\psi_0 U_{[a_1,a_2]} \psi_1 U_{[b_1,b_2]} \psi_2) \\ + \mu^s(\psi_0 U_{[a_2,b_1]} \psi_1 U_{[b_1,b_2]} \psi_2) + \mu^s(\psi_0 U_{[a_2,b_1]} \psi_1 U_{[a_2,b_1]} \psi_2) \qquad (3)\end{aligned}$$

The first three terms are equivalent to the case with non-overlapping intervals. The last term involves having both the $[\![\psi_0]\!]_M \rightarrow [\![\psi_1]\!]_M$ and $[\![\psi_1]\!]_M \rightarrow [\![\psi_2]\!]_M$ transitions in the same interval $[a_2, b_1]$ in the correct order. This may be evaluated by integrating the probabilities over the time of the first transition.

$$\mu^s(\psi_0 U_{[a_2,b_1]} \psi_1 U_{[a_2,b_1]} \psi_2) = \pi_s P_{0,0}(a_2) I_0 \int_{a_2}^{b_1} P_{0,0}(t - a_2) I_0 Q_{0,1} I_1 P_{1,2}(b_1 - t) I_2 dt$$

It is clear that since the integrand involved algebraic terms and and exponentials in algebraic complex numbers and t, the definite integral with rational limits can be written in the form of a sum of exponentials of algebraic numbers with algebraic coefficients. Hence, this term is in $E_{\mathcal{A}}$. The other three terms in equation 3 correspond to forms equivalent to the non-overlapping intervals case, and hence already satisfy the decidability criteria. ∎

4 Conclusions and Future Work

We have defined a logic for specifying properties of finite state continuous time Markov chains. The model checking problem for this logic was shown to be decidable through a combination of results in algebraic and transcendental number theory. In practise we believe that a model checker can be built using conventional numerical methods for computing probabilities of events in continuous Markov chains.

In the future, we intend to study synthesis of specifications in the logic. We are planning to use some of the techniques used in this paper to derive decidability results for analyzing dynamical systems which evolve using exponential laws.

References

1. R. Alur, C. Courcoubetis, and D. Dill. Model Checking for Real-Time Systems. In *Proc. IEEE Symposium on Logic in Computer Science*, pages 414–425, 1990.
2. R. Alur, C. Courcoubetis, and D. Dill. Model Checking for Probabilistic Real Time Systems. In *Proc. of the Colloquium on Automata, Languages, and Programming*, pages 115–126, 1991.

3. C. Courcoubetis and M. Yannakakis. Verifying Temporal Properties of Finite State Probabilistic Programs. In *Proc. IEEE Symposium on the Foundations of Computer Science*, pages 338–345, 1988.
4. E. A. Emerson. Temporal and Modal Logic. In J. van Leeuwen, editor, *Formal Models and Semantics*, volume B of *Handbook of Theoretical Computer Science*, pages 996–1072. Elsevier Science, 1990.
5. J. H. Ewing. *Numbers*. Springer-Verlag, 1991.
6. H. Hansson and B. Jonsson. A Logic for Reasoning about Time and Reliability. *Formal Aspects of Computing*, 6:512–535, 1994.
7. T. Kaliath. *Linear Systems*. Prentice-Hall, 1980.
8. I. Niven. *Irrational Numbers*. John-Wiley, 1956.
9. S. Ross. *Stochastic Processes*. Wiley, 1983.
10. H. L. Royden. *Real Analysis*. Macmillan Publishing, 1989.
11. A. Tarski. *A Decision Procedure for Elementary Algebra and Geometry*. University of California Press, 1951.

Verifying Safety Properties of Differential Equations

Mark R. Greenstreet, mrg@cs.ubc.ca
Department of Computer Science
University of British Columbia
Vancouver, BC V6T 1Z4
Canada

Abstract

This paper presents an approach to verifying that a circuit as described by a continuous, differential equation model is a correct implementation of a discrete specification. The abstraction from continuous trajectories to discrete traces is based on topological features of the continuous model rather than quantizing the continuous phase space. An practical verification method based on numerical integration is presented. The method is demonstrated by the verification that a toggle circuit satisfies a discrete specification.

1 Introduction

Most research in hardware verification has been based on discrete models for circuit behavior [Gup92]. In many high performance designs, discrete models are inadequate. Details of transition times, slew rates, capacitive coupling, etc. can be crucial for the correct operation of such designs. Accurate models for these phenomena are typically expressed as systems of non-linear differential equations. Thus, it becomes important to verify that a circuit modeled by non-linear differential equations is a valid implementation of a discrete specification.

This paper presents an approach to this problem based on methods from dynamical systems theory. Safety properties of the continuous model are verified by demonstrating the existence of suitable invariant manifolds in the continuous phase space. Interface specifications are expressed as constraints on the relationship between signals and their time derivatives, and continuous trajectories are mapped to discrete traces rather than attempting to discretize the continuous phase space to define discrete states. This topological approach to describing behavior is described in section 3.

Section 5 describes the verification method. The continuous system is verified by computing a conservative bound on the reachable region of the system throughout a continuous integration. The reachable region is represented by its projection onto planes defined by pairs of the continuous variables. This approach allows standard computational geometry algorithms to be used to maintain the data structure for the reachable region, and it avoids the exponential complexity of explicitly representing a high-dimensional object. A numerical integrator is

used to determine the evolution of the reachable region. This approach is demonstrated by verifying that a toggle circuit implements its discrete specification. The circuit is described in section 4, and section 6 presents its verification.

Recently, there has been a large interest in the verification of continuous systems. Much of this is based on linear automata models [OSY94] which cannot be applied directly to the non-linear models of VLSI circuits. Henzinger and Ho [HH95] showed how these methods could be applied to non-linear systems by constructing asymptotically equivalent linear descriptions. The approach presented here differs from theirs in that the verification is performed directly on the system on non-linear differential equations. This facilitates using ideas from dynamical systems theory both for the specification and the verification of the design. Kurshan and McMillan [KM91] presented the verification of an arbiter circuit using a circuit model similar to the one used this paper. They partitioned the phase space into fixed boxes and computed a next-state relation between these boxes by integrating the non-linear circuit model for fixed time steps. This leads to an interaction of the time step size and the box size that is avoided by the methods presented here.

2 A Discrete Toggle

In this paper, discrete behaviors are described using finite state automata. A finite state automaton is described by a quadruple (I, O, Δ, Q_0), where I is a set of binary valued inputs and O is a set of of binary valued outputs. The state space, S, is $2^{I \cup O}$. For $s \in S$, $v(s)$ denotes the value of input or output v in state s. The set of initial states is given by Q_0, with $Q_0 \subseteq S$, and Δ is the state transition relation with $\Delta \subseteq S \times S$. A **trace** is a sequence of states, $s_0, s_1, \ldots$, such that $s_0 \in Q_0$ and $\forall i.\, (i \geq 0) \Rightarrow (s_i, s_{i+1}) \in \Delta$. The state transition relation is partitioned into circuit actions and environment actions. A circuit action only changes the values of outputs and an environment action only changes the values of inputs. More formally,

$$\forall (s_1, s_2) \in \Delta.\, (\forall v \in I.\, v(s_1) = v(s_2)) \vee (\forall v \in O.\, v(s_1) = v(s_2))$$

A simple toggle element has a clock input Φ and two outputs a and b. the singleton initial state set $\{(F, F, F)\}$, where F denotes the logical value false and state triples are written $(\Phi, \mathtt{a}, \mathtt{b})$. The state transition relation is

$$\{\ ((F,F,F),\ (T,F,F)),\ ((T,F,F),\ (T,T,F)),\ ((T,T,F),\ (F,T,F)),\\ ((F,T,F),\ (F,T,T)),\ ((F,T,T),\ (T,T,T)),\ ((T,T,T),\ (T,F,T)),\\ ((T,F,T),\ (F,F,T)),\ ((F,F,T),\ (F,F,F))\ \}$$

Figure 1 depicts the state space and state transition relation of this toggle embedded in a binary 3-cube. The salient features of this circuit description include:

Environment assumption: The clock input Φ only changes in states where no circuit actions are enabled.

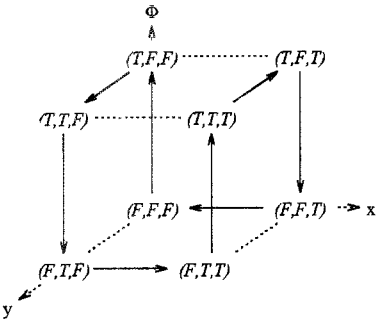


Fig. 1. A Discrete Toggle Element

Toggling: The period of the cycle of states for the toggle is twice the period of the clock input.

Compositional elements: The a output makes exactly one low-to-high and one high-to-low transition during each cycle of states. Likewise for the b output. This allows, for example, a counter to be constructed by connecting the output of one toggle element to the input of another as long as the environment assumption can be shown to be satisfied.

Continuous behaviors can be described using ordinary differential equations (ODEs). By analogy with a finite state automaton, we describe ODEs with a tuple, $(\mathcal{I}, \mathcal{P}_\mathcal{I}, \mathcal{O}, \delta, \mathcal{Q}_0)$. $\mathcal{I}$ is a set of real-valued inputs, and $\mathcal{P}_\mathcal{I}$ is a condition that must be satisfied by the inputs. For example, we assume that inputs are a continuous, bounded functions of time; additional conditions for inputs are described in section 3. $\mathcal{O}$ is a set or real-valued outputs. If the model has d_i inputs and d_o outputs, then the state space is R^d where $d = d_i + d_o$. The derivative function, $\delta : R^d \to R^{d_o}$, gives the time derivative of each output as a function of the current state. The initial point is any point in $\mathcal{Q}_0$ where $\mathcal{Q}_0 \subseteq R^d$. We assume that δ is Lipschitz, and that the inputs are continuous functions of time. These conditions guarantee that the outputs are uniquely defined for any inputs and initial state. We call a tuple, $(\mathcal{I}, \mathcal{P}_\mathcal{I}, \mathcal{O}, \delta, \mathcal{Q}_0)$ a **continuous model**.

A continuous model defines a set of trajectories. A trajectory, η is a differentiable function from time to R^d where

$$\begin{aligned} & \eta(0) \in \mathcal{Q}_0 \\ \wedge\ & \mathcal{P}_\mathcal{I}(\eta) \\ \wedge\ & \frac{d}{dt}\mathcal{O}(\eta) = \delta(\mathcal{I}(\eta), \mathcal{O}(\eta)) \end{aligned}$$

Given a continuous model Ω and a finite state automaton F, an abstraction function maps trajectories of Ω to traces of F. We say that Ω is an implementation of F with abstraction function A iff for every trajectory η of Ω, $A(\eta)$ is a valid trace of F. Note that abstractions map continuous trajectories to discrete

traces rather than states to states. In this approach, discrete behaviors can be understood as topological properties of families of trajectories which allows concepts from dynamical systems theory to be applied to the problem of verifying that a continuous model of a circuit satisfies a discrete specification.

3 Continuous Realizations of Discrete Behaviors

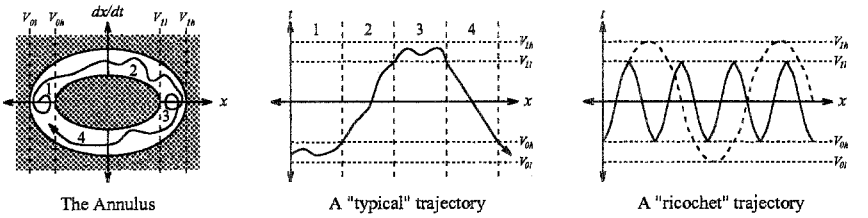


Fig. 2. Brockett's Annulus

Figure 2 depicts an annulus proposed by Brockett [Bro89] that provides a topological basis for mapping continuous trajectories to discrete behaviors. When a variable is in region 1, its value is constrained but its derivative may be either positive or negative. When the variable leaves region 1, it must enter region 2. Because the derivative of the variable is positive in this region, it makes a monotonic transition leading to region 3. Regions 3 and 4 are analogous to regions 1 and 2 corresponding to logically high and monotonically falling signals respectively. Because transitions through regions 2 and 4 are monotonic, traversals of these regions are distinct events. This provides a topological basis for discrete behaviors. Furthermore, the horizontal radii of the annulus define the maximum and minimum high and low levels of the signal (i.e. V_{0l}, V_{0h}, V_{1l}, and V_{1h} in figure 2). The maximum and minimum rise time for the signal correspond to trajectories along the upper-inner and upper-outer boundaries of the annulus respectively. Likewise, the lower-inner and lower-outer boundaries of the annulus specify the maximum and minimum fall times.

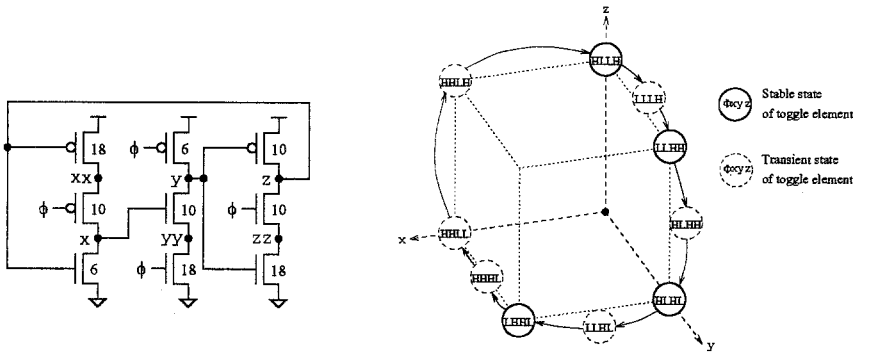
If the annulus is given by two ellipses, then the trajectories corresponding to the inner and outer boundaries of the annulus are sine waves, and it is tempting to think of these as giving upper and lower bounds for the period of the signal. This is not the case. First, note that a signal may remain in regions 1 or 3 arbitrarily long. This is essential when verifying the toggle where we must show that the output satisfies the constraints assumed of the input, even though the period of the output is twice that of the input. Furthermore, the signal is not required to spend any time in regions 1 and 3. The minimum period signal corresponds to a "ricochet" trajectory as depicted by the solid curve in the right most plot of figure 2. The period of this signal can be much less than that of the

sine wave corresponding to the outer boundary of the annulus (the dashed curve). It is desirable to independently specify constraints on signal levels, transition times, and period. We achieve this by imposing minimum times that a signal must remain in regions 1 and 3. This construction allows a large class of input signals to be described in a simple and natural manner.

To verify safety properties of a continuous model, we establish the existence of an invariant manifold in R^d. We then show that all trajectories starting from points in the initial region are contained in this manifold, and that all trajectories in this manifold satisfy the specification. This technique is the continuous analog of using discrete invariants to verify properties of state transition systems [LS84].

For the toggle element, all trajectories in the invariant manifold should have a period twice that of the clock signal. This notion can be formalized using a Poincaré section [PC89]. Let ϕ be the continuous signal corresponding to Φ, and let c be some constant with $V_{0h} < c < V_{1l}$. Consider the intersection of the manifold with the $\phi = c$ hyperplane. These intersections form a Poincaré map. This intersection must consist of four disjoint regions (two for rising ϕ crossing c, and two falling crossings) and all trajectories must visit these four regions in the same order. Continuous trajectories can be mapped to discrete sequences of the discrete toggle described in section 2 by mapping regions of the manifold to states of the discrete toggle. The toggle element is compositional if it there is an output variable such that for all trajectories in the manifold, the value and derivative of this variable satisfy the constraints of the input ring. It must also be shown that this output satisfies the minimum high and low time constraints.

4 A toggle circuit



Schematic State Transition Diagram

Fig. 3. Yuan and Svensson's Toggle

Figure 3 shows a toggle circuit that was originally published by Yuan and

Svensson [YS89]. The operation of this circuit can be understood by using a simple switch model starting from a state where the ϕ input is low. In this case, $\mathbf{y}$ will eventually become high, $\mathbf{z}$ is floating, and $\mathbf{x}$ is the logical negation of $\mathbf{z}$. If we assume that the value stored on node $\mathbf{z}$ is a well-defined logical value, then the circuit has two possible states when ϕ is low: $(\mathbf{x}, \mathbf{y}, \mathbf{z}) = (L, H, H)$, and $(\mathbf{x}, \mathbf{y}, \mathbf{z}) = (H, H, L)$. Starting from these two states, we can derive the corresponding stable successor states for when ϕ is high. If the circuit is allowed to reach a stable state before each transition of the ϕ input, then it implements a toggle as illustrated by the state transition diagram shown on the right half of figure 3.

The analysis presented in this paper is based on a simple circuit model using a standard, first-order transistor model [GD85] with three regions of operation: cut-off, saturation, and "linear." Capacitors are of fixed value, and all capacitances are to ground. Using basic circuit analysis techniques, we obtain a system of non-linear differential equations that is our continuous model for the circuit. A more detailed description of this model is given in [GC94].

5 Verification

Let Ω be a continuous model. Properties of Ω can be verified by finding a manifold that contains all trajectories of Ω. This can be done by starting with the initial region of the model and integrating the system of differential equations to compute a bounding region at each step. In the present work, variations in the input signal and initial state are considered, but the model parameters are fixed. Because the non-linear equations that arise from circuit models cannot be integrated analytically, this integration is performed numerically. Thus, this verification requires an assumption of the validity of the numerical integrator. The verification described in this paper uses a fourth-order Runge-Kutta integrator adapted from [PF+88].

The Brockett annulus provides a convenient way to perform this integration. When the input signal ϕ is in the first or third region of figure 2, either the N-channel or the P-channel transistors controlled by ϕ are in cut-off. For typical CMOS circuits including the toggle this ensures that each node is either floating in which case the time derivative of its voltage is zero, or that it there is a conducting path to either Vdd or ground, but not both. In this case the voltage of the node asymptotically approaches the corresponding power supply value. Given a bounding region for trajectories upon entry to the first or third region of the annulus, we integrate for the minimum low or high time respectively and then determine the bounding box for the reachable region. For nodes that are asymptotically approaching a power supply value, the box is expanded to include that value. The expanded box is used as the starting region for the next phase of integration. When ϕ is in the second or fourth region, its value is changing monotonically. This allows the integration to be performed with respect to ϕ which reduces the dimension of the phase space by one and reflects the natural dependence of the circuit on its input.

At each integration step, the reachable region is implicitly represented by a set of two types of constraints: simple and polygonal. Simple constraints give upper and lower bounds for the value of a single variable. Polygonal constraints give bounds on the values of pairs of variables: the polygon is a projection of the reachable region onto the plane corresponding to the two variables. In the current implementation, polygonal constraints are represented by simple, rectilinear polygons without convexity restrictions. Each polygon corresponds to a prism in the complete phase space, and the reachable region is represented by the intersection of these prisms clipped by the simple constraints. This approach avoids the exponential growth in complexity that would occur if the reachable region were explicitly constructed and it allows efficient algorithms from two-dimensional computational geometry to be utilized. The representation is conservative; accordingly, our verification method is sound but not complete.

At each integration step, a conservative estimate of the bounding region is computed. For each face of the reachable region, a maximum outward translation is determined. This translation is an upper bound on the outward normal component of any trajectory starting from some point on the face. Since the entire face is translated outward by this amount, this bounds all trajectories starting from that face. By performing this computation for each face, a conservative estimate is obtained for the bounding region at the end of the integration step.

Focusing on the non-degenerate cases[1], each face of the reachable region corresponds to a bound of a simple constraint or an edge from a polygonal constraint. If the constraint corresponds to a polygon edge, it gives an exact value for one variable and a bounding interval for the other. If the constraint is an upper or lower bound of a simple constraint, it gives a value for that variable. Given these explicit constraints, bounds on other variables can be derived from the polygonal constraints. In this way, we compute bounding intervals for each variable for each face. From this, maximum and minimum values are computed for the outward component of the derivative vector for points on the face. For models arising from CMOS circuit models, this requires calculating upper and lower bounds for the drain current of each transistor, and the monotonicity of the transistor model simplifies this calculation. Because a fourth-order integration algorithm is used, four of these derivative calculations are performed at each step, and an error estimate is calculated to adjust the step size.

There are several details that must be considered. First, as the integration is performed, some polygon edges will grow. To avoid excessively conservative estimates of the reachable region, edges are split into smaller edges when they exceed a pre-specified length. Conversely, when adjacent edges become sufficiently short, they are conservatively merged into a single edge for efficiency. If two polygonal constraints involve the same variable, then they each have edges corresponding to the maximum and minimum values of this variable. When one of these extremal edges is split, then it may be possible to compute a tighter bound for its outward derivative than for the corresponding edge of the other

[1] Zero-length polygon edges and coincident constraints can occur as a consequence of constraint splitting described shortly.

polygon. In this case, the unsplit edge may acquire an infeasible value at the end of an integration step. When this occurs, the algorithm solves the system of constrains and moves the infeasible edge to its maximally outward feasible position.

In addition to the change of variables to integrate with respect to ϕ, it is sometimes convenient to perform additional changes of the variable of integration. Once it is shown that some variable, u, changes monotonically with respect to ϕ, then u can be used as the variable of integration. This can provide tighter estimates of the reachable region, but it requires a relation to bound ϕ given u. This relation can be represented by another polygon, and the polygon manipulation routines for the integrator can be used to perform this change of variables as well.

6 Verifying the Toggle Circuit

The toggle element of section 4 can be verified by chosing an initial region and integrating that region through two periods of the clock input as described in section 5. An invariant manifold is identified by showing that the reachable region at the end of this integration is contained in the initial region. By computing the intersection of this manifold with the $\phi = 2.5$volts hyperplane, it is shown that the manifold has a period that is twice that of ϕ as required. We use z as the output of the toggle, and by computing bounds on z and dz/dt at each integration step, we show that z satisfies the same ring constraints as the input. Details of this process are described in the remainder of this section.

The specification for the ϕ input is an annulus whose inner-boundary corresponds to a 100 MHz., 4.5 volt peak-to-peak sine wave centered at 2.5 volts. The outer boundary corresponds to a 700 MHz. 5.5 volt peak-to-peak sine wave also centered at 2.5 volts. The large difference between these frequencies demonstrates the robustness of the toggle to variations in the input signal. The minimum high and low times for ϕ are each 1 nano-second. This yields a minimum period of ϕ of ~ 2.87 nano-seconds which corresponds to a maximum frequency of 348 MHz.

The circuit model is simplified by assuming that the capacitances at nodes xx, yy, and zz are negligible, and a four-terminal device model is used for pairs of transistors of the same type in series. A 160 femtofarad load is added to the z node to simulate the effect of driving the ϕ input of another toggle element. "Typical" values for the MOSIS 2μ n-well CMOS process were used for the analysis. All transistors have a 2μ gate length and shape factors are shown in figure 3. Diffusion and gate capacitances are included in the model; for simplicity, interconnect capacitance is ignored.

The initial region is given by the constraints: $\phi = 0.25$; $-0.1 \leq x \leq 0.1$; $4.9 \leq y \leq 5.1$; and $4.8 < z < 5.1$. The integration starts with ϕ entering the second region of Brockett's annulus. The integration is performed in four phases: (1) ϕ rising and high, (2) ϕ falling and low, (3) ϕ rising and high, (4) ϕ falling and low. Each phase is started with a new bounding box, and at the end of each phase, we verify that the reachable region is contained in the initial bounding box

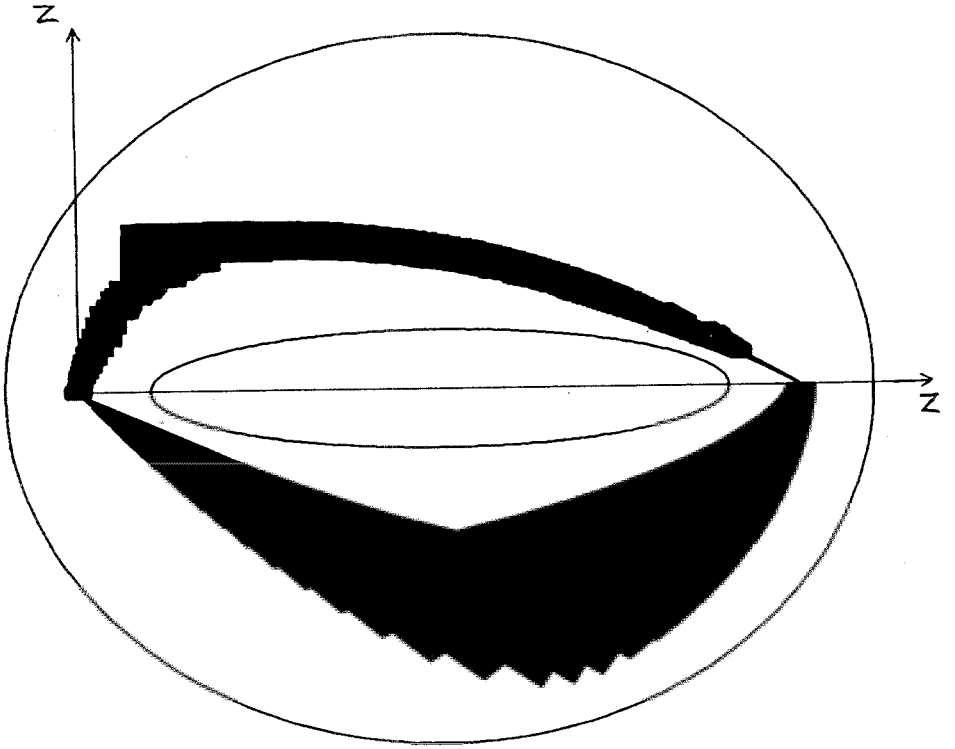


Fig. 4. Reachable region for z and dz/dt

for the next phase. This allows the four phases to be verified separately. After integration for two periods of ϕ, the reachable region satisfies the constraints: $\phi = 0.25$; $-1.57*10^{-13} \leq x \leq 1.61*10^{-13}$; $4.99 \leq y \leq 5.00$; and $4.83 < z < 5.05$. This demonstrates the existence of an invariant manifold as required.

In most of the phases, only a single variable has any large change in its value, and it is sufficient to approximate the reachable region by a bounding box. However, in the phase where ϕ and z make low-to-high transitions, x and y also make high-to-low transitions (see figure 3). The correct operation of the toggle requires that y complete its transition before x goes too far low. In this phase, coupling of each pair of variables with polygonal constraints was required along with an additional change of variable of integration from ϕ to y.

At each step of the integration, bounds on z and dz/dt are computed. These are shown in figure 4 with the annulus used to specify the input. It can be seen that z satisfies the specification for an input to the toggle. Furthermore, the integration shows that the minimum low-time for z is at least 2.74 nanoseconds and the minimum high time is at least 2.16 nanoseconds. Thus, z satisfies the requirements for an input signal to the toggle.

The current implementation of the verification algorithm is only for proof of concept and no effort has been for optimization. The run-time is dominated by the time for integration when z makes its low-to-high transition. When regions are estimated using polygons with a 0.25 volt nominal edge length, this step takes about forty minutes on a 50 MHz SPARC 10. When the nominal edge length is increased to 0.5 volts, the verification can be performed in just over five minutes. Further work will be required to optimize the implementation and characterize its performance on a larger set of examples.

7 Conclusions

This paper has presented a method for verifying that a circuit modeled by a system of non-linear differential equations satisfies a discrete specification. The approach is based on topological properties of the continuous model. Verification of the continuous model is performed by numerical integration to determine a manifold containing all feasible trajectories. Properties of the trajectories can be derived from the manifold by using methods from dynamical systems theory such as Poincaré sections.

The method has been applied to a toggle element. It was shown that the toggle operates correctly for a large class of input signals, and that its output satisfies the constraints required of its input. This means that these toggle elements can be connected in a chain to form a verified ripple counter. Although the toggle is a relatively simple circuit, its complexity is comparable to that of many cells in a typical standard-cell library. A potential application of these methods would be to verify that such a library has been properly designed and characterized.

Acknowledgements

I would like to express my appreciation to Peter Cahoon, Nick Pippenger, and Jim Varah for helpful suggestions throughout this work. Further thanks to the anonymous referees whose comments have contributed to the clarity of this paper.

References

[Bro89] R. W. Brockett. Smooth dynamical systems which realize arithmetical and logical operations. In Hendrik Nijmeijer and Johannes M. Schumacher, editors, *Three Decades of Mathematical Systems Theory: A Collection of Surveys at the Occasion of the 50th Birthday of J. C. Willems*, volume 135 of *Lecture Notes in Control and Information Sciences*, pages 19–30. Springer-Verlag, 1989.

[GC94] Mark R. Greenstreet and Peter Cahoon. How fast will the flip flop? In *Proceedings of the 1994 International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 77–86, Salt Lake City, November 1994. IEEE Computer Society Press.

[GD85] Lance A. Glasser and Daniel W. Dobberpuhl. *The Design and Analysis of VLSI Circuits.* Addison-Wesley, 1985.

[Gup92] Aarti Gupta. Formal hardware verification methods: A survey. *Formal Methods in System Design*, 1(2/3):151–258, October 1992.

[HH95] Thomas A. Henzinger and Pei-Hsin Ho. Algorithmic analysis of nonlinear hybrid systems. In *Proceedings of CAV 95*, pages 225–238, 1995.

[KM91] R.P. Kurshan and K.L. McMillan. Analysis of digital circuits through symbolic reduction. *IEEE Transactions on Computer-Aided Design*, 10(11):1356–1371, November 1991.

[LS84] Leslie Lamport and Fred B. Schneider. The 'Hoare logic' of CSP, and all that. *ACM Transactions on Programming Languages*, 6(2), April 1984.

[OSY94] A. Olivero, J. Sifakis, and S. Yovine. Using abstractions for the verification of linear hybrid systems. In David L. Dill, editor, *Proceedings of 1994 Workshop on Computer Aided Verification*, volume 818 of *Lecture Notes in Computer Science*, pages 81–94. Springer-Verlag, June 1994.

[PC89] Thomas S. Parker and Leon O. Chua. *Prectical Numerical Algorithms for Chaotic Systems.* Springer-Verlag, New York, 1989.

[PF+88] William H. Press, Brian P. Flannery, et al. *Numerical Recipes in C: The art of numerical computing.* Cambridge University Press, 1988.

[YS89] Jiren Yuan and Christer Svensson. High-speed CMOS circuit technique. *IEEE Journal of Solid-State Circuits*, 24(1):62–70, February 1989.

Temporal Verification by Diagram Transformations*

Luca de Alfaro and Zohar Manna

Computer Science Department
Stanford University
Stanford, CA 94305, USA

Abstract. This paper presents a methodology for the verification of temporal properties of systems based on the gradual construction and algorithmic checking of *fairness diagrams*. Fairness diagrams correspond to abstractions of the system and its progress properties, and have a simple graphical representation.
In the proposed methodology, a proof of a temporal property consists of a chain of diagram transformations, starting from a diagram representing the original system and ending with a diagram that either corresponds directly to the specification, or that can be shown to satisfy it by purely algorithmic methods. Each diagram transformation captures a natural step of the gradual process of system analysis and proof discovery. The structure of fairness diagrams simplifies reasoning about progress properties, and the graphical representation provided by the diagrams enables the user to direct the construction of the proof. The resulting methodology is complete for proving specifications written in first-order linear-time temporal logic, provided no temporal operator appears in the scope of a quantifier.

1 Introduction

This paper presents a methodology for the verification of temporal properties of fair transition systems based on the gradual construction and algorithmic checking of *fairness diagrams*. Fairness diagrams represent abstractions of the system, and provide a graphical formalism for the study of its temporal properties. Fairness diagrams are graphs whose vertices are labeled by first-order assertions and whose edges are labeled by first-order transition relations. Their progress properties are represented by *fairness constraints*, which generalize the classical concepts of fairness [8].

In the proposed methodology, a proof of a temporal specification consists of a chain of diagram transformations, starting with a fairness diagram representing the original system and ending with a fairness diagram that either corresponds

* This research was supported in part by the National Science Foundation under grant CCR-92-23226, the Advanced Research Projects Agency under NASA grant NAG2-892, the United States Air Force Office of Scientific Research under grant F49620-93-1-0139, and the Department of the Army under grant DAAH04-95-1-0317.

directly to the specification, or that can be shown to satisfy it by purely algorithmic methods. Since the transformations preserve containment of system behaviors, the existence of this chain of transformations implies that the set of behaviors of the original system is a subset of the set of behaviors that satisfy the specification.

Fairness diagram transformations are intended to capture the step-by-step nature of the process of system analysis and proof construction. We introduce two types of transformations. The first type relies on the construction of simulation relations between diagrams, and provides a flexible way to analyze the safety properties of the system. The second type relies on the proof of new progress properties of a fairness diagram. Once proved, these properties can be represented as fairness constraints and added to the diagram. The form of the fairness constraints has been chosen to make it possible to use simple but complete rules to reason about progress properties. The resulting methodology is complete for proving specifications written in first-order linear-time temporal logic, provided no temporal operator appears in the scope of a quantifier.

Related work. Methods based on stepwise system transformations for the study of branching-time temporal properties of finite-state systems have been proposed in [3], and the use of simulation relations to study the temporal behavior of fair transition systems has been discussed in [4]. A related approach to the proof of temporal properties of systems is based on the use of verification diagrams [10, 1]. Like fairness diagrams, verification diagrams are graphs labeled with first-order assertions, and enable the proof of general temporal properties. Unlike fairness diagrams, verification diagrams represent a completed proof, and trade the advantage of gradual proof construction for conciseness of proof representation.

2 Fairness Diagrams

A *fairness diagram* (diagram, for short) $A = (\mathcal{V}, \Sigma, V, \rho, \tau, \Theta, \mathcal{F})$ consists of the following components:

1. A set $\mathcal{V}$ of typed variables, called *state variables.*
2. A *state space* Σ: each state $s \in \Sigma$ is a type-consistent value assignment of all the variables in $\mathcal{V}$. For $x \in \mathcal{V}$, we denote by $s(x)$ the value of x at state s.
3. A set V of *vertices.*
4. A mapping $\rho : V \mapsto 2^{\Sigma}$, that labels each vertex $v \in V$ with a subset $\rho(v) \subseteq \Sigma$. The set $\rho(v)$ represents the possible states of the system when the diagram is at vertex v, and is specified by a first-order formula $\widehat{\rho}(v)$ over $\mathcal{V}$, such that $\rho(v) = \{s \in \Sigma \mid s \models \widehat{\rho}(v)\}$.
5. A mapping $\tau : V^2 \mapsto 2^{\Sigma \times \Sigma}$, that labels each edge $(u, v) \in V^2$ with a relation $\tau(u, v) \subseteq \Sigma^2$. The relation is specified by a formula $\widehat{\tau}(u, v)$ over $\mathcal{V}, \mathcal{V}'$, such that $\tau(u, v) = \{(s, s') \mid (s, s') \models \widehat{\tau}(u, v)\}$, where (s, s') interprets $x \in \mathcal{V}$ as $s(x)$ and $x' \in \mathcal{V}'$ as $s'(x)$.
6. An initial region Θ. Regions are defined below.
7. A *fairness set* $\mathcal{F}$, defined below.

Locations and runs. A *location* of a diagram is a pair $(v, s) : v \in V, s \in \rho(v)$ composed of a vertex and of a corresponding state. We denote by $loc(A) = \{(v, s) \mid v \in V, s \in \rho(v)\}$ the set of all the locations of a diagram A. A location represents an instantaneous configuration of the diagram, similarly to a state of an FTS. A *run* of a diagram is an infinite sequence of locations (v_0, s_0), (v_1, s_1), (v_2, s_2), ..., such that $s_0 \in \Theta(v_0)$, and $(s_i, s_{i+1}) \in \tau(v_i, v_{i+1})$ for all $i \geq 0$.

Regions. A *region* Φ is a set of locations. We denote by $\Phi(v)$ the set of states $\{s \in \Sigma \mid (v, s) \in \Phi\}$ that are part of Φ at vertex v. A region Φ is represented by the set of formulas $\{\widehat{\Phi}(v)\}_{v \in V}$, where for each $v \in V$ the formula $\widehat{\Phi}(v)$ over $\mathcal{V}$ defines the set $\Phi(v)$. We say that Φ is an *integral region* if $\Phi(v)$ is equal to either $\emptyset$ or $\rho(v)$ for every $v \in V$. We can specify an integral region Φ by the set of vertices $\{v \in V \mid \Phi(v) \neq \emptyset\}$.

Modes. A *mode* $\lambda : V^2 \mapsto 2^{\Sigma \times \Sigma}$ labels each edge $(u, v) \in V^2$ with a transition relation $\lambda(u, v) \subseteq \tau(u, v)$. For $u, v \in V$, $\lambda(u, v)$ is represented by a formula $\widehat{\lambda}(u, v)$ over $\mathcal{V}, \mathcal{V}'$. A mode represents a subset of the possible transitions between locations of the diagram. An *integral mode* is a mode λ such that $\lambda(u, v)$ is either $\emptyset$ or $\tau(u, v)$, for all $u, v \in V$. We can specify an integral mode λ by listing the set of edges $\{(u, v) \mid \lambda(u, v) \neq \emptyset\}$.

Fairness constraints. A *fairness constraint* (constraint, for short) is a triple (J, C, G), where J, C are regions s.t. $C \subseteq J$ and G is a mode. Constraints are used to specify the fairness properties of the diagram, and the *fairness set* $\mathcal{F}$ is a set of constraints.

A diagram must satisfy the *consecution* condition, which states that if a transition is taken from a location, it will lead to another location: formally, for all $u, v \in V$ and for all $s, t \in \Sigma$,

$$s \in \rho(u) \wedge (s, t) \in \tau(u, v) \rightarrow t \in \rho(v) .$$

In the following, we denote by ϕ' the formula obtained from a first-order logic formula ϕ by replacing each free $x \in \mathcal{V}$ with $x' \in \mathcal{V}'$. With this convention, the consecution condition holds iff the logical implication $\widehat{\rho}(u) \wedge \widehat{\tau}(u, v) \rightarrow \widehat{\rho}'(v)$ is valid for all $u, v \in V$.

The computations of a diagram are defined in terms of its accepting runs.

Definition 1. A run $\sigma : (v_0, s_0), (v_1, s_1), (v_2, s_2), \ldots$ of a diagram A is an *accepting run* if the following condition holds.

> *For each constraint $(J, C, G) \in \mathcal{F}$, if there is $n \geq 0$ such that $(v_i, s_i) \in J$ for all $i \geq n$ and $(v_i, s_i) \in C$ for infinitely many $i \geq 0$, then there are infinitely many $j \geq 0$ s.t. $(s_j, s_{j+1}) \in G(v_j, v_{j+1})$.*

If $\sigma : (v_0, s_0), (v_1, s_1), (v_2, s_2), \ldots$ is an accepting run of A, the sequence of states $s_0, s_1, s_2, \ldots$ is a *computation* of A. We denote by $Runs(A)$, $\mathcal{L}(A)$ the sets of accepting runs and computations of A, respectively. ∎

According to the above definition, the informal reading of a constraint (J, C, G) is that every accepting run that stays in J forever and visits C infinitely often must follow a transition in G infinitely often. The chosen names J, C and G reflect the notions of *Justice* set, *Compassion* set and *Gratify* action that describe fairness of transition systems in [8].

A fair transition system (FTS), defined as in [9], can be represented by a diagram having only one vertex.

Construction 2 (from FTS to diagram). Let $S = (\mathcal{V}, \Sigma, \theta, \mathcal{T}, \mathcal{J}, \mathcal{C})$ be an FTS, where $\mathcal{V}$ is the set of state variables, Σ is the state space, $\theta \subseteq \Sigma$ is the initial condition, $\mathcal{T} = \{\gamma_1, \ldots, \gamma_m\}$ is the set of transitions, and $\mathcal{J} \subseteq \mathcal{T}$, $\mathcal{C} \subseteq \mathcal{T}$ are the just and compassionate transitions, respectively. The FTS can be represented by a diagram $fd(S) = (\mathcal{V}, \Sigma, V, \rho, \tau, \Theta, \mathcal{F})$, where $\mathcal{V}$, Σ are as in the FTS, $V = \{v_0\}$, $\rho(v_0) = \Sigma$, $\Theta(v_0) = \theta$, and τ and $\mathcal{F}$ are defined as follows.

1. $\tau(v_0, v_0) = \{(s, s) \mid s \in \Sigma\} \cup \bigcup_{i=1}^{m} \{(s, t) \in \Sigma^2 \mid t \in \gamma_i(s)\}$.
2. For $1 \leq i \leq m$, let $E_i(v_0) = \mathrm{Dom}(\gamma_i)$, $G_i(v_0) = \{(s, t) \in \Sigma^2 \mid t \in \gamma_i(s)\}$. If $\gamma_i \in \mathcal{J}$ (resp. $\gamma_i \in \mathcal{C}$) we add (E_i, E_i, G_i) (resp. $(loc(A), E_i, G_i)$) to $\mathcal{F}$. ∎

We assume that an FTS S includes among its transitions the *idling* transition, that does not change the state. Given an FTS S, we will indicate with $\mathcal{L}(S)$ the computations admitted by S. Comparing the definitions of FTSs and diagrams, we have the following theorem.

Theorem 3. *For an FTS S, $\mathcal{L}(S) = \mathcal{L}(fd(S))$.*

3 Fairness Diagram Transformations

The temporal behavior of a diagram is studied by means of diagram transformations. These transformations preserve containment of behaviors, and they are reminiscent of the preorders of [3]. If a diagram A can be transformed into a diagram B by using one of the transformations, we write $A \Rightarrow B$. Since the transformations preserve containment of behaviors, $A \Rightarrow B$ implies $\mathcal{L}(A) \subseteq \mathcal{L}(B)$. We will denote by $\stackrel{*}{\Rightarrow}$ the reflexive transitive closure of $\Rightarrow$.

3.1 Simulation Transformations

Simulation transformations transform a diagram into a new one, such that the second diagram is capable of simulating the first one. These transformations modify the set of vertices of a diagram, rearranging the grouping of states in the vertices, and are used to study the safety properties of the diagram.

A *simulation relation* between two diagrams A_1 and A_2 is a function $V_1 \mapsto 2^{V_2}$ from the vertices of A_1 to those of A_2, which induces a simulation relation that maps each location (u, s) of A_1 into the subset $\bigcup_{v \in \mu(u)} (v, s)$ of locations of A_2. The following rule determines whether there is a simulation relation between two diagrams.

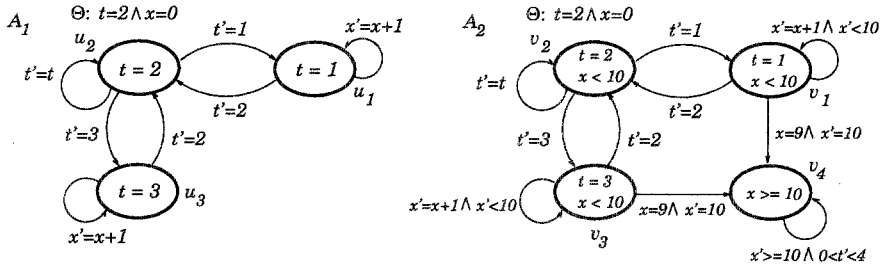


Fig. 1. Fairness diagram A_1 and A_2, related by the simulation relation arising from $\mu(u_1) = \{v_1, v_4\}$, $\mu(u_2) = \{v_2, v_4\}$, $\mu(u_3) = \{v_3, v_4\}$. Variables not mentioned in the transition relations are left unchanged.

Rule 4 (simulation). Let $A_1 = (\mathcal{V}, \Sigma, V_1, \rho_1, \tau_1, \Theta_1, \mathcal{F}_1)$, $A_2 = (\mathcal{V}, \Sigma, V_2, \rho_2, \tau_2, \Theta_2, \mathcal{F}_2)$ be two diagrams sharing the same variables and state space. We say that A_2 *simulates* A_1, written $A_1 \preceq A_2$, if there is a mapping $\mu : V_1 \mapsto 2^{V_2}$ such that the following logical assertions are valid.

1. For all $u \in V_1$, $\widehat{\Theta}_1(u) \rightarrow \bigvee_{v \in \mu(u)} \widehat{\Theta}_2(v)$.
2. For all $u, u' \in V_1$ and $v \in \mu(u)$, $\left(\widehat{\rho}_1(u) \wedge \widehat{\rho}_2(v) \wedge \widehat{\tau}_1(u, u')\right) \rightarrow \bigvee_{v' \in \mu(u')} \widehat{\tau}_2(v, v')$.
3. For each $(J_2, C_2, G_2) \in \mathcal{F}_2$ there is $(J_1, C_1, G_1) \in \mathcal{F}_1$ such that the following conditions are satisfied, for all $u \in V_1$ and $v \in \mu(u)$:
   (a) $\left(\widehat{J}_2(v) \wedge \widehat{\rho}_1(u)\right) \rightarrow \widehat{J}_1(u)$, and $\left(\widehat{C}_2(v) \wedge \widehat{\rho}_1(u)\right) \rightarrow \widehat{C}_1(u)$;
   (b) for all $u' \in V_1$, $\left(\widehat{\rho}_1(u) \wedge \widehat{\rho}_2(v) \wedge \widehat{G}_1(u, u')\right) \rightarrow \bigvee_{v' \in \mu(u')} \widehat{G}_2(v, v')$. ∎

Theorem 5. *If A_1, A_2 are two diagrams s.t. $A_1 \preceq A_2$, then $\mathcal{L}(A_1) \subseteq \mathcal{L}(A_2)$.*

Proof. Conditions 1 and 2 insure that for each run σ_1 of A_1 there is a related run σ_2 of A_2. Condition 3 insures additionally that if σ_1 is accepting, there is a related σ_2 of A_2 which is also accepting. The result then follows from the fact that the simulation relation is an identity with respect to the state space Σ. ∎

Transformation 6 (simulation transformation). Given two diagrams A_1, A_2, if $A_1 \preceq A_2$ we can transform A_1 into A_2. ∎

Example 7. Consider the diagrams A_1 and A_2 of Figure 1. With A_1 are associated the fairness constraints
$C_1^{(1)} = \left(\{u_1, u_2, u_3\}, \{u_1\}, \{(u_1, u_1)\}\right)$, $C_2^{(1)} = \left(\{u_1, u_2, u_3\}, \{u_3\}, \{(u_3, u_3)\}\right)$,
$C_3^{(1)} = \left(\{u_2\}, \{u_2\}, \{(u_2, u_1), (u_2, u_3)\}\right)$,
represented with the convention for integral regions and modes. With A_2 are associated the constraints

$$C_1^{(2)} = \left(\{v_1, v_2, v_3\}, \{v_1\}, \{(v_1, v_1), (v_1, v_4)\}\right),$$
$$C_2^{(2)} = \left(\{v_1, v_2, v_3\}, \{v_3\}, \{(v_3, v_3), (v_3, v_4)\}\right),$$
$$C_3^{(2)} = \left(\{v_2\}, \{v_2\}, \{(v_2, v_1), (v_2, v_3)\}\right).$$

Since the function μ of Figure 1 satisfies the conditions of Rule 4, A_1 can be transformed into A_2 using a simulation transformation. ∎

The proof of $A_1 \preceq A_2$ using Rule 4 fails if there is a constraint $(J_2, C_2, G_2) \in \mathcal{F}_2$ for which there is no constraint $(J_1, C_1, G_1) \in \mathcal{F}_1$ that satisfies conditions (3a) and (3b). In this case, to construct the simulation relation we must first add a suitable constraint to A_1. This can be done using fairness transformations.

3.2 Fairness Transformations

Fairness transformations analyze the structure of a diagram to derive new constraints that can be added to the set $\mathcal{F}$ without restricting the set of accepting runs of the diagram. These constraints are said to be *compatible*, and they represent progress properties of the diagram.

Definition 8. Given a diagram $A = (\mathcal{V}, \Sigma, V, \rho, \tau, \Theta, \mathcal{F})$ and a constraint (J, C, G), let $A' = (\mathcal{V}, \Sigma, V, \rho, \tau, \Theta, \mathcal{F} \cup \{(J, C, G)\})$ be the diagram obtained from A by adding (J, C, G) to the set $\mathcal{F}$. We say that the constraint (J, C, G) is *compatible* with A if $Runs(A) = Runs(A')$. ∎

Transformation 9 (fairness transformation). If diagram A' is obtained from diagram A by adding a compatible constraint, we can transform A into A'. ∎

To prove that a constraint is compatible with a diagram, we present verification rules. These rules are related to the rules for response and reactivity properties presented in [7]. The structure of the constraints enables two simplifications. First, separate rules for response and reactivity properties are not needed, since constraints can represent both types of properties. Second, it is possible to decompose the rules of [7] into simpler ones while retaining completeness.

We present three rules for proving the compatibility of constraints. The first rule shows the compatibility of a constraint independently from other constraints. The second rule uses one or two constraints in $\mathcal{F}$ to show that a third one is compatible, and can be thought as a rule to concatenate constraints. The third rule can be used to show that the union of constraints in $\mathcal{F}$ is compatible. Before presenting the rules, we introduce the notion of ranking functions.

Definition 10. A *well-founded domain* is a set D together with an order relation $>$, such that there is no infinite descending chain $d_0 > d_1 > d_2 > \ldots$ of elements of D. A *ranking function* $\delta : loc(A) \mapsto D$ for a diagram A is a function mapping pairs $(v, s) \in loc(A)$ into elements of a well-founded domain D. A ranking function δ is represented by the family of terms $\{\widehat{\delta}(u)\}_{u \in V}$ on $\mathcal{V}$, where term $\widehat{\delta}(u)$ denotes a function $\rho(u) \mapsto D$. ∎

Rule 11 (single constraint). A constraint (J, C, G) is compatible with the given diagram if there is a ranking function δ such that the assertions

$$\widehat{J}(u) \wedge \widehat{\tau}(u, v) \rightarrow \widehat{G}(u, v) \vee \widehat{\delta}(u) \geq \widehat{\delta}'(v) \vee \neg \widehat{J}'(v)$$

$$\widehat{C}(u) \wedge \widehat{\tau}(u, v) \rightarrow \widehat{G}(u, v) \vee \widehat{\delta}(u) > \widehat{\delta}'(v) \vee \neg \widehat{J}'(v)$$

are valid for all $u, v \in V$. ∎

Justification. Assume that the conditions of the rule are satisfied and assume, towards the contradiction, that there is an accepting run σ that beyond position $k \geq 0$ stays forever in J and visits C infinitely often, without taking any transition in G. Beyond k, the value of δ along σ will not increase, and will decrease each time σ is in C. Since C is visited infinitely often, this contradicts the fact that the domain of δ is well-founded. ∎

Rule 12 (concatenation of constraints). A constraint (J, C, G) is compatible with the given diagram if there is a constraint $(J_0, C_0, G_0) \in \mathcal{F}$ with $\widehat{J}(u) \to \widehat{J}_0(u)$ for all $u \in V$ and a ranking function δ such that the following logical assertions are valid.

1. For all $u, v \in V$,

$$\begin{aligned} \widehat{J}(u) \wedge \widehat{\tau}(u,v) &\to \widehat{G}(u,v) \vee \neg\widehat{J}'(v) \vee \widehat{\delta}(u) \geq \widehat{\delta}'(v) \\ \widehat{J}(u) \wedge \widehat{G}_0(u,v) &\to \widehat{G}(u,v) \vee \neg\widehat{J}'(v) \vee \widehat{\delta}(u) > \widehat{\delta}'(v) \,. \end{aligned}$$

2. Either $\widehat{C}(u) \to \widehat{C}_0(u)$ for all $u \in V$, or there is $(J_1, C_1, G_1) \in \mathcal{F}$ such that for all $u, v \in V$, $\widehat{J}(u) \to \big[\widehat{J}_1(u) \vee \widehat{C}_0(u)\big]$, $\widehat{C}(u) \to \big[\widehat{C}_1(u) \vee \widehat{C}_0(u)\big]$, and

$$\widehat{J}(u) \wedge \widehat{G}_1(u,v) \to \widehat{G}(u,v) \vee \widehat{C}'_0(v) \vee \neg\widehat{J}'(v) \,. \quad ∎$$

Justification. Assume that the conditions of the rule are satisfied and assume, towards the contradiction, that beyond a certain position $k \geq 0$ an accepting run σ stays forever in J and visits C infinitely often without taking transitions in G. From the first condition of the rule, beyond position k the value of δ will not increase, and it will decrease whenever a transition in G_0 is taken. Thus, if we can prove that infinitely many transitions in G_0 are taken we reach the desired contradiction.

If $C \subseteq C_0$, this follows from $J \subseteq J_0$ and $(J_0, C_0, G_0) \in \mathcal{F}$. If $C \not\subseteq C_0$, the region $C - C_0$ is non-empty, and there are two cases. If σ visits infinitely often C_0, the result follows as before. If σ beyond position $j \geq 0$ visits infinitely often $C - C_0$ without entering C_0, then σ after j will be confined to J_1 and will visit C_1 infinitely often, and the result follows from $(J_1, C_1, G_1) \in \mathcal{F}$ and from the condition on G_1 in the rule. ∎

Rule 13 (union of constraints). Given n constraints $(J_1, C_1, G_1), \ldots, (J_n, C_n, G_n) \in \mathcal{F}$ and a region J_0, the constraint (J, C, G) defined by

$$J = J_0 \cup \bigcup_{i=1}^{n} J_i \qquad C = \bigcup_{i=1}^{n} C_i \qquad \forall u, v \in V : \; G(u,v) = \bigcup_{i=1}^{n} G_i(u,v)$$

is compatible with the given diagram if there is a ranking function δ such that the following assertions are valid, for $1 \leq i \leq n$:

$$\begin{aligned} \widehat{J}_i(u) \wedge \widehat{\tau}(u,v) \wedge \widehat{J}'_i(v) &\to \widehat{G}(u,v) \vee \widehat{\delta}(u) \geq \widehat{\delta}'(v) \\ \widehat{J}_i(u) \wedge \widehat{\tau}(u,v) \wedge \neg\widehat{J}'_i(v) &\to \widehat{G}(u,v) \vee \widehat{\delta}(u) > \widehat{\delta}'(v) \vee \neg\widehat{J}'(v) \\ \widehat{J}_0(u) \wedge \widehat{\tau}(u,v) &\to \widehat{G}(u,v) \vee \widehat{\delta}(u) \geq \widehat{\delta}'(v) \vee \neg\widehat{J}'(v) \,. \end{aligned} \quad ∎$$

Justification. Assume that the conditions of the rule are satisfied and, towards the contradiction, that beyond a certain position $k \geq 0$ an accepting run σ stays forever in J visiting C infinitely often without taking transitions in G. As beyond k the value of δ never increases, and decreases every time σ leaves a region J_i, $1 \leq i \leq n$, σ can leave only finitely many times every J_i. Since $C_i \subseteq J_i$ for all $1 \leq i \leq n$ and σ visits infinitely often $\bigcup_{i=1}^{n} C_i$, there must be $m \in [1..n]$ s.t. eventually σ is confined to J_m and visits C_m infinitely often. The contradiction then follows from the fact that $(J_m, C_m, G_m) \in \mathcal{F}$. ∎

From the justifications of the rules, we have the following theorem.

Theorem 14 (soundness). *If the conditions of each of the rules 11, 12 or 13 are satisfied, the constraint (J, C, G) is compatible with the diagram under consideration.*

Example 15. Consider diagram A_2 of Example 7. Using Rule 13, it is possible to add to it the compatible constraint

$$C_4^{(2)} = \big(\{v_1, v_2, v_3\}, \{v_1, v_3\}, \{(v_1, v_1), (v_1, v_4), (v_3, v_3), (v_3, v_4)\}\big)$$

resulting from the union of $C_1^{(2)}$ and $C_2^{(2)}$. By Rule 12, with $C_4^{(2)}$ for (J_0, C_0, G_0) $C_3^{(2)}$ for (J_1, C_1, G_1), and ranking function $\widehat{\delta}(v_1) = \widehat{\delta}(v_2) = \widehat{\delta}(v_3) = 10 - x$, $\widehat{\delta}(v_4) = 0$, it is possible to add the constraint

$$C_5^{(2)} = \big(\{v_1, v_2, v_3\}, \{v_1, v_2, v_3\}, \{(v_1, v_4), (v_3, v_4)\}\big).$$

Let A_2' be the diagram obtained by adding $C_4^{(2)}$ and $C_5^{(2)}$ to A_2. Intuitively, $C_5^{(2)}$ represent the temporal progress property $\Diamond(x \geq 10)$, satisfied by A_2'. ∎

3.3 Completeness and Complexity Results

The transformations introduced in the previous sections are complete for proving the compatibility of fairness constraints, as the following theorem states.

Theorem 16 (completeness for constraints). *Given a diagram A and a constraint (J, C, G), if (J, C, G) is compatible there is a sequence of transformations $A \stackrel{*}{\Rightarrow} B$, where the diagram B is obtained from A by adding (J, C, G) to $\mathcal{F}$.*

The proof of this theorem is rather lengthy, and follows the general line of the completeness proof for reactivity and response rules presented in [7]. The complete proof is given in [2]. To state the completeness theorem for transition systems we need an additional definition.

Definition 17. A diagram $A = (\mathcal{V}, \Sigma, V, \rho, \tau, \Theta, \mathcal{F})$ is *state-deterministic* if for all $u, v, w \in V$ with $v \neq w$ it is $\Theta(v) \cap \Theta(w) = \emptyset$ and $\tau(u, v) \cap \tau(u, w) = \emptyset$. ∎

Theorem 18 (completeness for transition systems). *Let $A = fd(S)$ for an FTS S, and B be a state-deterministic diagram. If $\mathcal{L}(S) \subseteq \mathcal{L}(B)$, there is a chain of transformations $A \stackrel{*}{\Rightarrow} B$.*

The proof of this theorem relies on Theorem 16 for the fairness part, and follows otherwise from the existence of chains of simulation relations between diagrams derived from FTSs and deterministic diagrams.

Complexity of transformations. To establish a simulation transformation $A_1 \Rightarrow A_2$ using Rule 4, the number of logical formulas to be considered is $O(|V_1|^2 \cdot |V_2|)$, where V_1, V_2 are the set of vertices of A_1, A_2 respectively.

To add a constraint to a diagram A, the number of logical formulas to be considered is $O(|V|^2)$ using Rules 11, 12, and $O(n|V|^2)$ using Rule 13, where n is the number of constraints whose union is taken. These bounds, however, refer to the worst-case complexity. If these transformations are used to do a local analysis of a diagram that involves only few vertices, the number of non-trivial logical formulas to be proved does not necessarily increase when the size of the diagram increases.

4 Proving Linear Temporal Logic Properties

Let TL_s be the class of temporal formulas obtained by combining first-order logic formulas using propositional connectives, the future temporal operators $\bigcirc$ (next), $\Box$ (always), $\Diamond$ (eventually), $\mathcal{U}$ (until), and of the corresponding past ones $\ominus$, $\boxminus$, $\diamondminus$ and $\mathcal{S}$ [8]. Note that in a formula $\phi \in TL_s$, no temporal operator occurs in the scope of a quantifier.

Given an FTS S and $\phi \in TL_s$, in this section we present two methods for proving that all computations of S satisfy ϕ, written $S \models \phi$. According to the first method, we construct from ϕ a deterministic Streett automaton M_ϕ, we translate it into a diagram $fd(M_\phi)$, and we show that $fd(S) \stackrel{*}{\Rightarrow} fd(M_\phi)$. According to the second method, we construct a nondeterministic Streett automaton $N_{\neg\phi}$ representing $\neg\phi$ and we show that $fd(S) \stackrel{*}{\Rightarrow} B$, where B is a diagram s.t. $\mathcal{L}(B) \cap \mathcal{L}(N_{\neg\phi}) = \emptyset$ can be shown using algorithmic methods. The Streett automata used in the above methods are a first-order version of the classical ones [11].

Definition 19 (Streett automaton). A (first-order) *Streett automaton* A consists of the components $(\mathcal{V}, \Sigma, (V, E), \rho, Q, \mathcal{A})$, where $\mathcal{V}$, Σ, ρ are as in a diagram; (V, E) is a directed graph with set of vertices V and set of edges $E \subseteq V^2$; $Q \subseteq V$ is the set of *initial vertices*, and $\mathcal{A}$, called the *acceptance list*, is a set of pairs $(P, R) : P, R \subseteq V$.

A run σ of A is an infinite sequence of locations $(v_0, s_0), (v_1, s_1), (v_2, s_2), \ldots$ such that $v_0 \in Q$ and $s_i \in \rho(v_i)$, $(v_i, v_{i+1}) \in E$ for all $i \geq 0$. Run σ is an accepting run of A if the following condition holds:

> *For each pair* $(P, R) \in \mathcal{A}$, *either* $v_i \in R$ *for infinitely many* $i \in \mathbb{N}$, *or there is* $k \in \mathbb{N}$ *such that* $v_i \in P$ *for all* $i \geq k$.

The set of accepting runs (resp. computations) of a Streett automaton A is denoted by $Runs(A)$ (resp. $\mathcal{L}(A)$). ∎

Given a Streett automaton M, we can construct a fairness diagram $fd(M)$ such that $\mathcal{L}(fd(M)) = \mathcal{L}(M)$.

Construction 20 (from Streett Automaton to diagram). Given a Streett automaton $M = (\mathcal{V}, \Sigma, (V, E), \rho, Q, \mathcal{A})$ we can construct a fairness diagram $fd(M) = (\mathcal{V}, \Sigma, V, \rho, \tau, \Theta, \mathcal{F})$ as follows.

1. For all $u, v \in V$, $\tau(u,v) = \rho(u) \times \rho(v)$ if $(u,v) \in E$, and $\tau(u,v) = \emptyset$ otherwise.
2. $\Theta = \{(u,s) \mid u \in Q \wedge s \in \rho(u)\}$.
3. $\mathcal{F}$ consists of all constraints (J,C,G) such that there is $(P,R) \in \mathcal{A}$ for which $J = C = \{(u,s) \mid u \in V - P \wedge s \in \rho(u)\}$, and for all $u,v \in V$, $G(u,v) = \tau(u,v)$ if $v \in R$, and $G(u,v) = \emptyset$ otherwise. ∎

4.1 The Transformation Method

For a temporal logic formula $\phi \in TL_s$, let $\mathcal{L}(\phi)$ be the set of computations that satisfy ϕ. Let M_ϕ be a deterministic Streett automaton such that $\mathcal{L}(M_\phi) = \mathcal{L}(\phi)$. This automaton can be constructed from ϕ with the methods explained in [6, 11, 12]. We can thus formulate the first proof strategy.

Proof Strategy 1. To prove $S \models \phi$ for FTS S and a formula $\phi \in TL_s$, construct a chain of transformations $fd(S) \stackrel{*}{\Rightarrow} fd(M_\phi)$. ∎

Theorem 21. *Proof Strategy 1 is sound and complete for proving $S \models \phi$ for an FTS S and $\phi \in TL_s$.*

Proof. The soundness result follows from $\mathcal{L}(S) = \mathcal{L}(fd(S)) \subseteq \mathcal{L}(fd(M_\phi)) = \mathcal{L}(\phi)$. Since M_ϕ is deterministic, $fd(M_\phi)$ is state-deterministic, and the completeness result follows from Theorem 18. ∎

The drawback of Strategy 1 is that, in the worst case, the number of vertices of M_ϕ is doubly exponential in the size $|\phi|$ of the specification ϕ.

4.2 The Product Method

Given a temporal formula $\phi \in TL_s$, it is possible to construct a nondeterministic Streett automaton $N_{\neg\phi}$ s.t. $\mathcal{L}(N_{\neg\phi}) = \mathcal{L}(\neg\phi)$. The automaton $N_{\neg\phi}$ has number of vertices that is singly exponential in $|\phi|$. To prove $\mathcal{L}(S) \subseteq \mathcal{L}(\phi)$ for an FTS S, it suffices to construct a chain of transformations $fd(S) \stackrel{*}{\Rightarrow} B$ ending with a diagram B s.t. $\mathcal{L}(B) \cap \mathcal{L}(fd(N_{\neg\phi})) = \emptyset$ can be shown with algorithmic methods. The emptiness problem of $\mathcal{L}(A) \cap \mathcal{L}(B)$ for diagrams A, B is undecidable: in the following, we give a computable sufficient condition for $\mathcal{L}(A) \cap \mathcal{L}(B) = \emptyset$.

Let FL be the first-order logic language with interpreted function and predicate symbols in which the assertions labeling the diagrams and the first-order part of the temporal specifications are written. Assume that we have a proof procedure $\vdash$ for FL that always terminates, and that is able to prove a subset of the valid sentences that includes all substitution instances of propositional tautologies. Given $\phi \in FL$, if $\vdash$ terminates with a proof of ϕ we write $\vdash \phi$; otherwise we write $\nvdash \phi$. To obtain a sufficient condition for the emptiness of the intersection of diagram languages, we construct the *graph product* of the diagrams.

Construction 22 (graph product of diagrams). Given two diagrams $A_1 = (\mathcal{V}, \Sigma, V_1, \rho_1, \tau_1, \Theta_1, \mathcal{F}_1)$, $A_2 = (\mathcal{V}, \Sigma, V_2, \rho_2, \tau_2, \Theta_2, \mathcal{F}_2)$ their *graph product* $A_1 \otimes A_2 = ((V,E), V_{in}, \mathcal{G})$ consists of a graph (V,E), of a subset $V_{in} \subseteq V$ of initial vertices, and of a set $\mathcal{G}$ of triples of the form $(P,Q,R) : P,Q \subseteq V, R \subseteq V^2$. These components are defined in terms of the components of A_1 and A_2 as follows.

1. $V = \{(v_1, v_2) \in V_1 \times V_2 \mid \not\vdash \neg(\widehat{\rho}(v_1) \wedge \widehat{\rho}(v_2))\}$.
2. $V_{in} = \{(v_1, v_2) \in V \mid \not\vdash \neg(\widehat{\Theta}_1(v_1) \wedge \widehat{\Theta}_2(v_2))\}$.
3. $E = \left\{\left((u_1, u_2), (v_1, v_2)\right) \in V^2 \mid \not\vdash \neg(\widehat{\tau}_1(u_1, v_1) \wedge \widehat{\tau}_2(u_2, v_2))\right\}$.
4. For $i = 1, 2$, to each constraint $(J, C, G) \in \mathcal{F}_i$ corresponds the triple $(\kappa(J, i), \kappa(C, i), \pi(G, i))$, where κ and π are defined by:

$$\kappa(\Phi, i) = \left\{(u_1, u_2) \in V \mid \vdash (\widehat{\Phi}(u_i) \leftrightarrow \widehat{\rho}_i(u_i))\right\}$$

$$\pi(\lambda, i) = \left\{\left((u_1, u_2), (v_1, v_2)\right) \in E \mid \not\vdash \neg\widehat{\lambda}(u_i, v_i)\right\},$$

for all regions Φ and modes λ of A_i. The set $\mathcal{G}$ is then

$$\mathcal{G} = \left\{\left(\kappa(J, i), \kappa(C, i), \pi(G, i)\right) \mid 1 \leq i \leq 2 \wedge (J, C, G) \in \mathcal{F}_i\right\}. \qquad \blacksquare$$

Theorem 23. *Given two diagrams A, B, let $A \otimes B = ((V, E), V_{in}, \mathcal{G})$. A sufficient condition for $\mathcal{L}(A) \cap \mathcal{L}(B) = \emptyset$ is that for each strongly connected component $U \subseteq V$ of (V, E) reachable from V_{in} there is $(P, Q, R) \in \mathcal{G}$ s.t. $U \subseteq P$, $U \cap Q \neq \emptyset$, and $(u, v) \notin R$ for all $u, v \in U$.*

Proof Strategy 2. To prove $S \models \phi$ for an FTS S and a formula $\phi \in TL_s$, construct a chain of transformations $fd(S) \stackrel{*}{\Rightarrow} B$ to a diagram B s.t. $\mathcal{L}(B) \cap \mathcal{L}(fd(N_{\neg\phi})) = \emptyset$ can be proved using the condition of Theorem 23. $\blacksquare$

Example 24. If $\phi : \Diamond(x \geq 10)$, the Streett automaton $N_{\neg\phi}$ will consist of only one vertex, labeled with $x < 10$. Using Theorem 23, it is easy to check that the graph product of $N_{\neg\phi}$ and diagram A_2' of Example 15 has empty language. $\blacksquare$

Theorem 25. *Proof Strategy 2 is sound and complete for proving $S \models \phi$ for an FTS S and $\phi \in TL_s$.*

Proof. The soundness part is a consequence of the previous definitions. Let $fd(M_\phi)$ be the deterministic diagram corresponding to ϕ, as in the previous strategy. If $S \models \phi$, there is a chain of transformations $fd(S) \stackrel{*}{\Rightarrow} fd(M_\phi)$, and from the construction of $fd(M_\phi)$ it can be seen that the graph product $fd(M_\phi) \otimes fd(N_{\neg\phi})$ satisfies the condition for emptiness of Theorem 23. $\blacksquare$

Note that we still need a complete deductive system for the interpreted first-order language FL to perform the diagram transformations, in order to retain the completeness results expressed by Theorems 16, 18 and 25.

From the above proof, we see that there is a final diagram B for strategy 2 with number of vertices at most doubly exponential in $|\phi|$. In fact, if the state space Σ of the FTS if finite, it is possible to show that there is a diagram B with number of vertices bound by $|\Sigma|$, so that the number of vertices of $B \otimes fd(N_{\neg\phi})$ is at most singly exponential in $|\phi|$, similarly to the case of finite-state model checking.

For systems with an infinite number of reachable states, the worst-case complexity of strategy 2 is not better than the one of Proof Strategy 1. However, in most cases an FTS S will satisfy a specification ϕ by exhibiting a set of computations $\mathcal{L}(S)$ significantly smaller than $\mathcal{L}(\phi)$. Thus, the diagram B of Proof Strategy 2 in general is smaller than $fd(M_\phi)$, so that Proof Strategy 2 is often more convenient than Proof Strategy 1.

5 Conclusions

Fairness diagrams provide a methodology for the proof of temporal specifications of systems. They can also be used as a graphical specification language. Since both vertices and edges are labeled with first-order assertions, fairness diagrams have the advantage over traditional temporal logic (and similarly to TLA [5]) of providing a simpler representation for specifications that involve conditions on both system states and actions.

While we have given completeness results on the existence of chains of transformations, we have not discussed how to obtain guidance for their construction. When the specification has a simple temporal form, the graphical representation of the diagrams often captures enough intuition about the system to guide the construction of the chain of transformations. We intend to address the question of guidance and heuristics for chain constructions in future work.

Acknowledgements. We would like to thank Anca Browne, Henny Sipma and Tomás Uribe for helpful comments and suggestions.

References

1. I.A. Browne, Z. Manna, and H.B. Sipma. Generalized verification diagrams. In *Found. of Software Technology and Theoretical Comp. Sci.*, volume 1026 of *Lect. Notes in Comp. Sci.*, pages 484–498. Springer-Verlag, 1995.
2. L. de Alfaro and Z. Manna. Temporal verification by diagram transformations. Technical report, Stanford University, 1996.
3. O. Grumberg and D.E. Long. Model checking and modular verification. *ACM Trans. Prog. Lang. Sys.*, 16(3):843–871, May 1994.
4. Y. Kesten, Z. Manna, and A. Pnueli. Temporal verification of simulation and refinement. In *Proc. of the REX Workshop "A Decade of Concurrency"*, volume 803 of *Lect. Notes in Comp. Sci.*, pages 273–346. Springer-Verlag, 1994.
5. L. Lamport. The temporal logic of actions. *ACM Trans. Prog. Lang. Sys.*, 16(3):872–923, May 1994.
6. O. Lichtenstein and A. Pnueli. Checking that finite state concurrent programs satisfy their linear specifications. In *Proc. 12th ACM Symp. Princ. of Prog. Lang.*, pages 97–107, 1985.
7. Z. Manna and A. Pnueli. Completing the temporal picture. *Theor. Comp. Sci.*, 83(1):97–130, 1991.
8. Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification.* Springer-Verlag, New York, 1991.
9. Z. Manna and A. Pnueli. Models for reactivity. *Acta Informatica*, 30:609–678, 1993.
10. Z. Manna and A. Pnueli. Temporal verification diagrams. In *TACS 94*, Lect. Notes in Comp. Sci. Springer-Verlag, 1994.
11. S. Safra. On the complexity of ω-automata. In *Proc. 29th IEEE Symp. Found. of Comp. Sci.*, 1988.
12. S. Safra and M.Y. Vardi. On ω-automata and temporal logic. In *Proc. 21th ACM Symp. Theory of Comp.*, pages 127–137, 1989.

Protocol Verification by Aggregation of Distributed Transactions *

Seungjoon Park and David L. Dill

Computer Systems Laboratory
Stanford University
park@turnip.stanford.edu dill@cs.stanford.edu

Abstract. We present a new approach for using a theorem-prover to verify the correctness of protocols and distributed algorithms. The method compares a state graph of the implementation with a specification which is a state graph representing the desired abstract behavior. The steps in the specification correspond to atomic transactions, which are not atomic in the implementation.
The method relies on an *aggregation function*, which is a type of an abstraction function that aggregates the steps of each transaction in the implementation into a single atomic transaction in the specification. The key idea in defining the aggregation function is that it must *complete* atomic transactions which have committed but are not finished.
We illustrate the method on a simple but nontrivial example. We have successfully used it for other examples, including the cache coherence protocol for the Stanford FLASH multiprocessor.

1 Introduction

Protocols for distributed systems often simulate atomic transactions in environments where atomic implementations are impossible. We believe that this observation can be exploited to make formal verification of protocols and distributed algorithms using a theorem-prover much easier than it would otherwise be. Indeed, we have used the techniques described below to verify safety properties of two significant examples: the cache coherence protocol for the FLASH multiprocessor (currently being designed at Stanford), and for a majority consensus algorithm for multiple copy databases.

The method proves that an implementation state graph is consistent with a specification state graph that captures the abstract behavior of the protocol, in which each transaction appears to be atomic. The method involves constructing an abstraction function which maps the distributed steps of each transaction to the atomic transaction in the specification. We call this *aggregation*, because the abstraction function reassembles the distributed transactions into atomic transactions.

This method addresses the primary difficulty with using theorem proving for verification of real systems, which is the amount of human effort required to complete a proof, by making it easier to create appropriate abstraction functions. Although our

* This research was supported by the Advanced Research Projects Agency through NASA grant NAG-2-891.

work is based on using the PVS theorem-prover from SRI International [ORSvH95], the method is useful with other theorem-provers, or manual proofs.

Although finite-state methods (e.g. [McM93, DDHY92]) can solve many of the same problems with even less effort, they are basically limited to finite-state protocols. Finite-state methods have been applied to non-finite-state systems in various ways, but these techniques typically require substantial pencil-and-paper reasoning to justify. Theorem-provers make sure that such manual reasoning is indeed correct, in addition to making available the full power of formal mathematics for proof, so they can routinely deal with problems that cannot yet be solved by any finite-state methods.

For our method to be applicable, the description must have an identifiable set of transactions. Each transaction must have a unique commit point, at which a state change first becomes visible to the specification. The most important idea in the method is that the aggregation function can be defined by completing transactions that have committed but not yet completed. In general, the steps to complete separate transactions are independent, which simplifies the definition of this function. In our experience, this guideline greatly simplifies the definition of an appropriate aggregation function.

The same idea of aggregating transactions can be applied to reverse-engineer a specification where none exists, because the specification with atomic transactions is usually consistent with the intuition of the system designer.

If the extracted specification is not considered as a complete specification, or is not obviously correct, it can instead be regarded as a model of the protocol having an enormously reduced number of states. The amount of reduction is much more than other reduction methods used in model checking, such as partial order reduction, mainly because the reduced system is based on the only state variables relevant to the specification, without variables such as local states and communications buffers.

The method described here has been successfully applied to the verification of several protocols for distributed systems including the FLASH cache coherence protocol [KOH+94, Hei93]. The FLASH cache coherence protocol, consisting of more than a hundred different kinds of implement steps, can be reduced to a specification with six kinds of atomic transactions [PD96]. It is then simple to prove interesting properties of the (much smaller) specification, such as the consistency of data at the user level.

Related work

The idea of using abstraction functions to relate implementation and specification state graphs is very widely used, especially when manual or automatic theorem-proving is used [Lyn88, LS84] (indeed, whole volumes have been written on the subject [dBdRR90]). The idea has also been used with finite-state techniques [Kur94, DHWT91].

Ladkin, et al. [LLOR96] have used a refinement mapping [AL91] to verify a simple caching algorithm. Their refinement mapping hides some implementation variables, which may have the effect of aggregating steps if the specification-visible variables do not change. Our aggregation functions generalize on this idea by merging steps even when specification-visible variables change more than once.

A more limited notion of aggregation is found in [Lam82, Lam83], where a state function undoes or completes an unfinished process. The method only aggregates se-

quential steps within a local process, while our method aggregates steps across distributed components. The idea of an aggregated transaction has been used to prove a protocol for data base systems [PKP91], where aggregation is obtained in a local process by showing the commutativity of actions from simple syntactic analysis.

Cohen used an idea similar to aggregation to prove global progress properties by combining progress properties of local processes [Coh93]. The idea of how to construct our aggregation function was inspired by a method of Burch and Dill for defining abstraction functions when verifying microprocessors [BD94].

In the next section, the basic verification procedure is presented. To illustrate it, we use a distributed list protocol which is a fragment of a distributed cache coherence protocol. In section 3, the protocol is described and we explain how to construct an aggregation function and prove that it has the necessary properties.

2 The verification method

The verification method begins with a description in higher-order logic of the state graph of the implementation of a distributed computation, and a logical description of the state graph of the specification. The implementation description contains a set of *state variables*, which are partitioned into *specification variables* and *implementation variables*. The set Q of *states* of the implementation is the set of assignments of values to state variables. The description of the implementation also includes a logical formula defining the relation between a state and its possible successors. The relation is represented by a set of functions, $\mathcal{F} : 2^{Q \to Q}$, each of which maps a given implementation state to its next state. The implementation is nondeterministic if this set has more than one function.

The description of the specification state graph is similar. A specification state is an assignment of values to the specification variables of the implementation (implementation variables do not appear in the specification). Also, every state in the specification has a transition to itself. We call these *idle* transitions. The idle transitions are necessary for following implementation steps that do not change specification variables. We adopt the convention that components of the specification are primed, so the set of states of the specification is Q', the set of functions is $\mathcal{F}'$, etc.

The verification method is based on the usual notion of an abstraction function. The function, which we call abs, maps implementation states to specification states and must satisfy a commutativity property

$$\forall q \in Q \quad \forall N \in \mathcal{F} \quad \exists N' \in \mathcal{F}' : abs(N(q)) = N'(abs(q)). \tag{1}$$

The most interesting part of the method is how the aggregation idea is used to define this function.

The method relies on the notion that there is a set of *transactions* which the computation is supposed to implement, which are atomic at the specification level (meaning that a transaction occurs during a single state transition in the specification), but non-atomic at the implementation level. Indeed, the transactions in the implementation may involve many steps that are executed in several different components of the implementation. Formally, the transactions in the specification are the specification transition functions.

The method requires that each transaction in the implementation have an identifiable *commit point*. Intuitively, when tracing through the steps of a transaction, the commit point is the implementation step that first causes a change in the specification variables. Implementation states that occur before the transaction or during the transaction but before the commit point are called *pre-commit states* for that transaction. The transaction is *complete* when the last specification variable change occurs as part of the transaction. The states after the commit point but before the completion of the transaction are called *post-commit states* for the transaction. A state where every committed transaction has completed is called a *clean* state.

Formally, all of the above concepts can be derived once the pre-commit states are known for each transaction. The post-commit states for the transaction are the states that are not pre-commit; the commit point for an transaction is the transition from a pre-commit state to a post-commit state for that transaction; and the completion point is the transition from a post-commit state to a pre-commit state. A state is clean if it is a pre-commit state for *every* transaction.

An aggregation function consists of two parts: a *completion function* which changes the state as though the transaction had completed, and a *projection* which hides the implementation variables, leaving only the specification variables.

Once a purported aggregation function has been defined, the user must prove that it meets the commutativity requirement (1). The proof consists of a sequence of standard steps, many of which are or could be automated[2]. The initial $\forall q$ and $\forall N$ can be eliminated automatically by *Skolemization*, which is substituting a new symbolic constant for q throughout (when we Skolemize in this presentation, we will not change the name of the quantified variable). This yields a subgoal of the form

$$(N \in \mathcal{F}) \Rightarrow \exists N' \in \mathcal{F}' : abs(N(q)) = N'(abs(q)). \tag{2}$$

The set of implementation steps $\mathcal{F}$ will often be defined with a logical formula of the general form $\exists \boldsymbol{p} : N = N_1(\boldsymbol{p}) \vee N = N_2(\boldsymbol{p}) \vee \ldots$, where $\boldsymbol{p}$ is a tuple of parameters (perhaps ranging over an unknown number of components), and each N_j is a different kind of implementation step. Since the $\exists \boldsymbol{p}$ is in the antecedent of an implication, it can be Skolemized automatically, and the resulting disjunction can be proved by proving a collection of subgoals

$$(N = N_j(\boldsymbol{p})) \Rightarrow \exists N' \in \mathcal{F}' : abs(N(q)) = N'(abs(q)). \tag{3}$$

The existential quantifier $\exists N'$ can be eliminated by the user by manually substituting the definition of the appropriate function for N'. Given j and $\boldsymbol{p}$, the user must supply proper instantiation j' and $\boldsymbol{p}'$ such that the resulting subgoals

$$abs(N_j(\boldsymbol{p})(q)) = N'_{j'}(\boldsymbol{p}')(abs(q)) \tag{4}$$

are provable.

The number of subgoals is equal to the number of transition functions in the implementation. In most cases, the required specification step $N'_{j'}(\boldsymbol{p}')$ is the idle step; indeed,

[2] We base this comment on our use of the PVS theorem prover, but we believe the same basic method would be used with others.

the only non-idle step is that corresponds to the commit step in the implementation. We have no global strategy for proving these theorems, although most are very simple.

The above discussion omits an important point, which is that not all states are worthy of consideration. Theorem (1) will generally not hold for some absurd states that cannot actually occur during a computation. Hence, it is usually necessary to provide an *invariant* predicate, which characterizes a superset of all the reachable states. If the invariant is Inv, Theorem (1) can then be weakened to

$$\forall q \in Q \quad \forall N \in \mathcal{F} \quad \exists N' \in \mathcal{F}' : Inv(q) \Rightarrow abs(N(q)) = N'(abs(q)). \qquad (5)$$

In other words, abs only needs to commute when q satisfies the Inv.

Use of an invariant incurs some additional proof obligations. First, we must prove that the initial states of the protocol satisfy Inv, and second, that the implementation transition functions all preserve Inv.

3 The Distributed List Protocol

We illustrate the concepts of the previous section on a small but somewhat nontrivial example, which we call the "distributed list protocol." The protocol is an abstraction of part of a multiprocessor cache coherence protocol, which maintains a singly-linked list of processors which share a cache line.

The finite-state techniques we have applied do not scale especially well for this protocol. We have tried explicit state methods (specifically our Murφ verifier) with techniques such as symmetry reduction, reversible rule reduction [ID96], and special verification methods for parameterized families of protocols, as well as BDD-based techniques. None of these methods has allowed us to verify systems with more than about 5 list cells, because we do not have a good way of compressing or abstracting states containing linked lists. However, using the method described here, we have verified the protocol for arbitrary or even infinite numbers of list cells.

3.1 The transactions of the protocol

The protocol maintains a circular, singly-linked list of list cell processes, called *cells*. There is a special process called the *head cell* which is always in the list. Cells not in the list may request to be added to the list, and cells in the list may request to be removed. The cells communicate by sending messages over a network that is reliable, but does not preserve the sending order of messages.

Every message used in the protocol has a field *src* that contains the index of the sending cell, and a field *dst* that contains the address of the cell to which it was sent. Additional fields, *old* and *new*, are used in some message types to hold the indices of other cells.

Every cell has state variables for its control state, *state*, and the index of the next cell in the list *next*. When a cell is not in the list, its *next* variable contains the index of the cell itself. The *next* variable of each cell is a specification variable, because the list structure is important for the correctness of the protocol. The variable *state* is an implementation

variable. There are also variables associated with the cells to hold messages that are in transmission.

A cell, other than the head cell, can perform two types of transactions: *add* and *delete*. There is an add_i transaction and a $delete_i$ transaction for each cell i in the protocol (i.e., if there are n cells, there are $2n$ transactions, not 2 transactions). In the following, let i be the index of the cell initiating the transaction.

An *add* transaction can occur when cell i is not in the list, and when the state of cell i is *normal*. The cell i will be added at the head of the list. The transaction consists of three steps:

1. Cell i sends an *add* message to the head cell; cell i changes its state to *w_head* ("wait for *head* message").
2. The head cell sends a *head* message containing the *next* value of the head cell to cell i. Then the head cell stores i in its *next* variable.
3. When cell i receives the *head* message, it stores the value in the message into its *next* variable. Cell i then changes its state back to *normal*.

The specification state variables consist of the collection of *next* pointers of the cells. The *add* transaction in the specification inserts cell i at the front of the list, updating the *next* variables of the head cell and cell i in a single atomic step.

The commit step for the add_i transaction occurs in step 2, which is the first point where a specification variable is modified (*next* of the head cell). Step 1 only modifies implementation variables *state* and *network*, so it begins and ends in pre-commit states for add_i. The state between step 2 and 3 is a post-commit state. Step 3 completes the transaction; it is the point where a specification variable changes for the last time in the transaction. Hence, the state following step 3 is again a pre-commit state for add_i.

The $delete_i$ transaction can occur when a cell's *next* points to a cell other than i (meaning i is in the list) and its *state* is *normal*. The problem with deleting in a distributed singly-linked list is that there is no easy way for cell i to determine its predecessor in the list, which is unfortunate since *next* of the predecessor must be changed to point to the *next* of cell i.

The solution to this problem is to have another message *pred* which circulates around the list at all times[3]. When cell i receives the *pred* message, it can determine its predecessor by examining the *src* field of the message. So, the steps of the $delete_i$ transaction are:

1. Cell i changes its *state* to *w_pred* ("wait for *pred* message").
2. When cell i receives a *pred* message, it sends a *chnext* message ("change next") to the source of the *pred* message which is usually the predecessor of i in the list. The *chnext* message has i in its *old* field and the *next* of cell i in its *new* field. Cell i changes *state* to *w_delack* ("wait for delete-acknowledgment").
3. When a cell j receives the *chnext* message there are several possible cases. The subtleties of these rules handle difficult scenarios, such as the predecessor deleting itself and then being in the midst of adding itself again between cell i's receipt of the *pred* message and the receipt of the *chnext* message.

[3] There is another version of distributed list protocol, in which *pred* message is generated only when necessary.

(a) If the *state* of cell j is not *normal* or *w_pred*, the *chnext* message remains in the network (progress occurs when some other message arrives at cell j).
(b) Otherwise, if the *old* field of the message matches the *next* variable of cell j, the cell changes its *next* to be the *new* of the *chnext* message (*next* of i). Then cell j sends a *delack* message to cell i (*src* of the *chnext* message). Cell j then sends a *pred* message to its *next* cell.
(c) Otherwise, cell j forwards the *chnext* message to its *next* cell. In this case, the cell receiving the *chnext* message is the head cell and one or more new cells were inserted at the head of the list while cell i was being deleted, so the predecessor of cell i is now somewhere further down the list. The true predecessor will eventually receive the *chnext*, causing the case (b) to occur.

4. When cell i receives a *delack*, it changes its *next* variable to i, and changes *state* to *normal*.

The commit step of the $delete_i$ transaction is in case (b) of step 3 above. Step 3 may be repeated several times because of case (c) before a commit occurs, so a state immediately following step 3(c) is a pre-commit state. Step 4 completes the transaction.

The specification handles the delete transaction atomically by removing cell i from the list in the obvious way: it sets the *next* of the predecessor of i to the *next* of i, then sets *next* of i to i.

The *pred* message circulates around the list constantly except when it temporarily disappears during processing of a *chnext* during a delete transaction, so each cell has rules for propagating it. However, processing a *pred* message never affects a specification variable, so there are no transactions associated with it. It is necessary to reason about the processing of *pred* messages during the proof of invariants (discussed below), and also for liveness properties (which are not discussed here).

The above description of the protocol traces through individual transactions. It is easier to make sure that a description is complete if the behavior is described for each component, not each transaction (and, indeed, the above description is not complete). Table 1 gives the rules of cell behavior in pseudo-code on per-cell basis.

3.2 The aggregation function

Here, we define the aggregation function abs for the distributed list example. The key question is how to complete all committed transactions in the current state, especially since the number of cells, and hence the number of committed transactions, is unknown. The general strategy, which has worked for our larger examples as well, is to define a per-component completion function, which is then generalized to a completion function for all of the cells in the system. This is possible because the post-commit steps of different nodes are generally independent.

It is quite simple to complete a committed transaction for a particular cell. If a *head* message destined for cell i exists, an add_i transaction must be completed by simulating the effect of cell i processing the *head* message it receives at the end of the transaction. This processing changes *next* to point to the value *new* field in the message. Changes to implementation variables, such as removing messages from the network, can be omitted from the completion function, as they do not affect the corresponding specification state.

Step	Condition	Action
Initiate Add	$i \neq$headptr $\wedge$ next[i]=i $\wedge$ state[i]=normal	Send *add*⟨src=i⟩ to headptr state[i] := w_head
Process *add*	*add* sent to headptr	Send *head*⟨new=next[headptr]⟩ to *add.src* next[headptr] := *add.src*
Process *head*	*head* sent to i	next[i] := *head.new* state[i] := normal
Initiate Delete	$i \neq$headptr $\wedge$ next[i]$\neq i$ $\wedge$ state[i]=normal	state[i] := w_pred

Process *pred*	*pred* sent to i	if state[i]=normal:
		Send *pred*⟨src=i⟩ to next[i]
		if state[i]=w_pred:
		state[i] := w_delack,
		Send *chnext*⟨old=i, new=next[i]⟩
		to *pred.src*

Process *chnext*	*chnext* sent to i $\wedge$ *chnext.old*$\neq$next[i] $\wedge$ state[i]$\in${normal,w_pred}	Send *chnext* to next[i]
Process *chnext*	*chnext* sent to i $\wedge$ *chnext.old*=next[i] $\wedge$ state[i]$\in${normal,w_pred}	next[i] := *chnext.new* Send *delack* to *chnext.old* Send *pred*⟨src=i⟩ to *chnext.new*
Process *delack*	*delack* sent to i	next[i] := i, state[i] := normal

Table 1. Formal Description of Distributed List Protocol: The action of a step is executed if its condition holds. Each process consumes the message that triggers it. A message consists of a record with fields *src, new, old*. When a message is created, we use $m\langle f{=}a'\rangle$ to denote that message m has value a' for its record field f. We use $m.f$ to refer to the value of field f in message m. State variables for cells are kept in arrays, state and next.

All of this computation is done solely in cell i, without the involvement or interference of other cells. If there is a *delack* message for cell i, a $delete_i$ transaction must be completed by setting *next* to i. Otherwise, the completion function does nothing.

It is easy to generalize the completion function for one cell to a completion function for all of the cells because the completions do not interact. The global implementation state is an array of cell state records, indexed by the cell indices. Let $cc(q[i])$ be a completion function for cell i, which modifies the state variables for i in the record $q[i]$, and returns a new record of the state variables as modified by the completion of the transaction.

If $cc(q[i])$ completes committed transactions on cell i, the completion function for all cells is $\lambda q.\lambda i.cc(q[i])$. When this function is supplied a state q, it returns $\lambda i.cc(q[i])$,[4] which is an array of the completed cell states, i.e., the desired clean global state. The aggregation function is simply the completion function, followed by a projection which eliminates all implementation variables.

[4] The notation may be a bit confusing. $\lambda i.cc(q[i])$ is a function, which when applied to a particular value of i, say i_0, returns $cc(q[i_0])$, which is the completed state for cell i_0. This is effectively the same as indexing into an array of completed cell states.

3.3 Extracting specification

Reverse engineering of a specification can be illustrated on the distributed list protocol (indeed, we had to do this). Given only an implementation description, the first step is to identify the specification variables. In the distributed list protocol, we decided that they were the *next* variables for the cells. The next step is to trace through a transaction, concatenating the implementation steps, simplifying by substituting values forward through intermediate assignments, and then eliminating statements that only change implementation variables.

For an add_i transaction in the protocol, the sequence of steps is "initiate add," "process *add*," and "process *head*." The result obtained by the procedure is

$$\begin{aligned} &\mathit{Atomic\_Add}(i)\text{: if } i \neq \text{headptr} \wedge \text{next}[i] = i \text{ then} \\ &\qquad \text{next}[i] := \text{next[headptr]; next[headptr]} := i. \end{aligned}$$

Similarly, $delete_i$ transaction corresponds to the sequence of steps, "initiate delete," "process *pred*," "process *chnext*," and "process *delack*." The atomic transaction obtained by aggregation is

$$\begin{aligned} &\mathit{Atomic\_Delete}(c, i)\text{: if } i \neq \text{headptr} \wedge \text{next}[i] \neq i \wedge \text{next}[c] = i \text{ then} \\ &\qquad \text{next}[c] := \text{next}[i]; \text{next}[i] := i. \end{aligned}$$

With the two atomic transactions and idle steps in the specification, we instantiate the subgoals (4) for each implementation steps. The proper instantiation for the proof is shown in table 2.

Implementation step at node i	Specification transactions
Initiate Add	ε
Process *add*	*Atomic_Add*(*add.src*)
Process *head*	ε
Initiate Delete	ε
Process *pred*	ε
Process *chnext* (Forward)	ε
Process *chnext* (Commit)	*Atomic_Delete*(*i*, *chnext.old*)
Process *delack*	ε

Table 2. Corresponding specification steps for implementation steps in the distributed list protocol

3.4 The invariant

The proofs of the subgoals (4) corresponding to each row in table 2 are simple. PVS can handle them almost automatically. Among the eight subgoals, four have been proved automatically for any state q. However, the rest of the subgoals need some assertions on the state in the system to satisfy the commutativity property. The invariant consisting of several assertions that we need to prove the subgoals is listed below.

- The head cell is always *normal*.
- If a cell is in *normal* or *w_pred* state, there is no *add* message from the cell, *delack* message to the cell, or *chnext* message with *old* field equal to the cell.
- If there is an *add* message from or *head* message to a cell i, then the *next* of the cell is i.
- In a *chnext* message, the *next* of the cell contained in the *old* field of the message must be the same as the *new* field of the message.
- There is at most one message in the network for each transaction currently in progress, and there must be no more than one *pred* message in the network.

The only manual step occurs when proving subgoals of the form $(\forall j : Inv(j)) \Rightarrow Q(i)$, where i is a cell index, which requires eliminating the $\forall j$ by substituting i for j to obtain $Inv(i) \Rightarrow Q(i)$, which can be handled automatically.

Part of the reason that the proof is simple is that we have chosen to represent the network in a non-obvious way. We observe that there is at most one message pertaining to any particular transaction at any time. So the network can be represented with one variable per cell (sometimes associated with the source, sometimes with the destination), plus a single variable for the *pred* message. Hence, instead of proving that there is only one message of a certain type in the network for cell i at any time, we register an error whenever a message in a variable for the network is about to be overwritten, and verify that no error occurs. The description can read a message by accessing the variable instead of choosing one and removing it from a set of messages, which is a bit more difficult to deal with in PVS. It is possible to use similar tricks in the other examples we have done, including the large FLASH protocol.

4 Concluding Remarks

Although, aggregation as described can be applied to many protocols, we have only tried a few. It may need to be generalized (and many generalizations are conceivable).

We have not considered the important problem of proving liveness properties here. We do not expect that it will prove to be particularly difficult, however.

From this and many other efforts, it has become clear that finding invariants the most time consuming part of many verification problems. More computer assistance is needed, especially for large problems.

Acknowledgments

We would like to thank Sam Owre and Natarajan Shankar at SRI International for their help with PVS system.

References

[AL91] Martín Abadi and Leslie Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82:253–284, 1991.

[BD94] Jerry Burch and David Dill. Automatic verification of pipelined microprocessor control. In *Computer Aided Verification, 6th International Conference, CAV'94*, pages 68–80, June 1994.

[Coh93] Ernest Cohen. *Modular progress proofs of asynchronous programs*. PhD thesis, University of Texas at Austin, 1993.

[dBdRR90] J. de Bakker, W. de Roever, and G. Rozenberg, editors. *Stepwise Refinement of Distributed Systems. Models, Formalisms, Correctness.: LNCS 430*. Springer-Verlag, 1990.

[DDHY92] D. Dill, A. Drexler, A. Hu, and C. Yang. Protocol verification as a hardware design aid. In *International Conference on Computer Design: VLSI in Computers*. IEEE Computer Society, 1992.

[DHWT91] D. Dill, A. Hu, and H. Wong-Toi. Checking for language inclusion using simulation relation. In *Computer Aided Verification, 3rd International Workshop*, pages 255–265, July 1991.

[Hei93] Mark Heinrich. *The FLASH Protocol*. Internal document, Stanford University FLASH Group, 1993.

[ID96] C. Norris Ip and David Dill. State reduction using reversible rules. In *Proceedings of 33rd Design Automation Conference*, June 1996.

[KOH+94] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy. The Stanford FLASH multiprocessor. In *Proc. 21st International Symposium on Computer Architecture*, pages 302–313, April 1994.

[Kur94] Robert Kurshan. *Computer-Aided Verification of Coordinating Processes: The Automata-Theoretic Approach*. Princeton, 1994.

[Lam82] Leslie Lamport. An assertional correctness proof of a distributed algorithm. *Science of Computer Programming*, 2:175–206, 1982.

[Lam83] Leslie Lamport. Specifying concurrent program modules. *ACM Transactions on Programming Language and Systems*, 5(2):190–222, April 1983.

[LLOR96] P. Ladkin, L. Lamport, B. Olivier, and D. Roegel. Lazy caching: An assertional view. *Distributed Computing*, 1996. To appear.

[LS84] S. Lam and A. Shankar. Protocol verification via projection. *IEEE Transactions on Software Engineering*, 10(4):325–342, July 1984.

[Lyn88] N. Lynch. I/O automata: A model for discrete event systems. In *22nd Annual Conference on Information Science and Systems*, March 1988. Princeton University.

[McM93] Ken McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993. Boston.

[ORSvH95] S. Owre, J. Rushby, N. Shankar, and F. von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, February 1995.

[PD96] Seungjoon Park and David Dill. Verification of FLASH cache coherence protocol by aggregation of distributed transactions. In *Proc. 8th ACM Symposium on Parallel Algorithms and Architectures*, June 1996.

[PKP91] D. Peled, S. Katz, and A. Pnueli. Specifying and proving serializability in temporal logic. In *Proc. 6th Annual IEEE Symposium on Logic in Computer Science*, pages 232–244, July 1991.

Atomicity Refinement and Trace Reduction Theorems

E. Pascal Gribomont
Institut Montefiore, Université de Liège, Sart-Tilman B 28, B-4000 Liège (Belgium)
gribomon@montefiore.ulg.ac.be

Abstract. Assertional methods tend to be useable for abstract, coarse-grained versions of concurrent algorithms, but quickly become intractable for more realistic, finer-grained implementations. Various trace-reduction methods have been proposed to transfer properties of coarse-grained versions to finer-grained versions. We show that a more direct approach, involving the explicit construction of an (inductive) invariant for the finer-grained version, is theoretically more powerful, and also more appropriate for computer-aided verification.

1 Introduction

Recents improvements in methods and tools for testing the validity of propositional and predicate logic formulas have revived the interest in assertional methods for concurrent system verification. Indeed, at least as far as safety properties are concerned, Hoare's logic and Dijkstra's predicate transformer calculus reduce the correctness problem for programs to the validity problem for logical formulas.

However, as soon as loops occur in programs, creativity is needed to discover appropriate invariants. This task is reasonably feasible for coarse-grained, abstract concurrent systems, but often becomes intractable for fine-grained, reasonably efficient implementations.

A standard technique is to deal first with a coarse-grained version of the system to be verified, and then to attempt (in a more or less formal way) to adapt the conclusion to a finer-grained implementation of the system. This is called *atomicity refinement*. In this paper, we compare two frequently used techniques for atomicity refinement, from both theoretical and practical point of view.

The problem solved by these techniques is as follows. Some concurrent system has been proved correct with respect to some safety property. Some statement is replaced by an equivalent sequence of more elementary statements. Due to possible interference between processes, this atomicity refinement is not always correct. How can such a refinement be validated (or disproved)? Let us consider a two-process system **S**, where the (cyclic) concurrent processes are

$$Loop(S_1; S_2) \quad \text{and} \quad Loop(T_1; T_2)$$

There is some initial condition A and some safety property J, validated with some invariant I. Otherwise stated, there is an assertion I such that $A \Rightarrow I$, $I \Rightarrow J$, and, for each state σ satisfying I, if any of the transitions S_1, S_2, T_1 and T_2 can be executed from state σ, then the resulting state ρ also satisfies I. As a consequence, any **S**-computation whose initial state satisfies A reaches only states satisfying J.

Now we replace a transition, say T_2, by an equivalent sequence, say $T'; T''$ (transition T_2 can lead from state σ_1 to state σ_2 if and only if sequence $T'; T''$ can lead from σ_1 to σ_2). The question is, is the new system **S'** still correct w.r.t. the safety property J?

There is clearly no problem with *primary* **S'**-computations, such that any execution of T' is *immediately* followed by an execution of T'', without interference from

S_1 or S_2. Let us call B the assertion which holds in all states but those "between" some execution of T' and the corresponding execution of T''. It is clear that J still holds in *relevant states* (those satisfying B), that is, that $B \Rightarrow J$ remains true throughout the computation.

Now, let us consider the general case where some execution(s) of S_1 and S_2 take(s) place between an execution of T' and an execution of T'', for instance

$$S_1; T_1; T'; S_2; T''; T_1; S_1; T'; S_2; S_1; T''; \ldots$$

It is not always the case that $B \Rightarrow J$ remains true throughout the computation.

The *trace reduction method* guarantees that $B \Rightarrow J$ remains a safety property, provided that T' is a *right-mover*, i.e., the following holds: if $(T'; S_1)$ can lead from some state σ to some state ρ, then $(S_1; T')$ can also lead from σ to ρ, and the same with S_1 replaced by S_2. (Instead of requiring T' to be a right-mover, we can require T'' to be a *left-mover*.) This method is of easy application and has led to successful non-trivial designs; it is especially useful to convert centralized concurrent systems into distributed ones. The drawback is that the method is not complete; some correct atomicity refinements cannot be validated that way.

The *invariant adaptation method* consists in finding some invariant I' of $\mathbf{S}'$ which reduces to I in every relevant state. This method is complete in the following sense: if J is a safety property of $\mathbf{S}$ that remains true in all relevant states of $\mathbf{S}'$, then adequate invariants I and I' exist. The knowledge of I is a big help for the construction of the adapted invariant I', but this construction often turns to be a complicated task nevertheless.

A usual policy for validating atomicity refinements is therefore to try the trace reduction method first, and, only in case of failure, to try the invariant adaptation method. The purpose of this paper is to show that success cases for the reduction method always are elementary cases for the invariant adaptation method, whereas some elementary cases for the invariant adaptation method are still failure cases for the reduction method. As a result, it might be better to use only the invariant adaptation method, especially for computer-aided design/verification.

The paper goes on as follows. An abstract framework for atomicity refinement is introduced in Section 2, where the trace reduction method is presented as a special case of the invariant adaptation method. Both methods are compared in a more general way in Section 3, where success cases for the trace reduction method are proved to correspond to cases of easy invariant adaptation. Section 4 shows that a failure case for the reduction method can turn to be an easy case for the invariant adaptation method. Section 5 is a conclusion and mentions related works.

2 Theorems about atomicity refinement

We introduce an abstract framework for atomicity refinement and show that, from the theoretical point of view, the trace reduction method is a particular case of the invariant adaptation method. More specifically, we recall the main theorem about trace reduction and give a theorem connecting the invariant of a system before and after the atomicity refinement. The former appears as a mere corollary of the latter.

2.1 Relational notation

Let $\mathcal{R}$ and $\mathcal{S}$ be binary relations on a non-empty set Γ, and let $\gamma \in \Gamma$ and $A \subseteq \Gamma$. The following notation is used in the sequel.

$$\begin{array}{ll}
\mathbf{1}_\Gamma =_{def} \{(\gamma,\gamma) : \gamma \in \Gamma\}, & \text{(identical relation)}, \\
\mathcal{R};\mathcal{S} =_{def} \{(\gamma,\delta) : \exists \rho\,[(\gamma,\rho) \in \mathcal{R} \wedge (\rho,\delta) \in \mathcal{S}]\}, & \text{(sequential composition)}, \\
\mathcal{R}^0 =_{def} \mathbf{1}_\Gamma,\ \mathcal{R}^{n+1} =_{def} (\mathcal{R}^n;\mathcal{R}),\ \mathcal{R}^* =_{def} \bigcup_{n\geq 0} \mathcal{R}^n, & \text{(iteration, closure)}, \\
\gamma\mathcal{R} =_{def} \{\delta : (\gamma,\delta) \in \mathcal{R}\},\ A\mathcal{R} =_{def} \bigcup_{\rho\in A} \rho\mathcal{R}, & \text{(set of successors, postset)}.
\end{array}$$

Comments. A binary relation on Γ is simply a subset of $\Gamma \times \Gamma$. The notation $\gamma\mathcal{R}\delta$ usually stands for $(\gamma,\delta) \in \mathcal{R}$. The sequential composition $\mathcal{R};\mathcal{S}$ is also noted $\mathcal{S} \circ \mathcal{R}$. Note that $A\mathcal{R}\mathcal{S} = A(\mathcal{R};\mathcal{S})$. An element γ is an $\mathcal{R}$-*predecessor* of δ if δ is an $\mathcal{R}$-successor of γ, that is, if $(\gamma,\delta) \in \mathcal{R}$.

2.2 Abstract transition systems

An *abstract transition system* [18, 28] is a couple $\mathbf{Ats} = (\Gamma, \{\mathcal{R}_1,\ldots,\mathcal{R}_n\})$ where Γ is a non-empty *state space* and where $\{\mathcal{R}_1,\ldots,\mathcal{R}_n\}$ is a finite non-empty set of *actions*, i.e., binary relations on Γ. A *state* is an element $\gamma \in \Gamma$. A *predicate* is a subset $A \subseteq \Gamma$.

Comment. Predicates are usually represented as assertions, so we will write $\gamma \models A$ ("γ satisfies A", "A is true at γ") instead of $\gamma \in A$. Similarly, we write $\neg A$, $A \wedge B$ and $A \vee B$ instead of $\Gamma \backslash A$, $A \cap B$ and $A \cup B$, respectively. An assertion C is *valid* if the corresponding set is Γ; we write $\models C$ instead of $(\forall \gamma \in \Gamma)\,(\gamma \models C)$; the inclusion $A \subseteq B$ therefore becomes $\models (A \Rightarrow B)$. Last, the successor set $A\mathcal{R}$ and the reachability set $A\mathcal{R}^*$ are modelled by the assertions $sp[A;\mathcal{R}]$ ("strongest postcondition") and $sin[A;\mathcal{R}]$ ("strongest invariant") respectively.

An $(\mathbf{Ats}, A)$-*traced computation*, or simply a *traced computation*, is a sequence

$$\mathcal{C} = (\gamma_0, r_1, \gamma_1, r_2, \ldots, r_m, \gamma_m, \ldots),$$

where $\gamma_0 \models A$ and, for all $i > 0$, r_i is an **Ats**-action (a member of $\{\mathcal{R}_1,\ldots,\mathcal{R}_n\}$) such that $\gamma_{i-1} r_i \gamma_i$. The underlying sequence of states $\mathcal{C}_s$ is a *computation*, and the sequence of actions $\mathcal{C}_a$ is a *trace*. Assertion A is the *initial condition*.

Comments. The union of a set of actions is an action, so the abstract transition system $(\Gamma, \{\mathcal{R}_1,\ldots,\mathcal{R}_n\})$ can be replaced by $(\Gamma, \{\mathcal{R}\})$, where $\mathcal{R} =_{def} \bigcup_{i=1}^n \mathcal{R}_i$, without changing the set of computations; a sequence $\mathcal{C}_s = (\gamma_n)$ is a computation if γ_i is an $\mathcal{R}$-successor of γ_{i-1}, for all $i > 0$. A computation can be finite if it reaches a state without successor.

An **Ats**-*invariant*, or simply an *invariant*, is a predicate I such that every successor of every state satisfying I also satisfies I; this is denoted $\{I\}\,\mathcal{R}\,\{I\}$, or $\{I\}\,\mathbf{Ats}\,\{I\}$, or $\models (sp[I;\mathcal{R}] \Rightarrow I)$. An $(\mathbf{Ats}, A)$-*safety property*, or simply a *safety property*, is a predicate J such that, for every computation $\mathcal{C}_s = (\gamma_0, \gamma_1, \ldots)$, if $\gamma_0 \models A$, then $\gamma_n \models J$ for all n.[1]

[1] In our framework, the connection between Hoare's logic and Dijkstra's calculus is simple: expressions $\{A\}\mathcal{S}\{B\}$, $\models (sp[A;\mathcal{S}] \Rightarrow B)$ and $\models (A \Rightarrow wlp[\mathcal{S};B])$ are equivalent. A useful property of *sp* (and *wlp*) is *monotonicity*. If $\mathcal{R}_1 \subseteq \mathcal{R}_2$ and $\models (A_1 \Rightarrow A_2)$, then $\models (sp[A_1;\mathcal{R}_1] \Rightarrow sp[A_2;\mathcal{R}_2])$. Similarly, if $\models (A_2 \Rightarrow A_1)$, $\mathcal{R}_2 \subseteq \mathcal{R}_1$ and $\models (B_1 \Rightarrow B_2)$, then $\{A_1\}\mathcal{R}_1\{B_1\}$ implies $\{A_2\}\mathcal{R}_2\{B_2\}$.

If $\models (A \Rightarrow I)$ and if I is an invariant, then I is necessarily a safety property, but it should be emphasized that the converse is not true: safety properties usually are not invariants. For instance, if **Ats** is a correct mutual exclusion algorithm, the assertion J which expresses mutual exclusion is a safety property but is not an invariant.

Comment. With the restrictive definition given above, a computation C can be checked for some safety property by considering only isolated states, and a safety property is simply (modelled by) a subset of Γ. Safety properties can be defined in a more general way [1] and modelled by subsets of Γ^* (Γ^* denotes the set of finite sequences of states). However, it is always possible, at least theoretically, to include all the preceding states in any state of the computation, so the restriction is not essential: any information about a computation prefix $(\gamma_0, \ldots, \gamma_n)$ can be retrieved from the state γ_n. In practice, special auxiliary variables, called *history variables*, are used for that purpose.

The following classical result (an early reference is [9]) asserts the completeness of the invariant method and states the connection between invariants and safety properties.

Theorem. The system $(\Gamma, \{\mathcal{R}\})$ satisfies the safety property J for the initial condition A if and only if an invariant I exists such that $\models [(A \Rightarrow I) \wedge (I \Rightarrow J)]$.

Sketch of proof. The strongest possible choice for I is $sin[A; \mathcal{R}]$, i.e., the set of states that can be accessed from A (in finitely many computation steps). This predicate represents the set of $\mathcal{R}^*$-successors of all states satisfying A; it is an invariant, so J is a safety property if and only if $\models (sin[A; \mathcal{R}] \Rightarrow J)$. □

Comment. Invariant are *inductive* safety properties, which can be proved by an induction argument. The standard technique for proving a (non-inductive) safety property is to construct a stronger, inductive one (i.e., an invariant). A similar situation frequently occurs in number theory. If some property $P(n)$ of natural numbers cannot be proved by induction, it is sometimes possible to discover a stronger property $Q(n)$ that can be proved by induction. Invariants are also named *stable properties*, e.g. in [6], where the word "invariant" refers to a stable property satisfied in some specified set of initial states.

2.3 Atomicity refinement: the abstract framework

Let A, B be predicates on Γ, and let **Old** $= (\Gamma, \{\mathcal{S}, \mathcal{R}\})$, **New** $= (\Gamma, \{\mathcal{S}_1, \mathcal{S}_2, \mathcal{R}\})$ be two abstract transition systems. Condition A is the *initial condition*, and B is the *refinement condition*. States satisfying B are called *relevant states*; those not satisfying B are *transient states*. We assume the following conditions:

1. $\models (A \Rightarrow B)$;
2. $\mathcal{S}_1$-successors of relevant states are transient states;
3. relevant states have no $\mathcal{S}_2$-successor;
4. $\mathcal{S}_2$-successors of transient states are relevant states;
5. transient states have no $\mathcal{S}_1$-successor;
6. $\mathcal{R}$-successors of relevant states are relevant states;
7. $\mathcal{R}$-successors of transient states are transient states;
8. $\mathcal{S} = \mathcal{S}_1; \mathcal{S}_2$ (sequential consistency).

These conditions[2] guarantee that, in any **New**-trace, actions $\mathcal{S}_1$ and $\mathcal{S}_2$ appear strictly in turn, and that $\mathcal{S}_1$ appears first. Predicate A is the initial condition of both **Old** and **New** (only computations whose initial state satisfies A are of interest). Predicate B is the *refinement condition*, which is true in relevant states and false in transient states. Let $\mathcal{C} = (\gamma_0, r_1, \gamma_1, r_2, \ldots, r_m, \gamma_m, \ldots)$ be a **New**-traced computation (so $r_i \in \{\mathcal{S}_1, \mathcal{S}_2, \mathcal{R}\}$, for all i). A state γ_k is relevant if $\mathcal{S}_1$ and $\mathcal{S}_2$ occur equally many times in the trace prefix $\mathcal{P} = (r_1, \ldots, r_k)$; otherwise, γ_k is transient (and $\mathcal{S}_1$ occurs one more time than $\mathcal{S}_2$ in $\mathcal{P}$).

Comments. We assume the existence of an atomicity refinement condition B. The simplest and most frequent case of atomicity refinement is the replacement of a transition (ℓ_0, S, ℓ_1) by (ℓ_0, S_1, m) and (m, S_2, ℓ_1), where $S_1; S_2$ is "sequentially equivalent" to S and where m is a new label. The natural choice for the refinement condition is $B =_{def} \neg at\ m$ (the control does not lie at control point m, between S_1 and S_2). However, we also require that **Old** and **New** share the same state space Γ, and therefore the same assertion language. To ensure this, we assume that the new location predicate *at m* already existed in the old assertion language, even if no state satisfying it could be reached. Any assertion J about **Old**, in particular the initial condition and the invariant, will be (maybe implicitly) rewritten as $J \wedge \neg at\ m$.

A **New**-trace is *primary* if every occurrence of $\mathcal{S}_1$ is immediately followed by an occurrence of $\mathcal{S}_2$. For most practical purposes, primary **New**-traces can be assimilated to **Old**-traces. The idea underlying trace reduction theorems is that, provided some hypotheses are satisfied, every **New**-trace has an equivalent **New**-primary trace, so **New** itself is equivalent to **Old**. The problem is, the stronger the equivalence notion, the stronger the required hypotheses. As a result, several trace reduction theorems have been proposed, with more or less restrictive hypotheses and equivalence notions.

2.4 Theorems

The trace reduction method allows to assert that some properties of **Old**-computations are preserved in **New**-computations. Even with restricting to safety properties, one cannot hope that all of them are preserved. For instance, with the notation of § 2.3, the refinement condition B is an (**Old**,A)-safety property (and also an **Old**-invariant) but cannot be a (**New**,A)-safety property since B is false in any transient state. However, if some hypothesis is satisfied, any **Old**-safety property J gives rise to the **New**-safety property $B \Rightarrow J$. Otherwise stated, safety properties are preserved in relevant states, but nothing is known about transient states. Such a result is useful when J is trivially true in transient states, i.e., when $\neg B \Rightarrow J$ is valid. This is a very frequent case; for instance, 2-process mutual exclusion and partial correctness are expressed by assertions that trivially hold in transient states, since critical states and final states (if any) always are relevant states.

The preservation theorem for safety property is an old result, originating from the ideas of [18] and [26]. The first formal presentation is probably [12]; [19] and [23] contain more results about atomicity refinement and the trace reduction method.

[2] Conditions 2 to 8 can be expressed as $\{B\}\,\mathcal{S}_1\,\{\neg B\}$, $\{B\}\,\mathcal{S}_2\,\{false\}$, $\{\neg B\}\,\mathcal{S}_2\,\{B\}$, $\{\neg B\}\,\mathcal{S}_1\,\{false\}$, $\{B\}\,\mathcal{R}\,\{B\}$, $\{\neg B\}\,\mathcal{R}\,\{\neg B\}$, and $sp[X;\mathcal{S}] \equiv sp[sp[X;\mathcal{S}_1];\mathcal{S}_2]$ for all X, respectively. Two useful corollaries are $\{true\}\,\mathcal{S}_1\,\{\neg B\}$ and $\{true\}\,\mathcal{S}_2\,\{B\}$.

A definition is introduced first:

Definition. A relation $\mathcal{R}_1$ *right-commutes* with a relation $\mathcal{R}_2$ (and relation $\mathcal{R}_2$ *left-commutes* with relation $\mathcal{R}_1$) if $\mathcal{R}_1;\mathcal{R}_2 \subseteq \mathcal{R}_2;\mathcal{R}_1$.

Theorem 1. If **Old**, **New**, A and B are as introduced in § 2.3, if J is a predicate on Γ and if $\mathcal{S}_1$ right-commutes with $\mathcal{R}$, then $B \Rightarrow J$ is a (**New**, A)-safety property if and only if J is an (**Old**, A)-safety property.

Proof of theorem 1. The "only if" part is trivial. A direct proof of the "if" part is given in [12] and [23]; it is also a corollary of theorem 2 given below. □

Comment. Theorem 1 has a dual version, where requirement $\mathcal{S}_1$ right-commutes with $\mathcal{R}$ is replaced by $\mathcal{S}_2$ left-commutes with $\mathcal{R}$.

In order to compare the trace reduction technique and the invariant adaptation technique, we specify the connection between **Old**-invariants and **New**-invariants, when the reduction hypothesis holds.

Theorem 2. If **Old**, **New** and B are as introduced in § 2.3, if I is a predicate on Γ such that $\models (I \Rightarrow B)$, and if $\mathcal{S}_1;\mathcal{R} \subseteq \mathcal{R};\mathcal{S}_1$ (that is, $\mathcal{S}_1$ right-commutes with $\mathcal{R}$), then predicate $I \vee sp[I;\mathcal{S}_1]$ is a **New**-invariant if and only if I is an **Old**-invariant.

Proof of theorem 2. Let Φ be the predicate $I \vee sp[I;\mathcal{S}_1]$. We first assume that Φ is a **New**-invariant, and observe that $\Phi \wedge B$ is I. (Indeed, formula $\Phi \wedge B$ reduces to $(I \vee sp[I;\mathcal{S}_1]) \wedge B$, i.e., to $(I \wedge B) \vee (sp[I;\mathcal{S}_1] \wedge B)$, and the second disjunct is identically false.) From $\{\Phi\}\,\mathcal{R}\,\{\Phi\}$ and $\{B\}\,\mathcal{R}\,\{B\}$, we therefore deduce $\{I\}\,\mathcal{R}\,\{I\}$; from $\{\Phi\}\,\mathcal{S}_1\,\{\Phi\}$, $\{\Phi\}\,\mathcal{S}_2\,\{\Phi\}$ and $\{\mathit{true}\}\,\mathcal{S}_2\,\{B\}$ we deduce $\{\Phi\}\,\mathcal{S}_1;\mathcal{S}_2\,\{\Phi\}$ and $\{B\}\,\mathcal{S}_1;\mathcal{S}_2\,\{B\}$, and then $\{I\}\,\mathcal{S}_1;\mathcal{S}_2\,\{I\}$, therefore $\{I\}\,\mathcal{S}\,\{I\}$. As $\{I\}\,\mathcal{R}\,\{I\}$ and $\{I\}\,\mathcal{S}\,\{I\}$ both hold, I is an **Old**-invariant.

We now assume that I is an **Old**-invariant. In order to prove that Φ is a **New**-invariant, we check separately the triples $\{\Phi\}\,\mathcal{S}_1\,\{\Phi\}$, $\{\Phi\}\,\mathcal{S}_2\,\{\Phi\}$ and $\{\Phi\}\,\mathcal{R}\,\{\Phi\}$.

1. From the triples $\{I\}\,\mathcal{S}_1\,\{sp[I;\mathcal{S}_1]\}$ and $\{sp[I;\mathcal{S}_1]\}\,\mathcal{S}_1\,\{\mathit{false}\}$, we deduce $\underline{\{I \vee sp[I;\mathcal{S}_1]\}\,\mathcal{S}_1\,\{sp[I;\mathcal{S}_1] \vee \mathit{false}\}}$
2. $\{B\}\,\mathcal{S}_2\,\{\mathit{false}\}$ and $\{sp[I;\mathcal{S}_1]\}\,\mathcal{S}_2\,\{I\}$ lead to $\underline{\{B \vee sp[I;\mathcal{S}_1]\}\,\mathcal{S}_2\,\{I \vee \mathit{false}\}}$
3. Since sp is monotonic, we get from the reduction hypothesis $\mathcal{S}_1;\mathcal{R} \subseteq \mathcal{R};\mathcal{S}_1$ $sp[I;(\mathcal{S}_1;\mathcal{R})] \Rightarrow sp[I;(\mathcal{R};\mathcal{S}_1)]$, i.e., $\{sp[I;\mathcal{S}_1]\}\,\mathcal{R}\,\{sp[sp[I;\mathcal{R}];\mathcal{S}_1]\}$. We have also $\{I\}\,\mathcal{R}\,\{I\}$, hence $\underline{\{I \vee sp[I;\mathcal{S}_1]\}\,\mathcal{R}\,\{I \vee sp[sp[I;\mathcal{R}];\mathcal{S}_1]\}}$

In every case the precondition is weaker than Φ and the postcondition is stronger, so the three required triples follow by monotonicity. (For the third postcondition, observe that $sp[I;\mathcal{R}] \Rightarrow I$, hence $sp[sp[I;\mathcal{R}];\mathcal{S}_1] \Rightarrow sp[I;\mathcal{S}_1]$.)

Comment. Let Ψ be the strongest **New**-invariant which is implied by I, that is, the predicate $sin[I;(\mathcal{R} \cup \mathcal{S}_1 \cup \mathcal{S}_2)]$. A state γ satisfies Ψ if and only if there exists a **New**-computation $(\gamma_n : n = 0, 1, \ldots)$ such that $\gamma_0 \models I$ and $\gamma_k = \gamma$ for some $k \geq 0$. As Φ is a **New**-invariant implied by I, we have $\models (\Psi \Rightarrow \Phi)$; besides, $\models (\Phi \Rightarrow \Psi)$ also holds, since any state γ satisfying Φ can be chosen as an initial state of computation (if $\gamma \models I$) or reached in a single step (if $\gamma \models sp[I;\mathcal{S}_1]$). This gives an interesting operational interpretation to the reduction hypothesis: every reachable transient state can be reached from some relevant state in exactly one step.

Comment. Here is the dual version of theorem 2. If **Old**, **New** and B are as introduced above, if I is a predicate on Γ such that $\models (I \Rightarrow B)$, and if $\mathcal{S}_2$ left-commutes with $\mathcal{R}$,

then predicate $I \vee wlp[\mathcal{S}_2; I]$ is a **New**-invariant if and only if I is an **Old**-invariant. The operator *wlp* (weakest liberal precondition) is defined as follows: $\gamma \models wlp[\mathcal{R}; J]$ if and only if every $\mathcal{R}$-successor of γ satisfies J. Although the computation of $wlp[\mathcal{S}_2; I]$ can be easier than the computation of $sp[I; \mathcal{S}_1]$, we prefer to use the latter, which leads to a stronger **New**-invariant; as a program invariant is a formal description of its behaviour, the stronger is usually the better.

We can now show that, when the reduction hypothesis holds, the connection between the safety properties of **Old** and **New** is a mere consequence of the connection between the invariants of **Old** and **New**.

Proposition. The "if" part of theorem 1 is a corollary of theorem 2.[3]

Proof. If J is an (**Old**, A)-safety property, then, due to the completeness of the invariant method, there exists an **Old**-invariant I such that $\models (A \Rightarrow I)$ and $\models (I \Rightarrow (B \wedge J))$.[4] If $\mathcal{S}_1$ right-commutes with $\mathcal{R}$ then (theorem 2), $\Phi =_{def} (I \vee sp[I; \mathcal{S}_1])$ is a **New**-invariant. Besides, it is easy to check[5] $\models (I \Rightarrow \Phi)$, $\models (A \Rightarrow \Phi)$, and $\models (\Phi \Rightarrow (B \Rightarrow J))$; as a result $B \Rightarrow J$ is a logical consequence of an (initially true) invariant, and therefore a (**New**, A)-safety property. □

Comment. The fact $\models (\Phi \Rightarrow (B \Rightarrow J))$ will be useful later.

3 Trace reduction technique vs. invariant adaptation

In paragraph 2.4, the invariant adaptation method has been used to *justify* the trace reduction method. In this section, we would like to show that the invariant adaptation method can *replace* the trace reduction method. We will first show that, when an atomicity refinement can be validated by the trace reduction method, it can as easily be validated by the invariant adaptation method. Afterwards, we show that validation by invariant adaptation may happen to be tractable even when the reduction hypothesis is not satisfied.

3.1 The easy case of atomicity refinement

The data of the atomicity refinement problem are Γ, $\mathcal{S}$, $\mathcal{S}_1$, $\mathcal{S}_2$, $\mathcal{R}$, **Old**, **New**, A and B, satisfying the 8 conditions stated in paragraph 2.3. Furthermore, we suppose that J is an (**Old**, A)-safety property, validated by an **Old**-invariant I. The question is to determine whether $B \Rightarrow J$ is a (**New**, A)-safety property.

If we use the trace reduction technique, we have to verify that the reduction hypothesis $\mathcal{S}_1; \mathcal{R} \subseteq \mathcal{R}; \mathcal{S}_1$ holds. Theorem 2 asserts that a byproduct of this verification is the fact that $\Phi =_{def} (I \vee sp[I; \mathcal{S}_1])$ is a **New**-invariant. This fact alone is sufficient to validate the refinement (last comment of § 2.3). So, instead of checking whether the reduction hypothesis holds, we can check whether Φ is a **New**-invariant. In fact, we can do a bit less, as indicated by the next theorem.

Theorem 3. The assertion Φ is a **New**-invariant if and only if the assertion $sp[I; \mathcal{S}_1]$ is $\mathcal{R}$-invariant, i.e., if the triple $\{sp[I; \mathcal{S}_1]\}\, \mathcal{R}\, \{sp[I; \mathcal{S}_1]\}$ holds.

[3] Recall that the "only if" part of theorem 1 is trivial.

[4] Recall that B characterizes relevant states, and therefore is a safety property of **Old**; transient states appear only in **New**-computations.

[5] Just consider separately the cases where B is true and where B is false; indeed, Φ can also be written as $(B \Rightarrow I) \wedge (\neg B \Rightarrow sp[I; \mathcal{S}_1])$.

Proof. Let us recall first that the assertion Φ reduces to I when B holds (relevant states) and to $sp[I; \mathcal{S}_1]$ when $\neg B$ holds (transient states). As a result, Φ is a **New**-invariant if and only if the following triples hold:

$1.\ \{I\}\mathcal{R}\{I\},\ 2.\ \{I\}\mathcal{S}_1\{sp[I; \mathcal{S}_1]\},\ 3.\ \{sp[I; \mathcal{S}_1]\}\mathcal{S}_2\{I\},\ 4.\ \{sp[I; \mathcal{S}_1]\}\mathcal{R}\{sp[I; \mathcal{S}_1]\}.$

Triple 2 is a tautology and triples 1 and 3 express that I is an **Old**-invariant, so with this hypothesis triple 4 holds if and only Φ is a **New**-invariant. □

Comment. Validity of triple 4 is a weaker condition than the reduction hypothesis (theorem 2); furthermore, its verification can be easier. Indeed, the reduction hypothesis holds if and only if the implication

$$sp[P; (\mathcal{S}_1; \mathcal{R})] \Rightarrow sp[P; (\mathcal{R}; \mathcal{S}_1)]$$

holds for each assertion P, whereas triple 4 can be rewritten in

$$sp[I; (\mathcal{S}_1; \mathcal{R})] \Rightarrow sp[I; \mathcal{S}_1],$$

i.e., an implication that must be true only for one specific assertion.

The conclusion is, when the trace reduction technique applies, the invariant adaptation technique also applies, with no more verification work.

3.2 The general case of atomicity refinement

The trace reduction technique might fail to validate a correct atomicity refinement, since this technique takes all states into account, even unreachable ones. (A notion of *context* has been introduced in [2] to deal with this problem.)

However, the invariant method might be useful even when theorem 2 does not apply. To investigate this, we have the following general theorem, which can be seen as a completeness theorem for atomicity refinement. It states that an atomicity refinement is correct if and only if some formula is an invariant.

Theorem 4. If **Old**, **New** and B are as introduced above, and if I is an **Old**-invariant such that $\models (I \Rightarrow B)$, then $B \Rightarrow I$ is a (**New**, I)-safety property if and only if formula $\Phi^* =_{def} (I \vee sp[I; (\mathcal{S}_1; \mathcal{R}^*)])$ is a **New**-invariant.

Comment. Even when $B \Rightarrow I$ is a (**New**, I)-safety property, it is usually not inductive; it is therefore not a **New**-invariant, but only the logical consequence of some **New**-invariant.

Comment. If $\mathcal{S}_1; \mathcal{R} \subseteq \mathcal{R}; \mathcal{S}_1$, then formula Φ^* reduces to $\Phi =_{def} (I \vee sp[I; \mathcal{S}_1])$.

Proof of theorem 4. If $B \Rightarrow I$ is a (**New**, I)-safety property, then any reachable relevant state satisfies I. Let γ be a reachable transient state; there exist $n \geq 0$ and a traced computation prefix

$$\mathcal{C} =_{def} (\gamma_0, \mathcal{S}_1, \gamma_1, \mathcal{R}, \ldots, \gamma_i, \mathcal{R}, \ldots, \mathcal{R}, \gamma_{n+1})$$

such that $\gamma_0 \models I$ and $\gamma_{n+1} = \gamma$. As a result, $\gamma \models sp[I; (\mathcal{S}_1; \mathcal{R}^n)]$ and therefore $\gamma \models \Phi^*$. Any reachable state satisfies Φ^* and, clearly, any state satisfying Φ^* is reachable; so Φ^* is the set of reachable states, and therefore an invariant.

Conversely, if Φ^* is an invariant, it is also the set of reachable states, so all relevant reachable states satisfy $\Phi^* \wedge B$, that reduces to I. □

Theorem 4 can be the basis of a complete technique for validating atomicity refinements, but the problem is, computing $sp[I; (\mathcal{S}_1; \mathcal{R}^*)]$ is not easy in general.

We can now outline a more general comparison between trace reduction and invariant adaptation. Some notation is introduced first.

$$T_n =_{def} sp[I; (\mathcal{S}_1; \mathcal{R}^n)],$$
$$U_n =_{def} \bigvee_{i \leq n} T_i.$$
$$U^* =_{def} \bigvee_{i \geq 0} T_i.$$

The sequence (U_n) is monotonic ($U_n \Rightarrow U_{n+1}$ holds for all n). An atomicity refinement is correct (theorem 4) if and only if $I \vee U^*$ is a **New**-invariant. A (correct) refinement is *stationary* if U^* reduces to U_n for some n. The preceding theorems imply that the trace reduction method works only if $U_0 = U^*$; even then, the notion of context introduced in [2] may be needed. The invariant-based technique is complete but, in practice, the computation of U^* is likely to be intractable, except when U^* reduces to U_n for a small value of n. Three cases are of special interest:

1. U^* reduces to U_0 and the trace reduction method does work.
2. U^* reduces to U_0 and the trace reduction method does not work (except when contexts are used).
3. U_1 is weaker (i.e., greater) than U_0, and U^* reduces to U_1; the trace reduction method does not work, but the invariant method remains tractable.

Case 2 is briefly illustrated in paragraph 4, where an example of case 3 is also mentioned.

3.3 Computer-aided verification

CAVEAT [16] is a tool for invariant validation. It also supports atomicity refinement, in so far only *sp*-calculus is used to produce invariant candidates U_0 and U_1. The practical bottleneck is that atomicity refinement induces quick size growing of the invariant, and therefore of the verification conditions. The general form of these conditions in CAVEAT is $(h_1 \ldots h_n) \Rightarrow c$, and the validation module becomes very slow when n is big. A possible solution is to rank the hypotheses $h_1, \ldots, h_n$ according to their relevance to the conclusion c. Typically, very few hypotheses are really relevant, and even an elementary ranking program can speed up the validation process. Preliminary results are reported in [17].

4 Applications

When some requirements are satisfied, it is possible to solve (approximately) a fixpoint system of equations (e.g., on the domain of real numbers) like

$$\begin{cases} x = f(x,y) \\ y = g(x,y) \end{cases} \tag{1}$$

in a concurrent way, using two processes X and Y and two boolean variables h_x and h_y, initialized to *true* [5, 11]. The processes are:

Process X	*Process Y*
while $(h_x \vee h_y)$ do	while $(h_x \vee h_y)$ do
if $x \simeq f(x,y)$	if $y \simeq g(x,y)$
then $h_x := \mathit{false}$	then $h_y := \mathit{false}$
else $x := f(x,y)$;	else $y := g(x,y)$;
$(h_x, h_y) := (\mathit{true}, \mathit{true})$	$(h_x, h_y) := (\mathit{true}, \mathit{true})$

(2)

The system terminates when both h_x and h_y are false; we would like that, on termination, both conditions $e_x =_{def} (x \simeq f(x,y))$ and $e_y =_{def} (y \simeq g(x,y))$ are satisfied.

In the coarser-grained version, there are only two transitions (and a single location for each process, say X_0 and Y_0 respectively). The transitions executed by process X are

$$(X_0\,,\ (h_x \vee h_y) \wedge e_x \longrightarrow h_x := \mathit{false}\,,\ X_0)\,,$$
$$(X_0\,,\ (h_x \vee h_y) \wedge \neg e_x \longrightarrow (x, h_x, h_y) := (f(x,y), \mathit{true}, \mathit{true})\,,\ X_0)\,.$$

Comment. The relevant effect of the statement $x := f(x,y)$ is to assign unknown boolean values to *both* conditions e_x and e_y.

An appropriate invariant of this coarse-grained version is $(h_x \vee e_x) \wedge (h_y \vee e_y)$. This formula is true initially (since h_x and h_y are both true) and respected by all transitions (h_x and h_y become false only when e_x and e_y are true, respectively, and every time x or y is touched, both variables h_x and h_y become true again). On termination, the invariant reduces to $e_x \wedge e_y$.

As a first atomicity refinement, we split the "else" part of process X, i.e., we replace

$$(X_0\,,\ (h_x \vee h_y) \wedge \neg e_x \longrightarrow (x, h_x, h_y) := (f(x,y), \mathit{true}, \mathit{true})\,,\ X_0)\,.$$

by

$$(X_0\,,\ (h_x \vee h_y) \wedge \neg e_x \longrightarrow x := f(x,y)\,,\ X_1)\,,$$
$$(X_1\,,\ (h_x, h_y) := (\mathit{true}, \mathit{true})\,,\ X_0)\,.$$

It is not possible to apply the reduction principle, since $x := f(x,y);\ y := g(x,y)$ and $y := g(x,y);\ x := f(x,y)$ may lead to distinct states; similarly, $(h_x, h_y) := (\mathit{true}, \mathit{true})$, in process X, and $h_y := \mathit{false}$ (in process Y) do not commute either. Nevertheless, the refinement is correct. To see this, we compute the first terms of the sequence (T_n) introduced in paragraph 3.2.

The data are:

$I_0 : \mathit{at}\ X_0 \wedge \mathit{at}\ Y_0 \wedge (h_x \vee e_x) \wedge (h_y \vee e_y)$
$\mathcal{S}_1 : (X_0\,,\ (h_x \vee h_y) \wedge \neg e_x \longrightarrow x := f(x,y)\,,\ X_1)\,,$
$\mathcal{R} : \mathcal{R}_t \cup \mathcal{R}_f$, where
$\quad \mathcal{R}_t =_{def} (Y_0\,,\ (h_x \vee h_y) \wedge e_y \longrightarrow h_y := \mathit{false}\,,\ Y_0)\,,$
$\quad \mathcal{R}_f =_{def} (Y_0\,,\ (h_x \vee h_y) \wedge \neg e_y \longrightarrow (y, h_x, h_y) := (g(x,y), \mathit{true}, \mathit{true})\,,\ Y_0)\,.$

For $n = 0$, the disjunctive term $T_n =_{def} sp[I; (\mathcal{S}_1; \mathcal{R}^n)]$ reduces to $T_0 = sp[I_0; \mathcal{S}_1]$, i.e.

$$\mathit{at}\ X_1 \wedge \mathit{at}\ Y_0 \wedge h_x\,.$$

For $n = 1$, the disjunctive term $sp[I_0; (\mathcal{S}_1; \mathcal{R}^n)]$ reduces to $T_1 = sp[I_0; (\mathcal{S}_1; \mathcal{R})]$, and further to $sp[sp[I_0; \mathcal{S}_1]; \mathcal{R}_t] \vee sp[sp[I_0; \mathcal{S}_1]; \mathcal{R}_f]$, that is

$$\mathit{at}\ X_1 \wedge \mathit{at}\ Y_0 \wedge h_x \wedge [(e_y \wedge \neg h_y) \vee h_y]\,,$$

which further results in

$$\mathit{at}\ X_1 \wedge \mathit{at}\ Y_0 \wedge h_x \wedge (e_y \vee h_y)\,.$$

As T_1 is stronger then T_0, there is no need to compute further terms; $\bigvee T_n$ reduces to T_0. An acceptable invariant is now $I_1 =_{def} (I_0 \vee T_0)$, which can be simplified into

$$[(h_x \vee e_x) \wedge (h_y \vee e_y)] \vee (\mathit{at}\ X_1 \wedge h_x)\,.$$

This is an instance of case 2, since U^* reduces to U_0

Symmetrically, if the "else" part of of process Y is split, then the invariant is adapted into

$$[(h_x \vee e_x) \wedge (h_y \vee e_y)] \vee (at\, X_1 \wedge h_x) \vee (at\, Y_1 \wedge h_y).$$

A generalized version of algorithm (2) exists, which involves n processes and allows the distributed solution of n-equation systems. However, the validation of atomicity refinements becomes more complicated, and involves several instances of case 3 (see [14] for details).

Comment. It should be emphasized that, for specific concurrent systems, easier validity proofs can be found for atomicity refinements. This paper is concerned only with the systematic techniques, applying to a broad class of concurrent systems.

5 Conclusion and related work

Two widely used methods for the validation of atomicity refinements have been compared. It is known for a long time that the invariant adaptation method is complete whereas the trace reduction method is not, but also assumed that, in some cases, the trace reduction method is easier to use. This assumption turns to be false and, as far as safety properties are concerned, the invariant-based method has definite advantages. Especially, many refinements encountered in classical examples are correct but outside the scope of the trace reduction techniques. Note, however, that the trace reduction method might still be useful to prove properties like termination and freeness of individual starvation; besides, other reduction methods (relying not only on traces) have been proposed.

The trace reduction technique has been successfully used especially in the area of (deterministic) parallel programming [2, 4]. The invariant adaptation technique is used e.g. in [10, 20]; a systematic presentation is [15]. Incremental construction of invariants, using approximation sequences like (U_n), originates from [8, 7, 29]. Systematic approaches are [21] and [14].

Our main goal in this paper was to validate the decision made in CAVEAT, where the trace reduction method is not implemented (we plan to rely on invariant adaptation only). The program notation used in CAVEAT and in this paper is classical and allows for a convenient version of the reduction theorem and related results. From the theoretical point of view, however, these problems are better investigated at a more abstract, purely semantical level. An adequate framework for doing this is Lamport's TLA (Temporal Logic of Actions). In this formalism, both statements and assertions are represented as logical formulas; this leads to elegant and general formulations of results which, like the reduction theorem and other refinement theorems, involve more than one version of a program [22]. (TLA is also appropriate for more practical problems, especially in program specification; see [22, 25] for more details.) As pointed out by reviewers, the construction of the invariant of the refined version of a concurrent system in terms of the invariant of the reduced version can also be achieved in TLA, at a purely semantic level, as reported in an unpublished working paper [24]. The form given in the present paper (theorem 2) relies only on the elementary predicate transformer *sp*, and not on the higher-level predicate transformers *win* and *sin* used in [24], which cannot be implemented easily as such.

Acknowledgment. It is a pleasure to thank Yih-Kuen Tsay for improving the demonstration of theorem 2, and for a careful and critical reading of the manuscript.

References

1. B. Alpern and F. Schneider, Recognizing safety and liveness, *Distributed Computing* **2** (1987) 117-126.
2. R.-J. Back, A Method for Refining Atomicity in Parallel Algorithms, *Lect. Notes in Comput. Sci.* **366** (1989) 199-216.
3. R.-J. Back and R. Kurki-Suonio, Decentralization of Process Nets with Centralized Control, *Distributed Computing* **3** (1989) 73-87.
4. R.-J. Back and R. Sere, Stepwise Refinement of Parallel Algorithms, *Sci. Comput. Programming* **13** (1990) 133-180.
5. E. Best, A Note on the Proof of a Concurrent Program, *Inform. Processing lett.* **9**, pp. 103-104, 1979
6. K.M. Chandy and J. Misra, *Parallel Program Design : A Foundation* (Addison-Wesley, Reading, MA, 1988).
7. E.M. Clarke, Synthesis of Resource Invariants for Concurrent Programs, *ACM Trans. Programming Languages Syst.* **2** (1980) 338-358.
8. P. Cousot and R. Cousot, Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints, *Proc. 4th ACM Symp. on Principles of Progr. Languages* (1977) 238-252.
9. J.W. De Bakker and L.G.L.T. Meertens, On the Completeness of the Inductive Assertion Method, *Jl. of Computer and Syst. Sci.* (1975) 323-357.
10. E.W. Dijkstra and al., On-the-Fly Garbage Collection : An Exercise in Cooperation, *Comm. ACM* **21** (1978) 966-975.
11. E.W. Dijkstra, Finding the Correctness Proof of a Concurrent Program, *Lect. Notes in Comput. Sci.* **69** (1979) 24-34.
12. T.W. Doeppner, Parallel Program Correctness Through Refinement, *Proc. 4th ACM Symp. on Principles of Progr. Languages* (1977) 155-169.
13. E.P. Gribomont, Synthesis of parallel programs invariants, *Lect. Notes in Comput. Sci.* **186** (1985) 325-338.
14. E.P. Gribomont, Stepwise refinement and concurrency : the finite-state case, *Sci. Comput. Programming* **14** (1990) 185-228.
15. E.P. Gribomont, Concurrency without toil : a systematic method for parallel program design, *Sci. Comput. Programming* **21** (1993) 1-56.
16. E.P. Gribomont and D. Rossetto, CAVEAT : technique and tool for Computer Aided VErification And Transformation, *Lect. Notes in Comp. Sci.* **939** (1995) 70-83.
17. E.P. Gribomont, Preprocessing for invariant validation, *submitted to AMAST'96.*
18. R.M. Keller, Formal Verification of Parallel Programs, *C. ACM* **19** (1976) 371-384.
19. Y.S. Kwong, On reduction of asynchronous systems, *Th. Comp. Sci.* **15** (1977) 25-50.
20. L. Lamport, An Assertional Correctness Proof of a Distributed Algorithm, *Sci. Comput. Programming* **2** (1983) 175-206.
21. L. Lamport, *win* and *sin* : Predicate Transformers for Concurrency, *ACM Trans. Programming Languages Syst.* **12** (1990) 396-428.
22. L. Lamport, The Temporal Logic of Actions, DEC SRC Report 79, 1989.
23. L. Lamport and F.B. Schneider, Pretending Atomicity, DEC SRC Report 44, 1989.
24. L. Lamport and F.B. Schneider, The Reduction Theorem, unpublished TLA note, available on `http://www.research.digital.com/SRC/tla/notes.html` , 1992.
25. L. Lamport and al., Introduction, papers and notes about TLA, available on `http://www.research.digital.com/SRC/tla/` .
26. R.J. Lipton, Reduction : A method of proving properties of parallel programs, *Comm. ACM* **18** (1975) 717-721.
27. G.L. Peterson, Myths about the mutual exclusion problem, *Information Proc. Lett.* **12** (1981) 115-116.
28. J. Sifakis, A unified approach for studying the properties of transition systems, *Theoret. Comput. Sci.* **18** (1982) 227-259.
29. A. van Lamsweerde and M. Sintzoff, Formal derivation of strongly correct concurrent programs, *Acta Inform.* **12** (1979) 1-31.

Powerful Techniques for the Automatic Generation of Invariants

Saddek Bensalem[1]*, Yassine Lakhnech[2] **, and Hassen Saidi [1]***

[1] VERIMAG, Miniparc-Zirst, Rue Lavoisier 38330 Montbonnot St-Martin, France.
[2] Institut für Informatik und Praktische Mathematik Christian-Albrechts-Universität zu Kiel, Preußerstr. 1-9, D-24105 Kiel, Germany.

Abstract. When proving invariance properties of programs one is faced with two problems. The first problem is related to the necessity of proving tautologies of the considered assertion language, whereas the second manifests in the need of finding sufficiently strong invariants. This paper focuses on the second problem and describes techniques for the automatic generation of invariants. The first set of these techniques is applicable on sequential transition systems and allows to derive so-called local invariants, i.e. predicates which are invariant at some control location. The second is applicable on networks of transition systems and allows to combine local invariants of the sequential components to obtain local invariants of the global systems. Furthermore, a refined strengthening technique is presented that allows to avoid the problem of size-increase of the considered predicates which is the main drawback of the usual strengthening technique. The proposed techniques are illustrated by examples.

1 Introduction

Model checking [17, 4, 13, 20] is by now a well-known method for proving properties of reactive programs. The main reason for its success is that it works fully automatically, i.e. without any intervention of the user. The price to pay for this feature is that it can only be applied on finite-state, or restricted classes of infinite state, programs.

On the other hand, there exist deductive methods to prove safety properties of reactive programs. These methods are based on a proof rule which can be formulated as follows. To prove that some given predicate P is an *invariant* of a given program S, i.e. that every reachable state of S satisfies P, it is necessary and sufficient to find a predicate Q with the following properties: 1.) Q is stronger than P, 2.) Q is preserved by every transition of S, i.e. for every states s and s', if s satisfies Q and s' is reachable from s by a transition, then also s' satisfies Q, and 3.) Q is satisfied by every initial state of S. The predicate Q is called *auxiliary* predicate.

* Bensalem@imag.fr, Currently visiting the Computer Science Laboratory, SRI International
** yl@informatik.uni-kiel.de
*** Saidi@imag.fr

Although, this rule is sound and (relatively) complete, it provides only a partial answer to the verification problem of safety properties. For it leaves open (i) how to find the auxiliary predicate Q and (ii) how to prove that Q is preserved by every transition of S and satisfied by the initial states. Problem (ii) is related to the problem of proving tautologies of the underlying assertion language.

In this work, we describe techniques for automatically generating auxiliary predicates. We present the following strategies:

- *Generalized reaffirmed invariance:* This applies to transitions for which the value of the guard and of the expressions occurring on the right hand side of its assignment are not changed by the transition itself, i.e. they have the same value before and after the transition. This is more general than the one called reaffirmed invariants in [15, 14].
- *Propagation of invariants:* This technique allows to propagate an assertion that holds whenever control is at some fixed control location to other control locations. We consider two instances of this technique. The most general one allows to propagate even in the presence of loops. Again our technique is applicable in cases not covered by the propagation techniques presented in e.g. [15, 14].
- *Refined strengthening:* One of the most used techniques for strengthening invariants is by calculating the weakest (liberal) precondition [6] w.r.t. the considered invariant and taking it as a conjunct. A drawback of this method is that it increases the complexity of the considered predicate, and hence, after few steps its application leads in many cases to unmanageable predicates. We present a refined version of this method that allows to attenuate the blow up caused by applying this useful strengthening method.
- *Combining Invariants:* This method allows to combine invariants developed separately for the components of a given network $S_1 \parallel \cdots \parallel S_n$ of transition systems to an invariant of the global system.

All predicates that can be generated by these strategies are proved to be invariant by construction. The use of these techniques for various mutual exclusion algorithms shows that they are promising. For instance, in case of the Bakery algorithm [12, 15], which is an infinite-state program, we generate an invariant that is sufficiently strong to prove the required property.

It is also important to note that these techniques are local in the sense that, in order to apply them, they do not require the full transition system to satisfy some restrictions, but rather subsets of control locations and variables are required to satisfy some condition.

The problem of automatically constructing invariants from program description has been intensively investigated in the seventieth leading to results reported in e.g. [11, 9, 3, 7][3]. Here, we present results which are to our knowledge new or extensions of existing ones. Other interesting recent results are reported in [2].

These techniques represent an important component of a tool which is being developed to support the computer-aided verification of safety properties of

[3] This list of references is far from being exhaustive. See [15] for other references.

reactive programs. Here, we give a brief description of this tool (See [10] for a detailed discussion). It consists of the following components:

- **Front-end:** The front-end takes as input a description of a transition system written as a program in a simple programming language and a predicate to be proved as invariant of the described transition system. Then, it produces a PVS-theory [16] that mainly contains the verification conditions to be proved. The front-end analyses also the program and generates a file containing information needed to decide, for each control location, whether some invariant generation procedure can be applied.
- **Automatic Invariant Generation:** This is a module that contains procedures implementing several invariant generation techniques. In this paper, we present some of these techniques.
- **Proof Manager:** The user can try to prove that P is an invariant fully automatically. In this case, the system tries to prove that P is inductive, that is, P is preserved by each transition of the program. In case of success, this is reported to the user. Otherwise, the system tries to prove the invariance of P using predicates which are obtained by calling some invariant generating procedures. These predicates are guaranteed to be invariant by construction. In case the system is unable to prove the invariance of P, it may either do some strengthening or enter the interactive modus and requires the user's guidance. This choice is made by the user.
- **PVS** is the theorem prover developed at SRI [16]. It is used during the automatic- as well as interactive proof procedure to discharge the verification conditions.

2 Transition Systems and Invariance Properties

We assume an underlying assertion language $\mathcal{A}$ that includes first-order predicate logic and interpreted symbols for expressing the standard operations and relations over some concrete domains. We assume to have the set of integers among these domains. Assertions (we also say predicates) in $\mathcal{A}$ are interpreted in states that assign values to the variables of $\mathcal{A}$. Let Σ denote the set of states. Given a state s and a predicate P, we use the notation $s \models P$ to denote that s satisfies P, and use $[\![P]\!]$ to denote the set of states that satisfy P. Henceforth, we identify P and its characteristic set $[\![P]\!]$.

Definition 1. A transition system is a structure $S = \langle X, pc : DC, T, \mathrm{Init}\rangle$, where

- X is a finite set $\{x_1 : D_1, \ldots, x_n : D_n\}$ of typed data variables. Each variable x_i ranges over data domain D_i. We assume that the variables in X form a subset of those in $\mathcal{A}$.
- pc is a control variable (or program instruction counter). It ranges over the finite domain DC. We assume that $pc \notin X$.
- T is a finite set of transitions. A transition t is characterized by a quadruple $(pc = d, g(\mathbf{Y}), \mathbf{Z'} = \mathbf{e}(\mathbf{U}), pc' = d')$[4], where $\mathbf{Y}, \mathbf{Z}, \mathbf{U} \subseteq X$. The variables in

[4] $\mathbf{Z'}$ can be empty; this is the case when no variable is affected

$\mathbf{Z}$ are called the *variables affected* by transition t, and we denote by $sour(t)$ (resp. $tar(t)$) the value d (resp. d'). These definitions are easily generalized to sets of transitions. Given a transition $t = (pc = d, g(\mathbf{Y}), \mathbf{Z}' = \mathbf{e}(\mathbf{U}), pc' = d')$, and states s and s', s' is called *t-successor of s*, denoted by $s \rightarrow_t s'$, if the following conditions are satisfied: *1.) s satisfies the* enabledness condition $pc = d \wedge g(\mathbf{Y})$ *of transition t and 2.) s' satisfies $s'(z_i) = e_i(s(\mathbf{U}))$, for each $z_i \in \mathbf{Z}$, $s'(x) = s(x)$, for each x with $x \notin \mathbf{Z}$, and $s'(pc) = d'$.*

- Init is of the form $I(X) \wedge pc = d_0$. The conjunct $I(X)$ specifies the initial condition on data variables, whereas $pc = d_0$ specifies the initial value of the control variable. We call I *the initial predicate of* S and d_0 its *initial control location.*

A transition system generates a set of sequences of states. Since we are only interested in invariance properties, we only consider finite sequences. A finite sequence $\sigma = s_0, \cdots, s_n$ of states is called *computation of* S, if s_0 satisfies Init and, for every $i \in \{0, \ldots, n-1\}$, there exists a transition t in T with $s_i \rightarrow_t s_{i+1}$.

To define the semantics of the parallel construct, we define the product of two transition systems. Let $S_i = \langle X_i, pc_i : DC_i, T_i, \mathrm{Init}_i \rangle$, for $i = 1, 2$, be transition systems. The product of S_1 and S_2, denoted $S_1 \bigotimes S_2$, is a transition system $\langle X, pc : DC, T, \mathrm{Init} \rangle$, where

- $X = X_1 \cup X_2$ is the set of program variables.
- pc ranges over $DC = DC_1 \times DC_2$.
- A transition $(pc = (d_1, d_2), g(\mathbf{Y}), \mathbf{Z}' = \mathbf{e}(\mathbf{U}), pc = (d_1', d_2'))$ is in T iff either
  - $(pc_1 = d_1, g(\mathbf{Y}), \mathbf{Z}' = \mathbf{e}(\mathbf{U}), pc_1' = d_1') \in T_1$ and $d_2' = d_2$ or
  - $(pc_2 = d_2, g(\mathbf{Y}), \mathbf{Z}' = \mathbf{e}(\mathbf{U}), pc_2' = d_2') \in T_2$ and $d_1' = d_1$.
- $\mathrm{Init} = I_1 \wedge I_2 \wedge pc = (d_{1,0}, d_{2,0})$, where $\mathrm{Init}_i = I_i \wedge pc_i = d_{i,0}$, for $i = 1, 2$.

Then, the set of computations of $S_1 \parallel S_2$ is defined to be that of $S_1 \bigotimes S_2$.

Invariance Properties We consider a class of properties, named *invariance properties* (cf. [15]). Intuitively, a property P is an invariant of a transition system S, if in each state of the system S this property holds. In other words, each state that is reached during a computation of S satisfies P.

Definition 2. A state s is called *reachable* (accessible) in the transition system S, if there exists a computation $s_0, \cdots, s_n$ of S such that $s_n = s$. We denote the set of reachable states by $Reach(S)$. A predicate P is called *invariance property of S* (or *invariant* of S) iff $Reach(S) \subseteq [\![P]\!]$. For $d \in DC$, we say that P is *an invariant of S at d*, if $P \vee \neg(pc = d)$ is an invariant of S.

Next, we briefly recall the basic idea for proving invariance properties of programs. This idea underlies many proof rules formulated in different settings (e.g. [8, 1, 15]). To do so, we recall the definition of some predicate transformers.

Definition 3. Given $\rho \subseteq \Sigma \times \Sigma$, the predicate transformers $pre[\rho]$, $\widetilde{pre}[\rho]$, and $post[\rho]$ are defined by $pre[\rho](P) = \{s \in \Sigma \mid \exists s' \in P \cdot (s, s') \in \rho\}$, $\widetilde{pre}[\rho](P) = \neg pre[\rho](\neg P)$, and $post[\rho](P) = \{s' \in \Sigma \mid \exists s \in P \cdot (s, s') \in \rho\}$

Thus, $pre[\rho](P)$ is the set of predecessors of P by ρ, $post[\rho](P)$ is the set of successors of P, and $\widetilde{pre}[\rho](P)$ is the set of states which either do not have successors by ρ or all their successors are in P. Note that the $\widetilde{pre}[\rho]$ and $post[\rho]$ are the *weakest liberal precondition* and *strongest postcondition* predicate transformers [6].

The main principle used in the literature for proving that a predicate P is an invariant of a system S, consists on finding an *auxiliary* predicate Q such that *1.) Q is stronger than P, 2.) every initial state satisfies Q, and 3.) Q is* inductive, *i.e. for all transitions $t \in T$, we have $[\![Q]\!] \subseteq \widetilde{pre}[\rightarrow_t](Q)$, or equivalently, $post[\rightarrow_t](Q) \subseteq [\![Q]\!]$.*

This proof rule is unsatisfactory because it does not tell us how to find the auxiliary predicate Q. Finding Q is often the hard part in the proof of invariance properties.

In the next section, we present a set of techniques that, given a transition system S and a predicate P, automatically generate an auxiliary predicate that is by construction an invariant. In some cases, the generated predicate is strong enough to prove that P is an invariant.

3 Automatic Generation of Auxiliary Predicates

In this section we present some of the strategies for deriving auxiliary predicates we implemented in our tool. We concentrate on strategies which are to our knowledge new or extensions of strategies presented in other works (e.g. [9, 11, 15, 14, 2]). The auxiliary predicates derived using our strategies are proved to be invariant by construction.

Generalized Reaffirmed Invariance without Cycles We begin with a strategy that can be applied to a control location d to derive an invariant under the assumption that all transitions that lead to d satisfy some restrictions we define below. This is a generalization of the reaffirmed invariance strategy presented in [15, 14].

Let $S = \langle X, pc : DC, T, I \wedge pc = d_0 \rangle$ be given. For $\alpha \subseteq DC$, let $L(\alpha)$ denote the set of transitions t with $tar(t) \in \alpha$. Thus, $L(\alpha)$ is the set of transitions changing the value of the control variable to a value in α. We write $L(d)$ instead of $L(\{d\})$.

Consider a transition $t = (pc = d_1, g(\mathbf{Y}), \mathbf{Z'} = \mathbf{e}(\mathbf{U}), pc' = d)$, with $\mathbf{Z} \cap \mathbf{U} = \emptyset$. Then, for every states s and s', if $s \rightarrow_t s'$, then $s'(\mathbf{Z}) = \mathbf{e}(s'(\mathbf{U}))$ and $s'(\mathbf{U}) = s(\mathbf{U})$. This suggests to take the predicate $\mathbf{Z} = \mathbf{e}$ as invariant at d.

To formulate the general case, given a transition t as above, we denote by $aff(t)$ the predicate $\mathbf{Z} = \mathbf{e}(\mathbf{U})$ and by $gu(t)$, the guard $g(\mathbf{Y})$. Let, for $d \in DC$, $Ass_S(d) = \bigvee_{t \in L(d)} (gu(t) \wedge aff(t))$, if $d \neq d_0$; and $I \vee \bigvee_{t \in L(d)} (gu(t) \wedge aff(t))$, if $d = d_0$, where I is the initial predicate of S and d_0 its initial control location.

Lemma 4. *Let S be a given transition system with $Init = I \wedge pc = d_0$ and let $D \subseteq DC$ be such that for each $d \in D$ and transition $(pc = d_1, g(\mathbf{Y}), \mathbf{Z'} = \mathbf{e}(\mathbf{U}), pc' = d)$ in $L(d)$ we have $\mathbf{Z} \cap (\mathbf{Y} \cup \mathbf{U}) = \emptyset$. Then, for each $d \in D$, the predicate $Ass_S(d)$ is an invariant of S at d.*

We can actually formulate a strategy that generalizes the one above by relaxing the condition $\mathbf{Z} \cap (\mathbf{Y} \cup \mathbf{U}) = \emptyset$. Let $Ass'_S(d)$ be defined as in Figure 1. Then, for each $d \in DC$, $Ass'_S(d)$ is an invariant of S at d. Henceforth, let **aff-indep** denote

$$Ass'_S(d) = \begin{cases} \bigvee_{t \in L(d)} (gu(t) \wedge aff(t)) & ; \text{if } d \neq d_0 \text{ and } \mathbf{Z} \cap (\mathbf{Y} \cup \mathbf{U}) = \emptyset \\ I \vee \bigvee_{t \in L(d)} (gu(t) \wedge aff(t)) & ; \text{if } d = d_0 \text{ and } \mathbf{Z} \cap (\mathbf{Y} \cup \mathbf{U}) = \emptyset \\ \bigvee_{t \in L(d)} aff(t) & ; \text{if } d \neq d_0,\ \mathbf{Z} \cap \mathbf{U} = \emptyset \text{ and } \mathbf{Z} \cap \mathbf{Y} \neq \emptyset \\ I \vee \bigvee_{t \in L(d)} aff(t) & ; \text{if } d = d_0,\ \mathbf{Z} \cap \mathbf{U} = \emptyset \text{ and } \mathbf{Z} \cap \mathbf{Y} \neq \emptyset \\ \bigvee_{t \in L(d)} gu(t) & ; \text{if } d \neq d_0,\ \mathbf{Z} \cap \mathbf{Y} = \emptyset \text{ and } \mathbf{Z} \cap \mathbf{U} \neq \emptyset \\ I \vee \bigvee_{t \in L(d)} gu(t) & ; \text{if } d = d_0,\ \mathbf{Z} \cap \mathbf{Y} = \emptyset \text{ and } \mathbf{Z} \cap \mathbf{U} \neq \emptyset \\ true & ; \text{otherwise} \end{cases}$$

Fig. 1. Definition of $Ass'_S(d)$

the function that for a given transition system S returns as result the predicate $\bigwedge_{d \in D} pc = d \Rightarrow Ass'_S(d)$.

Generalized Reaffirmed Invariance with Cycles Consider the situation described in Figure 2. Then, function **aff-indep** yields the predicate $x = 2 \vee y = 1$ as invariant at d. It is easy to see, however, that the stronger predicate $x = 2$ is also invariant at d. We develop a technique that extends the previous one and covers situations similar to that of Figure 2.

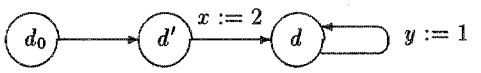


Fig. 2. Generalized Reaffirmed Invariance

A *path* from d to d' in S is a sequence $d_1, t_1, \cdots, t_{n-1}, d_n$ with $n \geq 2$, $d_1 = d$, and $d_n = d$. We say that a path $d_1, t_1, \cdots, t_{n-1}, d_n$ from d to d' *goes through* d'', if $d_i = d''$, for some $i \in \{1, \cdots, n\}$.

Definition 5. Given a transition system S, a control location d of S, and a set α of control locations of S with $d \in \alpha$. We say that α is *guarded by* d, if the following conditions are satisfied:

- The initial control location of S is not in α or it is d.
- For every transition $t \in L(\alpha) \setminus \{d\}$, $sour(t) \in \alpha$.
- Each path from d to $d' \in \alpha$ goes only through control locations in α.

Let $Tr(S, \alpha, d)$ denote the set $L(\alpha) \setminus \{t \mid t \in L(d), sour(t) \notin \alpha\}$.

Example 1. Consider the system S given in Figure 3, where d_0 is the initial control location. Then, $\alpha_1 = \{d_1, d_2, d_3, d_4, d_5\}$ and $\alpha_2 = \{d_1, d_4, d_5\}$ are guarded by d_1, while $\alpha_3 = \{d_1, d_2, d_4, d_5\}$ and $\alpha_4 = \{d_1, d_2, d_3\}$ are not because the second respectively third condition are violated. We have $Tr(S, \alpha_1, d_1) = \{t_1, t_2, t_3, t_4, t_5, t_6\}$ and $Tr(S, \alpha_2, d_1) = \{t_4, t_5, t_6\}$.

Definition 6. Given a transition system S and $d \in DC$. We say that d is *safe* with respect to a set V of variables and a set α of control locations, if α is guarded by d and for every $t \in Tr(S, \alpha, d)$, t does not affect any variable in V.

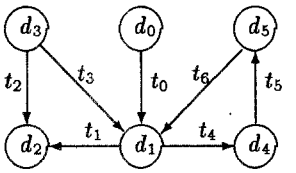


Fig. 3.

Then we have the following lemma.

Lemma 7. *Consider a transition system S, a control location d, a set α of control locations, and a set V of variables such that d is safe w.r.t. V and α. Let S' denote the transition system obtained from S by removing the transitions in $Tr(S, \alpha, d)$. For every predicate Q with free variables V, if Q is an invariant of S' at d, then Q is an invariant of S at every $d' \in \alpha$.*

The lemma above suggests a procedure to derive an invariant **aff-cyc**(S) from the description of the transition system S: For each $d \in DC$, determine a maximal set α of control locations for which d is safe with respect to the variables affected by transitions in $\{t \mid t \in L(d), sour(t) \notin \alpha\}$; in case d is the initial control location, we have to check also w.r.t. the free variables of I. If this is the case, record $Ass'_{S'}(d)$, where S' is as above, as an invariant of S at d' for each $d' \in \alpha$, otherwise, record $Ass'_S(d)$ as an invariant of S at d.

Remark.

1. A possible variant of the algorithm **aff-cyc** concerns the case where the initial control location is considered. Instead of requiring that d is safe w.r.t. the free variables of I, we hide those which could be affected by some transition in $Tr(S, \alpha, d)$ by existential quantification.
2. Clearly, determining the maximal set α which is guarded by d and then checking whether d is safe w.r.t. this set and the variables affected by transitions in $\{t \mid t \in L(d), sour(t) \notin \alpha\}$ does not always allow to derive the strongest possible predicate. One can, however, have a procedure which depends on some given set V of variables and which computes the maximal set α such that d is safe w.r.t. V and α.
3. Until now we considered a single transition system S and **aff-cyc** has been formulated for this case. When n transition systems $S_1 \parallel \cdots \parallel S_n$ in parallel are considered, we have to strengthen the notion of d being safe w.r.t. a set V of variables and a set α of control locations; and require that all variables in V are only written by the system S_i to which d belongs. Henceforth, whenever we refer to **aff-cyc** when a parallel program is considered, we mean the algorithm obtained by strengthening this notion and taking into account the variation suggested in 1.

Next, we present a technique that allows to propagate predicates that have been proved to be invariant at some control points of the system, i.e. for some value of pc. We first start with the basic idea.

Propagation without cycles Given a transition system S, a predicate Q with $\mathbf{V}$ as free variables and a transition t of S, we say that transition t *does not affect* Q, if $\mathbf{Z} \cap \mathbf{V} = \emptyset$, where $\mathbf{Z}$ are the variables affected by t.

Consider a transition system S and a control location $d \in DC$ which is not the initial one. Let $\{d_1, \cdots, d_n\} = sour(L(d))$ and assume that, for each $i \in \{1, \cdots, n\}$, $Q_i(\mathbf{V_i})$ is an invariant of S at d_i. If for each $t \in L(d)$ and $i \in \{1, \cdots, n\}$, with $sour(t) = d_i$, t does not affect $\mathbf{V_i}$, then $\bigvee_{i=1}^{n} Q(\mathbf{V_i})$ is an invariant at d. For the case where d is the initial control location, $\bigvee_{i=1}^{n} Q(\mathbf{V_i}) \vee I$, where I is the initial predicate, is an invariant at d. The correctness of this observation is guaranteed by the following lemma.

Lemma 8. *Consider a transition system S and a predicate P that is an invariant of S. Let $d \in DC$ be a control location of S with $L(d) = \{t_1, \cdots, t_m\}$ and $d_i = sour(t_i)$. Let also $Q_1, \cdots, Q_m$ be predicates such that $P \wedge pc = d_i$ implies Q_i, with $i = 1, \cdots, m$. If d is not the initial control location of S, then the predicate $P \wedge (l = d \Rightarrow \bigvee_{i=1}^{m} post[\rightarrow_{t_i}](Q_i))$ is an invariant of S, otherwise $P \wedge (l = d \Rightarrow (\bigvee_{i=1}^{m} post[\rightarrow_{t_i}](Q_i) \vee I))$ is an invariant of S.*

Note that in case that transition t does not affect Q, we have $post[\rightarrow_t](Q) \Rightarrow Q$, and therefore, the correctness of our technique is implied by the lemma above and the fact that if P' is an invariant of S and P' implies Q', then Q' is also an invariant of S.

The implementation of this technique is a function, denoted **propg**, that takes as input a transition system S and a predicate P of the form $\bigwedge_{d \in DC} pc = d \Rightarrow Q_d(\mathbf{V}'_d)$. Then, computes for each control location d, the set of variables affected by any transition in $L(d)$. Let $\mathbf{V}_d$ denote the intersection of this set with $\mathbf{V}'_d$. As result, this function yields, for each control location d, as a local invariant at d the predicate $Q_d(\mathbf{V}'_d) \wedge \exists \mathbf{V}_d \cdot \bigvee_{d' \in L(d)} Q_{d'}(\mathbf{V}'_{d'})$.

Propagation with cycles Consider now the situation described in Figure 4. An application of the simple propagation technique does not allow to strengthen the predicate $\bigwedge_{i=1}^{m} pc = d_i \Rightarrow x = i$. For, we would add as a conjunct the predicate $pc = d \Rightarrow true \vee \bigvee_{i=1}^{m} x = i$, which is equivalent to true. Yet, it is clear that $\bigvee_{i=1}^{m} x = i$ is an invariant at d. We develop the next technique which captures similar situations.

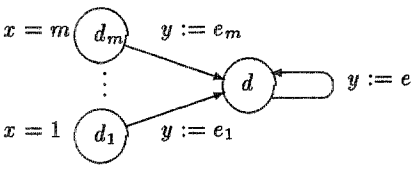


Fig. 4. Propagation with cycles

Consider a control location d and a set α of control locations which is guarded by d. Let $\{d_1, \cdots, d_m\} = sour(L(d)) \setminus \alpha$. Then, if for each $i = 1, \cdots, m$, $Q_i(\mathbf{V_i})$ is an invariant of S at d_i and if d is safe w.r.t. $\bigcup_{i=1}^{m} \mathbf{V_i}$ and α, we can conclude by Lemma 7 and Lemma 8 that $\bigvee_{i=1}^{m} Q_i(\mathbf{V_i})$ is an invariant at each $d' \in \alpha$.

Mixing generalized reaffirmed invariance and propagation Until now we considered propagation and reaffirmed invariance separately. Whereas propagation assumes a given invariant P and propagates local invariants from control locations

to others, reaffirmed invariance does not assume such a predicate. We now present a technique that combines propagation and reaffirmed invariance.

Consider a transition system S and an invariant P of S. Let d be a control location of S such that $\{t_1, \cdots t_m\} = L(d)$ and $d_i = sour(t_i)$, for $i = 1, \cdots, m$. Suppose that for each $i = 1, \cdots, m$, $P \wedge pc = d_i$ implies $Q_i(\mathbf{V_i})$. If, for each transition t_i and each j with $d_j = sour(t_i)$, $Q_i(\mathbf{V_i})$ implies $\mathbf{e}(\mathbf{U_i}) = \mathbf{C_i}$ and $\mathbf{Z_i} \cap \mathbf{V_j} = \emptyset$, where $\mathbf{Z_i}' = \mathbf{e}(\mathbf{U_i})$ is $aff(t_i)$ and $\mathbf{C}$ is a list of constants, then we can conclude that $\bigvee_{i=1}^{m}(Q_i(\mathbf{V_i}) \wedge \mathbf{Z_i} = \mathbf{C_i})$ is invariant at d. Correctness of this observation is again a consequence of Lemma 8 and Lemma 8.

Refined Strengthening Suppose we are given a proposed invariant P for transition system S with transitions T. Suppose also that the proof of $P \Rightarrow \widetilde{pre}[\rightarrow_t](P)$ fails for $t_1, \cdots, t_m$. The method of strengthening invariants (e.g. [15]) proposes to try as next invariant $P_1 = P \wedge \bigwedge_{i=1}^{m} \widetilde{pre}[\rightarrow_{t_i}](P)$. Thus, one has to try to prove for each transition t the implication $P \wedge Q \Rightarrow \widetilde{pre}[\rightarrow_t](P \wedge Q)$, where $Q = \bigwedge_{i=1}^{m} \widetilde{pre}[\rightarrow_{t_i}](P)$. The main drawback of this method is that, in general, each strengthening step increases the size of the considered invariant which in some cases leads to unreadable predicates.

We propose a variant of this method that is theoretically equivalent, i.e. it leads to logically equivalent verification conditions, but which allows to reduce the number of applications of $\widetilde{pre}$ and to save redoing proofs.

Suppose that the attempt of proving $\forall t \in T \cdot (P \Rightarrow \widetilde{pre}[\rightarrow_t](P))$ fails for the transitions $t_1, \cdots, t_m$, and that one gets subgoals $Q_1, \cdots, Q_m$, which are logically equivalent to $P \Rightarrow \widetilde{pre}[\rightarrow_{t_i}](P)$, $i = 1, \cdots, m$. We propose to take in the next step the predicate $P_1' = P \wedge \bigwedge_{i=1}^{m} Q_i$ instead of P_1. The next lemma implies soundness of our method but also proves that if P_1 is inductive, then also P_1'.

Lemma 9. *Let $P_1 = P \wedge \bigwedge_{i=1}^{m} \widetilde{pre}[\rightarrow_{t_i}](P)$, Q_i be equivalent to $P \Rightarrow \widetilde{pre}[\rightarrow_{t_i}](P)$, and let $P_1' = P \wedge \bigwedge_{i=1}^{m} Q_i$. Then, P_1 and P_1' are equivalent.*

It is worth to note that soundness of our method does not depend on the fact that Q_i is equivalent to $P \Rightarrow \widetilde{pre}[\rightarrow_{t_i}](P)$ but it suffices, if it is stronger.

To see that our method indeed avoids the blow-up of the considered predicates which is due to the repeated application of the predicate transformer $\widetilde{pre}$, let us look at the predicates to be considered at step i when each of the strengthening and refined strengthening methods are applied in turn. In case of the strengthening method, one has to consider at step i the predicate $P_i = P_0 \wedge \widetilde{pre}(P_0) \wedge \cdots \widetilde{pre}^i(P_{i-1})$ and to prove $P_i \Rightarrow \widetilde{pre}(P_i)$. In case of the refined strengthening method, however, one has to consider the predicate Q_i which is obtained as a subgoal in step i, and then, to prove $Q_0 \wedge \cdots \wedge Q_i \Rightarrow \widetilde{pre}(Q_i)$. Thus, in the refined strengthening method, at each step $\widetilde{pre}$ has to be applied only once. Another advantage of this method is that Q_i is usually of the form $pc = d \Rightarrow Q$ which can be explained by the fact that Q_i is the predicate that is obtained when the proof of $Q_0 \wedge \cdots \wedge Q_{i-1} \Rightarrow \widetilde{pre}(Q_{i-1})$ for some fixed transition with $pc = d$ as part of the enabling condition has been attempted. Now, when a predicate Q of the form $pc = d \Rightarrow Q'$ is considered in order to prove that Q is preserved by all transitions, it suffices to consider only those in $L(d)$.

Combining Invariants Consider a network $S = S_1 \parallel \cdots \parallel S_n$ of transition systems. Given a predicate P, in order to prove that P is an invariant of S, one can calculate the product $S_1 \bigotimes \cdots \bigotimes S_n$ and then prove that P is an invariant of the resulting sequential transition system. This method is, however, not applicable for large transition systems because of the big size of the obtained system. Indeed, the resulting transition system mainly codes all possible interleaving of the transition steps in the network S. In this section, we present techniques we use to prove invariance properties of networks without calculating the product. These techniques have been successfully applied to many mutual exclusion algorithms, e.g. the Bakery mutual exclusion algorithm [12, 15] in three different versions and Szymanski's mutual exclusion algorithm [18, 19] both parameterized and for two processes.

Definition 10. Given a transition system S, a predicate P is called *history-independent assertion at $d \in DC$*, if $post[t](true) \subseteq [\![P]\!]$ holds for each $t \in L(d)$, and moreover, if d is the initial control location of S, then *Init* implies P.

An history-independent assertion at d is true whenever computation reaches d independently on how this happens, in particular it does not dependent on the state in which the transition is taken.

Consider transition systems S_1 and S_2 with $S_i = \langle X_i, pc_i : DC_i, T_i, I_i \wedge pc_i = d_{i,0} \rangle$, for $i = 1, 2$. Moreover, consider predicates Q_i, for $i = 1, 2$, and $(d_1, d_2) \in DC_1 \times DC_2$. Assume we know that Q_i is an history-independent assertion at d_i. Then, we can conclude that $Q_1 \vee Q_2$ is an invariant of $S_1 \parallel S_2$ at (d_1, d_2). This leads to the following heuristic formulated in the next lemma.

Lemma 11. *Let $S_i = \langle X_i, pc_i : DC_i, T_i, I_i \wedge pc_i = d_{i,0} \rangle$, for $i = 1, 2$, be transition systems and let Q_i be predicates. Then, for each $(d_1, d_2) \in DC_1 \times DC_2$ such that Q_i is an history-independent assertion of S_i at d_i, for $i = 1, 2$, the predicate $Q_1 \vee Q_2$ is an invariant of $S_1 \parallel S_2$ at (d_1, d_2).*

If the predicates Q_1 and Q_2 constraint only variables which are affected only in S_1, respectively, S_2, then we can even conclude that the stronger predicate $Q_1 \wedge Q_2$ is an invariant at (d_1, d_2).

The implementation of both observations above is realized by a single function **comp** which takes as arguments the transition systems S_1 and S_2 as well as two predicates P_1 and P_2 for S_1 and S_2, respectively, which are of the form $\bigwedge_{d_j \in DC_i} pc = d_j \Rightarrow P_i(d_j)$, $i = 1, 2$. The result of the application of this function is a predicate of the form $\bigwedge_{d \in DC} pc = d \Rightarrow Q(d)$, where $DC = DC_1 \times DC_2$ and for $d = (d_1, d_2)$, $Q(d)$ is defined in Figure 3.

Remark. It is worth to note that each invariant Q obtained by applying the function **aff-indep** is history-independent.

In a concrete implementation, the predicate obtained by an application of the function **comp**, can be encoded by adding to each local invariant $P_i(d_i)$ at d_i two bits. The first one encodes whether $P_i(d_i)$ is history-independent and the second whether it refers to a variable affected in S_j with $j \neq i$.

$$Q(d) = \begin{cases} P_1(d_1) \vee P_2(d_2) & \text{; if for } i = 1,2,\ P_i \text{ is an history-independent assertion at} d_i \\ & \quad \text{and one of the predicates } P_1 \text{ or } P_2 \text{ refers to a variable} \\ & \quad \text{affected in } S_2 \text{ respectively } S_1 \\ P_1(d_1) \wedge P_2(d_2) & \text{; if for } i = 1,2,\ P_i \text{ is an history-independent assertion at } d_i \\ & \quad \text{and predicate } P_1 \text{ respec. } P_2 \text{ does not refer to any variable} \\ & \quad \text{affected in } S_2 \text{ respec. } S_1 \\ \mathit{true} & \text{; otherwise} \end{cases}$$

Fig. 5. Definition of **comp**

The next lemma shows how given $d_i' \in DC_i$ and a predicate Q that is history-independent at d_i', we can deduce a predicate Q' which is also history-independent at d_i' and which does not refer to variables affected in S_j with $j \neq i$.

Lemma 12. *Let S_1 and S_2 be transition systems and let $d_1 \in DC_1$ (resp. $d_2 \in DC_2$) be a control location of S_1 (resp. S_2). If Q is a history-independent assertion at d and $\mathbf{Y}$ are the variables occurring in Q which are affected in S_2 (resp, S_1), then $\exists \mathbf{Y} \cdot Q$ is a history-independent assertion at d.*

Clearly, the predicate $\exists \mathbf{Y} : \mathbf{D} \cdot Q$ does not refer to variables affected in S_j. Let **abst** be a function that takes as arguments two transition systems S_1 and S_2 and a predicate P for S_1, and returns a predicate Q for S_1 such that Q is obtained from P by applying the observation above.

Next we present the tactic we apply to synthesize an invariant from a given network $S_1 \parallel S_2$. This is presented by an algorithm written in pseudo-code and which uses the heuristics presented above.

Input: $S_1 \parallel S_2$
Output: An invariant

1. $P_i :=$ **aff-indep**(S_i); for $i = 1, 2$
2. $P :=$ **comp**(S_1, S_2, P_1, P_2)
3. $Q_1 :=$ **abst**(S_1, S_2, P_1), $Q_2 :=$ **abst**(S_2, S_1, P_2)
4. $Q_i := Q_i \wedge$ **propg**(S_i, Q_i), for $i = 1, 2$
5. return $P \wedge Q_1 \wedge Q_2$

4 Example

The example we consider is the Bakery mutual exclusion algorithm [12, 15]. Two processes are competing to enter their respective critical sections represented by location 4. Thus, the invariant we are going to prove is given by the predicate $INV = \neg(pc_1 = 4 \wedge pc_2 = 4)$.

It can easily be checked that this invariant is not inductive. Moreover, calculating the set of reachable states using the *post* operator does not terminate (no fix-point can be reached in a finite number of steps). Calculating the weakest invariance property that is contained in INV does terminate after 8 steps (cf. [14]). We can automatically generate by our techniques an invariant that is inductive and that allows to prove that INV is indeed an invariant.

Transition system S_1	Transition system S_2
$pc_1 = 1 \longrightarrow pc_1' = 2$	$pc_2 = 1 \longrightarrow pc_2' = 2$
$pc_1 = 2 \longrightarrow y_1' = y_2 + 1, pc_1' = 3$	$pc_2 = 2 \longrightarrow y_2' = y_1 + 1, pc_2' = 3$
$pc_1 = 3 \wedge (y_2 = 0 \vee y_1 \leq y_2) \rightarrow pc_1' = 4$	$pc_2 = 3 \wedge (y_1 = 0 \vee y_2 < y_1) \rightarrow pc_2' = 4$
$pc_1 = 4 \longrightarrow pc_1' = 5$	$pc_2 = 4 \longrightarrow pc_2' = 5$
$pc_1 = 5 \longrightarrow y_1' = 0, pc_1' = 1$	$pc_2 = 5 \longrightarrow y_2' = 0, pc_2' = 1$
$Init = (y_1 = y_2 = 0 \wedge pc_1 = pc_2 = 1)$	

Applying generalized reaffirmed invariance without cycles for S_1 (resp. S_2) yields the predicate P_1 (resp. P_2) with:

$P_1 = (pc_1 = 1 \Rightarrow y_1 = 0 \vee y_1 = 0 \wedge y_2 = 0) \wedge (pc_1 = 3 \Rightarrow y_1 = y_2 + 1) \wedge$
$(pc_1 = 4 \Rightarrow y_2 = 0 \vee y_1 \leq y_2)$

$P_2 = (pc_2 = 1 \Rightarrow y_2 = 0 \vee y_1 = 0 \wedge y_2 = 0) \wedge (pc_2 = 3 \Rightarrow y_2 = y_1 + 1) \wedge$
$(pc_2 = 4 \Rightarrow y_1 = 0 \vee y_2 < y_1)$

Combining the predicates P_1 and P_2 according to function **comp** results in a predicate equivalent to

$P = (pc = (1,1) \Rightarrow y_1 = 0 \vee y_2 = 0) \wedge (pc = (1,3) \Rightarrow y_1 = 0 \vee y_2 = y_1 + 1) \wedge$
$(pc = (1,4) \Rightarrow y_1 = 0 \vee y_2 < y_1) \wedge (pc(3,1) \Rightarrow y_1 = y_2 + 1 \vee y_2 = 0) \wedge$
$(pc = (3,3) \Rightarrow y_1 = y_2 + 1 \vee y_2 = y_1 + 1) \wedge (pc = (3,4) \Rightarrow y_1 = 0 \vee y_2 < y_1) \wedge$
$(pc = (4,1) \Rightarrow y_2 = 0 \vee y_1 \leq y_2) \wedge (pc = (4,3) \Rightarrow y_2 = 0 \vee y_2 < y_1)$

In the sequel, we write $pc_1 = d_1 \wedge pc_2 = d_2$ for $pc = (d_1, d_2)$.

Next, we apply the abstraction function **abst** on P_1 and P_2 to obtain:

$Q_1 = (pc_1 = 1 \Rightarrow y_1 = 0) \wedge (pc_1 = 3 \Rightarrow y_1 \geq 1)$

$Q_2 = (pc_2 = 1 \Rightarrow y_2 = 0) \wedge (pc_2 = 3 \Rightarrow y_2 \geq 1)$

Then, we apply our propagation technique without cycles. It can easily be checked that we can propagate from control location 1 to 2, from 3 to 4, and from 4 to 5, which yields the following predicates:

$Q_1' = (pc_1 = 1 \vee pc_1 = 2 \Rightarrow y_1 = 0) \wedge (pc_1 = 3 \vee pc_1 = 4 \vee pc_1 = 5 \Rightarrow y_1 \geq 1)$

$Q_2' = (pc_2 = 1 \vee pc_2 = 1 \Rightarrow y_2 = 0) \wedge (pc_2 = 3 \vee pc_2 = 4 \vee pc_2 = 5 \Rightarrow y_2 \geq 1)$

Then, we can show $P \wedge Q_1' \wedge Q_2' \wedge INV \Rightarrow \widetilde{pre}[\rightarrow_t](INV)$, for each transition t of $S_1 \parallel S_2$.

5 Discussion and Future Work

This paper provides a set of techniques for the automatic generation of auxiliary predicates to prove invariants of programs. The use of these heuristics for the verification of various mutual exclusion algorithms shows that they are promising. They have been applied to different versions of the Bakery, Dekker, Peterson, and Szymanski algorithms (see [15] for a recent presentation of many of these algorithms and for references). Concerning Szymanski's mutual exclusion algorithm, we verified the parameterized as well as the unparameterized case. We intend to combine our techniques with others as abstract interpretation [5] to discover relationships between program variables that can be used to derive invariants and to investigate heuristics and strategies for the decomposition of large programs.

Acknowledgements We thank J. Sifakis who continuously encouraged and supported this work. Many interesting discussions with S. Graf and A. Pnueli helped clarifying and fixing our ideas. We also thank the anonymous referees for judecious comments.

References

1. K.R. Apt. Ten years of Hoare's logic : a survey, part I. *ACM Trans. on Prog. Lang. and Sys.*, 3(2):431–483, 1981.
2. N. Bjøner, A. Browne, and Z. Manna. Automatic generation of invariants and intermediate assertions. In U. Montanari, editor, *1st Int. Conf. on Principles and Practice of Constraint Programming*, 1995.
3. M. Caplain. Finding invariant assertions for proving programs. In *Proc. Int. Conf. on Reliable Software*, Los Angeles, CA, 1975.
4. E.M. Clarke, E.A. Emerson, and E. Sistla. Automatic verification of finite state concurrent systems using temporal logic specifications: A practical approach. In *POPL'83*. ACM, 1983.
5. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *4th ACM symp. of Prog. Lang.*, pages 238–252. ACM Press, 1977.
6. E. W. Dijkstra. Guarded commands, nondeterminacy, and formal derivation. *Comm. ACM*, 18(8):453–457, 1975.
7. B. Elspas. The semiautomatic generation of inductive assertions for proving program correctness. Research report, SRI, Menlo Park, CA, 1974.
8. R. W. Floyd. Assigning meanings to programs. In *In. Proc. Symp. on Appl. Math. 19*, pages 19–32. American Mathematical Society, 1967.
9. S. M. German and B. Wegbreit. A synthesizer of inductive assertions. *IEEE Trans. On Software Engineering*, 1:68–75, March 1975.
10. S. Graf and H. Saidi. Verifying invariants using theorem proving. In *In this volume*, 1996.
11. S. Katz and Z. Manna. A heuristic approach to program verification. In *Proc. 3rd Int. Joint Conf. on Artificial Intelligence*, Stanford,CA, 1976.
12. L. Lamport. A new solution of Dijkstra's concurrent programming problem. *Comm. ACM*, 17(8):453–455, 1974.
13. O. Lichtenstein and A. Pnueli. Checking that finite state concurrent programs satisfy their linear specification. In *POPL*, pages 97–107, 1985.
14. Z. Manna, A. Anuchitanukul, N. Bjøner, A. Browne, E. Chang, M. Colon, L. De Alfaro, H. Devarajan, H. Sipma, and T. Uribe. STeP : The Stanford Temporal Prover. Technical report, Stanford Univ., Stanford, CA, 1995.
15. Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems: Safety.* Springer-Verlag, 1995.
16. S. Owre, J. Rushby, N. Shankar, and F. von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 1995.
17. J. P. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *Proc. 5th Int. Sym. on Programming*, volume 137 of *Lecture Notes in Computer Science*, pages 337–351. Springer-Verlag, 1982.
18. B. K. Szymanski. A simple solution to Lamport's concurrent programming problem verification. In *Proc. Intern. Conf. on Supercomputing Sys.*, pages 621–626, 1988.
19. B. K. Szymanski and J. M. Vidal. Automatic verfication of a class of symmetric parallel programs. In *Proc. 13th IFIP World Computer Congress*, 1994.
20. M.Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *LICS'86*. IEEE, 1986.

Saving Space by Fully Exploiting Invisible Transitions

Hillel Miller* and Shmuel Katz**

Department of Computer Science
The Technion
Haifa, Israel
hillelm,katz@cs.technion.ac.il

Abstract. Checking that a given finite state program satisfies a linear temporal logic property suffers from a severe space and time explosion. One way to cope with this is to reduce the state graph used for model checking. We present an algorithm for constructing a state graph that is a projection of the program's state graph. The algorithm maintains the transitions and states that affect the truth of the property to be checked. The algorithm works in conjunction with a partial order reduction algorithm. We show a substantial reduction in memory over current partial order reduction methods, both in the precomputation stage, and in the result presented to a model checker, with a price of a single additional traversal of the graph obtained with partial order reduction. As part of our space-saving methods, we present a new way to exploit Holzmann's Bit Hash Table, which assists us in solving the revisiting problem.

1 Introduction

In order to reduce the space needed for model checking of linear temporal logic properties, several pre-processing techniques have been suggested to construct smaller graphs such that a property to be checked is true of the original state graph iff it is true of the reduced graph.

In particular, partial order methods such as [GW91],[Val90], and [Pel94] exploit the fact that certain operations are independent of other operations, and that not all interleavings of independent operations need to be explicitly examined. Here, we exploit the fact that the specification –the property to be proven– is independent of certain operations to obtain a further reduction. Invisible operations are those that do not affect the truth of any of the atomic propositions of the specification, while visible operations do affect them. A node is considered visible if some edge corresponding to a visible operation enters it, and invisible otherwise. We also exploit the fact that an operation can be invisible or visible, depending on the state from which it is executed. In this paper, a program's projected visible state space relative to a specification is constructed through a

* Presently with Motorola Semiconductors–Israel, Herzlia, Israel
** Supported by the Technion V.P.R. Fund–Promotion of Sponsored Research

DFS traversal, and the invisible states are eliminated. Thus we present to the model checker a much smaller structure that represents the program.

The construction of the visible state space requires a linear traversal of a state graph that is somewhat reduced from the original, but can still be large in some cases. This is still worthwhile because a standard temporal logic model checker requires space and time complexity which is the multiplication of the size of the state space by a term exponential in the length of the formula. Thus for a formula of length 20, the time and memory complexity for the model checker are multiplied by 10^6. We are therefore motivated to reduce the state graph given to the model checker.

Moreover, we will show that the reduced structure can be produced with a low space overhead. During any such pre-processing, and also during a traversal of the state-space for purposes of reachability analysis and deadlock detection, the question arises of whether to record for future reference that a particular state was already visited. Not indicating that a state was visited saves space, but may be costly in time: if the state is later reached again along another path, its descendants must be recomputed unnecessarily. We can define the revisiting degree of a state as the number of incoming edges not including those that close loops. (Those that close loops are on the stack used for a depth-first traversal, and thus are easily identified with no additional space needed). The question is whether a state should be identifiable as having already been visited even after backtracking from that state, in the DFS traversal. For graphs with many states having a large revisiting degree, the time can increase exponentially if states are reexpanded each time they are reached. Identifying such states avoids this problem, but can lead to memory overflow. This trade-off can be called the state revisiting problem.

In [GHP92], the state revisiting problem is considered, for reachability analysis, in the context of a partial order reduction method. Their conclusion is that in that context, the state revisiting problem can be ignored, states should not be saved after visiting, and that the price to be paid in recomputation is tolerable (3 to 4 times a single traversal, for their examples). In general model checking, however, the number of recomputations can be unacceptably large.

It is clear that partial order reductions as in [GHP92] lessen the state revisiting problem because one cause of reaching the same state by different paths is that independent operations are executed in a different order. Nevertheless, recomputation is still sometimes necessary both because such methods only eliminate some of the redundancy of various orderings of independent operations, and because sometimes the same state is reached through truly different sequences of operations. Partial order methods consider operations dependent and/or influencing the specification, if they *might* have such an influence. Here we check more carefully whether an occurrence of an operation actually affects atomic clauses in the specification, as in [KP92], and thus can have greater savings.

Another approach to the state revisiting problem was proposed in [Hol88] by using a hash table where the keys are the states themselves, to indicate whether a state has already been visited, without saving the full states. The difficulty

with this approach is that there may be a hash conflict, and then a state can map to a hash entry indicating that it has been visited, even if it has not been, and thus some parts of the graph may never be explored. Here we both achieve a greater reduction in the graph to be used for model checking, and overcome the state revisiting problem while exploring all of the state space, at a cost of a single additional traversal of the graph obtained by the partial order expansion, beyond the one needed by the partial order method itself.

In order to overcome the state recomputation problem, a preliminary DFS traversal is used to compute the revisiting degree of each state, so that it is clear, on the second traversal, when a state should be retained, and when it can be eliminated. Thus, we can manage a caching method that does not randomly free memory. During this preliminary traversal, a hashing method similar to that in [Hol88] can be used. A significant difference is that when conflicts do occur in the hash table, the worst effect will be some additional recomputation, but the entire graph will ultimately be examined.

In the following section some preliminary definitions are given, the visible state graph is defined and theorems with its properties are given. In Section 3 the basic traversal algorithm to eliminate invisible states is first described, then related to a partial order method, and finally combined with a preliminary traversal to determine revisiting degrees. In Section 4 we summarize the memory and time complexity both of the pre-computation, and of the graph presented to the model checker, and present some simulation results.

In Figures 1 to 3, an example program, and several of its state-space graphs are shown. The specification is of mutual exclusion and of liveness. Y_1 and Y_2 represent flags that are true when processes P_1 and P_2 respectively are in crucial sections. The assertion is that Y_1 and Y_2 are never both true at the same time, and that if one flag is true, the other will eventually become true. The program is represented as a labeled set of guarded commands, for each of the two processes. Each process has its own program counter (denoted PC_i), to control the internal flow of the process. A command is enabled if its guard is true and if the program counter of the process containing it is equal to the command's number. Figure 2 shows the full state-space graph, while the graph without the grey nodes and the dotted lines is the reduced graph after the partial order method of [Pel94] is applied. In Figure 3 the graph is shown in an intermediate stage, after some of the invisible states have been removed, with the candidates for elimination in the rest of the algorithm indicated in gray. The graph without the grey nodes, and with edges connected to their successors is the fully reduced graph relative to the given specification. Note that the original graph has 30 states, the one after partial order reduction has 26, and the graph that fully exploits the elimination of invisible states has only 14. The stages in this reduction will be explained later in the paper.

In this toy example, the specification includes both of the program variables; and only operations involving the control counters are invisible. When the program is more realistic, and the property to be proven only involves part of the variables, much greater savings can be expected, as is shown in the simulations summarized in Section 4.

This paper demonstrates that a careful combination of a partial order reduction method with an algorithm to eliminate states not relevant for the specification, along with a hashing technique to save only relevant information about which states have already been considered, can yield a result that makes previously infeasible problems treatable.

- Global State Representation = (PC_1, PC_2, Y_1, Y_2)
- Initial State = $(0, 0, F, F)$
- Specification Checked = $\Box\neg((Y_1 = T) \wedge (Y_2 = T)), \Box((Y_1 = T) \Rightarrow \Diamond(Y_2 = T))$

PROCESS 1	PROCESS 2
PC1	PC2
0: PC_1:=1 **{a1}**	0: PC_2:=1 **{b1}**
1: PC_1:=2; Y_2:=T **{a2}**	1: PC_2:=2; Y_1:=T **{b2}**
2: (Y_1=T) $\Rightarrow$ PC_1:=3; Y_1:=F **{a3}**	2: (Y_2=T) $\Rightarrow$ PC_2:=3;Y_2:=F **{b3}**
3: PC_1:=0 **{a4}**	3: PC_2:=0 **{b4}**

Fig. 1. Example of a program P.

2 Preliminaries

A finite state program P is a triple $< T, Q, I >$ where T is a finite set of operations, Q is a finite set of states, and $I \in Q$ is the initial state. The enabling condition $en_\alpha \subseteq Q$ of an operation $\alpha \in T$ is the set of states from which α can be executed. Each operation $\alpha \in T$ is a partial transformation $\alpha : Q \mapsto Q$ which needs to be defined at least for each $q \in en_\alpha$. For simplicity we assume that for each $q \in Q$ there exists an operation $\alpha \in T$ such that $q \in en_\alpha$.

An interleaving sequence of a program is an infinite sequence of operations $v = \alpha_0\alpha_1 \ldots$ that generates the sequence of states $\zeta = q_0q_1q_2 \ldots$ from Q such that (1) $q_0 = I$, (2) for each $0 \leq i, q_i \in en_{\alpha_i}$ and $q_{i+1} = \alpha_i(q_i)$.

A nexttime-free LTL formula (denoted LTL-X) is composed of atomic propositions from a set AP, boolean operators $(\wedge, \neg, \vee)$ and the usual temporal modals $\Box$ ('always'), $\Diamond$ ('eventually') and U ('Until') but not the modal $\bigcirc$ ('next').

Definition 1. A *state graph*, $G_P = (\hat{s}, S, E)$, for a program P is a directed, rooted graph, such that :

1. S is a finite set of nodes, $\hat{s} \in S$ is the graph's root and E is a finite set of edges (we denote an edge from node s to node t as $s \rightarrow t$).
2. The graph is total, i.e. from every node there is an exiting edge.

3. There is an injective homomorphism $st : S \rightarrow Q$ that maps nodes to program states such that: $st(\hat{s}) = I$ and if $s \rightarrow t \in E$ then there exists an operation α such that $st(s) \in en_\alpha$ and $st(t) = \alpha(st(s))$.
4. The graph is maximal, i.e, for each state s in a sequence of states generated from some sequence of operations from P, and for each operation α enabled at s such that $\alpha(s) = t$, we have that $st^{-1}(s) \rightarrow st^{-1}(t) \in E$.

We will identify a state and a node with this mapping.

Definition 2. $M = (G_P, V)$ is a *model for a program P and a specification φ* iff G_P is a state graph of P and V is a function $V : S \rightarrow 2^{AP}$ (where AP are the atomic propositions of φ) such that for all nodes $s \in S$, $V(s) = \{a \mid a \in AP$ and a is true in state $st(s)\}$.

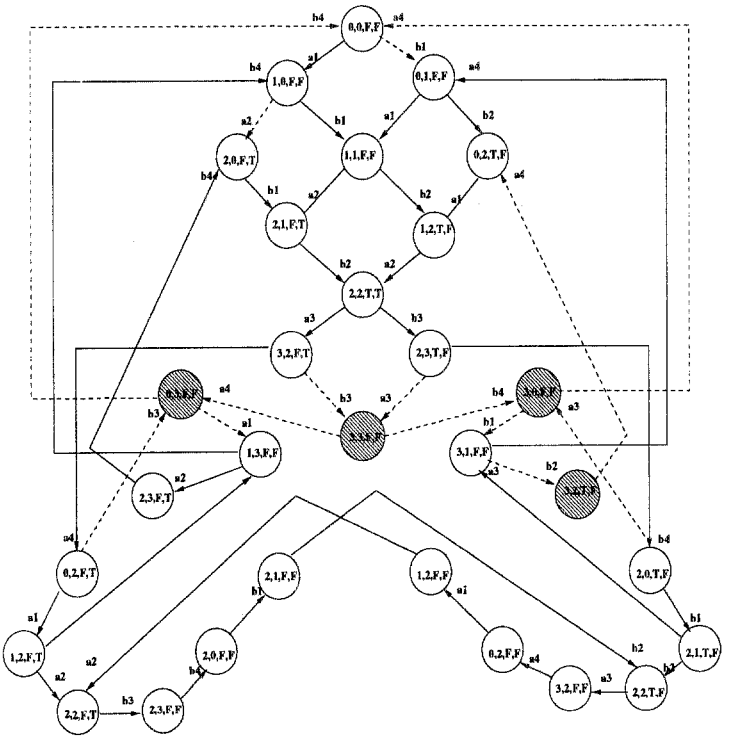


Fig. 2. P's full and partial ordered state graph.

Definition 3. Let $M(G_P(\hat{s}, S, E), V)$ be a model of a program P and specification φ, $s \rightarrow s' \in E$ is a *visible edge* iff $V(s) \neq V(s')$. A node t is a *visible node* iff 1) $t = \hat{s}$ (i.e. the initial node) or, 2) there is a visible edge entering t.

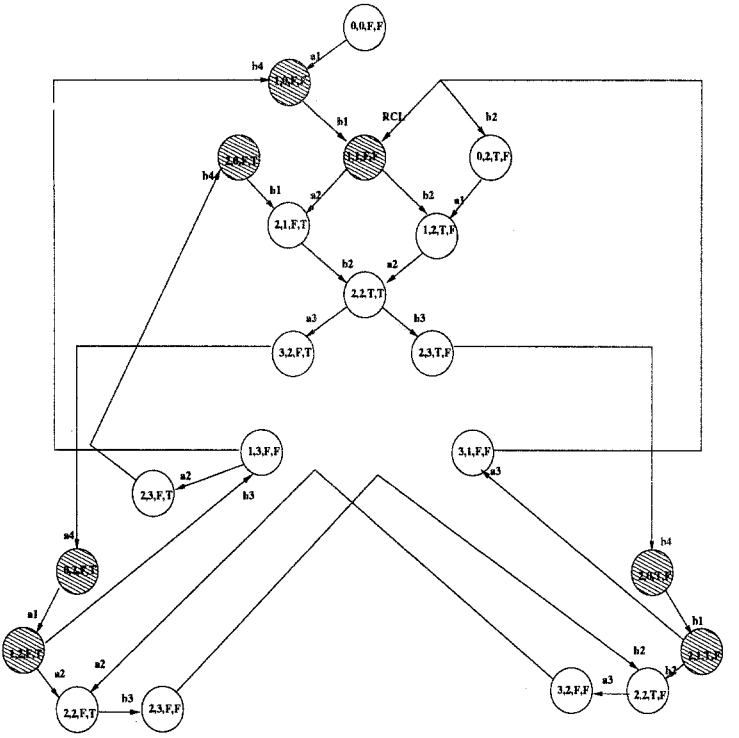


Fig. 3. An intermediate stage in the reduction algorithm.

The visible state graph G_V is the state graph of an abstraction of a program P. Its set of nodes is exactly the visible nodes of the state graph G_P (denoted $VIS(G_P)$). Its edges satisfy the following two properties:

P1 For any 2 nodes $s, s' \in \text{VIS}(G_P)$, there is a sub-path $ss_1 \ldots s_n s'$ of a path of G_P such that $V(s) = V(s_1) = \ldots = V(s_n)$ and $V(s) \neq V(s')$ iff there is subpath $st_1t_2 \ldots t_m s'$ of a path of G_V such that $V(s) = V(t_1) = \ldots = V(t_m)$ and $V(s) \neq V(s')$

P2 For any node $s \in VIS(G_P)$, there is a suffix $ss_1s_2 \ldots$ of a path of G_P such that $V(s) = V(s_1) = V(s_2) = \ldots$ iff there is a suffix $st_1t_2 \ldots$ of a path of G_V such that $V(s) = V(t_1) = V(t_2) = \ldots$.

These properties guarantee that paths in the two graphs have the same properties when repetitions of the relevant truth values are ignored, i.e., they are *stuttering equivalent* [LAM83] (denoted by $\pi \sim \pi'$).

Theorem 4. *Let G_P and G_V be the state graph and a visible state graph respectively of a program P. For each path π in G_P there exists a path π' in G_V such that $\pi \sim \pi'$. For each path π in G_V there exists a path π in G_P such that $\pi \sim \pi'$.*

The proof is done by building a linear stuttering equivalent relation based on properties P1 and P2, and using the fact that LTL-X is insensitive to stuttering.

3 Visible state graph generation

In this section we first show the basic algorithm (Figure 4) which generates the visible state graph. In the algorithm we do not keep an indicator whether invisible nodes have been visited (after backtracking from it). Therefore the time may increase exponentially for graphs that have many invisible nodes with high revisiting degrees. Later we show how to solve this problem.

In the algorithm we abstract from implementation details. We create a visible state graph G, which is represented by a set of nodes S and a set of edges E. We operate on the sets with the standard set operations $(\cup, \cap, \setminus)$ and operands $(\in, \subseteq)$. The search path is kept on a stack. In intermediate stages, S will contain both the visible nodes already examined and all nodes on the stack. The algorithm uses the following functions and indicators:

- $s \in S$ - Either s is a visible node already examined or is on the search stack.
- $s \rightarrow s' \in E$ - An edge from node s to visible node s' was created.
- **en(s)** - The set of operations enabled at state $st(s)$.
- $\alpha(s)$ - The node obtained after executing an operation α on state $st(s)$.
- **visible(s)** - The node s is visible (only for nodes in S).
- **RCL**$(s \rightarrow s')$ - The edge $s \rightarrow s'$ must be reconnected when backtracking from s'.
- **open(s)** - Node s is on the search stack.

The algorithm is based on a standard DFS traversal, implemented in a recursive procedure: At each node s we calculate its set of successors. We then recursively examine all successors that are not indicated as having already been examined (remember we sometimes reexamine a node more than once). The reduction comes when backtracking from a successor s' of the node s. If s' is invisible we replace each edge exiting s' with an edge that exits s and that enters the same target. We then remove the set of edges exiting s', which is followed by the removal of the invisible node s' (lines 10 - 16). In lines 13-14 before removing an edge that is marked RCL (see explanation below) and exiting s' we mark the respective replacing edge that exits s as RCL. In line 11 if s' has a self loop we give s a self loop. This maintains diverging sequences. Note that even if the state later proves to be visible when approached along a different path, and is therefore reintroduced, the edges we remove are invisible. When s' is visible we add the edge $s \rightarrow s'$ to E (line 18).

If a successor s' of s is in S then this indicates that either s' is visible and has been examined or s' is open (i.e. s' closes a loop), thus we add the edge $s \rightarrow s'$ to E. If s' closes a loop and $s \rightarrow s'$ is invisible (i.e $V(s) = V(s')$) we mark that edge RCL, standing for *reconnect later*. In the DFS traversal when we arrive at an invisible open node s' from s, there may be successors of s' that have not yet been examined. We therefore do not know all the visible successors of s' and we

cannot know which are s's successors (that go through s' in the original graph) in the visible graph. Hence, we indicate (i.e. RCL($s \rightarrow s'$) := TRUE in line 24) that we still have to update the set of edges exiting s. Finally when we backtrack from s', the sub-tree from s' has been examined. Thus we know which are s''s visible successors. We then replace the set of edges which enter s' and that are marked RCL (lines 25-28) by edges that enter s''s visible successors. Note that any edge entering s' that is marked RCL when we backtrack from s' is from a visible node or a self loop from s' (because an edge marked RCL closed a loop, the node it came from has already been backtracked from, and was removed if it was invisible).

```
1   procedure expand(s)
2     open(s) := TRUE
3     foreach α ∈ en(s) do
4       s':=α(s)
5       if not (s' ∈ S) then
6         visible(s'):=FALSE
7         S := S ∪ {s'}
8         expand(s')
9         if (not visible(s')) and (V(s) = V(s')) then
10          foreach u such that (s' → u) ∈ E
11            if s' = u then u := s
12            E := E ∪ {s → u}
13            if RCL(s' → u)and(s' ≠ u) then
14              RCL(s → u) := TRUE ;RCL(s' → u) := FALSE ;
15            E := E \ {s' → u}
16          S := S \ {s'}
17        else
18          E := E ∪ {s → s'}
19          visible(s') := TRUE
20      else
21        if V(s) ≠ V(s') then visible(s') := TRUE
22        E := E ∪ {s → s'}
23        if open(s') and (V(s) = V(s')) then
24          RCL(s → s') := TRUE
25    foreach u such that RCL(u → s)
26      foreach v such that (s → v) ∈ E
27        E := E ∪ {u → v}
28        if RCL(s → v) then RCL(u → v) := TRUE
29      RCL(u → s) := FALSE
30    open(s) := FALSE
31  end
```

Fig. 4. Algorithm for generating the visible state graph.

We demonstrate the backtracking in Figure 2 where nodes are represented by a 4-tuple of values (PC1,PC2,Y1,Y2). In Figure 2 the algorithm first starts backtracking when it does a step from (2,1,F,F) and arrives a second time at node (2,2,T,F) (on the lower right). This node is visible because it has a visible operation (b2) entering it (e.g., the truth value of the atomic proposition "Y1=T" is changed). Therefore it is not deleted when backtracking from it. On the other hand, node (2,1,F,F) has only an invisible operation (b1) entering it (because only the program counter, irrelevant to the specification, is changed). Therefore it is deleted when backtracking. Its successors are now added to the successor set of its predecessor (state (2,0,F,F)). Next, node (2,0,F,F) is also deleted because it is an invisible node. Its successor (i.e. node (2,2,T,F)) will now become a successor of the visible node (2,3,F,F). The result of the above can be seen in Figure 3.

The use of RCL can be seen in Figure 3 where invisible node (1,1,F,F) (the second from the top center) has an edge marked RCL entering it. When backtracking from node (1,1,F,F), before removing it we reconnect node (3,1,F,F) to the nodes (2,1,F,T), and (1,2,T,F).

To show the algorithm correct we must prove that properties P1 and P2 hold with respect to the full state graph and the graph constructed (i.e, it is a visible state graph). This is done by induction on the set of backtracking steps executed by the algorithm. For each step of the induction we look at: (1) the (intermediate) full state graph obtained from the edges backtracked from, (2) the (intermediate) graph obtained from edges in the set E (see algorithm). We then show that an intermediate version of P1 and P2 hold for these two graphs. When the algorithm terminates, the "full" version of P1 and P2 hold.

We can combine our algorithm with any algorithm for partial order reduction, e.g., A1 from [Pel94]. In that algorithm we execute a DFS traversal of a program's state space. At each state only a subset of the enabled transitions (called the ample set) are expanded. This is due to the fact that expanding all enabled transitions will lead to a graph with more than one interleaving per partial order. The only change to our algorithm is that instead of expanding all the enabled operations from a particular state s, we expand only those operations that belong to the ample set of state s. Therefore, we replace line 3 in Figure 4 with: **foreach** $\alpha \in$ **ample(s) do**. Other algorithms differ in the way that a subset of enabled transitions are selected, but can be used in the same way.

To solve the revisiting problem, we present an algorithm that pre–processes the state space. The algorithm calculates the revisiting degree of each state. This information is passed on to the algorithm that generates the visible state graph, which then can more selectively delete states. The preliminary DFS algorithm traverses the state space in a partial order manner, which is the exact same order used in the later reduction algorithm (both are deterministic).

We use a Hash Table [Hol88] (called the *revisited* hash table), as a revisiting degree counter for each node. The hash table is accessed with a hash function whose argument is a state. When visiting a node in the DFS traversal, we check if its revisiting degree is zero. If this is the case we set its revisiting degree counter

to 1, and then we recursively calculate the revisiting degree of all its successors (from the *ample* set of the underlying partial order traversal). Otherwise, we increase the counter of the node by 1, and backtrack from that node. Note that all this only relates to revisits that do not close a loop, for reasons explained already in the Introduction.

Here we use Holzmann's hash table to assist us in calculating the revisiting degree of each node. This is a novelty in itself: until now the use of this technique was problematic, because of the small probability of a hash collision when model checking (resulting in not checking part of the state space). Here in the worst case, a hash collision will cause us to calculate an incorrect revisiting degree, resulting in additional state recomputation in the latter DFS.

Now, in the latter DFS that generates the visible state graph, when backtracking from a state, we check if the node will be revisited (according to the *revisited* hash table). If an invisible node will not be revisited we remove it and all its pointers to its successors from internal memory, otherwise we maintain it and pointers to its successors in memory. (If we were in error because of hash conflicts, we might have to regenerate the node and its pointers later on, when we reach that state along another path.) If a visible node will not be revisited we remove it from the internal memory and store it on external memory, otherwise we maintain it in memory. The pointers of a visible node to its successors are always stored in external memory, because they are not needed for later revisits in the traversal. When we revisit such a node we decrement its counter in the *revisited* hash table.

When we run out of memory, we can do a form of garbage collection: For each node s in memory (stored in a balanced tree) we check if s's counter is zero. If this is the case, we delete that state from the internal memory, and store it on the external memory (if it is visible). Note that some nodes may have the same entry in the *revisited* hash table. This means that they can only be deleted together (i.e. when they all will not be revisited anymore). Thus we must execute a garbage collection to dispose of these kinds of nodes. This is a better caching method for memory management because states are not deleted randomly.

Note that this method of an initial traversal of the state space can be applied to all current state space generation algorithms. For example [Pel94] presented an on-line model checker, by traversing the product of the state space and specification graphs. We also can initially traverse the product and calculate the revisiting degree of all nodes, saving space as shown in the following section.

4 Memory and Time Complexity

For analysis of memory and time complexity we distinguish between two stages: 1) The memory and time complexity of the algorithm that constructs the visible state graph (denoted VSG), and 2) The memory and time complexity of the algorithm for the model checking. In both cases the analysis is relative to the complexity of the algorithms that construct and model check the graph obtained by applying a partial order method (denoted POG). When we refer to

memory complexity, our intention is internal memory. We assume that we have an unrestricted amount of external memory (used for the caching method). For the model checking itself, the savings is in checking a smaller model. Standard model checking of linear time temporal logic specifications has time and space complexity $O(|VSG| \cdot 2^{|\varphi|})$, whereas previously we had the same formula over the original state graph or the POG.

As for the preprocessing stage to compute the revisiting degree and then generate the visible state graph, the time complexity is the same as for existing methods for partial order reductions, namely $O(pe \cdot log(ps))$, where pe is the number of edges in the partial order graph that would be produced by that method alone, and ps is the number of nodes in that graph. As explained previously, our algorithm makes one additional traversal.

The partial order method we considered uses $O(m \cdot ps + log(m) \cdot pe)$ space in its original form, where m is the space needed for a single state.

For our algorithm, the memory complexity of the first DFS is $O(m \cdot ss + ps)$ (where ss is the size of the stack). The data structures used are the search stack and the revisited hash table (using $O(ps)$ for the size of the hash table). The memory complexity for the subsequent reduction traversal is also $O(m \cdot ss + ps)$. Here the data structures used are the search stack, the revisited hash table and the intermediate stages of the visible state graph construction. We use a caching method, therefore we bound by a constant the memory needed for the full states retained in intermediate stages. Our simulations show that the number of states needed at any one time is actually small, and the cache will not cause extraneous recomputation. The simulation was constructed using the high level language ICON. We implemented the algorithms that calculate the revisiting degree of a program's state graph and that generate the visible state graph. The ample set for the partial order method is calculated according to [HP94].

In one test, we simulated a leader election protocol in a unidirectional ring from [DKR82] and several alternative specification formulas. The algorithm uses a local variable max_i in each process to show its version of the maximal value.

We executed our algorithm on the state space of the protocol for 5 processes for 5 different specification formulas. In Figure 6, we compare for each formula the original size of the state graph (first states, and then edges), the state graph that was obtained only with the partial method, and the state graph that was obtained with our method (which includes the partial order method). The last column of the table presents the number of full nodes that were in memory at any time, in addition to the hash table.

The fifth formula was especially complex, namely: $\Diamond(((max_1 = 5) \wedge \neg((max_2 = 5) \vee \ldots (max_5 = 5)))\ U\ (((max_1 = 5) \wedge (max_2 = 5)) \wedge \neg((max_3 = 5) \vee (max_4 = 5) \vee (max_5 = 5)))\ U\ \ldots((max_1 = 5) \wedge (max_2 = 5) \wedge (max_3 = 5) \wedge (max_4 = 5) \wedge (max_5 = 5))))$, i.e., the processes obtain the correct maximum in the fixed order 1,2,3,4,5. This formula's tableau state graph has on the order of 1000 states. In the model checking stage, multiplying this by the program's partial order state graph will result in a quarter of a million states, while multiplying this by the visible state graph will result in about 5,000 states. This formula is not satisfied by the protocol, because the order is actually random.

FORMULA	ORIG-S	ORIG-E	POG-S	POG-E	VSG-S	VSG-E	FULL
1	11099	68717	7030	21548	75	163	222
2	11099	68717	203	352	10	19	61
3	11099	68717	630	1373	42	98	81
4	11099	68717	723	1329	56	137	97
5	11099	68717	263	351	5	9	57

Fig. 5. Simulation results for leader election protocol.

In the table, for formula 1 our algorithm has reduced a state space of 11,099 states and 68,717 transitions to a state space of 75 states and 163 transitions, where the partial order method succeeded in reducing by less than one order of magnitude relative to the original. In addition to the hash table, only 222 full nodes were needed at any one time in the generation of the reduced graph. The reader can observe that the rest of the results are similarly impressive.

References

[DKR82] D. Dolev, M. Klawe and M.Rodeh. An O(nlogn) unidirectional distributed algorithm for extrema finding in a circle. *Journal of Algorithms*, volume 3, pages 245–260, 1992.

[GHP92] P. Godefroid, G.J Holzman and D. Pirottin. State space caching revisited. In *Proc. 4th International Conference on Computer Aided Verification*, LNCS 697, pages 178–191, Canada, June 1992.

[GW91] P.Godefroid and P.Wolper. A partial order approach to model checking. In *Proc. 6th Symposium on Logic in Computer Science*, pages 406–415, Amsterdam, July 91.

[Hol88] G.J.Holzmann. An improved protocol peachability analysis technique. *Software-Practice and Experience*, Vol 18(2), pages 137–161,February 1988.

[HP94] G.J. Holzmann and D. Peled. An improvement in formal verification. In *Proceedings FORTE 1994 Conference*, Switzerland, October 1994.

[KP92] S. Katz and D. Peled. Conditional independence using collapses. *Theoretical Computer Science*, volume 101, pages 337–359, 1992.

[LAM83] L. Lamport, What good is temporal logic? in: *Proc. IFIP 9th World Congress*, Paris, France (1983) 657–668.

[Pel94] D. Peled. Combining partial order reductions with on-the-fly model checking. In *Proc. 6th International Conference on Computer Aided Verification*, LNCS 818, pages 377–390, California, USA, June 1994.

[Val90] A. Valmari. A stubborn attack on state explosion. Formal methods in System Design 1, pages 297–322, 1992.

[VW86] M. Vardi and P.Wolper. An automata-theoretic approach to automatic program verification. In *Proc. 1st Symposium on Logic in Computer Science*, pages 322–331, Cambridge, June 1986.

Using On-The-Fly Verification Techniques for the Generation of Test Suites *

Jean-Claude Fernandez[1] Claude Jard[2] Thierry Jéron[2] and César Viho[2]

[1] Vérimag, Miniparc Zirst, rue Lavoisier, F-38330 Montbonnot Saint-Martin, France.
Jean-Claude.Fernandez@imag.fr
[2] IRISA/Pampa, Campus de Beaulieu, F-35042 Rennes, France.
(Claude.Jard, Thierry.Jeron, César.Viho)@irisa.fr

Abstract. In this paper we attempt to demonstrate that on-the-fly techniques, developed in the context of verification, can help in deriving test suites. Test purposes are used in practice to select test cases according to some properties of the specification. We define a consistency preorder linking test purposes and specifications. We give a set of rules to check this consistency and to derive a complete test case with preamble, postamble, verdicts and timers. The algorithm, which implements the construction rules, is based on a depth first traversal of a synchronous product between the test purpose and the specification. We shortly relate our experience on an industrial protocol with TGV, a first prototype of the algorithm implemented as a component of the CADP toolbox.

1 Introduction

It is widely recognized that testing is an essential component of the full lifecycle of communicating systems. However, the process of generating test suites is complicated, error-prone and expensive. The intrinsic difficulty comes from the black-box nature of the implementation: its behaviour is only observable and controlable at the interfaces. In that context, a formal framework is a prerequisite for giving precise and consistent meanings of test verdicts. The usual theoretical approach [Bri88] is to consider a formal specification of the intended behaviour of the Implementation Under Test (IUT). It permits to define the notion of *conformance* relation linking an implementation to the specification and the notion of *verdict* associated to the application of a test case (set of interaction sequences) to an implementation, w.r.t. the conformance relation. The problem is to automatically generate correct test cases from a formal specification of the IUT. A correct test case, applied to an IUT, will declare "fail" only implementations which do not conform to the specification (*soundness* property). We also require that an implementation which does not conform to the specification might be detected by repeating the application of a test case, under a fairness assumption on the implementation (property of exhaustivity).

* This work has been partially supported by an industrial contract with Verilog in a study for the french DGA (Direction Générale pour l'Armement)

During the last decade, testing theory and algorithms for the generation of tests have been developed from Labelled Transition System specifications (LTS). Test generation involves sub-problems of traversal, comparison or reduction of LTS, already addressed by verification. Consequently, we think, among others [CGPT95], that time is ripe for linking test and verification. The experience of practitioners tells them that it is not reasonable to try to validate all possible behaviours of their protocol. It is why they use informal test purposes. Basically, we traverse in a depth-first manner, the synchronous product of the IUT specification and of the test purpose. During the traversal, we check their mutual consistency. If so, an acyclic test graph is generated and decorated with verdicts and timers. The algorithm is an original extension of the on-the-fly verification kernel we developed a few years ago [JJ89, JJ91, FM91, FMJJ92]. It provides a complete treatment of the problem of test cases, including preambles and postambles, verdicts and timers management. It is now well known that depth-first traversals are the heart of some good verification algorithms, for behavioural comparison and reduction [FM91], as well as for model-checking [JJ89, CVWY90, JJ91]. We show that it is also true for test generation, which constitutes a good example of transfer from verification to testing.

The test generator, based on verification technology, has been prototyped in the context of an industrial consortium, linking Vérilog, Cap-Sesa, Cnet, Inria and the French Army. An experiment was performed on a real ISDN protocol specification. The results were very encouraging, confirming the interest of using this kind of algorithmic, which is now mature enough to be transfered in the industrial world, to deal with real formal specifications. Our approach is compatible with symbolic (or structural) ones like TVEDA [Pha94b] which may compute test purposes using reachability analysis.

The presentation is organized as follows. We start by defining the different models used for describing test purposes, test cases, the specification and the IUT. We define a consistency preorder between test purposes and specifications, and a test conformance relation linking implementations to specifications. We give the formal rules allowing the construction of a test case from a test purpose and a specification. We give some results concerning soundness and exhaustivity of our generated test cases. Finally, we give the main results gained during an experiment on an ISDN protocol.

2 Models

In this section we first describe the models used for the description of the different objects involved in the generation of test cases. They are used to define the notion of consistency relating a test purpose with a specification and the notion of conformance relating an implementation with a specification. These models are then used to define formal rules for the construction of test cases.

2.1 Input-Outputs labelled transition systems

The models used are all based on Input-Output Labelled Transition Systems (IOLTS) in which input and output actions are differentiated because of the asymmetrical nature of the testing activity.

We consider a finite alphabet of actions A, partitioned into two sets: *input actions* A_I and *output actions* A_O. We shall let α, β range over A, i, i' range over A_I and o, o' range over A_O. We consider finite IOLTS $\mathrm{M}=(Q^{\mathrm{M}}, A, T^{\mathrm{M}}, q^{\mathrm{M}}_{\mathrm{init}})$ where Q^{M} is the set of states, $q^{\mathrm{M}}_{\mathrm{init}}$ is the initial state, $T^{\mathrm{M}} \subseteq Q^{\mathrm{M}} \times A \times Q^{\mathrm{M}}$ is the transition relation.

We adopt the following notations and conventions: Let $\sigma \in A^*$, $p, q \in Q^{\mathrm{M}}$. We write $p \xrightarrow{\alpha}_{\mathrm{M}} q$ iff $(p, \alpha, q) \in T^{\mathrm{M}}$ and write $p \overset{\sigma}{\Longrightarrow}_{\mathrm{M}} q$ iff $\exists \alpha_1, \alpha_2 \cdots \alpha_n \in A$, $p_0, \cdots, p_n \in Q^{\mathrm{M}}$. $\sigma = \alpha_1.\alpha_2 \ldots \alpha_n$ and $p_0 = p, p_i \xrightarrow{\alpha_{i+1}}_{\mathrm{M}} p_{i+1}$ with $i < n, p_n = q$. $\mathcal{A}(q) = \{\alpha \mid \exists q' \text{ and } q \xrightarrow{\alpha}_{\mathrm{M}} q'\}$ is the set of immediate actions after q, $\mathcal{I}(q) = \mathcal{A}(q) \cap A_I$ is the set of inputs after q, and $\mathcal{O}(q) = \mathcal{A}(q) \cap A_O$ is the set of outputs after q. $\mathrm{Succ}_\alpha(q) = \{q' \mid q \xrightarrow{\alpha}_{\mathrm{M}} q'\}$ is the set of states reachable from q by means of a transition labelled by α. We write $\neg(p \xrightarrow{\alpha}_{\mathrm{M}})$ if there is no transition starting from p and labelled by α, $\neg(p \xrightarrow{\alpha}_{\mathrm{M}}) = (\mathrm{Succ}_\alpha(p) = \emptyset)$. We note p **after** $\sigma = \{q \in Q^{\mathrm{M}} \mid p \overset{\sigma}{\Longrightarrow}_{\mathrm{M}} q\}$ the set of states reachable from p by the sequence of transitions σ and $traces(p) = \{\sigma \in A^* | p \textbf{ after } \sigma \neq \emptyset\}$ the set of sequences starting from p and reaching a state in Q^{M}. In the sequel, we will not distinguish between a transition system and its initial state.

An IOLTS satisfies the *controlability condition* if and only if for each state, if an output is enabled, then there is exactly one outgoing transition. More formally, if $|X|$ denotes the cardinality of the set X, $\forall p \,.\, |\mathcal{O}(p)| = 0 \vee (|\mathcal{O}(p)| = 1 \wedge \mathcal{O}(p) = \mathcal{A}(p))$.

An IOLTS is *deterministic* if and only if $\forall p, \forall \sigma \,.\, |p \textbf{ after } \sigma| \leq 1$.

We consider four kinds of IOLTS: the specification, the implementation, the test purpose behaviour and the test cases which meanings are described below.

2.2 Specification and implementation

An IUT is placed in a test environment in which the tester can only interact with inputs and outputs. Thus the tester has an *external view* of the implementation. In contrast, a specification generally models the *internal view* of the system, i.e. the behaviour of the system with its internal actions, without considering the way it interacts with the environment. But this interaction should be taken into account in the test generation. As an example, if the implementation communicates asynchronously with its environment through several points of control and observation (PCOs), two subsequent and causally ordered outputs may be observed by the environment as two concurrent inputs if they occur on two different PCOs. In the following, we will consider the environmental point of view: outputs are *controlable* actions initiated by the environment (which may be the tester) and sent to the IUT whereas inputs are *observable* actions, initiated by the IUT and received by the environment.

While testing an IUT, we check for the conformance of the IUT in its environment with the specification in the same environment. Thus we first have to transform the specification into its external view. Internal actions which are not observable by the environment have to be hidden and replaced by a τ transition. Inputs are replaced by outputs and vice versa, taking care of concurrency which may be produced by asynchronous interaction. This is called the mirror image operation. After that, we have to apply a τ-reduction which suppresses τ transitions. The transition system of the resulting specification is then an IOLTS. This has been implemented on-the-fly in our prototype but, due to space limitations, we will not give more details of how this can be done effectively. Deadlocks are often supposed to be observable by a tester. In practice the tester uses timers to achieve this (see 3.2) and we have to suppose that a timeout occurs if and only if the implementation is deadlocked. This is why timeouts are considered as inputs of the tester. If the specification is allowed to deadlock in a particular state, this is modelled by a special transition δ considered as an input of the environment initiated by the system. This treatment of deadlocks is quite similar with what is done in [Tre95, Pha94a]. Finally, the last operation is determinization.

The resulting specification is a deterministic IOLTS $S = (Q^{\mathrm{S}}, A, T^{\mathrm{S}}, q^{\mathrm{S}}_{\mathrm{init}})$ with $A = A_I \cup A_O$ and $\delta \in A_I$ a distinguished input. Without loss of generality, we will suppose that S starts with outputs of the environment i.e. $\mathcal{A}(q^{\mathrm{S}}_{\mathrm{init}}) \subseteq A_O$. In the following, specification will always correspond to the external view S of the specification. Though the implementation is not necessarily a transition system (it may be a physical system), as in all testing theories, we have to reason formally about it and model its behaviour. As it is only considered by its interactions with the environment, it is also modelled as an IOLTS $I = (Q^{\mathrm{I}}, A^{\mathrm{I}}, T^{\mathrm{I}}, q^{\mathrm{I}}_{\mathrm{init}})$, with $A^{\mathrm{I}} = A^{\mathrm{I}}_I \cup A^{\mathrm{I}}_O$, $A_I \subseteq A^{\mathrm{I}}_I$.

2.3 Test Purpose

A test purpose defines a property on some particular interactions between the IUT and the tester. It consists in two parts : a behavioural part and a constraint part. The constraint part gives some property on the state of the implementation. It can be seen as computable by the environment and will be modelled by an input for the tester. Thus it is integrated in the behavioural part.

Definition 2.1 *(Test Purpose behaviour) A test purpose behaviour is a deterministic acyclic IOLTS* $TP = (Q^{\mathrm{TP}}, A, T^{\mathrm{TP}}, q^{\mathrm{TP}}_{\mathrm{init}})$ *satisfying the controlability condition and with a set of distinguished states* $\mathtt{Accept} \subseteq Q^{\mathrm{TP}}$ *with no successor.*

2.4 Test Cases

A *test case* is a set of sequences of actions describing all the interactions occuring between an IUT and a tester which wants to verify that an implementation conforms with the specification according to a test purpose. In an industrial context, test cases are often described using the Tree and Tabular Combined Notation (TTCN [ISO92]). Some transitions are decorated with verdicts with

the following informal meaning :
(PASS): means that the test purpose is satisfied by the current sequence. But a sequence leading to the initial state (Postamble) must be applied in order to carry on another test case. It is a temporary verdict as the application of the postamble may produce Fail verdicts.
PASS: this is a definitive verdict meaning that the initial state has been reached after a (PASS) verdict. The sequence between (PASS) and PASS is a Postamble.
FAIL: means non-conformance of the IUT.
INCONCLUSIVE: this verdict is used in practice when a reception is allowed in the specification but cannot lead to a (PASS) or leads to a behaviour that is not considered in the test case because testing cannot be exhaustive in practice.

Definition 2.2 *(Test Case) A test case is a deterministic acyclic IOLTS* $TC = (Q^{\mathrm{TC}}, A, T^{\mathrm{TC}}, q^{\mathrm{TC}}_{\mathrm{init}})$ *satisfying the controlability condition. A test suite is a set of test cases.*

2.5 Consistency and test conformance relation

In this section, we define what we mean by consistency of a test purpose w.r.t a specification and which conformance relation linking the implementation with its specification is considered.

A test purpose TP is said to be *consistent* w.r.t a specification S, denoted by $q^{\mathrm{TP}}_{\mathrm{init}} \prec q^{\mathrm{S}}_{\mathrm{init}}$, if the two following conditions are satisfied: the set of behaviours described by the test purpose is included (see the definition below) in the set of behaviours of the specification, and from each state of S corresponding to an **Accept** state of TP, there is a path in S to $q^{\mathrm{S}}_{\mathrm{init}}$.

Definition 2.3 *(Consistency preorder). A relation* $R \subseteq Q^{\mathrm{TP}} \times Q^{\mathrm{S}}$ *is a* consistency relation *if and only if* $R \subseteq \mathcal{F}(R)$ *where,*

$$
\begin{array}{l}
\mathcal{F}(R) = \{(p^{\mathrm{TP}}, p^{\mathrm{S}}) \mid \\
(\forall \alpha, \forall q^{\mathrm{TP}} \cdot p^{\mathrm{TP}} \xrightarrow{\alpha}_{\mathrm{TP}} q^{\mathrm{TP}} \Longrightarrow \exists q^{\mathrm{S}}, q^{\mathrm{S}}_1, \exists \sigma \in (A \setminus \{\alpha\})^* \cdot p^{\mathrm{S}} \overset{\sigma}{\Longrightarrow}_{\mathrm{S}} q^{\mathrm{S}}_1 \xrightarrow{\alpha}_{\mathrm{S}} q^{\mathrm{S}} \wedge \\
\qquad\qquad (q^{\mathrm{TP}}, q^{\mathrm{S}}) \in R \wedge (p^{\mathrm{TP}}, q^{\mathrm{S}}_1) \in R) \wedge \\
p^{\mathrm{TP}} \in \mathbf{Accept} \qquad \Longrightarrow \exists \sigma \in A^* \cdot p^{\mathrm{S}} \overset{\sigma}{\Longrightarrow}_{\mathrm{S}} q^{\mathrm{S}}_{\mathrm{init}}\}
\end{array}
$$

$q^{\mathrm{TP}}_{\mathrm{init}} \prec q^{\mathrm{S}}_{\mathrm{init}}$ *if and only if there is a relation* $R \subseteq \mathcal{F}(R)$, *containing* $(q^{\mathrm{TP}}_{\mathrm{init}}, q^{\mathrm{S}}_{\mathrm{init}})$

If the test purpose and the specification are consistent, we can derive sound test cases. A test case is sound if it gives a negative verdict only if the implementation is not correct w.r.t. the specification.

We consider a conformance relation quite similar to those in [Tre95, Pha94a]. Informally, the conformance relation states that outputs of the environment which are not accepted by the specification may be accepted by the implementation but inputs produced by the implementation must be also produced by the specification.

Definition 2.4 *(Test conformance relation) Let* S *and* I *be two IOLTS describing the external view of a specification and an implementation,*

$$I \textbf{ ioconf } S \textit{ if and only if } \forall \sigma \in traces(S), \mathcal{I}(I \textbf{ after } \sigma) \subseteq \mathcal{I}(S \textbf{ after } \sigma)$$

3 Construction rules

The essence of the on-the-fly method is to traverse a kind of synchronous product between two graphs, one for the specification and the other for the property to be checked. We first define this synchronous product. Then we give the rules for the test case construction, including decoration with verdicts and timers. Finally we give some properties of the generated test cases.

3.1 Synchronous product

A transition is firable in the product if either it is firable in the two components or it is firable only in the specification.

Definition 3.1 *(synchronous product) We define the product* $P = (Q^{\mathrm{P}}, A, T^{\mathrm{P}}, (q^{\mathrm{TP}}_{\mathrm{init}}, q^{\mathrm{S}}_{\mathrm{init}}))$, *with* $Q^{\mathrm{P}} \subseteq Q^{\mathrm{TP}} \times Q^{\mathrm{S}}$ *where* Q^{P} *and* T^{P} *are the smallest sets obtained by application of the following rules:*

- [Sync1] $(q^{\mathrm{TP}}_{\mathrm{init}}, q^{\mathrm{S}}_{\mathrm{init}}) \in Q^{\mathrm{P}}$,
- [Sync2] $$\frac{(p^{\mathrm{TP}}, p^{\mathrm{S}}) \in Q^{\mathrm{P}} \quad p^{\mathrm{TP}} \xrightarrow{\alpha}_{\mathrm{TP}} q^{\mathrm{TP}} \quad p^{\mathrm{S}} \xrightarrow{\alpha}_{\mathrm{S}} q^{\mathrm{S}}}{(q^{\mathrm{TP}}, q^{\mathrm{S}}) \in Q^{\mathrm{P}} \quad (p^{\mathrm{TP}}, p^{\mathrm{S}}) \xrightarrow{\alpha}_{\mathrm{P}} (q^{\mathrm{TP}}, q^{\mathrm{S}})}$$
- [Sync3] $$\frac{(p^{\mathrm{TP}}, p^{\mathrm{S}}) \in Q^{\mathrm{P}} \quad \neg(p^{\mathrm{TP}} \xrightarrow{\alpha}_{\mathrm{TP}}) \quad p^{\mathrm{S}} \xrightarrow{\alpha}_{\mathrm{S}} q^{\mathrm{S}} \quad (p^{\mathrm{TP}}, p^{\mathrm{S}}) \notin \mathbf{Accept} \times \{q^{\mathrm{S}}_{\mathrm{init}}\}}{(p^{\mathrm{TP}}, q^{\mathrm{S}}) \in Q^{\mathrm{P}} \quad (p^{\mathrm{TP}}, p^{\mathrm{S}}) \xrightarrow{\alpha}_{\mathrm{P}} (p^{\mathrm{TP}}, q^{\mathrm{S}})}$$

3.2 The test case construction

The algorithm is based on a depth first traversal of the synchronous product, definition 3.1. Two mains actions are performed : the consistency relation between the test purpose and the specification is checked while a *direct acyclic graph* DAG is synthesized, definition 3.2. More precisely, a stack stores the states of the current execution sequence. The algorithm proceeds as follows, starting from the initial state, $(q^{\mathrm{TP}}_{\mathrm{init}}, q^{\mathrm{S}}_{\mathrm{init}})$. Let $(p^{\mathrm{TP}}, p^{\mathrm{S}})$ be the current state, i.e. the top of the stack, and $(p^{\mathrm{TP}}, p^{\mathrm{S}}) \xrightarrow{\alpha}_{\mathrm{P}} (q^{\mathrm{TP}}, q^{\mathrm{S}})$ a transition not yet analyzed. If q^{TP} is an accepting state, then a postamble is computed by searching a shortest path from q^{S} to $q^{\mathrm{S}}_{\mathrm{init}}$; else if $(q^{\mathrm{TP}}, q^{\mathrm{S}})$ belongs to the DAG then the transition is added to the DAG; otherwise, if $(q^{\mathrm{TP}}, q^{\mathrm{S}})$ does not belong to the stack nor to the visited states, then the state $(p^{\mathrm{TP}}, p^{\mathrm{S}})$ is pushed on the stack. When all the transitions starting from the state $(p^{\mathrm{TP}}, p^{\mathrm{S}})$ are analyzed, then $(p^{\mathrm{TP}}, p^{\mathrm{S}})$ is popped in the set of visited states. The operator Comb is used in order to ensure the controlability condition. The algorithm terminates when the stack is empty and succeeds if $(q^{\mathrm{TP}}_{\mathrm{init}}, q^{\mathrm{S}}_{\mathrm{init}})$ belongs to the preamble and if $\mathbf{Accept} \times \{q^{\mathrm{S}}_{\mathrm{init}}\}$ is a subset of the visited states. The algorithm requires a time complexity linear with respect to the size of the transition relation of the synchronous product and a space complexity linear with respect to its state space.

Definition 3.2 *We define DAGs syntactically as* $n ::= 0 \mid 1 \mid \alpha; n \mid n + n$

We also define a predicate $\vdash^v$ *for* $v = 1, 2, 3$. $\vdash^v (p^{\mathrm{TP}}, p^{\mathrm{S}}) : n$ *means "the node associated with* $(p^{\mathrm{TP}}, p^{\mathrm{S}})$ *is* n *and belongs to the preamble, test case body*

and postamble if v is respectively 1, 2 or 3". We use $Node((p^{TP}, p^S))$ to denote the node currently associated to (p^{TP}, p^S). Initially $Node((p^{TP}, p^S)) = 0$. An operator Comb is used to accumulate the nodes in order to ensure the controlability condition:

$$Comb(m, \alpha; n) := \begin{cases} \alpha; n \text{ if } \alpha \in A_O,\ m = \Sigma \alpha_i; n_i \text{ and } \forall i, \alpha_i \in A_I \\ m \text{ if } m = \alpha'; n' \text{ and } \alpha' \in A_O \\ m + \alpha; n \text{ otherwise} \end{cases}$$

Preamble

$$\frac{v \in \{1,2\} \quad \vdash^v (q^{TP}_{init}, q^S) : n \quad (q^{TP}_{init}, p^S) \xrightarrow{\alpha}_P (q^{TP}_{init}, q^S)}{\vdash^1 (q^{TP}_{init}, p^S) : \text{Comb}(\text{Node}((q^{TP}_{init}, p^S)), \alpha; n)}$$

Test Case Body

$$\frac{\vdash^2 (q^{TP}, q^S) : n \quad q^{TP} \neq q^{TP}_{init} \quad (q^{TP}_{init}, p^S) \xrightarrow{\alpha}_P (q^{TP}, q^S)}{\vdash^2 (q^{TP}_{init}, p^S) : \text{Comb}(\text{Node}((q^{TP}_{init}, p^S)), \alpha; n)}$$

$$\frac{\vdash^2 (q^{TP}, q^S) : n \quad p^{TP} \neq q^{TP}_{init} \quad (p^{TP}, p^S) \xrightarrow{\alpha}_P (q^{TP}, q^S)}{\vdash^2 (p^{TP}, p^S) : \text{Comb}(\text{Node}((p^{TP}, p^S)), \alpha; n)}$$

$$\frac{\vdash^3 (q^{TP}, q^S) : n \quad p^{TP} \notin \texttt{Accept} \quad q^{TP} \in \texttt{Accept} \quad (p^{TP}, p^S) \xrightarrow{\alpha}_P (q^{TP}, q^S)}{\vdash^2 (p^{TP}, p^S) : \text{Comb}(\text{Node}((p^{TP}, p^S)), \alpha; n)}$$

Postamble

$$\vdash^3 (p^{TP}, q^S_{init}) : 1 \quad (p^{TP} \in \texttt{Accept})$$

$$\frac{\vdash^3 (p^{TP}, q^S) : n \quad (p^{TP}, p^S) \xrightarrow{\alpha}_P (p^{TP}, q^S) \quad p^{TP} \in \texttt{Accept}}{\vdash^3 (p^{TP}, p^S) : \text{Comb}(\text{Node}((p^{TP}, p^S)), \alpha; n)}$$

The test case verdicts We define the partial function `verdict` which assigns verdicts to some transitions in the DAG which construction is defined above. This function is defined by means of the rules below.

We complete the definition of the predicate $\vdash^v$ for $v = 4, 5$. $\vdash^4 (p^{TP}, p^S) : 1$ means that "the node associated with (p^{TP}, p^S) is 1 and is the ending state of a transition labelled by an Inconclusive. We need a new state *Fail_State* in the synchronous product and the axiom $\vdash^5 Fail\_State : 1$.

In the sequel, u, v range over $1, 2, 3, 4, 5$.

Pass : assigned to a transition in the DAG if the ending state is in the Postamble and the specification is in state q^S_{init}

$$\frac{\vdash^v (p^{TP}, p^S) : n \quad v \in \{2,3\} \quad \vdash^3 (q^{TP}, q^S_{init}) : 1 \quad (p^{TP}, p^S) \xrightarrow{\alpha}_P (q^{TP}, q^S_{init})}{\texttt{verdict}(n, \alpha, 1) = \text{Pass}}$$

(Pass) : assigned to a transition linking a state of the Test Case Body to a state in the Postamble.

$$\frac{\vdash^2 (p^{TP}, p^S) : m \quad \vdash^3 (q^{TP}, q^S) : n \quad (p^{TP}, p^S) \xrightarrow{\alpha}_P (q^{TP}, q^S)}{\texttt{verdict}(m, \alpha, n) = (\text{Pass})}$$

Inconclusive : add a new transition with verdict Inconclusive from a state in the DAG which allows an input reaching a state not in the DAG

$$\frac{\vdash^v (p^{TP}, p^S) : n \quad \not\vdash^u (q^{TP}, q^S) \quad u, v \in \{1,2,3\} \quad (p^{TP}, p^S) \xrightarrow{i}_P (q^{TP}, q^S) \quad i \in A_I}{\vdash^4 (q^{TP}, q^S) : 1 \quad (n, i, 1) \in DAG \quad \texttt{verdict}(n, i, 1) = \text{Inconclusive}}$$

Fail : in each state of the DAG, an input of the implementation which is not allowed in the specification should produce a Fail verdict. A new transition with verdict Fail is (virtually) added. In practice this corresponds to an *Otherwise Fail* in TTCN.

$$\frac{\vdash^{v} (p^{\mathrm{TP}}, p^{\mathrm{S}}) : n \quad v \in \{1,2,3\} \quad ((i \in A_I, \neg((p^{\mathrm{TP}}, p^{\mathrm{S}}) \xrightarrow{i}_{\mathrm{P}})) \text{ or } i \in A_I^{\mathrm{I}} \setminus A_I)}{\vdash^{5} \mathit{Fail\_State} : 1 \quad (n,i,1) \in \mathit{DAG} \quad \mathtt{verdict}(n,i,1) = \mathrm{Fail}}$$

Timers Timers are useful in practice in order to insure against implementation deadlocks. The management of timers is made on the DAG generated by the test generation rules 3.2. As timers depend on inputs, we associate a timer t_i to each input i labelling a transition in the test case. Three operations on a timer are available: $\mathtt{Start}(t_i)$ which initializes the timer and must be done as soon as input i is expected, $\mathtt{Cancel}(t_i)$ which is done when i is received or when, due to a choice, i is no more expected, and $\mathtt{Timeout}(t_i)$ which represents the observation of a deadlock when waiting for i.

Let $(p^{\mathrm{TP}}, p^{\mathrm{S}}) \xrightarrow{\alpha}_{\mathrm{T}} (q^{\mathrm{TP}}, q^{\mathrm{S}})$ be a transition of the synchronous product and $t : (n, \alpha, m)$ the corresponding transition in the DAG. Let $\mathtt{Ind}((p^{\mathrm{TP}}, p^{\mathrm{S}}))$ be the independency relation which represents the concurrency. The independency relation is a binary symmetrical relation defined on the inputs of a state: two inputs are independant if they may be received in any order. We denote by $\mathit{Running}(n)$ the set of timers that have been started in the sequences leading to n and have not yet been cancelled, Cl and St are sets of timers that have to be respectively cancelled or started after action α. Finally, $\mathtt{discard}(t)$ means that t is discarded from the DAG. The following rules specify the timers management:

Init As specifications, test cases start with an output, thus if r is the root of the DAG, $\mathit{Running}(r) = \emptyset$

Cancel and Start

$$\frac{t : (n, \alpha, m) \in \mathit{DAG} \quad \mathit{Running}(n) = R}{t' : (n, \alpha; \mathtt{Cancel}(Cl); \mathtt{Start}(St), m) \in \mathit{DAG} \quad \mathit{Running}(m) = (R \setminus Cl) \cup St \quad \mathtt{discard}(t)}$$

where

$$Cl = \{t_i | i \in \mathcal{I}((p^{\mathrm{TP}}, p^{\mathrm{S}})) \wedge (\alpha, i) \notin \mathtt{Ind}((p^{\mathrm{TP}}, p^{\mathrm{S}}))\}$$
$$St = \{t_i | i \in \mathcal{I}((p^{\mathrm{TP}}, p^{\mathrm{S}}))\} \setminus (R \setminus Cl)$$

i.e. all timers corresponding to inputs not concurrent with α must be cancelled and a timer must be started for each input available in m if it is not already running in n, except if it has just been cancelled.

Timeouts We suppose that $\delta \in A_I$. By the construction and verdict rules, in each node of the DAG, there is a transition labelled δ and its verdict may be (PASS) if δ is in the test purpose, Inconclusive if it is in the specification or Fail otherwise. If an input i (i may be δ) is possible in a state of the synchronous product, a transition labelled by $\mathtt{Timeout}(t_i)$ is added. The verdict assigned to this timeout must be the same as the verdict assigned to δ.

$$\frac{t : (n, i, m) \in \mathit{DAG} \quad i \in A_I \quad \mathtt{verdict}(t) \neq \mathit{Fail} \quad t' : (n, \delta, 1) \in \mathit{DAG}}{t'' : (n, \mathtt{Timeout}(t_i), 1) \in \mathit{DAG} \quad \mathtt{verdict}(t'') = \mathtt{verdict}(t')}$$

Discard δ For each transition $t : (n, \delta, 1) \in \mathit{DAG}$, apply $\mathtt{discard}(t)$

Another depth first search is performed on the DAG to generate the timers operations. Unlike the DAG construction, which works by synthesis (just around the pop operation) the operations on timers are generated before the exploration of the state successors (around the push operation). The running set associated with each state is initialized to empty set at the initial state. It is inherited from a state to its successor. During this step, on one hand, each transition of the DAG is decorated with **cancel** and **start** operations on timers, on the other hand some transitions labelled by **timeout** are added, following the previous rules.

3.3 Results

Proposition 3.1 *Let P be the synchronous product between S and TP (definition 3.1) and $\mathcal{T}(TP,S)$ be the DAG synthesized by applying the rules of definition 3.1. If $(q^{TP}_{init}, q^{S}_{init})$ is the root of the dag, then $q^{TP}_{init} \prec q^{S}_{init}$ else the test purpose and the specification are not consistent.*

Let $OT(S) = \{TP \in IOLTS \mid q^{TP}_{init} \prec q^{S}_{init}\}$ be the set of test purposes which are consistent with respect to the specification S. Let $TS(S) = \{\mathcal{T}(TP,S) \mid TP \in OT(S)\}$ i.e. the set of test cases (test suite) that can be constructed for a specification S. For $\mathcal{T} \in TS(S)$, we denote $Max\_traces(\mathcal{T}) = \{\sigma \mid \mathcal{A}(\mathcal{T}\ \mathbf{after}\ \sigma) = \emptyset\}$ the set of maximal traces of $\mathcal{T}$. For $\sigma = \sigma'.\alpha \in Max\_Traces(\mathcal{T})$, and for an implementation I, we define $\mathbf{verdict}(\sigma, I) = \mathbf{verdict}(\mathcal{T}\ \mathbf{after}\ \sigma', \alpha, \mathbf{1})$. Notice that $\mathcal{T}$ is deterministic, thus $\mathcal{T}$ **after** σ' is unique. We have the two following results:

Proposition 3.2 *(Soundness) Assuming that timeouts are produced if and only if the implementation is deadlocked, for every implementation I, if the application of a test case $\mathcal{T} \in TS(S)$ produces a Fail verdict then I does not conform with S:*
$(\exists \mathcal{T} \in TS(S), \exists \sigma \in Max\_Traces(\mathcal{T}), \mathbf{verdict}(\sigma, I) = Fail) \Rightarrow \neg(I\ \mathbf{ioconf} S)$.

This second proposition is not exactly the converse. Implementations can be non deterministic. Thus the application of the same sequence of actions of the tester may produce different verdicts. Thus, like other authors [Pha94a], we assume a bounded fairness hypothesis on implementations. This informally means that a bounded number of executions of a non deterministic implementation will show all its behaviours. For $n \in \mathbb{N}$, we define $\mathbf{verdict}^*(n, \sigma, I)$ to be Fail if one of the n applications of σ on I produces a Fail verdict, Pass otherwise.

Proposition 3.3 *(Exhaustivity) For every implementation I, if I does not conform with S, there exists a test case $\mathcal{T} \in TS(S)$ which can produce a Fail verdict:*
$\neg(I\ \mathbf{ioconf}\ S) \Rightarrow (\exists \mathcal{T} \in TS(S), \exists \sigma \in Max\_Traces(\mathcal{T}), \exists n \in I\!N, verdict^*(n, \sigma, I) = Fail)$.

4 Experimentation

The algorithms and transformations described in previous sections have been developed in the CADP toolbox [FGM+92] as a software component named TGV (for Test Generation using Verification techniques). In order to prove the feasibility of the approach, we have applied TGV to an industrial protocol, the DREX protocol.

4.1 TGV

As we were primary interested by demonstrating the feasibility of our approach before a real implementation, all algorithms are not yet combined into a unique on the fly algorithm. We have used the Geode simulator [ALHH93] from Verilog as an SDL [CCI88] front-end which produces state graphs representing the behaviour of a specification, constrained by the test purpose constraints.

Thus the inputs of TGV are a state graph produced by Geode (from a SDL specification of the protocol) and an automaton formalizing the behavioural part of a test purpose. The output is the behaviour description and constraints definitions of a test case in the standard TTCN format [ISO92].

Different steps bring out this output. The first step takes as input the state graph produced by Geode and transforms it into a graph representing the observable behaviour of the protocol specification in the testing environment (*external view graph*). Several transformations are performed in this step: abstraction of unobservable internal actions, determinization, mirror image which transforms inputs into outputs and vice versa and construction of diamonds modelling concurrency introduced by the asynchronous interaction between the tester and the IUT. The next step is the kernel of TGV. The output is the DAG which contains all informations needed in TTCN test cases. The last step takes as input the DAG. The algorithm extracts from the transition labels the message parameters and produces the constraint part in TTCN GR format. The remaining graph is unfolded into a tree describing the behavioural part of the test case in TTCN GR format. Finally the constraint and behavioural parts of the test case are translated into the graphical format TTCN GR.

4.2 Experiment with the DREX protocol

TGV has been used during an industrial contract for the *Direction Générale pour l'Armement*. The protocol used for the experiment was a military protocol called the DREX protocol which allows the access to the transit network Socrate of the French Army, defined in the framework of Integrated Service Military Network. This protocol has been chosen for three main reasons: firstly, we wanted to prove the feasibility of automatic test generation methods on realistic specifications; secondly, an SDL specification of a similar protocol was already available, and finally, hand written test suites had already been produced. This last point is important as hand written test cases have served as a basis for comparison with automatically generated test suites.

The SDL specification models the behaviour of the DREX protocol on the network, communicating asynchronously with two users by two PCOs. The size of the SDL specification was about 2000 lines. 54 test purposes have been considered and 54 corresponding test cases have been generated. The time needed for the generation of a test case has to be separated into two parts: the time needed for the graph generation with Geode which took between 3.5s and 400s and the test case generation with TGV which took between 1s and 2s.

We have compared automatic test suites generated by TGV with hand written test suites in a qualitative way. Even though TGV is just a prototype, all hand written test suites or similar ones have been generated. The differences that were observed were principally due to the fact that TGV treats systematically concurrency and timers. For example, in some hand written test cases, concurrency between events were forgotten and risked an incorrect verdict. Some differences were also due to the formal interpretation of test purposes. More details and other quantitative results of this study can be read in [FJJV96].

5 Conclusion

In this paper, we have shown how on-the-fly verification techniques could be used in the generation of test suites. Starting from an already known conformance relation and from the experiment gained with the analysis of hand written test cases, we have formally defined the rules allowing a construction of complete test cases, with preambles, postambles, verdicts and timers. These rules allowed us to prove that generated test cases are sound (correct implementations are not rejected) and exhaustive (if we assume a fairness hypothesis on implementations under test, incorrect implementation can be detected) with respect to the conformance relation. A depth first search algorithm implementing these rules has been described. A first version of this algorithm has been implemented in a prototype named TGV which produces TTCN test suites from SDL specifications. TGV has been experimented on an industrial protocol, proving the efficiency and maturity of the algorithm.

The next step in this study will be the development of a new prototype which will incorporate the algorithm described in this paper in a unique on-the-fly algorithm and its integration in a complete validation tool. Another continuation of the work is to deal with concurrent testing and links with interoperability testing.

References

[ALHH93] B. Algayres, Y. Lejeune, F. Hugonnet, and F. Hantz. The AVALON project : A VALidatiON Environment For SDL/MSC Descriptions. In *6th SDL Forum, Darmstadt*, 1993.

[Bri88] E. Brinksma. A theory for the derivation of tests. In S. Aggarval and K. Sabnani, editors, *Protocol Specification, Testing and Verification VIII, IFIP*, pages 63–74. Elsevier Science Publishers, B.V., North-Holland, 1988.

[CCI88] CCITT/SGx/WP3-1, Specification and Description Language, SDL. *CCITT Recommendation Z.100*, 1988.

[CGPT95] M. Clatin, R. Groz, M. Phalippou, and R. Thummel. Two approaches linking a test generation tool with verification techniques. In A. Cavalli and S. Budkowski, editors, *8th Int. Workshop on Protocols Test Systems, Evry, France*, pages 159–174, September 1995.

[CVWY90] C. Courcoubetis, M. Vardi, P. Wolper, and M. Yannakakis. Memory efficient algorithms for the verification of temporal properties. In E.M. Clarke and R.P. Kurshan, editors, *Computer Aided Verification, 2nd International Workshop, CAV'90, New Brunswick, NJ, USA*. Springer Verlag, LNCS 531, 1990.

[FGM+92] J.-C. Fernandez, H. Garavel, L. Mounier, A. Rasse, C. Rodriguez, and J. Sifakis. A Tool Box for the Verification of Lotos Programs. In *14th International Conference on Software Engineering*, Melbourne, Australia, May 1992.

[FJJV96] J.-C. Fernandez, C. Jard, T. Jéron, and G. Viho. An experiment in automatic generation of test suites for protocoles with verification technology. under revision for SCP, 1996.

[FM91] J.-C. Fernandez and L. Mounier. On the fly verification of behavioral equivalences and preorders. In K.G. Larsen and A. Skou, editors, *Computer Aided Verification, 3rd International Workshop, CAV'91, Aalborg, Denmark*, pages 181–190. Springer Verlag, LNCS 575, June 1991.

[FMJJ92] J.-C. Fernandez, L. Mounier, C. Jard, and T. Jéron. On-the-fly verification of finite transition systems. *Formal Methods in System Design*, 1:251–273, 1992. Kluwer Academic Publishers.

[ISO92] OSI-Open Systems Interconnection, Information Technology - Open Systems Interconnection Conformance Testing Methodology and Framework - Part 1 : General Concept - part 2 : Abstract Test Suite Specification - part 3 : The Tree and Tabular Combined Notation (TTCN). *International Standard ISO/IEC 9646-1/2/3*, 1992.

[JJ89] C. Jard and T. Jéron. On-line model-checking for finite linear temporal logic specifications. In J. Sifakis, editor, *Automatic Verification Methods for Finite State Systems, International Workshop, Grenoble, France*, pages 275–285. Springer–Verlag, LNCS 407, June 1989.

[JJ91] C. Jard and T. Jéron. Bounded memory algorithms for verification on the fly. In K.G. Larsen and A. Skou, editors, *Computer Aided Verification, 3rd International Workshop, CAV'91, Aalborg, Denmark*, pages 192–202. Springer Verlag, LNCS 575, June 1991.

[Pha94a] M. Phalippou. *Relations d'implantations et Hypothèses de Test sur des automates à entrées et sorties.* Thèse de doctorat, Université de Bordeaux, France, 1994.

[Pha94b] M. Phalippou. Test sequence using Estelle or SDL structure information. In *FORTE'94*, Berne, October 1994.

[Tre95] J. Tretmans. Testing Labelled Transition Systems with Inputs and Outputs. In A. Cavalli and S. Budkowski, editors, *8th Int. Workshop on Protocols Test Systems, Evry, France*, pages 461–476, September 1995.

Automatic Translation of Natural Language System Specifications into Temporal Logic

Rani Nelken[1] and Nissim Francez[2]

[1] Tel-Aviv University, Tel-Aviv 69978, Israel
[2] Computer Science Department, The Technion, Haifa 32000, Israel

Abstract. This paper presents a method for automatically translating natural language specifications into temporal logic. Using this method, users may express complex specifications in relatively free natural language, allowing multi-sentence specifications, the use of pronouns instead of repeating the description of previously mentioned objects and complex temporal relations. These specifications are translated into temporal logic, while ensuring the correctness of the translation. This approach overcomes a well-known obstacle of applying model-checking industrially. In contrast to prior attempts, the translation is linguistically based on a modern formalism for discourse representation. An implementation of this translation method is presented in one of the modern computational linguistics systems.

1 Introduction

Temporal logic based verification using model-checking has proved to be a useful and effective method for verifying finite-state concurrent systems. A recognized problem in industrially applying this verification technique is making the specification formalism more intuitive and easy to use. In practice, designers of computerized systems either lack sufficient training in formal methods, or find the specification formalism un-intuitive and inconvenient. Thus, the formulation of specifications often becomes a two stage process:

1. Specifications are written in natural language (NL).
2. These specifications are manually translated into temporal logic (TL). This is done according to the intuitive understanding of the translator, who has to contend with the imprecision and ambiguity of NL and with the subtle interpretation of TL formulae.

The frequent occurrence of imprecision in this translation is discussed in [4], who propose an automatic translation tool, based on a direct mapping between TL formulae and NL constructs. This method allows designers to express specifications in NL, but severely restricts the source language according to the structure of TL. Thus, the full power attainable by computational linguistic methods is not attained. It is not clear whether their system is grounded on any sound linguistic theory.

In [5], the translation of NL specifications of logic programs into Prolog is discussed. Input specifications in 'controlled' NL are automatically translated into Prolog clauses through an intermediary representation in a linguistic theory called *Discourse Representation Theory* (DRT) [8]. These clauses serve as a Prolog knowledge base, which may be executed and queried, and also paraphrased back in NL.

In this paper, we describe a novel translation method, which also uses DRT as an intermediate representation level. We largely enhance the NL constructs which may be translated, abstracting away from the target TL, and allowing the use of much freer, more natural language. Instead of isolated formula-like sentences as in [4], we deal with sequences of inter-connected specifications, which we term *specification discourses* (SDs). Our method allows a treatment of complex NL constructs such as:

- NP anaphora - the use of pronouns instead of repeating a full description of an object.
- Complex temporal expressions, which are crucial for TL verification. Expressing such properties is one of the major difficulties, which hinders designers from using TL directly. This is a major advancement over the treatment of temporal expressions in [5], who use the temporal capabilities of DRT in a very limited way. They distinguish between events and states and relate them with the utterance time, but do not explore temporal relations between events or states in discourse.

We further introduce a correctness criterion for the (second stage of the) translation, and formally prove that the translation method fulfills this criterion. Such a criterion and proof were lacking.

An implementation of this method, in the form of an interactive program is described. It accepts SDs, parses them, constructing a representation in DRT, called a *Discourse Representation Structure* (DRS), and then translates the DRS into a TL formula. The generated formulae can subsequently serve as input to a model-checker.

1.1 A Running Example

To illustrate the translation method, consider an allocator A that allocates a single resource to m different customer processes $C_1, C_2, \ldots, C_m$[3]. The communication between each C_i and A is done by a pair of shared boolean variables r_i (request) and g_i (grant). Following are statements of properties, that a designer may wish to formally specify and verify about the operation of this system. Each specification is given in both NL and our target temporal logic, ACTL[4,5] [6], a subset of Computational Tree Logic (CTL) [2]. These specifications will illustrate the problems encountered in NL translation as described in Sec. 1.2.

[3] This example is based on one in [14].

[4] The target formalism of [4] is called ACTL too. Nevertheless, it is a different formalism.

[5] We allow the use of the following additional operators:

(1) a. One cycle After r_i is activated, g_i should be asserted. r_i is deactivated one to six cycles later. Afterwards, it should be deasserted. (Response to requests)
$\mathbf{AG}\ [rise(r_i) \rightarrow \mathbf{AX}\ [g_i \& \mathbf{ABF}_{1..6}\ (fall(r_i) \& \mathbf{A}\ [\neg fall(g_i) \mathbf{U}\ fall(g_i)])]]$

b. If r_i is active and g_i is asserted one cycle later, then eventually r_i will be inactive. (Conformance with the protocol)
$\mathbf{AG}\ [r_i \rightarrow \mathbf{AX}\ (g_i \rightarrow \mathbf{AF}\ \neg r_i)]$

c. Whenever r_i and g_i are inactive, they remain inactive, until r_i is activated and g_i remains inactive. (Conformance with the protocol)
$\mathbf{AG}\ [\neg r_i \& \neg g_i \rightarrow \mathbf{A}\ [\neg r_i \& \neg g_i \mathbf{U}\ (rise(r_i) \& \neg g_i)]]$

d. Once r_i is activated, if g_j is asserted, then g_i will be activated before it is asserted again. (1-Bounded overtaking)
$\mathbf{AG}\ [rise(r_i) \rightarrow \mathbf{AG}\ [g_j \rightarrow \mathbf{A}\ [\neg rise(g_j) \mathbf{U}\ rise(g_i)]]]$

1.2 Problems in the Analysis of NL Specifications

Following are characteristics of SDs, which should be taken into account when translating NL. They stem from NL in general, the specific structure of SDs and the gap between the source and target languages.

Ambiguity and imprecision are exhibited by SDs and must be resolved in the translation into TL.
Example: in sentence (1c) it is unclear which two clauses are conjoined by the conjunction 'and'. The second conjunct is 'g_i remains inactive', but the first is either: 'r_i is activated' or 'they remain inactive until r_i is activated'.

Inter-sentential links The natural way of expressing specifications involves multi-sentence discourses. These are not just lists of isolated sentences but rather coherent objects.
Example: the second sentence of example (1a) is crucially dependent on its predecessor, i.e. the phrase 'one to six cycles later' can be interpreted only by reference to the first sentence.

The temporal structure of specifications Specifications exhibit a unique temporal structure, causing SDs to often be expressed in a generic present or future tense:
- *Disconnection from the present:* SDs are often expressed in a timeless language.
Example: Even though the tense of the verbs in sentence (1b) is simple present, it does not refer to the present moment per se (e.g. the time of writing the sentence).

- $rise(p)$ and $fall(p)$ for $p \in AP$, where AP is the set of atomic propositions - to represent a change from $\neg p$ to p and vice versa.
- $\mathbf{AX}_i\ f \equiv \underbrace{\mathbf{AX}\ (\mathbf{AX}\ (\ldots(\mathbf{AX}\ f)))}_{\times i}$
- $\mathbf{ABF}_{i..j}\ f \equiv \mathbf{AX}_i\ (f \vee \underbrace{\mathbf{AX}\ (f \vee \ldots \mathbf{AX}\ (f \vee \mathbf{AX}\ f))}_{\times(j-i)})$

- *Quantification over events:* Specifications are usually concerned with recurring events in the course of a system's operation. Consequently, SDs contain either explicit or implicit quantification over events and time periods.
  Example: Sentence (1b) is understood as referring to *each* time r_i is active.

Non determinism Our target formalism is ACTL, a branching-time TL. In practice, designers tend to specify the behavior of systems in linear-time terms. When translating SDs into a branching-time TL, this linearity has to be introduced into the formalism. A simpler solution to this problem would be choosing a linear time TL, e.g. LTL, as our target formalism. The choice of ACTL in this paper is motivated by its widespread and successful use (e.g. [15, 1]).
Example: the meaning of sentence (1d) is clear in a linear-time model. However, it may have several branching-time TL interpretations, e.g. assume a branching structure at the root of which r_i is asserted, but where g_j is asserted along only one path from the root. Along which paths should g_i be activated ?

We solve these problems through the use of DRT, presented in Sec. 2, and a restriction on the generated formulae, explained in Sec. 4.2.

2 Discourse Representation Theory

DRT [8, 9, 10] is a linguistic theory of the semantic content of general NL, which studies discourses. It combines a static logical view of meaning with a dynamic cognitive view. DRSs are defined as formulae of a formal language $\mathcal{L}$ consisting of:

1. an infinite set R of typed *markers*: $x\,y\,z$ for signals, $s, s_1, s_2, \ldots s_n$ for states. $e, e_1, e_2, \ldots e_n$ for events and i, j for integers.
2. for each $n \in \mathbb{N}$ an infinite set P^n of n-place predicates. $V = \bigcup_n P^n$
3. identity

Definition 1. 1. A DRS $K = \langle U_K, Con_K \rangle$, where $U_K \subset R$ is finite and Con_K is a finite set of DRS-conditions.
2. Let K, K_1, K_2 be DRSs, $r_1, r_2, \ldots r_n \in R$ and $P \in P^n$. A DRS-condition may be one of: $r_1 = r_2$, $P(r_1, r_2, \ldots, r_n)$,$\neg K$, $K_1 \Rightarrow K_2$, $K_1 \vee K_2 \vee \ldots \vee K_n$

DRSs are interpreted as partial models. A DRS is verified by a model[6] M, iff it may be embedded in it as follows:

Definition 2. A model for $\mathcal{L}$ is $M = \langle U_M, \mathcal{EV}, \mathbb{N}, Pred_M \rangle$ where:

1. U_M is a non-empty set.

[6] In Sec. 4.2, we make some changes in the following definition of a model.

2. $\mathcal{EV}$ is an event structure (see [10]).
3. $\mathbb{N}$ is the set of natural numbers.
4. $Pred_M$ maps each n-place predicate of $\mathcal{L}$ onto an n-place relation over U_M.

Definition 3. Let K be a DRS, γ a DRS-condition, and let f be an **embedding** from some subset of V into M, i.e. $f : R' \rightarrow U_M$, where $R' \subseteq R$.

1. f verifies the DRS K in M ($M \models_f K$) iff f verifies each $\gamma \in Con_K$ in M.
2. f verifies the condition γ in M ($M \models_f \gamma$) iff
   (a) γ is of the form $r_1 = r_2$, $r_1, r_2 \in Dom(f)$ and $f(r_1) = f(r_2)$.
   (b) γ is of the form $P(r_1, r_2, \ldots, r_n)$, $r_1, r_2, \ldots r_n \in Dom(f)$ and $\langle f(r_1), f(r_2), \ldots, f(r_n)\rangle \in Pred_M(P)$
   (c) γ is of the form $\neg K'$ and there is no embedding $g : R \rightarrow U_M$, which extends f, s.t. $Dom(g) = Dom(f) \cup U_{K'}$ and $M \models_g K'$
   (d) γ is of the form $K_1 \Rightarrow K_2$ and for every embedding g s.t. $Dom(g) = Dom(f) \cup U_{K_1}$ and $M \models_g K_1$, there is an extension h of g s.t. $Dom(h) = Dom(g) \cup U_{K_2}$ and $M \models_h K_2$.
   (e) γ is of the form $K_1 \vee K_2 \vee \ldots \vee K_n$ and for some i $1 \leq i \leq n$ $M \models_f K_i$.

DRSs are constructed from NL discourses by the *DRS-construction algorithm* (Fig. 1), which processes sentences one by one, incrementally updating a DRS according to a set of *DRS-construction rules*[7]. The algorithm models the way in which a human listener processes a discourse, understanding sentences one at a time. DRT provides an analysis of the 'glue' that holds a discourse together, most prominently, it is able to resolve the meaning of anaphoric pronouns in discourse. Full DRT also provides a thorough analysis of temporal relations within discourse [7, 17, 10, 16]. While this analysis is not described in this paper, it is illustrated through the DRS-construction for example (1a).

```
Input: Specification Discourse D = ⟨S_1, S_2, ..., S_n⟩
ContextDRS ← EmptyDRS
i ← 0
repeat
        DRS_i ← parse(S_i)
        ContextDRS ← UpdateContext(ContextDRS,DRS_i)
        i ← i + 1
until i = n
Output: ContextDRS
```

Fig. 1. DRS-construction algorithm

[7] These rules, i.e. the details of *UpdateContext* are given in [10] and are not presented here for lack of space.

3 DRS-Construction for NL Specifications

We illustrate the construction of DRSs for SDs through the stepwise construction of a DRS for example (1a), repeated here as (2).

(2) a. One cycle after r_i is activated, g_i should be asserted.
b. r_i is deactivated one to six cycles later.
c. Afterwards, it should be deasserted.

DRSs are depicted using a graphical box-notation. The top of the box corresponds to the universe of the DRS, and the rest to the DRS-conditions. The DRS-construction algorithm constructs the DRS[8] shown in Fig. 2.

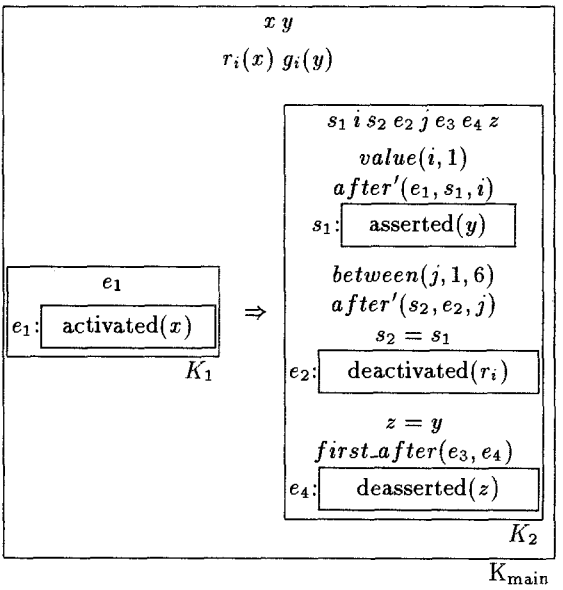


Fig. 2. Example DRS

a. Processing sentence (2a) introduces the discourse markers x, y into $U_{K_{main}}$, the first two DRS-conditions naming x and y r_i and g_i and the implication condition $K_1 \Rightarrow K_2$, excluding the part of K_2 following the condition: s_1:[...]. The embedding conditions of an implication $K_1 \Rightarrow K_2$ determine that it induces universal quantification over the discourse referents of U_{K_1} and existential over those of U_{K_2}, giving an interpretation that for each event e_1, in which x is activated, there is a state s_1, in which y is asserted,

[8] For reference purposes DRSs are labeled. The labeling is part of the meta-language used to discuss DRSs.

which follows the occurrence of e_1 by one time cycle. The introduction of the implication condition is due to the implicit quantification inherent in (2a).

b. Parsing sentence (2b) produces a new DRS. This DRS is combined with the DRS for the previous sentence, adding into K_2 the discourse markers s_2, e_2, j and the DRS-conditions $between(\ldots)$, $after'(s_2, e_2, j)$, e_2:[$\cdots$]. When combining the DRSs, the 'anonymous' eventuality marker s_2 is identified with s_1, thus determining that 'later' here means 'after the assertion of g_i'.

c. Parsing sentence (2c) generates an additional DRS, which is combined with the compound DRS of sentences (2a) and (2b) to form the completed DRS of Fig. 2. The treatment of 'afterwards' is identical to that of 'later'. This sentence also contains the pronoun 'it', which causes the introduction of the marker z, which is later identified with y by a similar technique.

We introduce a set of restrictions on the structure of DRSs generated for SDs, not described here for lack of space. A DRS thus restricted is called a *Specification DRS* (SPDRS), and the set of SPDRSs, SPDRT. These restrictions are a result of the subset of NL accepted by the translation method and the DRS-construction rules.

4 Translating DRSs into ACTL

4.1 The Translation Method

We sketch the translation procedure $trans : SPDRT \Rightarrow ACTL$. We illustrate its operation on the DRS of Fig. 2:

Translating K_{main}: $K_1 \Rightarrow K_2$ generates a formula starting in **AG** . This reflects the universal quantification on the elements of U_{K_1} conferred by the embedding[9].

Translating $K_1 \Rightarrow K_2$: The temporal relation[10] $after'(e_1, s_1, i)$ generates an operator $\mathbf{AX}_i$. Let f_{K_2} be the translation of the remaining conditions of K_2. The translation is $\mathbf{AG}\,[rise(r_i) \rightarrow \mathbf{AX}\,(g_i \& f_{K_2})]$, where $rise(r_i)$ is the translation of the *event* e_1, and g_i that of the *state* s_1.

Translating the remaining conditions of K_2: The translation is driven by the $after'$ and $first\_after$ conditions. The translation of the $first\_after$ generates the formula $f_2 = fall(r_i) \& \mathbf{A}\,[\neg fall(g_i) \mathbf{U}\ fall(g_i)]$. The translation of the first one generates the formula $g_i \& \mathbf{ABF}_{1..6}\,(fall(r_i) \& f_2)$.

The resulting translation is therefore the following, which is equivalent to (1a):

$$\mathbf{AG}\,[rise(r_i) \rightarrow \mathbf{AX}\,(g_i \& \mathbf{ABF}_{1..6}\,(fall(r_i) \& fall(r_i) \& \mathbf{A}\,[\neg fall(g_i) \mathbf{U}\ fall(g_i)]))]$$

[9] In general, the main DRS may contain several implications, in which case its translation is the conjunction of such formulae.

[10] Different temporal relations generate different operators.

4.2 Correctness

The translation method consists of two major transitions: from NL to DRT, and from DRT to TL. We define and prove a correctness criterion only for the second stage of the translation. No correctness criterion can be defined for the first stage of the translation. Given a NL SD and DRS, such a criterion would determine whether the DRS correctly represents the meaning of the NL specification. This criterion would require an alternative formal interpretation of the NL SD, the validity of which should also be somehow defined and proved. Therefore, we cannot hope for a mathematical correctness criterion for the DRS-construction algorithm. We can only check whether the constructed DRSs conform to our linguistic knowledge about the meaning of the NL SDs they are meant to represent. Thus, we benefit from the wealth of linguistic research dedicated for the purpose of providing a semantic analysis of NL.

Kripke Structure Induced DRT Models In order to ensure the correctness of the translation from DRT to TL, we link the models used for the interpretation of both theories. Through this linking, we solve the dichotomy between the linear-time interpretation of SDs and DRSs, and the branching-time interpretation of ACTL formulae.

Definition 4. Given a path $\pi = s_0, s_1, \ldots$ in a Kripke structure P, a *single path structure* $P(\pi)$ is a tuple $\langle S_\pi, s'_0, R_\pi, L_\pi \rangle$ such that

- $S_\pi = \{s'_0, s'_1, \ldots\}$ is an infinite set of (pairwise distinct) states.
- $R_\pi = \{(s'_i, s'_{i+1}) | i \geq 0\}$
- $\forall i \geq 0\ L_\pi(s'_i) = L(s_i)$

We expand the definition of satisfaction of CTL* formulae (defined only for structures having a finite set of states) to also include satisfaction of formulae by single path structures.

Definition 5. For any Kripke structure P, its induced DRT model M_P is constructed as follows:

- For each $p \in AP$,a binary signal of M_P
- For each each path $\pi \in \Pi_P$, the set of paths of P, a DRT path-model M_π as follows: for each state of $P(\pi)$, a state of M_π, and for each pair of consecutive states, an event.

Verification for DRT path models is as in Definition 3. We define $M_P \models K$ iff each of the induced path-models $M_\pi \models K$.

Based on this construction, we present the following correctness criterion:

Definition 6. An ACTL formula f is a correct translation of a DRS K iff for every Kripke structure P and its induced DRT model M_P: $M_P \models K \iff P \models f$

Theorem 7 shows that the translation method conforms to this correctness criterion. The proof of this theorem is based on the reduction of correctness to single paths.

Theorem 7. *Let K be an SPDRS, and $f = trans(K)$. f is a correct translation of K into ACTL.*

Reduction of Correctness to Single Paths In [6] three linearity properties for branching time temporal logics are defined: *strong linearity*, *sub-linearity* and *equi-linearity*. We take advantage of the strongest of these properties, strong-linearity, by restricting the formulae generated by the translation to the subset of strong-linear formulae.

Definition 8. [6] A formula $f \in$ ACTL is strong-linear, iff there is an ω-regular language $\mathcal{L}_f$, such that for every Kripke structure P, $P \models f \Longleftrightarrow \mathcal{L}(P) \subseteq \mathcal{L}_f$

Lemma 9. *If K is an SPDRS, then $trans(K)$ is a strong-linear ACTL formula.*

Lemma 10 asserts that for a strong-linear formulae f, the satisfaction of f by a Kripke structure P depends only on the satisfaction of f by each individual path of P, regardless of the way they are interleaved.

Lemma 10. *If $f \in$ ACTL is strong-linear, then for every Kripke structure P, $[\forall \pi \in \Pi_P, P(\pi) \models f] \Longleftrightarrow P \models f$*

Strong-linearity allows us to reduce the correctness of the translation of an SPDRS by a Kripke structure to its correctness relative to single path structures, as in the following lemma illustrated by Fig. 3.

Lemma 11. *Let f be a strong-linear ACTL formula and K an SPDRS. If for every Kripke structure P and its induced DRT model M_P: $\forall \pi \in \Pi_P[M_\pi \models K \Longleftrightarrow P(\pi) \models f]$ where M_π is the path model associated with the single path structure $P(\pi)$, then f is a correct translation of K.*

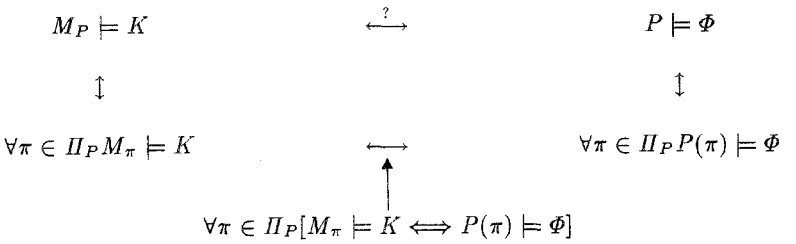


Fig. 3. Reduction of the correctness to correctness for paths

5 The Implementation

The translation method described above has been implemented as an interactive program **SpecTran 1.0**, which receives as input SDs, parses them, constructs DRSs and generates an ACTL formula from the DRS. The parser is written using the LexGram system [11, 12], which is based on a synthesis of ideas from *Lambek Categorial Grammar* [13] and *Head Driven Phrase structure Grammar* (HPSG) [18]. LexGram is written in a unification-based grammar formalism, CUF [3], and in Prolog.

In [11] a German grammar is implemented. The system parses sentences, constructing syntactic tree representations and semantic representations in the form of DRSs for them. In **SpecTran** the German grammar was replaced by an English one. The syntax part of this grammar was written by Esther König. **SpecTran**, parses input SDs sentence after sentence, processing each new sentence and updating a single DRS. In cases of ambiguity, the parser produces all the alternative parses of a sentence and their related DRSs.

SpecTran consults the user with respect to the resolution of the meaning of pronouns (e.g. 'it', 'they'), allowing a choice between a set of appropriate alternatives. A similar consultation is done with respect to the resolution of the meaning of words such as 'later'.

The completed DRS after the parsing of the full discourse, is passed in the form of a CUF data structure into the DRT to TL translation module. This module, written in CUF as well, implements the translation procedure described above. It accepts SPDRSs and translates them into strong-linear ACTL formulae.

6 Conclusion

In this paper, we have presented a translation method from NL specifications into TL, for the purpose of verification. Through the use of computational linguistic methods, we allow the expression of complex specifications in NL. While completely unrestricted NL is beyond the reach of current technology, and is arguably an undesirable medium for expressing specifications, we allow the use of relatively convenient language within certain restrictions. It is still necessary for designers to write specifications in precise and concrete language, but some tolerance is allowed in the use of flexible syntactic structure, multi-sentence discourse, pronominal anaphora and complex inner-sentential and inter-sentential temporal relations. By drawing on current linguistic research in the analysis of NL discourses, we enhance the applicability of an NL interface to a model-checker. It is our belief that the use of such an interface may facilitate the verification process in industrial practice, without harming the correctness of the verification. By introducing and proving a correctness criterion for the (second stage) of the translation and drawing on linguistic research for the first part of the translation, we are able to guarantee that the transformation from NL into TL does not introduce errors. Such a guarantee is lacking for the manual translation

process often exercised in practice, and from previous attempts of automatic translation.

Acknowledgments

The work of the second author was partially supported by a grant from the Israeli ministry of science "Programming languages induced computational linguistics", and by the fund for the promotion of research in the Technion. We also wish to thank the following people: Uwe Reyle, Esther König, Orna Grumberg, Ilan Beer and Shoham Ben-David.

References

1. E. M. Clarke, O. Grumberg, H. Hiraishi, S. Jha, D. E. Long, K. L. McMillan, and L. A. Ness. Verification of the futurebus+ cache coherence protocol. In L. Claesen, editor, *Proceedings of the Eleventh International Symposium on Computer Hardware Description Langugages and their Applications*, North Holland, 1993.
2. E.M. Clarke, E.A. Emerson, and A.P. Sistla. Expressibility results for linear time and branching time logic. *ACM transactions on Programming Languages and Systems*, 8(2):244–263, 1986.
3. J. Dörre and M. Dorna. Cuf - a formalism for linguistic knowledge representation. In J. Dörre, editor, *Computational Aspects of Constraint - Based Linguistic Description*, volume I. ILLC/Department of Philosophy, University of Amsterdam, Amsterdam, 1993. DYANA 2 deliverable R.1.2.A.
4. A. Fantechi, S. Gnesi, G. Ristori, M. Carenini, M. Vanocchi, and P. Moreschini. Assisting requirement formalization by means of natural language translation. *Formal Methods in System Design*, 4:243–263, 1994.
5. N. E. Fuchs and R. Schwitter. Specifying logic programs in controlled natural language. In *Proceedings of the Workshop on Computational Logic for Natural Language Processing. A Joint COMPULOGNET/ELSNET/EAGLES Workshop*, Edinburgh, 1995.
6. O. Grumberg and R.P. Kurshan. How linear can branching-time be? In D. M. Gabbay and H. J. Ohlbach, editors, *First International Conference on Temporal Logic (ICTL'94). Lecture Notes in Artificial Intelligence 827*, pages 180–194, Bonn, Germany, 1994. Springer-Verlag.
7. E. W. Hinrichs. Temporale anaphora im englischen. Unpublished Statexamen Thesis. University of Tuebingen., 1981.
8. H. Kamp. A theory of truth and semantic representation. In T. Janssen J. Groenendijk and M. Stokhof, editors, *Formal Methods in the Study of Language*, pages 277–322. Mathematical Center, Amsterdam, 1981.
9. H. Kamp and U. Reyle. A calculus for first order discourse representation structures. Bericht Nr.16-1991 Arbeitspapier des Sonderforschungsbereich 340, Institut für Maschinelle Sprachverarbeitung, Universität Stuttgart, 1991.
10. H. Kamp and U. Reyle. *From Discourse to Logic*, volume 42 of *Studies in Linguistics and Philosophy*. Kluwer Academic Publishers, 1993.
11. E. König. A study in grammar design. Arbeitspapier 54 des Sonderforschungsbereich 340. Institut für Maschinelle Sprachverarbeitung, Universität Stuttgart, 1994.

12. E. König. Lexgram - a practical categorial grammar formalism. In *Proceedings of the Workshop on Computational Logic for Natural Language Processing. A Joint COMPULOGNET/ELSNET/EAGLES Workshop*, Edinburgh, Scotland, 1995.
13. J. Lambek. The mathematics of sentence structure. *American mathematical monthly*, 65:154–170, 1958.
14. Z. Manna and A. Pnueli. *The temporal logic of reactive and concurrent systems.* Springer - Verlag, 1991.
15. K. L. Mcmillan. *Symbolic Model Checking: An Approach to the State Explosion Problem.* PhD thesis, Carnegie Mellon University, 1992.
16. R. Nelken and N. Francez. Splitting the reference time:temporal anaphora and quantification. In *Proceedings of the EACL '95 - The seventh meeting of the European Chapter of the Association for Computational Linguistics*, Dublin, Ireland, 1995. Also available as Technical Report LCL-94-10 of the Laboratory for Computational Linguistics, The Technion IIT.
17. B. Partee. Nominal and temporal anaphora. *Linguistics and Philosophy*, 7:243–286, 1984.
18. C. Pollard and I. A. Sag. *Head Driven Phrase Structure Grammar.* University of Chicago Press, Chicago, 1994.

Verification of Fair Transition Systems

Orna Kupferman[1] and Moshe Y. Vardi[2]

[1] Bell Laboratories, 600 Mountain Avenue, Murray Hill, NJ 07974, U.S.A.
Email: ok@research.att.com
[2] Rice University, Department of Computer Science, P.O. Box 1892, Houston, TX 77251-1892, U.S.A.
Email: vardi@cs.rice.edu, URL: http://www.cs.rice.edu/~vardi

Abstract. In *program verification*, we check that an *implementation* meets its *specification*. Both the specification and the implementation describe the possible behaviors of the program, though at different levels of abstraction. We distinguish between two approaches to implementation of specifications. The first approach is *trace-based implementation*, where we require every computation of the implementation to correlate to some computation of the specification. The second approach is *tree-based implementation*, where we require every computation tree embodied in the implementation to correlate to some computation tree embodied in the specification. The two approaches to implementation are strongly related to the linear-time versus branching-time dichotomy in temporal logic.

In this work we examine the trace-based and the tree-based approaches from a *complexity-theoretic* point of view. We consider and compare the complexity of verification of *fair transition systems*, modeling both the implementation and the specification, in the two approaches. We consider *unconditional*, *weak*, and *strong* fairness. For the trace-based approach, the corresponding problem is *language containment*. For the tree-based approach, the corresponding problem is *fair simulation*. We show that while both problems are PSPACE-complete, their complexities in terms of the size of the implementation do not coincide and the trace-based approach is more efficient. As the implementation is normally much bigger than the specification, we see this as an advantage of the trace-based approach. Our results are at variance with the known results for the case of transition systems with no fairness, where the tree-based approach is more efficient.

1 Introduction

In *program verification*, we check that an *implementation* meets its *specification*. Both the specification and the implementation describe the possible behaviors of the program, but the implementation is more concrete than the specification, or, equivalently, the specification is more abstract than the implementation (cf. [AL91]). This basic notion of verification suggests a top-down method for design development. Starting with a highly abstract specification, we can construct a sequence of "behavior descriptions". Each description refers to its predecessor as a specification, so it is less abstract than its predecessor. The last description contains no abstractions, and constitutes the implementation. Hence the name *hierarchical refinement* for this methodology (cf. [LS84, LT87, Kur94]).

We distinguish between two approaches to implementation of specifications. The first approach is *trace-based implementation*, where we require every computation of the implementation to *correlate* to some computation of the specification. The second approach is *tree-based implementation*, where we require every computation tree embodied in the implementation to correlate to some computation tree embodied in the specification. The exact notion of correct implementation then depends on how we interpret correlation. We can, for example, interpret correlation as identity. Then, correct trace-based implementation is one in which every computation is also a computation of the specification, and correct tree-based implementation is one in which every embodied computation tree is also embodied in the specification. Numerous interpretations of correlation are suggested and studied in the literature [Hen85, Mil89, AL91]. Here, we consider a very simple definition of correlation and interpret it as equivalence with respect to the variables joint to the implementation and the specification, as the implementation is typically defined over a wider set of variables, reflecting the fact that it is more concrete than the specification.

The tree-based approach is stronger in the following sense. If $\mathcal{I}$ is a correct tree-based implementation of the specification S, then $\mathcal{I}$ is also a correct trace-based implementation of S. As shown by Milner

[Mil80], the opposite direction is not true. The two approaches to implementation are strongly related to the linear-time versus branching-time dichotomy in temporal logic [Pnu85]. The temporal-logic analogy to the strength of the tree-based approach is the expressiveness superiority of ∀CTL*, the universal fragment of CTL*, over LTL [CD88]. Indeed, while a correct trace-based implementation is guaranteed to satisfy all the LTL formulas satisfied in the specification, a correct tree-based implementation is guaranteed to satisfy all the ∀CTL* formulas satisfied in the specification [GL94].

In this work we examine the traced-based and the tree-based approaches from a *complexity-theoretic* point of view. More precisely, we consider and compare the complexity of the problem of determining whether $\mathcal{I}$ is a correct trace-based implementation of $\mathcal{S}$, and the problem of determining whether $\mathcal{I}$ is a correct tree-based implementation of $\mathcal{S}$. The different levels of abstraction in the implementation and the specification are reflected in their sizes. The implementation is typically much larger than the specification and it is its size that is the computational bottleneck. Therefore, of particular interest to us is the *implementation complexity* of these problems; i.e., their complexity in terms of $\mathcal{I}$, assuming $\mathcal{S}$ is fixed.

We model specifications and implementations by *transition systems* [Kel76]. The systems are defined over the sets $AP_{\mathcal{I}}$ and $AP_{\mathcal{S}}$ of *atomic propositions* used in the implementation and specification, respectively. Thus, the alphabets of the systems are $2^{AP_{\mathcal{I}}}$ and $2^{AP_{\mathcal{S}}}$. Recall that usually the implementation has more variables than the specification. Hence, $AP_{\mathcal{I}} \supseteq AP_{\mathcal{S}}$. We therefore interpret correlation as equivalence with respect to $AP_{\mathcal{S}}$. In other words, associating computations and computation trees of the implementation with these of the specification, we first projet them on $AP_{\mathcal{S}}$. Within this framework, correct trace-based implementation corresponds to *trace containment* and correct tree-based implementation corresponds to *simulation* [Mil71]. Since simulation can be checked in polynomial time [Mil80, BGS92], whereas the trace containment problem is PSPACE-complete [MS72] [3], it seems that the tree-based approach is more efficient than the trace-based approach. This is reminiscent of the computational advantage of branching-time model checking over linear-time model checking [CES86, LP85, QS81, VW86].

Once, however, we want our implementations and specifications to describe behaviors that satisfy both *liveness* and *safety* properties, transition systems are too weak. Then, we need the framework of *fair transition systems*. We consider *unconditional*, *weak*, and *strong* fairness (also known as *impartiality*, *justice*, and *compassion*, respectively) [LPS81, Eme90, MP92]. Within this framework, correct trace-based implementation corresponds to *language containment* and correct tree-based implementation corresponds to *fair simulation* [BBLS92, ASB+94, GL94]. Hence, it is the complexity of these problems that should be examined when we compare the trace-based and the tree-based approaches.

We present a uniform method and a simple algorithm for solving the language-containment problem for all the three types of fairness conditions. Unlike [CDK93], we consider the case where both the specification and the implementation are nondeterministic, as is appropriate in a hierarchical refinement framework. We prove that the problem is PSPACE-complete for all the three types. For the case the implementation uses the unconditional or weak fairness conditions, our nondeterministic algorithm requires space logarithmic in the size of the implementation (regardless the type of fairness condition used in the specification). For the case the implementation uses the strong fairness condition, we suggest an alternative algorithm that runs in time polynomial in the size of the implementation. We show that these algorithms are optimal; thus the implementation complexity of language containment is NLOGSPACE-complete for implementations that use the unconditional or weak fairness conditions and is PTIME-complete for implementations that use the strong fairness condition. To prove the latter, we show that the nonemptiness problem for fair transition systems with a strong fairness condition is PTIME-hard, which is most likely harder than the NLOGSPACE bounds known for the unconditional and weak fairness conditions [VW94].

We also present a uniform method and a simple algorithm for solving the fair-simulation problem for all the three types of fairness conditions. Our algorithm uses the language-containment algorithm as a subroutine. We prove that the problem is PSPACE-complete for all the three types. Like Milner's algorithm for checking simulation [Mil90], our algorithm can be implemented as a calculation of a fixed-point expression, significantly improving its practicality. The running time of our algorithm is polynomial in the size of the implementation. We show that this is optimal; thus, the implementation complexity of

[3] The reduction in [MS72] considers containment of languages defined by regular expressions and can be extended to consider trace containment.

fair simulation is PTIME-complete for all types of fairness conditions. Proving the latter we prove that the implementation complexity of simulation (without fairness conditions) is PTIME-complete too.

Our results show that when we model the specification and the implementation by fair transition systems, the advantage of the tree-based approach disappears. Furthermore, when we consider the implementation complexity, then checking implementations that use unconditional or weak fairness conditions is easier in the trace-based approach.

2 Preliminaries

2.1 Fair Transition Systems

A *fair transition system* (*transition system*, for short) $S = \langle \Sigma, W, R, W_0, L, \alpha \rangle$ consists of an alphabet Σ, a set W of states, a total transition relation $R \subseteq W \times W$ (i.e., for every $w \in W$ there exists $w' \in W$ such that $R(w, w')$), a set W_0 of initial states, a labeling function $L : W \to \Sigma$, and a fairness condition α. We will define three types of fairness conditions shortly. A *computation* of S is a sequence $\pi = w_0, w_1, w_2, \ldots$ of states such that for every $i \geq 0$ we have $R(w_i, w_{i+1})$. In order to determine whether a computation is *fair*, we refer to the set $Inf(\pi)$ of states that π visits infinitely often. Formally,

$$Inf(\pi) = \{w \in W : \text{for infinitely many } i \geq 0, \text{ we have } w_i = w\}.$$

The way we refer to $Inf(\pi)$ depends in the fairness condition of S. Several types of fairness conditions are studied in the literature. We consider here three:

- *Unconditional* fairness (or *impartiality*), where $\alpha \subseteq W$, and π is fair iff $Inf(\pi) \cap \alpha \neq \emptyset$.
- *Weak* fairness (or *justice*), where $\alpha \subseteq 2^W \times 2^W$, and π is fair iff for every pair $\langle L, R \rangle \in \alpha$, we have that $Inf(\pi) \cap (W \setminus L) = \emptyset$ implies $Inf(\pi) \cap R \neq \emptyset$.
- *Strong* fairness (or *fairness*), where $\alpha \subseteq 2^W \times 2^W$, and π is fair iff for every pair $\langle L, R \rangle \in \alpha$, we have that $Inf(\pi) \cap L \neq \emptyset$ implies $Inf(r) \cap R \neq \emptyset$.

It is easy to see is that fair transition systems are essentially a notational variant of automata on infinite words [Tho90]. Thus, we will be able to use freely results from the theory of such automata. In particular, the unconditional and the strong fairness conditions correspond to the *Büchi* and *Streett* acceptance conditions.

For a state w, a w-computation is a computation $w_0, w_1, w_2, \ldots$ with $w_0 = w$. We use $\mathcal{L}(S^w)$ to denote the set of all words $\sigma_0 \cdot \sigma_1 \cdots \in \Sigma^\omega$ for which there exists a fair w-computation $w_0, w_1, \ldots$ in S with $L(w_i) = \sigma_i$ for all $i \geq 0$. The language $\mathcal{L}(S)$ of S is then defined as $\bigcup_{w \in W_0} \mathcal{L}(S^w)$. Thus, each transition system defines a subset of Σ^ω. We say that a transition system S is *empty* iff $\mathcal{L}(S) = \emptyset$; i.e., S has no fair computation. We sometimes say that S accepts w, meaning that $w \in \mathcal{L}(S)$.

The size of a transition system and its fairness condition, determine the *complexity* of solving questions about it. We define classes of transition systems according to these two characteristics. We write $\mathcal{U}, \mathcal{W}$, and $\mathcal{S}$ to denote the unconditional, weak, and strong fairness conditions, respectively. We measure the size of a transition system by its number of states (the number of edges is at most quadratic in the number of states) and, in the case of weak and strong fairness, also by the number of pairs in its fairness condition. For example, an unconditionally fair transition system with n states is denoted $\mathcal{U}(n)$. We also use a line over the transition system to denote the complementary transition system (one that accepts the complementary language). For example, the transition system complementing a strongly fair transition system with n states and m pairs is denoted $\overline{\mathcal{S}(n, m)}$.

2.2 The Language-Containment and the Fair-Simulation Problems

In this section we formalize correct trace-based and tree-based implementations in terms of language containment and fair simulation between an implementation $\mathcal{I}$ and a specification $\mathcal{S}$. Recall that $\mathcal{I}$ and $\mathcal{S}$ are given as fair transition systems over the alphabets $2^{AP_\mathcal{I}}$ and $2^{AP_\mathcal{S}}$ respectively, with $AP_\mathcal{I} \supseteq AP_\mathcal{S}$. For technical convenience, we assume that $AP_\mathcal{I} = AP_\mathcal{S}$; thus, the implementation and the specification are defined over the same alphabet. By taking, for each $\sigma \in 2^{AP_\mathcal{I}}$, the letter $\sigma \cap AP_\mathcal{S}$ instead the letter σ, all our algorithms and results are valid also for the case $AP_\mathcal{I} \supset AP_\mathcal{S}$.

Given two transition systems S and S' over the same alphabet, the *language containment* problem of S and S' is to determine whether $\mathcal{L}(S) \subseteq \mathcal{L}(S')$. That is, whether every word accepted by S is also accepted by S'. While language containment refers to each word in $\mathcal{L}(S)$ independently, *fair simulation* refers also to the branching structure of the transition system.

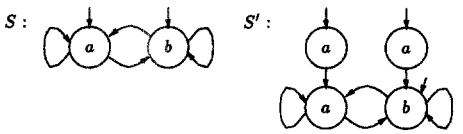
Let S and S' be two transition systems over the same alphabet and let $H \subseteq W \times W'$ be a relation over their states. It is convenient to extend H to relate also infinite computations of S and S'. For two computations $\pi = w_0, w_1, \ldots$ in S, and $\pi' = w'_0, w'_1, \ldots$ in S', we say that $H(\pi, \pi')$ holds iff $H(w_i, w'_i)$ holds for all $i \geq 0$. For a pair $\langle w, w' \rangle \in W \times W'$, we say that $\langle w, w' \rangle$ is *good in* H iff for every fair w-computation π in S, there exists a fair w'-computation π' in S', such that $H(\pi, \pi')$.

Let w and w' be states in W and W', respectively. A relation $H \subseteq W \times W'$ is a *simulation relation* from $\langle S, w \rangle$ to $\langle S', w' \rangle$ iff the following conditions hold:

(1) $H(w, w')$.
(2) For all t and t' with $H(t, t')$, we have $L(t) = L(t')$.
(3) For all t and t' with $H(t, t')$, the pair $\langle t, t' \rangle$ is good in H.

A simulation relation H is a *simulation from S to S'* iff for every $w \in W_0$ there exists $w' \in W'_0$ such that $H(w, w')$. If there exists a simulation from S to S', we say that S *simulates* S' and we write $S \leq S'$. Intuitively, it means that the transition system S' has more behaviors than the transition system S. In fact, every computation tree embodied in S is embodied in S'. The *fair-simulation problem* is, given S and S', to determine whether $S \leq S'$.

It is easy to see that fair simulation implies language containment. That is, if $S \leq S'$ then $\mathcal{L}(S) \subseteq \mathcal{L}(S')$. The opposite, however, is not true. In the figure below we present two transition systems S and S' such that the language of both transition systems is $(a+b)^\omega$. As such, $\mathcal{L}(S) \subseteq \mathcal{L}(S')$, but still, S does not simulate S'. Indeed, no initial state of S' can be paired, by any H, to the initial state labeled a of S.



3 The Complexity of the Language-Containment Problem

Theorem 1. *The language-containment problem $\mathcal{L}(S) \subseteq \mathcal{L}(S')$ for $S \in \{\mathcal{U}, \mathcal{W}, \mathcal{S}\}$ and $S' \in \{\mathcal{U}, \mathcal{W}, \mathcal{S}\}$ is PSPACE-complete.*

Proof: As there are three possible types for the transition system S and three possible types for the transition system S', we have nine containment problems to solve in order to prove a PSPACE upper bound. We solve them all using the same method:

(1) Translate the transition system S to an unconditionally fair transition system S_U.
(2) Construct an unconditionally fair transition system $\overline{S'_U}$ that complements the transition system S'.
(3) Check $\mathcal{L}(S_U) \cap \mathcal{L}(\overline{S'_U})$ for nonemptiness.

This is how we perform step (1) for the three possible types of S.

1. $\mathcal{U}(n) \rightarrow \mathcal{U}(n)$.
2. $\mathcal{W}(n, m) \rightarrow \mathcal{U}(nm)$ [easy, and will be proven in the full version].
3. $\mathcal{S}(n, m) \rightarrow \mathcal{U}(n2^{O(m)})$ [not hard, and will be proven in the full version].

This is how we perform step (2) for the three possible types of S'.

1. $\overline{\mathcal{U}(n)} \rightarrow \mathcal{U}(2^{O(n \log n)})$ [Saf88].
2. $\overline{\mathcal{W}(n,m)} \rightarrow \overline{\mathcal{U}(nm)} \rightarrow \mathcal{U}(2^{O(nm \log(nm))})$ [Saf88].
3. $\overline{\mathcal{S}(n,m)} \rightarrow \mathcal{U}(2^{O(nm \log(nm))})$ [Saf92].

For all the three types of S, going to S_U involves an at most exponential blow up. Similarly, for all the three types of S', going to $\overline{S'_U}$ involves an at most exponential blow up. Thus, the size of the product of S_U and $\overline{S'_U}$ is exponential in the sizes of S and S' [Cho74] and checking it for nonemptiness can be done in space polynomial in their sizes [VW94].

Hardness in PSPACE follows from the known PSPACE lower bound for the case where both S and S' are unconditionally fair [Wol82]. Since $\mathcal{U}(n) \rightarrow \mathcal{W}(n,1)$ and $\mathcal{U}(n) \rightarrow \mathcal{S}(n,1)$, we can not do better with weak or strong fairness. □

Recall that our main concern is the complexity in terms of the (much larger) implementation. We now turn to consider the implementation complexity of language containment.

Theorem 2. *The implementation complexity of checking $\mathcal{L}(S) \subseteq \mathcal{L}(S')$ for $S \in \{\mathcal{U}, \mathcal{W}\}$ and $S' \in \{\mathcal{U}, \mathcal{W}, \mathcal{S}\}$ is NLOGSPACE-complete.*

Proof: In the case where $S \in \{\mathcal{U}, \mathcal{W}\}$, the translation of S to S_U involves only a polynomial blow up. Thus, in this case, fixing the size of S', the nondeterministic algorithm described in the proof of Theorem 1 requires space logarithmic in the size of S. Since we can solve the nonemptiness problem of a transition system by checking its containment in a fixed-size empty transition system, hardness in NLOGSPACE follows from the NLOGSPACE lower bound for the nonemptiness problem of unconditionally fair transition systems [VW94]. □

So, for the case where the implementation does not use the strong fairness condition, our language-containment algorithm requires space that is only logarithmic in the size of the implementation. Clearly, this is not the case when the implementation does use the strong fairness condition. Then, our algorithm requires space that is polynomial in the size of the implementation and time that is exponential in the size of the implementation. We can, however, do better.

Theorem 3. *The implementation complexity of checking $\mathcal{L}(S) \subseteq \mathcal{L}(S')$ for $S \in \{\mathcal{S}\}$ and $S' \in \{\mathcal{U}, \mathcal{W}, \mathcal{S}\}$ is in PTIME.*

Proof: We are going to use the following known results.

1. For $S_1 \in \mathcal{S}(n_1, m)$ and $S_2 \in \mathcal{U}(n_2)$, there exists $S \in \mathcal{S}(n_1 n_2, m+1)$ such that $\mathcal{L}(S) = \mathcal{L}(S_1) \cap \mathcal{L}(S_2)$ [easy, and will be proven in the full version].
2. The nonemptiness problem for strongly fair transition systems can be solved in polynomial time [EL85].

Given S and S', we construct, as in the proof of Theorem 1, the unconditionally fair transition system $\overline{S'_U}$. Unlike the algorithm there, we do not translate the transition system S to an unconditionally fair system. Rather, we check the nonemptiness of $\mathcal{L}(S) \cap \mathcal{L}(\overline{S'_U})$. By 1 and 2 above, this can be done in time polynomial in the size of S. □

Note that the algorithm presented in the proof of Theorem 3 uses time and space exponential in the size of the specification, in contrast to the algorithm in the proof of Theorem 1 that uses space polynomial in the size of the specification. Nevertheless, as S' is usually much smaller than S, the algorithm in the proof of Theorem 3 may work better in practice. Can we do better and get the NLOGSPACE complexity we have for implementations that use the unconditional or weak fairness conditions? As we now show, the answer to this question is negative. To see this, we first need the following theorem.

Theorem 4. *The nonemptiness problem for strongly fair transition systems is PTIME-hard.*

Proof: We do a reduction from Propositional Anti-Horn Satisfiability (PAHS). Propositional Anti-Horn clauses are obtained from Propositional Horn clauses by replacing each proposition p with $\neg p$. Thus, a propositional anti-Horn clause is either of the form $p \rightarrow q_1 \vee \cdots \vee q_n$ (an empty disjunction is equivalent to false) or of the form $q_1 \vee \cdots \vee q_n$. As Propositional-Horn Satisfiability is PTIME-complete [Pla84], then clearly, so is PAHS.

Given an instance I of PAHS we construct the transition system $S_I = \langle W, W, W \times W, \{w_0\}, L, \alpha \rangle$, where W is the set of all the propositions in I, the initial state w_0 is an arbitrary state in W, $L(w) = w$ for $w \in W$, and α is the strong fairness condition defined as follows.

- For a clause $p \rightarrow q_1 \vee \ldots \vee q_n$ in I, we have $\langle \{p\}, \{q_1, \cdots, q_n\} \rangle$ in α.
- For a clause $q_1 \vee \ldots \vee q_n$ in I, we have $\langle W, \{q_1, \cdots, q_n\} \rangle$ in α.

We can view each computation of S_I as an assignment to the propositions in I. A proposition is assigned **true** iff the computation visits it infinitely often. The definition of α thus guarantees that I is satisfiable iff S_I is nonempty. □

So, unlike unconditionally or weakly fair transition systems, for which the nonemptiness problem is NLOGSPACE-complete, testing strongly fair transition systems for nonemptiness is PTIME-complete. Theorems 3 and 4 imply the following theorem.

Theorem 5. *The implementation complexity of checking $\mathcal{L}(S) \subseteq \mathcal{L}(S')$ for $S \in \{\mathcal{S}\}$ and $S' \in \{\mathcal{U}, \mathcal{W}, \mathcal{S}\}$ is PTIME-complete.*

4 The Complexity of the Fair-Simulation Problem

4.1 Upper Bound

Theorem 6. *The fair-simulation problem $S \leq S'$ for $S \in \{\mathcal{U}, \mathcal{W}, \mathcal{S}\}$ and $S' \in \{\mathcal{U}, \mathcal{W}, \mathcal{S}\}$ is in PSPACE.*

Proof (sketch): Given $S = \langle \Sigma, W, R, W_0, L, \alpha \rangle$ and $S' = \langle \Sigma, W', R', W'_0, L', \alpha' \rangle$, we show how to check in polynomial space that a candidate relation H is a simulation from S to S'. The claim then follows, since we can enumerate using polynomial space all candidate relations. First, we check, easily, that for every $w \in W_0$ there exists $w' \in W'_0$ such that $H(w, w')$. We then check, also easily, that for all $\langle w, w' \rangle \in H$, we have $L(w) = L(w')$. It is left to check that for all $\langle w, w' \rangle \in H$, the pair $\langle w, w' \rangle$ is good in H. To do this, we define, for every $\langle w, w' \rangle \in H$, two transition systems. The alphabet of both systems is W. The first transition system, A_w, accepts all the fair w-computations in S. The second transition system, $U_{w'}$, accepts all the sequences π in W^ω for which there exists a fair w'-computation π' in S' such that $H(\pi, \pi')$. Clearly, the pair $\langle w, w' \rangle$ is good in H iff $\mathcal{L}(A_w) \subseteq \mathcal{L}(U_{w'})$.

We define A_w and $U_{w'}$ as follows. The system A_w does nothing but tracing the w-computations of S, accepting these that satisfy S's acceptance condition. Formally, $A_w = \langle W, W, R, \{w\}, L'', \alpha \rangle$, where for all $w \in W$, we have $L''(w) = w$.

The transition system $U_{w'}$ has members of H as its set of states. Thus, each state has two elements. The second element of each state in $U_{w'}$ is a state in W' and it induces, according to R', the transitions. The first element in each state of $U_{w'}$ is a state in W and it induces the labeling. This combination guarantees that a computation $\pi'' \in H^\omega$ whose W'-elements form the computation $\pi' \in W'^\omega$ and whose states are labeled with $\pi \in W^\omega$, satisfies $H(\pi, \pi')$. Formally, $U_{w'} = \langle W, H, R'', W''_0, L'', \alpha'' \rangle$, where $W''_0 = (W \times \{w'\}) \cap H$, for every $\langle t, t' \rangle \in H$, we have $L''(\langle t, t' \rangle) = t$, the fairness condition α'' is adjusted to the new state space (i.e., each set $L \subseteq W'$ in α' is replaced by the set $(W \times L) \cap H$ in α''), and the transition relation R'' is also adjusted to the new state space (i.e., $R''(\langle t, t' \rangle, \langle q, q' \rangle)$ iff $R'(t', q')$). Note that R'' is not necessarily total. For that, we restrict $U_{w'}$ to states that have at least one R''-successor. Clearly, this does not effect the language of $U_{w'}$.

According to Theorem 1, checking that $\mathcal{L}(A_w) \subseteq \mathcal{L}(U_{w'})$ can be done in space polynomial in the sizes of A_w and $U_{w'}$, thus polynomial in S and S'. □

We note that our algorithm can be easily adjusted to check S and S' for fair *bisimulation*.

4.2 Lower Bound

For a transition system $S = \langle \Sigma, W, R, W_0, L, \alpha \rangle$, we say that S is *universal* iff $\mathcal{L}(S) = \Sigma^\omega$. The *universality problem* is to determine whether a given transition system is universal. Meyer and Stockmeyer proved that the problem of determining whether the language of an automaton over finite words is Σ^* is PSPACE-complete [MS72]. We give here the details of the proof, easily adjusted to infinite words.

Theorem 7. *The universality problem is PSPACE-hard.*

Proof (sketch): We do a reduction from polynomial-space Turing machines. Given a Turing machine T of space complexity $s(n)$, we construct a transition system S_T of size linear in T and $s(n)$ such that S_T is universal iff T does not accept the empty tape. We assume, without loss of generality, that once T reaches a final state it loops there forever. The system S_T accepts a word w iff w is not an encoding of a legal computation of T over the empty tape or if w is an encoding of a legal yet rejecting computation of T over the empty tape. Thus, S_T rejects a word w iff it encodes a legal and accepting computation of T over the empty tape. Hence, S_T is universal iff T does not accept the empty tape.

Below we give the details of the construction of S_T. Let $T = \langle \Gamma, Q, \rightarrow, q_0, F \rangle$, where Γ is the alphabet, Q is the set of states, $\rightarrow \subseteq Q \times \Gamma \times Q \times \Gamma \times \{L, R\}$ is the transition relation (we use $(q, a) \rightarrow (q', b, \Delta)$ to indicate that when T is in state q and it reads the input a in the current tape cell, it moves to state q', writes b in the current tape cell, and its reading head moves one cell to the left/right, according to Δ), q_0 is the initial state, and $F \subseteq Q$ is the set of accepting states. We encode a configuration of T by a word $\#\gamma_1\gamma_2 \ldots (q, \gamma_i) \ldots \gamma_{s(n)}\#$. That is, a configuration starts and ends with $\#$, all its other letters are in Γ, except for one letter in $Q \times \Gamma$. The meaning of such a configuration is that the j's cell in T, for $1 \leq j \leq s(n)$, is labeled γ_j, the reading head points on cell i, and T is in state q. For example, the initial configuration of T is $\#(q_0, b)b \ldots b\#$ where b stands for an empty cell. We can now encode a computation of T by a sequence of configurations (with only one $\#$ between two configurations).

Let $\Sigma = \{\#\} \cup \Gamma \cup (Q \times \Gamma)$ and let $\#\sigma_1 \ldots \sigma_{s(n)}\#\sigma'_1 \ldots \sigma'_{s(n)}\#$ be two successive configurations of T. For each triple $\langle \sigma_{i-1}, \sigma_i, \sigma_{i+1} \rangle$ with $1 \leq i \leq s(n)$, we know, by the transition relation of T, what σ'_i should be. Let $next(\langle \sigma_{i-1}, \sigma_i, \sigma_{i+1} \rangle)$ denote our expectation for σ'_i. For example, $next(\langle \gamma_{i-1}, \gamma_i, \gamma_{i+1} \rangle)$ is γ_i, and $next(\langle (q, \gamma_{i-1}), \gamma_i, \gamma_{i+1} \rangle)$ is γ_i, in the case $(q, \gamma_{i-1}) \rightarrow (q', \gamma'_{i-1}, L)$, and is (q', γ_i), in the case $(q, \gamma_{i-1}) \rightarrow (q', \gamma'_{i-1}, R)$. In addition, since we want the letter $\#$ to repeat exactly every $s(n) + 1$ letters, we define $next(\langle \sigma_{s(n)}, \#, \sigma'_1 \rangle)$ as $\#$. Consistency with $next$ now gives us a necessary condition for a word to encode a legal computation. In addition, the computation should start with the initial configuration.

In order to check consistency with $next$, S_T can use its nondeterminism and guess when there is a violation of $next$. Thus, S_T guesses $\langle \sigma_{i-1}, \sigma_i, \sigma_{i+1} \rangle \in \Sigma^3$, guesses a position in the word, checks whether the three letters to be read starting this position are σ_{i-1}, σ_i, and σ_{i+1}, and checks whether $next(\langle \sigma_{i-1}, \sigma_i, \sigma_{i+1} \rangle)$ is not the letter to come $s(n) + 1$ letters later. Once S_T sees such a violation, it goes to an accepting sink. In order to check that the first configuration is the initial configuration, S_T simply compares the first $s(n)+2$ letters with $\#(q_0, b)b \ldots b\#$. Finally, checking whether a legal computation is accepting is also easy; the computation has to reach an accepting configuration (one with $q \in F$). □

We would like to do a similar reduction in order to prove that the fair-simulation problem is PSPACE-hard. For every alphabet Σ, let S_Σ be the transition system $\langle \Sigma, \Sigma, \Sigma \times \Sigma, \Sigma, L_\Sigma, \alpha \rangle$, where $L_\Sigma(\sigma) = \sigma$ and α is such that all the computations of S_Σ are fair. That is, S_Σ is a universal transition system in which each state is associated with a letter $\sigma \in \Sigma$ and $\mathcal{L}(S^\sigma_\Sigma) = \sigma \cdot \Sigma^\omega$. For example, $S_{\{a,b\}}$ is the transition system S in Figure 2.2. It is easy to see that a transition system S over Σ is universal iff $\mathcal{L}(S_\Sigma) \subseteq \mathcal{L}(S)$. It is not true, however, that S is universal iff $S_\Sigma \leq S$. For example, the transition system S' in Figure 2.2 is universal yet $S_{\{a,b\}} \not\leq S'$. Our reduction overcomes this difficulty by defining S_T in such a way that if S_T is universal, then for each of its states w, we have $\mathcal{L}(S^w_T) = L(w) \cdot \Sigma^\omega$. For such S_T, we do have that S_T is universal iff $S_\Sigma \leq S_T$. Indeed, a relation that maps a state σ of S_Σ to all the states of S_T that are labeled with σ is a fair simulation.

Theorem 8. *The fair-simulation problem $S \leq S'$ for $S \in \{\mathcal{U}, \mathcal{W}, \mathcal{S}\}$ and $S' \in \{\mathcal{U}, \mathcal{W}, \mathcal{S}\}$ is PSPACE-hard.*

Proof (sketch): As in the previous proof, we do a reduction from polynomial space Turing machines. Given the Turing machine T, let T' be as follows. Whenever T reaches an accepting configuration, T' cleans the tape and starts from the beginning (i.e., empty tape and initial state at the left end of the tape). Thus, T accepts the empty tape iff T' has an infinite computation, in which case it visits the initial configuration infinitely often. We now define a transition system S_T with the following behavior. Reading a word w, the transition system S_T checks for a violation of the transition relation of T' in w (by guessing a violation of $next$). If it sees a violation, it goes to an accepting sink. If it does not see a violation, it continues to trace the computation of T' forever. We define the fairness condition of S_T such that it accepts w iff it reaches the accepting sink or it never sees the initial configuration in w. This acceptance condition can be specified by a pair $\langle g, b\rangle$ of states where g is simply the accepting sink and b is a state that S_T passes in whenever it traces the initial configuration in w (note that since the initial configuration starts with # and has no other # in it, it is very easy to being traced). A computation of S_T is fair with respect to $\langle g, b\rangle$ iff it eventually visits g and never visits b. This fairness condition can be easily translated to unconditional, weak, and strong fairness; e.g., by making g an accepting sink and b a rejecting sink. It follows that S_T does not accept a word w iff w has a finite prefix, not violating $next$, followed by an infinite computation of T' that passes in the initial configuration of T. Therefore, S_T is universal iff T does not accept the empty tape.

We, however, want more than universality test. We want to define S_T in such a way that if it is indeed universal, then for each of its states w, we have $\mathcal{L}(S_T^w) = L(w) \cdot \Sigma^\omega$. Let $S_T = \langle \Sigma, W, R, W_0, L, \alpha\rangle$. We define the transition system S'_T by adding to S_T transitions from all states to all the initial states, i.e., $S'_T = \langle \Sigma, W, R \cup (W \times W_0), W_0, \alpha\rangle$. We claim that the extension of S_T to S'_T preserves "non-universality". That is, if S'_T is universal, then so is S_T. Note that this is not the case for arbitrary transition systems. In S_T, however, if a word w is not accepted, then w is of the form yx where y is a prefix not violating $next$ and x is an infinite computation of T'. As such, all the suffixes of w are of that special form! Therefore, if w is not accepted by S_T, all its suffixes are also not accepted by S_T. Hence, w is not accepted by S'_T too.

We claim that S_T is universal iff for each state w of S'_T, we have $\mathcal{L}(S'^w_T) = L(w) \cdot \Sigma^\omega$. The direction from right to left follows from the fact that the extension of S_T to S'_T preserves non-universality and the fact that for every $\sigma \in \Sigma$, there exists $w_0 \in W_0$ with $L(w_0) = \sigma$. The second direction follows from the fact that each state w in S'_T has a transition to W_0.

We now have that $S_\Sigma \leq S'_T$ iff S_T is universal, thus $S_\Sigma \leq S'_T$ iff T does not accept the empty tape. Since the fairness conditions of both S_Σ and S'_T can be specified in terms of either unconditional, weak, or strong fairness, we are done.

□

Theorems 6 and 8 together imply the following.

Theorem 10. *The implementation complexity of checking $S \leq S'$ for $S \in \{\mathcal{U}, \mathcal{W}, \mathcal{S}\}$ and $S' \in \{\mathcal{U}, \mathcal{W}, \mathcal{S}\}$ complete.*

4.3 The Implementation Complexity of the Fair-Simulation Problem

So, fair simulation has the same complexity as language containment. In Theorem 10 below we show that when we consider the implementation complexity of fair simulation, the picture is different. Here, checking implementations that use the unconditional or weak fairness conditions is not easier than checking implementations that use the strong fairness condition. Hence, fair simulation is harder than language containment and the trace-based approach is more efficient.

Theorem 10. *The implementation complexity of checking $S \leq S'$ for $S \in \{\mathcal{U}, \mathcal{W}, \mathcal{S}\}$ and $S' \in \{\mathcal{U}, \mathcal{W}, \mathcal{S}\}$ is PTIME-complete.*

Proof: We start with the upper bound. Consider the algorithm presented in the proof of Theorem 6. It checks whether a candidate relation H is a simulation. Once we fix S', then, by Theorems 2 and 5, the complexity of checking each pair in the candidate relation is NLOGSPACE for $S \in \{\mathcal{U}, \mathcal{W}\}$ and is PTIME for $S \in \{\mathcal{S}\}$. Once we fix S', the number of pairs in each candidate relation is polynomial in the size of S. Thus, fixing S, the problem of checking a candidate relation H is in PTIME. Instead of guessing a relation H and checking it, we do a fixed-point computation as follows (cf. [Mil90]). Let $H_0 = \{\langle w, w' \rangle : w \in W, w' \in W', \text{ and } L(w) = L(w')\}$. Thus, H_0 is the maximal relation that satisfies condition (1) of fair simulation. Consider the monotonic function $f : 2^{W \times W'} \rightarrow 2^{W \times W'}$, where

$$f(H) = H \cap \{\langle w, w' \rangle : \langle w, w' \rangle \text{ is good in } H\}.$$

Thus, $f(H)$ contains all the pairs in H that are good with respect to the relation H. Let H be the greatest fixed-point of f when restricted to pairs in H_0. That is, $H = \nu z.H_0 \cap f(z)$. It can be shown that $S \leq S'$ iff for every $w \in W_0$, we have $(\{w\} \times W_0') \cap H \neq \emptyset$. Since $W \times W'$ is finite, we can calculate H iteratively, starting with H_0 until we reach a fixed-point. Now, as f is monotonic, we have to iterate it at most polynomially many times. Hence, out of the $2^{|W \times W'|}$ candidate relations for simulation, we actually check at most $|W \times W'|$ relations. Recall that if S' is fixed, the problem of checking a candidate relation is in PTIME. Also, if S' is fixed, we have only linearly many candidate relations to check. Hence, the problem is in PTIME.

We prove hardness in PTIME by reducing the NAND Circuit Value Problem (NANDCV), proved to be PTIME-complete in [Gol77, GHR95], to the problem of determining whether a transition system S simulates a fixed transition system S'. In the NANDCV problem, we are given a Boolean circuit α constructed solely of NAND gates of fanout 2, and a vector $\langle x_1, \ldots, x_n \rangle$ of Boolean input values. The problem is to determine whether the output of α on $\langle x_1, \ldots, x_n \rangle$ is 1. The idea of the reduction is as follows. We define a fixed transition system S' that embodies all the NAND circuits α and input vectors $\mathbf{x}$ for which the value of α on $\mathbf{x}$ is 1. Then, given a circuit α and an input vector $\mathbf{x}$, we translate them to a transition system S such that $S \leq S'$ iff the value of α on $\mathbf{x}$ is 1.

The transition system S' has 12 states. Eight states correspond to internal gates. Each of these states is an entry in the Truth Table of the operator NAND, attributed with a direction, either L or R. Thus, the "internal states" of S' are $\langle 001L \rangle$, $\langle 011L \rangle$, $\langle 101L \rangle$, $\langle 110L \rangle$, $\langle 001R \rangle$, $\langle 011R \rangle$, $\langle 101R \rangle$, and $\langle 110R \rangle$. Four more states correspond to the input gates of the circuit. Each of these states is a Boolean value, attributed with a direction. Thus, the "input states" are $\langle 0L \rangle$, $\langle 1L \rangle$, $\langle 0R \rangle$, and $\langle 1R \rangle$. The intuition is that an internal state $\langle l, r, val, d \rangle$ corresponds to a NAND gate that has the value l in its left input, has the value r in its right input, and whose output val can be only a d-input of other gates. Similarly, an input state $\langle val, d \rangle$ corresponds to an input gate with output val that can only be a d input of other gates.

Accordingly, the transitions from an internal state $\langle l, r, val, d \rangle$ correspond to the possible ways of having l and r as right and left inputs, respectively. Thus, we have transitions from this state to all (internal or input) states with either $val = l$ and $d = L$ or $val = r$ and $d = R$. For example, the internal state $\langle 100L \rangle$ has transitions to the states $\langle 001L \rangle$, $\langle 011L \rangle$, $\langle 101L \rangle$, $\langle 110R \rangle$, $\langle 1L \rangle$, and $\langle 0R \rangle$. It has transitions from all states $\langle l, r, val, d \rangle$ with $l = 0$. In addition, the input states have self loops.

We label an internal state by either L or R according to its direction element. We label an input state by both its value and direction. We define the initial states of S' to be these with $val = 1$, and we impose no fairness condition. Clearly, the size of S' is fixed.

Now, S is simply α with attributions of directions. That is, we duplicate all gates and inputs of α so that the output of each gate is either always a left input of other gates, in which case we label it with L, or always a right input of other gates, in which case we label it with R. In addition, we add self loops to the input gates and label them with their values.

It is not hard to prove that for a simulation relation H from S to S' and for every pair $\langle s, \langle l, r, val, d \rangle \rangle$ or $\langle s, \langle val, d \rangle \rangle$ in H, the output of the gate s on the vector $\mathbf{x}$ is val. Hence, the output of α on $\mathbf{x}$ is 1 iff S simulates S'. □

We note that our lower bound is different from the PTIME-hardness established for the bisimulation problem in [BGS92]. We consider simulation between two systems, one of them is fixed. Balcazar et al. consider bisimulation between the states of a single system, whose size is not fixed.

5 Discussion

We have examined the trace-based and the tree-based approaches to implementation from a complexity-theoretic point of view. Our results show that when we model the specification and the implementation by fair transition systems, the complexity of checking the correctness of a trace-based implementation coincides with that of checking the correctness a tree-based implementation. Furthermore, when we consider the implementation complexity, then checking implementations that use the unconditional or weak fairness condition is easier in the trace-based approach.

It is interesting to compare our results with the known complexities of LTL and $\forall$CTL* model checking. Trace-based implementations are part of the linear-time paradigm and correspond to LTL model checking. Tree-based implementations are part of the branching-time paradigm and correspond to $\forall$CTL* model checking. All the four problems are PSPACE-complete [SC85, EL85]. The model-checking algorithm of $\forall$CTL* uses as a subroutine the model-checking algorithm of LTL [EL85]. In a similar manner, our fair-simulation algorithm uses as a subroutine the language-containment algorithm. So, the implementation dichotomy and the temporal-logic dichotomy have a lot in common. When we turn to consider the program complexity of model checking, which is the analogue to our implementation complexity, this is no longer true. The program complexity of model checking for both LTL and $\forall$CTL* is NLOGSPACE-complete [VW86, BVW94]. In contrast, we saw here that implementation is easier in the trace-based approach.

References

[AL91] M. Abadi and L. Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82(2):253–284, 1991.

[ASB+94] A. Aziz, V. Singhal, F. Balarin, R. Brayton, and A.L. Sangiovanni-Vincentelli. Equivalences for fair kripke structures. In *Proc. 21st Int. Colloquium on Automata, Languages and Programming*, Jerusalem, Israel, July 1994.

[BBLS92] S. Bensalem, A. Bouajjani, C. Loiseaux, and J. Sifakis. Property preserving simulations. In *Proc. 4th Workshop on Computer Aided Verification*, volume 663 of *Lecture Notes in Computer Science*, Montreal, June 1992. Springer-Verlag.

[BGS92] J. Balcazar, J. Gabarro, and M. Santha. Deciding bisimilarity is P-complete. *Formal Aspects of Computing*, 4(6):638–648, 1992.

[BVW94] O. Bernholtz, M.Y. Vardi, and P. Wolper. An automata-theoretic approach to branching-time model checking. In D. L. Dill, editor, *Computer Aided Verification, Proc. 6th Int. Conference*, volume 818 of *Lecture Notes in Computer Science*, pages 142–155, Stanford, June 1994. Springer-Verlag, Berlin.

[CD88] E.M. Clarke and I.A. Draghicescu. Expressibility results for linear-time and branching-time logics. In *Proc. Workshop on Linear Time, Branching Time, and Partial Order in Logics and Models for Concurrency*, pages 428–437. Lecture Notes in Computer Science, Springer-Verlag, 1988.

[CDK93] E. M. Clarke, I. A. Draghicescu, and R. P. Kurshan. A unified approach for showing language containment and equivalence between various types of ω-automata. *Information Processing Letters* **46**, pages 301–308, (1993).

[CES86] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, January 1986.

[Cho74] Y. Choueka. Theories of automata on ω-tapes: A simplified approach. *Journal of Computer and System Sciences*, 8:117–141, 1974.

[EL85] E.A. Emerson and C.-L. Lei. Temporal model checking under generalized fairness constraints. In *Proc. 18th Hawaii International Conference on System Sciences*, Hawaii, 1985.

[Eme90] E.A. Emerson. Temporal and modal logic. *Handbook of theoretical computer science*, pages 997–1072, 1990.

[GHR95] R. Greenlaw, H.J. Hoover, and W.L. Ruzzo. *Limits of parallel computation.* Oxford University Press, 1995.

[GL94] O. Grumberg and D.E. Long. Model checking and modular verification. *ACM Trans. on Programming Languages and Systems*, 16(3):843–871, 1994.

[Gol77] L.M. Goldschlager. The monotone and planar circuit value problems are log space complete for p. *SIGACT News*, 9(2):25–29, 1977.

[Hen85] M. Hennessy. *Algebraic theory of Processes*. MIT Press, Cambridge, 1985.

[Kel76] R.M. Keller. Formal verification of parallel programs. *Comm ACM*, 19:371–384, 1976.

[Kur94] R.P. Kurshan. *Computer-Aided Verification of Coordinating Processesl*. Princeton Univ. Press, 1994.

[LP85] O. Lichtenstein and A. Pnueli. Checking that finite state concurrent programs satisfy their linear specification. In *Proceedings of the Twelfth ACM Symposium on Principles of Programming Languages*, pages 97–107, New Orleans, January 1985.

[LPS81] D. Lehman, A. Pnueli, and J. Stavi. Impartiality, justice, and fairness – the ethic of concurrent termination. In *Proc. 8th Colloq. on Automata, Programming, and Languages (ICALP)*, volume 115 of *Lecture Notes in Computer Science*, pages 264–277. Springer-Verlag, July 1981.

[LS84] S.S. Lam and A.U. Shankar. Protocol verification via projection. *IEEE Trans. on Software Engineering*, 10:325–342, 1984.

[LT87] N. A. Lynch and M.R. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *Proc. 6th ACM Symp. on Principles of Distributed Computing*, pages 137–151, 1987.

[Mil71] R. Milner. An algebraic definition of simulation between programs. In *Proceedings of the 2nd International Joint Conference on Artificial Intelligence*, pages 481–489, September 1971.

[Mil80] R. Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer Verlag, Berlin, 1980.

[Mil89] R. Milner. *Communication and Concurrecny*. Prentice-Hall, Englewood Clifs, 1989.

[Mil90] R. Milner. Operational and algebraic semantics of concurrent processes. *Handbook of theoretical computer science*, pages 1201–1242, 1990.

[MP92] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer-Verlag, Berlin, January 1992.

[MS72] A.R. Meyer and L.J. Stockmeyer. The equivalence problem for regular expressions with squaring requires exponential time. In *Proc. 13th IEEE Symp. on Switching and Automata Theory*, pages 125–129, 1972.

[Pla84] D.A. Plaisted. Complete problems in the first-order predicate claculus. *J. on Computer and System Sciences*, 29(1):8–35, 1984.

[Pnu85] A. Pnueli. Linear and branching structures in the semantics and logics of reactive systems. In *Proc. 12th Int. Colloquium on Automata, Languages and Programming*, pages 15–32. Lecture Notes in Computer Science, Springer-Verlag, 1985.

[QS81] J.P. Queille and J. Sifakis. Specification and verification of concurrent systems in Cesar. In *Proc. 5th Int'l Symp. on Programming*, volume 137, pages 337–351. Springer-Verlag, Lecture Notes in Computer Science, 1981.

[Saf88] S. Safra. On the complexity of omega-automata. In *Proceedings of the 29th IEEE Symposium on Foundations of Computer Science*, White Plains, October 1988.

[Saf92] S. Safra. Exponential determinization for ω-automata with strong-fairness acceptance condition. In *Proceedings of the 24th ACM Symposium on Theory of Computing*, Victoria, May 1992.

[SC85] A.P. Sistla and E.M. Clarke. The complexity of propositional linear temporal logic. *J. ACM*, 32:733–749, 1985.

[Tho90] W. Thomas. Automata on infinite objects. *Handbook of theoretical computer science*, pages 165–191, 1990.

[VW86] M.Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proceedings of the First Symposium on Logic in Computer Science*, pages 322–331, Cambridge, June 1986.

[VW94] M.Y. Vardi and P. Wolper. Reasoning about infinite computations. *Information and Computation*, 115(1):1–37, November 1994.

[Wol82] P. Wolper. *Synthesis of Communicating Processes from Temporal Logic Specifications*. PhD thesis, Stanford University, 1982.

Tools and Case Studies

The State of SPIN

Gerard J. Holzmann and Doron Peled

Bell Laboratories
700 Mountain Avenue
Murray Hill, NJ 07974
{gerard, doron}@research.bell-labs.com

Abstract. The number of installations of the SPIN model checking tool is steadily increasing. There are well over two thousand installations today, divided roughly evenly over academic and industrial sites. The tool itself also continues to evolve; it has more than doubled in size, and hopefully at least equally so in functionality, since it was first distributed in early 1991. The tool runs on most standard workstations, and starting with version 2.8 also on standard PCs.

In this overview, we summarize the design principles of the tool, and review its current state.

1 Background

SPIN is a general state-based model-checking tool designed for the efficient verification of logically distributed process systems. Processes in SPIN are always *asynchronous*. Synchronization, where desired, must be specified explicitly.

The native specification language of SPIN is called PROMELA. PROMELA is a non-deterministic guarded command language, in the tradition of [3] and [5] with a small influence from the language C [11]. The language was designed to encourage abstraction. The purpose of model checking in SPIN is to perform *design verification* well before the coding stage of a design is reached. A basic notion in the language is that of *executability*: every PROMELA statement can enforce synchronization constraints through the rules of executability. Whenever a statement is unexecutable, for instance, it blocks the execution of the corresponding process, unless alternative executions for that process were specified. The most recent version of the language supports data structures, interrupts, and a rich variety of both synchronous and asynchronous message passing primitives.

The semantics of a PROMELA model are based on the *interleaving model* of execution, where concurrently executed atomic operations from different processes are considered to be executable in any arbitrary time-order. The model is appropriate for modeling distributed software. For synchronous hardware a different semantics interpretation is usually chosen, e.g. [12]. Most model checkers today have adopted those alternative semantics. These systems can still simulate interleaving semantics at the language level, but the price to pay for this in efficiency can be substantial. The optimizations builtin to SPIN fully exploit the asynchronous process model.

1.1 State-based model-checking

SPIN's verification procedure is based on the reachability analysis of a model, using an optimized *depth-first-search* graph traversal method. A number of special-purpose algorithms are used to avoid a purely exhaustive search procedure (e.g., partial order reduction, state compression, and sequential bitstate hashing). We summarize some of the newer algorithms in the sequel.

1.2 Correctness properties

SPIN can verify both safety and liveness properties *on-the-fly*. By default, SPIN will check a set of basic properties such as absence of *deadlock* and *unreachable code*. It will also check that any user-defined *process assertions* or *invariants* cannot be violated, and that the system can only terminate in user-defined valid end-states.

The specification language includes two types of labels that can be used to define two complementary types of liveness properties: acceptance and progress. In the syntax of Linear Temporal Logic (LTL) [15], an acceptance property corresponds to formulae of the type $\Box\Diamond p$, where p is a user-defined accepting state. The violation of a progress property corresponds to formulae of the type $\Diamond\Box\neg p$ with p a user-defined progress state.

Correctness requirements can also be expressed directly in LTL syntax. For example, the formula $\Box(request \rightarrow \Diamond granted)$ asserts that at any point in the execution, if a request was made, it is eventually granted. SPIN versions 2.7 and later include a translation algorithm that converts LTL formulae like these into PROMELA *never-claims*. Never-claims formalize the potential violations of a correctness requirement, i.e., behavior that should *never* happen. More specifically, a never-claim can be used to represent a Büchi automaton (an automaton over infinite words), and it is this capability that is exploited by the LTL translator.

Although the expressive power of LTL is smaller than that of never-claims [17], the use of LTL can be simpler and more direct.

2 Basic Algorithms

2.1 Automata Intersection

SPIN uses finite automata based model-checking. Each process of the checked model is translated into a finite automaton. The checked property, representing the *violations* of correctness properties, is translated into a *property* automaton (i.e., the never-claim).

SPIN checks the given model against the given property by calculating the intersection of the corresponding automata. This is done *on-the-fly*, namely, the state space of the model intersected with the property automaton need only be built up to the point where the non-emptiness of the resulting automaton can be proven. A non-empty intersection means the possibility of violating a correctness requirement. An execution sequence that illustrates this is produced, which the

user can use to retrace the error, for instance, as a message sequence chart in XSPIN (an independent graphical user interface for SPIN, written in Tcl/Tk).

The computation of the intersection can either be done in a conventional exhaustive manner, or, when this proves to be impossible because of state space explosion, with an efficient proof approximation method based on bitstate hashing [6]. With a careful choice of hashing functions [9], the probability of an exhaustive proof remains very high. Both the exhaustive and the bistate modes of verification are based on a partial order reduction theory that limits the work to a small subset of states that suffices to render the proof (see also Section 2.3 below).

2.2 Cycle Detection

The model-checking is performed by checking the non-emptiness of the intersection of Büchi automata. The non-emptiness of the resulting automaton is examined by checking for the existence of at least one reachable cycle that contains an accepting state. Such a cycle constitutes a counterexample to the claim that no executions can exist that violate a correctness requirement.

The classical algorithm for finding a cycle in a graph is Tarjan's depth-first-search, which constructs the strongly connected components in linear time. If a reachable component contains an accepting state, a reachable acceptance cycle must exist. To find such a cycle requires searching the component for the accepting states, and constructing a path through at least one of them. An efficient alternative for this algorithm [7, 2] performs a nested depth-first-search, again visiting every state up to two times, but storing every state only once (with just two bits of overhead per state). The algorithm starts the search in the normal depth-first-order. Whenever it finishes processing an accepting state (i.e., in postorder), a search for a cycle is started in a logically separate statespace. SPIN uses a variant of this search procedure to prove acceptance and non-progress properties [10].

2.3 Partial Order Reduction

SPIN uses a *partial order reduction* algorithm [8, 13, 14] to reduce the state space explosion. The reduction is based on the observation that usually the checked property is insensitive to the order in which concurrent or independently executed events are interleaved. Thus, instead of generating a state-space which includes all the execution sequences, one can generate a reduced state space, with only representatives for classes of sequences that are undistinguishable from each other.

Reduction is achieved by changing the depth first search algorithm, used in the model-checking engine, such that from each state, only a sufficiently big subset of the successors are generated. This subset has to obey certain restrictions, guaranteeing that enough representative execution sequences is generated. The implementation is based on a careful analysis of the enabled atomic transitions from the currently analyzed state. To achieve a small overhead for doing the

extra checks, most of the analysis is done at the time of compiling the checked model. However, some reduction decisions are necessarily deferred to run time. The reduction algorithm as implemted was proven to preserve all safety and liveness properties [8, 1].

2.4 Linear Temporal Logic to Automata Translation

The translation of an LTL formula into an automaton is inherently exponential [16]. SPIN uses an algorithm [4] that in practice tends to produce small automata. The translation algorithm computes the states of an automaton by compiling the set of subformulas that need to hold in each state, and in each of its successors. The algorithm starts by converting the formula into a normal form, in which negations are pushed inward, to appear adjacent to only atomic propositions. An initial state is created, containing the formula to be translated and a dummy incoming edge. The automaton is then computed recursively, At each stage, a subformula that remains to be satisfied is taken and, according to the main logical operator, the current state may be split into two, with the two copies inheriting different parts of the subformula. In the last phase of the translation, some of the states are identified as accepting according to the presence or absence of subformulas of the type $\varphi U \psi$.

3 User Support

All the sources to the SPIN system and its graphical interface XSPIN are available for general educational use, free of charge, and for industrial applications for a small licensing fee. The latest version of the tool is always available via anonymous ftp from `netlib.att.com` from the directory `/netlib/spin`. Much of the documentation for SPIN is distributed together with the tool. The main reference on the implementation of the tool itself is the book [7]. The recent extensions, including the partial order reduction, and the LTL translation algorithm are reported in research papers [4, 8, 9]. A new book describing the automata theoretic background of SPIN is in preparation.

An informal SPIN *Users Group* was formed early in 1995. Anyone interested in the background of the tool, major applications, and extensions that are being planned, can subsribe to the mailing list of the group by sending a one line message `Subscribe` to `spin_list@research.bell-labs.com`.

A first SPIN *Workshop* was held in Montreal on 16 October 1995. The proceedings are available via: `http://netlib.att.com/netlib/spin/news.html`. A second workshop is held 5 August 1996, at Rutgers University.

References

1. C-T. Chou, D. Peled, Verifying a Model-Checking Algorithm, TACAS'96, Tools and Algorithms for the Construction and Analysis of Systems, Passau, Germany, March 1996.

2. C. Courcoubetis, M. Vardi, P. Wolper, M, Yannakakis, Memory-efficient algorithms for the verification of temporal properties, Formal methods in system design 1 (1992) 275–288.
3. E. Dijkstra, Guarded commands, nondeterminacy and formal derivation of programs, Comm. ACM, 18(8), 1975, 453–457.
4. R. Gerth, D. Peled, M.Y. Vardi, P. Wolper, Simple On-the-fly Automatic Verification of Linear Temporal Logic, PSTV95, Protocol Specification Testing and Verification, Warsaw, Poland. Chapman & Hall, Germany, 1995, 173–184.
5. C.A.R. Hoare, Communicating Sequential Processes, Comm. ACM, 21(8), 1978, 666–677.
6. G.J. Holzmann, An Improved Protocol Reachability Analysis Technique, Software Practice and Experience, Feb 1988, Vol 18, No 2, pp. 137-161.
7. G.J. Holzmann, Design and Validation of Computer Protocols, Prentice Hall, 1992.
8. G.J. Holzmann, D. Peled, An Improvement in Formal Verification, *7th* Int. Conf. on Formal Description Techniques, Berne, Switzerland, 1994, 177–194.
9. G.J. Holzmann, An Analysis of Bitstate Hashing, PSTV95, Protocol Specification Testing and Verification, Warsaw, Poland, Chapman & Hall, Germany, 1995, 301–314.
10. G.J. Holzmann, D. Peled, M. Yannakakis, On Nested Depth-First Search, In preparation, 1996.
11. B.W. Kernighan, D.M. Ritchie, The C programming Language, Prentice Hall, 1988.
12. R.P. Kurshan, Computer-Aided Verification of Coordinating Processes, Princeton University Press, 1994.
13. D. Peled, Combining Partial Order Reductions with On-the-fly Model-Checking, Proc. CAV'94, 6th International Conference on Computer Aided Verification, LNCS 818, Springer-Verlag, 377-390, 1994, Stanford CA, USA.
14. D. Peled, Partial Order Reduction: Model-Checking using Representatives, Proc. MFCS'96, 21st International Symposium on Mathamatical Foundations of Computer Science, September 1996, Cracow, Poland.
15. A. Pnueli, The temporal logic of programs, Proc. of the *18th* IEEE Symp. on Foundation of Computer Science, 1977, 46-57.
16. M.Y. Vardi, P. Wolper, An automata-theoretic approach to automatic program verification, Proc. of the *1st* Symposium on Logic in Computer Science, 1986, Cambridge, England, 322–331.
17. P. Wolper, Temporal Logic Can be More Expressive, Information and Control 56 (1983), 72–99.

The Murφ Verification System

David L. Dill

Computer Systems Laboratory
Stanford University
Email: dill@cs.stanford.edu

Abstract. This is a brief overview of the Murφ verification system.

The Murφ description language

Murφ is both a description language and a verifier for finite state concurrent systems [DDHY92]. It is appropriate for protocols and finite-state systems which can reasonably be modelled as a collection of processes that run at arbitrary speeds, where the steps of the processes interleave (only one process takes a step at any time), and where the processes interact by reading and writing shared variables. The Murφ verifier works by explicitly generating states and storing them in a hash table. We have put some effort into developing state reduction techniques, including symmetry reduction [ID93a, ID93b] , exploitation of reversible rules [ID96a], and verification of systems with varying numbers of replicated components [ID96b]. We have also investigated probabilistic verification techniques in Murφ [SD95c].

The Murφ description language was inspired by Misra and Chandy's Unity formalism [CM88]. A Murφ description consists of a collection of declarations of constants, data types such as subranges, records, and arrays, global variables, transition rules (which are guarded commands), start rules, and invariants.

The rules are similar to compound statements Pascal or Modula. Indeed, a rule can be arbitrarily complex, yet it is still executed atomically, meaning that the other rules cannot interfere. A *state* consists of the current values of the global variables. An *execution* of a Murφ program is any sequence of states that can be generated by starting in one of the states generated by a start rule, then repeatedly selecting a rule and executing it. Executing a rule generally changes the state, because the rule assigns to the global variables. Murφ is nondeterministic: there can be many executions, varying according to which rule was selected at each step of the execution.

A user can encode one of several concurrent processes by declaring variables for the process state and providing rules to capture its behavior. The behavior of several processes can be simulated by forming the union of the state variables and rules into a single Murφ program. Rule selection then simulates scheduling choices (the process whose rule is chosen runs next) as well as nondeterministic choice within a process.

Verification

The basic Murφ verifier generates all of the reachable states systematically, using a standard search algorithm such as breadth-first search. The search uses two data structures: a set of states whose descendants must be explored, and a table of states which have been previously encountered. When the search generates a state that is already in the table, the search is cut off. The invariant, which is a predicate which

reads the state variables, is evaluated in each newly generated state. If the result is *false*, verification halts and an error message is generated. The same effect can be achieved by an execution of an **error** statement in a rule. Similarly, if a state has no successors other than itself, the verifier halts and reports an error. In either event, the verifier also prints an execution from a start state to the offending state, to help with debugging.

We believe that explicit state verifiers are still useful, even when there are highly efficient BDD-based verifiers. One reason is that they are more predictable – performance is more closely related to the number of states, so the behavior of the verifier is more stable than with clever symbolic representations. The other reason is that some protocols, notably the ones we were most interested in verifying, require great cleverness to attack with successfully with BDDs. A naive approach performs much worse than Murφ. It is necessary to use non-obvious representations of state, identify variables that are functions of other variables, and/or decompose BDDs in various ways [HD93b, HD93a, HYD94]. Thus, verifying a such protocol with BDDs requires more expert users than attacking the same protocol with Murφ.

The basic Murφ verifier has been applied very successfully to several problems. It is especially suitable for multiprocessor cache coherence problems, because those were the problems we were working on most intensively when we were designing and redesigning the verifier. However, it has also been used for link-level protocols, a hybrid byzantine agreement algorithm, mutual exclusion algorithms, memory model specifications, and probably numerous other examples.

Symmetry reduction

In the last few years, we have found several ways of improving the performance of Murφ. The first was to exploit *symmetry* [ID93a, ID93b]. In some cases (particularly high-level descriptions of multiprocessor cache coherence protocols), components or values of a type can be exchanged arbitrarily without affecting the future behavior of the protocol. We have exploited this in Murφ by adding a new data type, called a *ScalarSet*, which is a subrange type with the additional restriction that it cannot be used in any way that "breaks the symmetry" between elements of the type (for example, there are no literal constants of the type, and one value cannot be compared with another using $<$). The Murφ semantic analyzer enforces these constraints, so that symmetry cannot be broken in the description.

Symmetry is exploited in the verifier by doing *symmetry reduction*. A *canonicalization function* is constructed by the verifier, which maps all states which are equivalent up to rearrangement of the elements of a scalarset to a particular representative state (a simple example of normalization would be sorting an array whose index set is a scalarset, if there are no scalarsets in the array itself). States are canonicalized before they are looked up or stored in the state table, so a state is not inspected if it is equivalent to a state in the state table, even if the states are not identical. This optimization has resulted in 100-fold reductions in the numbers of states generated in some cache coherence protocols. In certain cases (when a scalarset is not used as an array index), systems with unbounded scalarsets can be verified. For example, this property can be used to verify cache coherence regardless of the number of data values, and, hence, the number of bits in each data value.

Recent improvements

More recently, we have found an optimization which avoids storing transient states in the state table. The optimization works by identifying rules that do not lose information

when they are executed. The verifier can execute the "backwards" to map normalize transient states by finding a unique non-transient progenitor state from which they evolved [ID96a].

Most recently, we have developed a way of verifying certain systems with arbitrary numbers of replicated components in Murφ [ID96b]. The replicated components are flagged by using a datatype *RepetitiveID*, which is similar to a scalarset type but even more restricted. The verifier exploits this by working in an abstract space, where every global state is mapped to an abstract state which keeps track of whether there are zero, one, or more than zero of the replicated components in each component state. This method can be used to show that cache coherence protocols work properly for any number of processors. The method can be combined with symmetry reduction and the method of the previous paragraph to yield truly massive reductions in the state explosion problem.

We have also been exploring probabilistic verification algorithms, originally based on ideas from Gerard Holzmann, Pierre Wolper, and Denis Leroy [Hol87, WL, WL93], in which a small signature for each state is entered into the hash table instead of the state itself, saving a great deal of space at the expensive of some probability of producing a false positive result. The key is to find a bound on this probability, as Leroy and Wolper did. We have found several ways to reduce this bound, by changing the search and hashing algorithms and doing a more refined analysis of the probability [SD95a, SD95b, SD96]. This work has culminated in a factor-of-four reduction in the number of bits required per state, compared with Wolper and Leroy's original result, while guaranteeing the same or lower probability of missing an error,

Liveness

A few years ago, we implemented a version of Murφ which could verify common forms of liveness properties, expressed in a subset of linear temporal logic, using quite efficient state exploration algorithms. However, we have not updated the liveness verifier to use symmetry reduction and subsequent optimizations.

The Murφ verifier is available free by anonymous ftp from `snooze.stanford.edu` (directory /pub/murphi), under very liberal licensing terms.

Acknowledgements

The Murφ system was designed, implemented, redesigned, reimplemented, etc. by many different people, including me and the following students: C. Han Yang, Alan J. Hu, Andreas Drexler, Ralph Melton, C. Norris Ip, Seungjoon Park, and Ulrich Stern.

References

[CM88] K. Mani Chandy and Jayadev Misra. *Parallel Program Design — a Foundation.* Addison-Wesley, 1988.

[DDHY92] David L. Dill, Andreas J. Drexler, Alan J. Hu, and C. Han Yang. Protocol verification as a hardware design aid. *IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 522–525, 1992.

[HD93a] Alan J. Hu and David L. Dill. Conjunctive partitioned invariants for efficient verification with bdds. *5th International Conference on Computer-Aided Verification*, June 1993.

[HD93b] Alan J. Hu and David L. Dill. Reducing BDD size by exploiting functional dependencies. In *30th Design Automation Conference*, pages 266–271, 1993. Dallas, Texas, June 14–18.

[Hol87] G. J. Holzmann. On limits and possibilities of automated protocol analysis. In *Protocol Specification, Testing, and Verification. 7th International Conference*, pages 339–344, 1987.

[HYD94] Alan J. Hu, Gary York, and David L. Dill. New techniques for efficient verification with implicitly conjoined BDDs. In *31st Design Automation Conference*, pages 276–282, 1994.

[ID93a] C. Norris Ip and David L. Dill. Better verification through symmetry. *11th International Symposium on Computer Hardware Description Languages and Their Applications*, pages 87–100, April 1993. Extended version with complete proofs and semantic analysis to appear in Formal Methods in System Design.

[ID93b] C. Norris Ip and David L. Dill. Efficient verification of symmetric concurrent systems. *IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 230–234, October 1993.

[ID96a] C. Norris Ip and David L. Dill. State reduction using reversible rules. *33rd Design Automation Conference*, June 1996.

[ID96b] C. Norris Ip and David L. Dill. Verifying systems with replicated components in murφ. In *this proceedings*, 1996.

[SD95a] U. Stern and D. L. Dill. Automatic verification of the SCI cache coherence protocol. In *IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, pages 21–34, 1995.

[SD95b] U. Stern and D. L. Dill. Improved probabilistic verification by hash compaction. In *IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, pages 206–224, 1995.

[SD95c] Ulrich Stern and David L. Dill. Improved probabilistic verification by hash compaction. In *Correct Hardware Design and Verification Methods: IFIP WG10.5 Advanced Research Working Conference Proceedings*, 1995.

[SD96] U. Stern and D. L. Dill. A new scheme for memory-efficient probabilistic verification. Submitted for publication, 1996.

[WL] P. Wolper and D. Leroy. Reliable hashing without collision detection. Unpublished revised version of [WL93].

[WL93] P. Wolper and D. Leroy. Reliable hashing without collision detection. In *Computer Aided Verification. 5th International Conference*, pages 59–70, 1993.

The NCSU Concurrency Workbench*

Rance Cleaveland Steve Sims

Department of Computer Science
North Carolina State University
Raleigh, NC 27695-8206, USA
e-mail: {rance@csc, stsims@eos}.ncsu.edu

Abstract. The NCSU Concurrency Workbench is a tool for verifying finite-state systems. A key feature is its flexibility; its modular design eases the task of adding new analyses and changing the language users employ for describing systems. This note gives an overview of the system's features, including its capacity for generating diagnostic information for incorrect systems, and discusses some of its applications.

1 Introduction

The NCSU Concurrency Workbench (NCSU-CWB) [1] supports the automatic verification of finite-state concurrent systems. The main goal of the system is to provide users with a tool that is flexible and easy to use and yet whose performance is competitive with that of existing special-purpose tools. In support of the former, and like its predecessor, the (Edinburgh) Concurrency Workbench [9, 15], the NCSU-CWB includes implementations of decision procedures for calculating a number of different behavioral equivalences and preorders between systems and for determining whether systems satisfy formulas written in an expressive temporal logic, the modal mu-calculus. In contrast with the Edinburgh CWB, and with other tools, however, the NCSU-CWB also supports the following.

- **Diagnostic information.** The NCSU-CWB provides appropriate diagnostic information for explaining why two systems fail to be related by a given semantic equivalence or preorder.
- **Language flexibility.** The design of the system exploits the language-independence of its analysis routines by localizing language-specific procedures (syntactic analyzers, semantic functions) in one module. This enables users to change the system description language of the CWB using the Process Algebra Compiler tool [8].

In order to enable the tool to handle large "real-world" systems we have also paid great attention to issues of time- and space-efficiency.

The remainder of this note provides an overview of the features of the NCSU-CWB and reviews some case studies to which the tool has been applied.

* Research supported by NSF grants CCR-9120995 and CCR-9402807, ONR Young Investigator Award N00014-92-J-1582, NSF Young Investigator Award CCR-9257963, and AFOSR grant F49620-95-1-0508.

2 System Overview

User Interface. The NCSU-CWB supports two user interfaces: one text-based, and hence capable of running on a variety of different platforms, and the other graphical, and hence easier to use. Each consists of a "command loop"; users enter commands either textually or by pushing buttons, and the system calculates and displays the result. To analyze a system, a user first creates a file containing the definition of the system in the language supported by the version of the NCSU-CWB at hand, invokes the tool, loads the file into the NCSU-CWB, and executes appropriate commands.

Commands. In addition to providing a system simulation facility, the NCSU-CWB can compute a number of behavioral equivalences and preorders and calculate whether or not systems satisfy mu-calculus formulas. The textual interface of the NCSU-CWB uses a shell-like syntax for commands corresponding to these procedures. For example, the command to check whether systems `Spec` and `ABP` are must equivalent is the following: `eq -Smust Spec ABP`. Here the qualifier for the `-S` flag indicates which semantic equivalence should be checked; other possible qualifiers include `sim` (simulation equivalence), `obseq` (observational equivalence), `trace` (trace equivalence), and several others. The general command for preorder checking, `le`, uses the same scheme for specifying which ordering to check. In any case, appropriate *diagnostic information* is returned to the user when the given systems are found to be unrelated. For example, two systems are must equivalent iff they must pass exactly the same tests (in a precisely defined sense). Thus if `Spec` and `ABP` are not must equivalent, there must be a test that distinguishes them; when this is the case the NCSU-CWB computes such a test and displays it to the user.

The NCSU-CWB includes two model checkers, each of which allow users to specify formulas in a particular subset of the modal mu-calculus; one accepts formulas in the alternation free fragment [10] while the other accepts formulas in the L_2 fragment [2].

The system also supports commands for *minimizing* systems with respect to certain equivalences; this proves to be very useful in fighting state explosion.

Implementation. The NCSU-CWB is implemented in SML of New Jersey and consists of approximately 18,000 lines of code.

3 System Design

As stated in the introduction, one of the main features of the NCSU-CWB is its flexibility; in particular, users should be able to 1) add new equivalences and preorders, and 2) alter the language used for defining systems. In support of the first goal, the design of the NCSU-CWB uses *generic* preorder/equivalence-checking algorithms in conjunction with *system and formula transformations* in order to compute equivalences and preorders. To illustrate the approach, we

explain how the tool calculates its response to the command **eq -Smust Spec ABP** mentioned in the previous section.

1. **Spec** and **ABP** are "compiled" into automata.
2. These automata are transformed into a type of deterministic automata [6].
3. The transformed automata are fed into a generic equivalence checker, which calculates whether or not the two automata are bisimulation equivalent [14, 16]. If they are not equivalent, the checker produces a formula in a special logic that distinguishes the two automata [5].
4. If a formula is produced, a formula transformer converts it into a test, which is then returned to the user [4].

This general scheme is robust: numerous equivalences may be computed, and relevant diagnostic information generated, using appropriate transformations in tandem with bisimulation equivalence. Consequently, to add an equivalence relation to the NCSU-CWB, one need only define an appropriate process transformation and, if diagnostic information is desired, a formula transformation. A similar scheme is used for preorders.

In support of the second goal, all language-specific information has been localized within one module inside the NCSU-CWB. This module may be thought of as encapsulating the parse trees resulting from processing a user-supplied system description; it exports parsing and unparsing functions, and semantic routines (such as those for calculating single-step transitions), for these trees to the rest of the tool. To change the system description language, then, one need only alter routines in this module. The Process Algebra Compiler [8] greatly eases this task by providing users with a high-level notation in which to define the syntax and (operational) semantics of their languages. Using this tool, front ends for CCS, CSP [12], Basic Lotos [8], a version of CCS with priorities [7], and other languages have been generated for the NCSU-CWB.

4 Applications and Future Work

To date the NCSU-CWB has been applied to the analysis of several different systems, including the following.

- The connection phase of the UNI (Version 3.0) protocol used in ATM networks was formalized in CCS in [3] and verified. The largest finite-state machine handled in the course of the analysis contained about 60,000 reachable states.
- The timing behavior of an active-structure control system was analyzed in [11]. The system was formalized in a real-time variant of CCS and contained in excess of 10^{19} states. By minimizing the system in a component-wise manner, however, the analysis was carried to completion.
- The functional behavior of different variations of a railway signaling system was analyzed in [7]. The language used to define the system borrowed constructs from several different process algebras, while the system's requirements were specified using mu-calculus formulas.

As future work, we plan to investigate techniques for computing and displaying diagnostic information when systems fail to satisfy temporal formulas. We have also partially implemented on-the-fly versions of the preorder/equivalence-checking routines with a view to comparing the performance of global and on-the-fly algorithms.

References

1. The NCSU Concurrency Workbench home page. URL http://www4.ncsu.edu/eos/users/r/rance/WWW/ncsu-cw.html.
2. G. Bhat and R. Cleaveland. Efficient local model checking for fragments of the modal μ-calculus. In Margaria and Steffen [13], pages 107–126.
3. U. Celikkan. *Semantic Preorders in the Automated Verification of Concurrent Systems.* PhD thesis, North Carolina State University, 1995.
4. U. Celikkan and R. Cleaveland. Computing diagnostic tests for incorrect processes. In *Proceedings of the IFIP Symposium on Protocol Specification, Testing and Verification*, pages 263–278, Lake Buena Vista, Florida, June 1992. North-Holland.
5. R. Cleaveland. On automatically explaining bisimulation inequivalence. In E.M. Clarke and R.P. Kurshan, editors, *Computer Aided Verification (CAV '90)*, volume 531 of *Lecture Notes in Computer Science*, pages 364–372, Piscataway, NJ, June 1990. Springer-Verlag.
6. R. Cleaveland and M.C.B. Hennessy. Testing equivalence as a bisimulation equivalence. *Formal Aspects of Computing*, 5:1–20, 1993.
7. R. Cleaveland, G. Luettgen, V. Natarajan, and S. Sims. Priorities for verifying distributed systems. In Margaria and Steffen [13], pages 278–297.
8. R. Cleaveland, E. Madelaine, and S. Sims. A front-end generator for verification tools. In E. Brinksma, R. Cleaveland, K.G. Larsen, and B. Steffen, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS '95)*, volume 1019 of *Lecture Notes in Computer Science*, pages 153–173, Aarhus, Denmark, May 1995. Springer-Verlag.
9. R. Cleaveland, J. Parrow, and B. Steffen. The Concurrency Workbench: A semantics-based tool for the verification of finite-state systems. *ACM Transactions on Programming Languages and Systems*, 15(1):36–72, January 1993.
10. R. Cleaveland and B. Steffen. A linear-time model-checking algorithm for the alternation-free modal mu-calculus. *Formal Methods in System Design*, 2:121–147, 1993.
11. W. Elseaidy, J. Baugh, and R. Cleaveland. Verification of an active control system using temporal process algebra. *Engineering with Computers*, 12:46–61, 1996.
12. J. Gray. A CSP interface for the concurrency workbench. Undergraduate Honors Thesis, Department of Computer Science, North Carolina State University, May 1996.
13. T. Margaria and B. Steffen, editors. *Tools and Algorithms for the Construction and Analysis of Systems (TACAS '96)*, volume 1055 of *Lecture Notes in Computer Science*, Passau, Germany, March 1996. Springer-Verlag.
14. R. Milner. *Communication and Concurrency.* Prentice-Hall, London, 1989.
15. F. Moller and P. Stevens. *The Edinburgh Concurrency Workbench (Version 7.0).* University of Edinburgh, November 1994.
16. R. Paige and R.E. Tarjan. Three partition refinement algorithms. *SIAM Journal of Computing*, 16(6):973–989, December 1987.

The Concurrency Factory: A Development Environment for Concurrent Systems*

Rance Cleaveland
Dept. of Computer Science
N.C. State University
Raleigh, NC 27695-8206
rance@csc.ncsu.edu

Philip M. Lewis, Scott A. Smolka, Oleg Sokolsky
Dept. of Computer Science
SUNY at Stony Brook
Stony Brook, NY 11794–4400
{pml, sas, oleg}@cs.sunysb.edu

Abstract. The *Concurrency Factory* supports the specification, simulation, verification, and implementation of real-time concurrent systems such as communication protocols and process control systems. While the system uses *process algebra* as its underlying design formalism, the primary focus of the project is practical utility: the tools should be usable by engineers who are not familiar with formal models of concurrency, and it should be capable of handling large-scale systems such as those found in the telecommunications industry.
This paper serves as a status report for the Factory project and briefly describes a case-study involving the GNU UUCP i-protocol.

1 Introduction

The *Concurrency Factory* is an integrated toolset for specification, simulation, verification, and implementation of real-time concurrent systems such as communication protocols and process control systems. Two themes underpin the work done on the project: the use of *process algebra* [Mil89, BK84, Hoa85] as a formal design notation, and the provision of *practical* support for formal design analysis. Our goal is to make the Factory usable by system engineers who may not be familiar with formal verification as well as applicable to problems of the size found in industrial applications.

In order to achieve these aims, the Factory includes the following major components.

- A graphical editor, VTView [Tre92], and a simulator, VTSim [Jai93], for for hierarchically structured networks of finite-state processes. The graphical language, GCCS, resembles informal design diagrams drawn by engineers but possesses a formal, process-algebra-based semantics. We are currently extending the GUI to allow processes to be embedded in states of other processes, thereby permitting compact specifications such as those found in statecharts [Har87].

* Research supported in part by NSF Grants CCR–9120995, CCR–9208585, CCR–9257963, and CCR–9402807, AFOSR Grants F49620-93-1-0250 and F49620-95-1-0508, and ONR Grant N00014-92-J-1582.

- Support for system designs expressed in a programming-language-inspired design notation, VPL [Sok96]. VPL is a simple language for concurrent processes that communicate values from a finite data domain; as is the case with GCCS, however, the language features an underlying process-algebraic semantics. A compiler translates VPL programs into networks of finite-state processes.
- A collection of *analysis routines* that currently includes linear-time local and global model checker for the alternation-free fragment of the modal mu-calculus [CS93, Sok96], a local model checker for a real-time extension of this logic [SS95], and strong and weak bisimulation checkers.
  Care is being taken to ensure that these algorithms are efficient enough to be used on real-life systems. For example, we are investigating how these algorithms can be parallelized [ZS92, lic94], and made to perform incrementally [SS94].
- A *compiler* for transforming VTView and VPL specifications into executable code. The current Factory prototype produces Facile [GMP89] code, a concurrent language that symmetrically integrates many of the features of Standard ML [Mil84] and CCS [Mil89]. We are also considering adding a concurrent extension of C++ as another target language. The compiler relieves the user of the burden of manually recoding their designs in the target language of their final system.

The Concurrency Factory is written in C++ and executes under X-Windows, using Motif as the graphics engine, so that it is efficient, easily extendible, and highly portable. It is currently running on SUN SPARCstations under SunOS Release 4.1.

The remainder of this note describes VTView and VTSim and briefly discusses the i-protocol study. A fuller account of the system may be found in [CGL+94] and at URL `http://www.cs.sunysb.edu/~concurr/`.

2 VTView, GCCS and VTSim

The graphical user interface of the Concurrency Factory consists of the graphical editor, VTView [Tre92], and the graphical simulator, VTSim [Jai93]. VTView [Tre92] supports the design of hierarchically structured systems of communicating tasks expressed in GCCS, a graphical specification language. GCCS provides system builders with intuitive constructs (buses, ports, links, a subsystem facility, etc.) for concurrent systems, and it allows for both top-down and bottom-up development methodologies. The tool maintains an internal representation of systems as they are being created; this internal representation may then be manipulated by other tools in the system.

In contrast with other graphical languages [Har87, Mar89], GCCS is designed to model systems in which processes execute asynchronously (although communication between processes is synchronous). The language is equipped with two semantics: one involving a translation into Milner's CCS [Mil89], and another in

the form of a structural operational semantics, à la Plotkin [Plo81]. The latter semantics has been "implemented" in the Factory as a collection of methods that compute the set of transitions that are possible for a system in a given state. encapsulating the semantics of VTView objects, all tools within the Factory, including the simulator and model checkers, are guaranteed to interpret GCCS systems.

VTSim [Jai93] permits users to simulate graphically the execution of GCCS systems built using VTView. The tool provides both interactive and automatic modes of operation, and it also includes features such as breakpoints and reverse execution. The user may view the simulated execution of a system at different levels in the structure; one can either choose to observe the simulation at the interprocess level and watch the flow of messages, or one can look at individual processes in order to see why messages are sent when they are.

3 A Case Study: The i-protocol

The most sophisticated case study undertaken to date involved the use of the Concurrency Factory's local model checker to uncover and correct a subtle livelock in the i-protocol, a bidirectional sliding-window protocol implemented in the GNU UUCP file transfer utility. We analyzed a version of the protocol whose window size was 2; in the course of the analysis, the model checker explored 1.079×10^6 states out of a total estimated global state space of 1.473×10^{12}.

One key to the successful outcome of the case study was the use of an abstraction to reduce the message sequence number space from 32 — the constant defined in the protocol's C-code — to $2W$, where W is the window size. This insight underscores a central feature of practical use of formal verification: user understanding of the system being analyzed is crucial.

4 Future Work

We plan to extend the Factory in several directions, including the generation of simulator-based diagnostic information for verification routines, the development of improved state-space management techniques based on the underlying process-algebraic model, the support of languages besides Facile by the design compiler, and broader support for real-time systems.

References

[BK84] J. A. Bergstra and J. W. Klop. Process algebra for synchronous communication. *Information and Computation*, 60:109–137, 1984.

[CGL+94] R. Cleaveland, J. N. Gada, P. M. Lewis, S. A. Smolka, O. Sokolsky, and S. Zhang. The concurrency factory — practical tools for specification, simulation, verification, and implementation of concurrent systems. In G.E. Blelloch, K.M. Chandy, and S. Jagannathan, editors, *Proceedings of DIMACS*

Workshop on Specification of Parallel Algorithms, volume 18 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 75–90, Princeton, NJ, May 1994. American Mathematical Society.

[CS93] R. Cleaveland and B. U. Steffen. A linear-time model checking algorithm for the alternation-free modal mu-calculus. *Formal Methods in System Design*, 2, 1993.

[GMP89] A. Giacalone, P. Mishra, and S. Prasad. Facile: A symmertric integration of concurrent and functional programming. *International Journal of Parallel Programming*, 18(2), 1989.

[Har87] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, London, 1985.

[Jai93] S. Jain. VTSIM: A graphical simulator for finite-state networks. Master's thesis, Department of Computer Science, North Carolina State University, 1993.

[lic94] *Ninth Annual Symposium on Logic in Computer Science (LICS '94)*, Versailles, France, July 1994. Computer Society Press.

[Mar89] F. Maraninchi. Argonaute, graphical description, semantics and verification of reactive systems by using a process algebra. In *Proc. CAV '89*, volume 407 of *Lecture Notes in Computer Science*, pages 38–53, Grenoble, June 1989. Springer-Verlag.

[Mil84] R. Milner. A proposal for standard ML. Technical Report CSR-157-83, Department of Computer Science, University of Edinburgh, 1984.

[Mil89] R. Milner. *Communication and Concurrency*. International Series in Computer Science. Prentice Hall, 1989.

[Plo81] G. D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Computer Science Department, Aarhus University, 1981.

[Sok96] O. Sokolsky. *Efficient Graph-Based Algorithms for Model Checking in the Modal Mu-Calculus*. PhD thesis, Department of Computer Science, SUNY at Stony Brook, April 1996.

[SS94] O. Sokolsky and S. A. Smolka. Incremental model checking. In *Proceedings of the 6th International Conference on Computer-Aided Verification*. American Mathematical Society, 1994.

[SS95] O. Sokolsky and S. A. Smolka. Local model checking for real-time systems. In *Proceedings of the 7th International Conference on Computer-Aided Verification*. American Mathematical Society, 1995.

[Tre92] V. Trehan. VTVIEW: A graphical editor for hierarchical networks of finite-state processes. Master's thesis, Department of Computer Science, North Carolina State University, December 1992.

[ZS92] S. Zhang and S. A. Smolka. Efficient parallelization of equivalence checking algorithms. In M. Dias and R. Groz, editors, *Proceedings of FORTE '92 – Fifth International Conference on Formal Description Techniques*, pages 133–146, October 1992.

XVERSA: An Integrated Graphical and Textual Toolset for the Specification and Analysis of Resource-Bound Real-Time Systems *

Duncan Clarke, Hanêne Ben-Abdallah, Insup Lee and Hong-liang Xie
Department of Computer and Information Science
University of Pennsylvania
Philadelphia, PA 19104-6389
{dclarke,hanene,lee,hxie}@saul.cis.upenn.edu

Oleg Sokolsky
Computer Command and Control Company
Philadelphia, PA 19104
sokolsky@cccc.com

Abstract. We present XVERSA, a set of tools for the specification and analysis of resource-bound real-time systems. XVERSA facilitates the use of the Algebra of Communicating Shared Resources (ACSR), a real-time process algebra with explicit notions of resources and priority. A text based user interface supports syntax checking, analysis based on equivalence checking, state space exploration, and algebraic rewriting. A graphical user interface allows systems to be described and analyzed using intuitive pictorial representations of ACSR language elements.

1 Introduction

There has been significant progress in the development of formal methods for the design of real-time systems in an effort to increase safety and reliability. Formal approaches to the specification and analysis of real-time systems have taken many forms, including state machines, logics, and process algebras. We focus here on tools to support the algebraic paradigm. The algebra we use is the Algebra of Communicating Shared Resources (ACSR)[LBGG94].

ACSR is a timed process algebra that facilitates the description of concurrent real-time systems with serially reusable resources. Most concurrent real-time process algebras adequately capture delays due to process synchronization, e.g., timed extensions of the classic untimed process algebras CSP and CCS[BB91, MT90, NS94]. However, these algebras abstract out resource-specific delays and priority arbitration mechanisms. In contrast, the computation model of ACSR is based on the view that the notions of resource and priority are central to real-time systems. The use of shared resources is modeled by timed actions whose executions are subject to the availability of resources. Contention for synchronization and resources is arbitrated according to the priorities of the competing actions.

To facilitate the use of ACSR in the design and analysis of real-time systems we have created GCSR [BALC95], a graphical language that captures the semantics of ACSR in an intuitive pictorial representation, and XVERSA, a toolset that automates the analysis of ACSR and GCSR system models.

The remainder of this paper is organized as follows. Section 2 introduces the ACSR and GCSR languages. Section 3 describes the XVERSA toolset. Section 4 presents some concluding remarks and information on how to obtain further information and an executable copy of the toolset.

2 The ACSR and GCSR Formalisms

ACSR is a timed process algebra based on the synchronization model of CCS that includes features for representing synchronization, time, temporal scopes[LG85], resource requirements, and priorities. The semantics of ACSR has been developed for both dense time and discrete time models. However, the tools presented in this paper use the discrete time model exclusively.

The semantics of an ACSR process is defined in terms of a prioritized labeled transition system. Edges are labeled with prioritized *events* of the form (e, p), or *actions* that represent sets of prioritized resources to be

* This research was supported in part by NSF CCR-9415346, AFOSR F49620-95-1-0508, and ARO DAAH04-95-1-0092.

consumed for one time unit, e.g., $\{(r_1, p_1), \cdots, (r_n, p_n)\}$. Events model synchronization between concurrent process terms in a manner similar to that used in CCS. Actions represent resource allocation during the synchronous passage of one unit of discrete time. When several events and/or actions are offered simultaneously, a preemption relation determines which events or actions are allowed by pruning low priority edges.

ACSR offers two basic notions of behavior equivalence that are defined over the prioritized labeled transition system. The first equivalence relation is based on strong bisimulation, $\sim_\pi$, which insures that equivalent processes match one another's labeled transitions; it is a congruence relation. The second is based on weak bisimulation, $\approx_\pi$, which insures that equivalent processes match one another's non-τ events but allows one process to make transitions on τ that an equivalent process does not match.

A sound and complete set of approximately 30 $\sim_\pi$-preserving algebraic laws has been developed for ACSR. These laws can be used to derive proofs of properties of ACSR process expressions.

The Graphical Communicating Shared Resources (GCSR) language addresses a shortcoming of ACSR (and process algebras in general): textual, mathematical notations often produce obtuse descriptions. GCSR was developed to support the modular, hierarchical, and thus scalable, specification of real-time systems. In GCSR, the visibility scope of communication events, which reflect potential dependencies between system components, can be limited. Furthermore, GCSR's notion of hierarchy is *structured* in the sense that no edge can cross node boundaries and there is a graphical distinction between control transfer due to an interrupt versus an exception, i.e., involuntary versus voluntary release of control. These two syntactic features, in addition to the explicit representation of resources and priorities, distinguishes GCSR from other graphical languages for real-time systems, e.g., Statecharts[Har87], Modechart[JM94] and Communicating Real-time State Machines[Sha92].

The GCSR and ACSR languages are well integrated as there is a sound translation both from GCSR descriptions to ACSR processes and vice versa[BA96]. Thus, the theory of ACSR (including its semantic model, notions of equivalence, and set of algebraic laws) is directly applicable to GCSR. For instance, the algebraic laws of ACSR can be used to restructure a GCSR description to a graphically more succinct, e.g., fewer edges and nodes, yet, equivalent GCSR description. In addition, the sound integration between GCSR and ACSR makes it is possible to mix the graphical and textual notations; for example, to specify the high-level view of a system graphically and then fill the details of components textually.

3 The XVERSA Toolset

We have implemented a toolset with a graphical user interface to facilitate the use of ACSR and GCSR for modeling and analysis of real-time systems. Figure 1 shows the overall structure of the XVERSA system. The user's view of the tool is provided by the GCSR GUI and an X-Windows interface. The analysis of ACSR specifications is carried out by the VERSA system[CLX95b] that is accessed through these interfaces.

The user interfaces are responsible for management of input/output streams. They allow processes to be entered as graphical GCSR process descriptions or as ACSR processes using a text-based notation. Graphical input of GCSR specifications is managed with drawing support functions, syntax-checking functions, and automated translation from GCSR to ACSR. The text-based notation accepted by the X-Windows interface enhances the ACSR process algebra with the facility to define macros (e.g., to define manifest constants) and indexing which can be used to emulate value-passing.

Within VERSA there are four major functional areas for analyzing processes: term rewriting, state space exploration, equivalence testing, and interactive execution.

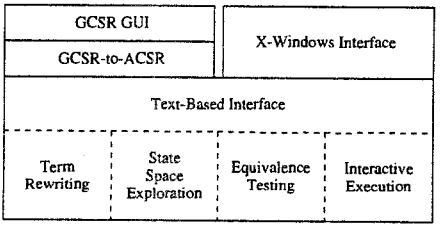


Fig. 1. The XVERSA Toolset

The rewrite system facilitates the rewriting of ACSR process expressions according to sound algebraic laws that preserve prioritized strong equivalence, a bisimulation relation that respects priority. At the direction of the user, the rewrite system applies pre-defined algebraic laws to one or more processes, producing a new process that may be bound to a new, or pre-existing process variable. In this way, algebraic proofs of the equivalence of process expressions may be developed. The tool aids this process by automatically determining and applying laws applicable to a highlighted ACSR term.

State space exploration, equivalence testing and interactive execution operate on a labeled transition system (LTS) representation of the system being analyzed. The LTS for one or more processes is produced by an algorithm that expands the process to produce a labeled transition system representing all possible executions. The LTS construction algorithm also prunes edges made unreachable by the semantics of the prioritized transition system, in most cases reducing the size of the resulting LTS.

State space exploration analysis can be used to determine key properties of a system's LTS. These include (1) number of states and transitions; (2) presence of deadlocked states; (3) states capable of *Zeno* behaviors (*i.e.*, infinite sequences of instantaneous events); (4) states that require synchronization to take place before time can progress; and (5) reachability of specific externally observable events.

Process equivalence can be tested using a number of different notions of equivalence including syntactic equivalence, a weaker syntactic equivalence which allows renaming of process variables and simple changes in structure, prioritized strong equivalence, and prioritized weak equivalence. In the order listed, these notions of equivalence increase in computational complexity and decrease in "strength" (i.e., equate more terms).

The interactive execution feature allows user-directed execution of process specifications. The user may interactively step through the LTS one action at a time, produce traces from random executions of the LTS, save process configurations to a stack for later analysis while an alternate path is explored, and analyze the size and deadlock characteristics of the LTS resulting from their process.

The XVERSA toolset has been used successfully to model and analyze railroad crossing systems[LBAC96], airport taxiways[BALC95], real-time schedulability analysis problems[CLX95a], the Philips audio control protocol[BPV94], the production cell case study[LL95, BA96], and to verify the correctness of a Sunshine ATM switching network[CL95].

4 Summary

We have presented XVERSA, a toolset that supports the formal analysis of resource-bound real-time systems. XVERSA offers a graphical process description language and X-Windows based tools that automate time consuming and error-prone analysis tasks. Our research into the theory of ACSR/GCSR and supporting tools is ongoing. Current goals include (1) the enhancement of ACSR/GCSR with value-passing; (2) the development of a refinement theory for ACSR/GCSR processes; and (3) implementation of alternative semantic representations.

Further information on ACSR and GCSR is available on the World Wide Web at

`http://www.cis.upenn.edu/~rtg/home.html`.

The XVERSA tools and descriptions of several case studies using XVERSA are available from

`http://www.cis.upenn.edu/~lee/duncan/versa.html`.

Questions about downloading and installing the tools should be addressed to `versa@saul.cis.upenn.edu`.

References

[BA96] Hanêne Ben-Abdallah. *Graphical Communicating Shared Resources: A Language for the Specification, Refinement, and Analysis of Real-Time Systems.* PhD thesis, Department of Computer and Information Science, University of Pennsylvania, 1996.

[BALC95] Hanêne Ben-Abdallah, Insup Lee, and Jin-Youg Choi. A graphical language with formal semantics for the specification and analysis of real-time systems. In *Proc. of IEEE Real-Time Systems Symposium*, Pisa, Italy, December 1995.

[BB91] J.C.M. Baeten and J.A. Bergstra. Real Time Process Algebra. *Formal Aspects of Computing*, 3(2):142–188, 1991.

[BPV94] D. Bosscher, I. Polak, and F. Vaandrager. Verification of an Audio Control Protocol. In H. Langmaack, W.-P. de Roever, and J. Vytopil, editors, *FTRTFT '94: Formal Techniques in Real-time and Fault-tolerant Systems*, pages 170–192. LNCS 863, Springer-Verlag, 1994.

[CL95] Duncan Clarke and Insup Lee. A hybrid approach to formal verification applied to an ATM switching system. Technical report, Dept. of CIS, Univ. of Pennsylvania, Dec 1995.

[CLX95a] J-Y. Choi, I. Lee, and H-L Xie. The Specification and Schedulability Analysis of Real-Time Systems using ACSR. In *Proc. of IEEE Real-Time Systems Symposium*, December 1995.

[CLX95b] D. Clarke, I. Lee, and H. Xie. VERSA: A tool for the specification and analysis of resource-bound real-time systems. *Journal of Computer and Software Engineering*, 3(2), April 1995.

[Har87] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

[JM94] F. Jahanian and A.K. Mok. Modechart: A specification language for real-time systems. *IEEE Transactions on Software Engineering*, 20(12):933–947, December 1994.

[LBAC96] Insup Lee, Hanêne Ben-Abdallah, and Jin-Young Choi. A process algebraic method for the specification and analysis of real-time systems. In Constance Heitmeyer and Dino Mandrioli, editors, *Formal Methods for Real-Time Computing*, chapter 7. John Wiley & Sons, Chichester, January 1996.

[LBGG94] I. Lee, P. Brémond-Grégoire, and R. Gerber. A Process Algebraic Approach to the Specification and Analysis of Resource-Bound Real-Time Systems. *Proceedings of the IEEE*, 82(1):158–171, January 1994.

[LG85] I. Lee and V. Gehlot. Language Constructs for Distributed Real-Time Programming. In *Proc. IEEE Real-Time Systems Symposium*, 1985.

[LL95] Claus Lewerentz and Thomas Linder, editors. *Formal Development of Reactive Systems: Case Study Production Cell*, volume 891 of *Lecture Notes in Computer Science*. Springer-Verlag, 1995.

[MT90] F. Moller and C. Tofts. A Temporal Calculus of Communicating Systems. In *Proc. of CONCUR '90*, pages 401–415. LNCS 458, Springer Verlag, August 1990.

[NS94] X. Nicollin and J. Sifakis. The Algebra of Timed Processes ATP: Theory and Application. *Information and Computation*, 114(1):131–178, October 1994.

[Sha92] A. Shaw. Communicating Real-time State Machines. *IEEE Transactions on Software Engineering*, September 1992.

EVP: Integration of FDTs for the Analysis and Verification of Communication Protocols*

P. Merino, J.M. Troya**
Depto. Lenguajes y Ciencias de la Computación.
ETSI Informática-ETSI Telecomunicación
University of Málaga , 29071 Málaga , SPAIN

Abstract

EVP is an integrated tool-set for specification and the analysis of communication protocols and distributed systems. Specifications in standard Formal Description Techniques (FDTs) like SDL or LOTOS are translated into a common language, and then analysed with tools for this internal representation. The common language is a concurrent logic language specially designed for distributed programming. The analysis consists in interactive simulation and automatic verification.

1 Introduction

It appears to be accepted that none of the current FDTs fulfils all the needs of the communication software cycle, although most of the desired properties are available in some of these languages. The search for a unique kernel language to be used for the integration of several standard FDTs seems to be one of the trends in the protocol engineering field. This approach allows the integration of different languages in a multi-paradigm environment, by translating user specifications into a common language, giving the user the chance to use the formal description technique to fit the protocol complexity, his experience and the available tools. New tools have to be developed for this single language, and all the effort can be oriented to process this common language. Projects like SEDOS[4] and SPECS[9] follow this approach.

Logic languages have been shown to be useful as formal description techniques in modelling and analysing concurrent systems, especially communication protocols [11, 5], and could be suitable as common languages. In the Spanish project TEMA [6, 7], the new concurrent logic language DRL [2, 3] is designed to be the kernel language of a distributed multi-paradigm environment (EVP) for specifying and analysing communication protocols. The concurrent and distributed programming oriented nature of DRL allows translation from SDL and LOTOS specifications. Its

* This work was supported by Spanish projects PLANBA/TEMA and CICYT TIC-1301-PB

** Other members of the research group GISUM who have contributed to the tool are: C. Canal, M. Díaz, L. Fuentes, M.M. Gallardo, A.J. Nebro, E. Pimentel, M. Roldan, B. Rubio

distributed implementation is also suitable for prototyping and for constructing a distributed verification tool to check errors in DRL specifications. The topic of this paper is the current status and applications of the EVP environment.

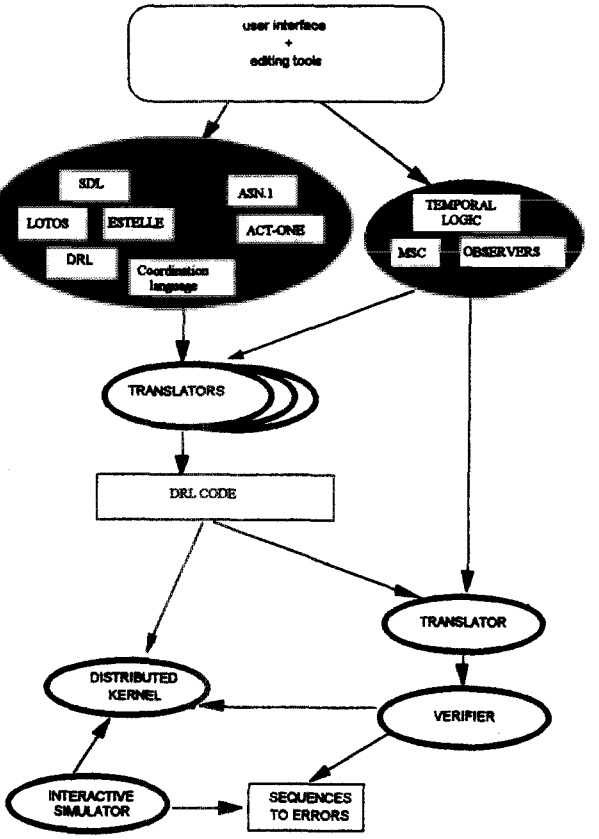


Figure 1. The EVP architecture

2 Overview of EVP

Figure 1 shows the intended complete EVP architecture. The formal specification of the protocol behaviour can be described by using standard FDTs and the new language DRL. Correctness requirements (properties) are expressed using different formalisms (temporal logic formulas, observers or MSCs). Data types are described with ASN.1 and ACT-ONE. The set of specifications are linked using a coordination language, and different translators produce a unique DRL description involving data, behaviour and requirements. A verifier is also produced to check both general and particular correctness requirements. Following this, simulation and verification tools

analyse the common DRL code and report the errors related to the original FDT. Problematic execution sequences produced by simulation and/or verification are registered, and can be used to find the origin of the error by interactive simulation.

The kernel language DRL is based on Parlog[1]. Like other concurrent logic languages, it naturally supports synchronisation through matching, communication via shared variables, fine granularity concurrence and non-determinism due to the don't care selection of clauses. It enhances Parlog with modularity, real time, process control facilities, explicit channels and effective distribution mechanisms.

3 Analysis and Verification

The analysis of the internal DRL specification can detect errors in the source FDT specification. Two tools have been developed for this purpose. The simulator is an interactive debugging system used to run some execution sequences in the protocol and to check errors in these sequences. The verifier automatically makes an on-the-fly checking of general and particular properties by reachability analysis. General properties include deadlock, undefined code, unspecified receptions and unreachable code. Particular properties refers to general temporal properties and assert violations. Since particular requirements for a protocol can be specified, a specialised verifier is automatically produced in order to obtain more efficiency.

To deal with the problem of the state space explosion, special annotations of DRL code have been introduced to allow different grain sizes in order to obtain the reachability graph. This kind of annotation gives the user the chance to mark piece of code to be executed both sequentially and in an atomic fashion, removing several interleavings with equivalent behaviours. Annotations are also automatically produced in the translation phase, by taking into account the source FDT semantics.

4 Current Status

A first version of the whole environment has been implemented and tested with several application level protocols specified in SDL and DRL. Current implementation runs on a network of workstations, and is composed of a distributed implementation of the kernel language, a translator of SDL-88 into DRL, a DRL oriented simulator, and a distributed verifier to check general errors in DRL specifications. The tool-set is integrated with an X-Window user-interface.

Simulation and verification tools have been constructed with DRL, using the meta-interpreter technique. This approach makes the development of a prototype environment easier, and also exploits the distributed implementation of DRL to produce a distributed verifier. This tool is useful for concurrently exploring different sequences in the protocol graph, and also for using the memory of several computers. Verification is carried out with a depth-first algorithm and state compactation using Holzmann's supertrace method. Errors currently detected include deadlock, undefined behaviour, assert violation and unreachable code.

The environment has been applied using standard FDTs and also the kernel language as the specification language. The most representative work using SDL is the specification, translating, simulation and verification of the AVP protocol [10], designed within the framework of the TEMA project.

DRL, as a user level FDT, is being employed in the specification and verification of AVP and also with the Multipoint Communication Service (MCS) [8]. In both cases, the specification is a validation oriented model, where only the relevant aspects of the behaviour of the protocols is considered.

5 Conclusions and Further Work

The GISUM group has designed and constructed an environment for the analysis of protocol specifications using different FDTs. User level specifications are translated into a common language, and then analysed with simulation and verification tools oriented to this common language. Instead of using classical internal representations like Petri nets or CFSM, we use a logic based language designed to represent distributed systems.

These experiences with real-life protocols show that these kinds of common languages are well suited for this purpose, but more improvements have to be made. Current work is mainly focused on the verification of temporal properties by extending the kernel language in order to express particular requirements of the protocols. However, other tasks are being carried out in the GISUM group, such as the design of a coordination language to link user level specifications and the integration of LOTOS and languages for data specification.

References

[1] Clark K.L, Gregory S. *Parlog: parallel programming in logic.* ACM Transactions on Programming Languages and Systems, 1986.

[2] Shapiro E.. *The Family of Concurrent Logic Programming Languages.* ACM Computing Surveys, Vol. 21, nº 3, pp. 413-510. Sept, 1989.

[3] Díaz M., Troya J.M. *A Parlog Based Real-Time Distributed Environment.* Future Generation Computer Systems 9. North-Holland. 1993.

[4] Diaz M., Vissers C., Ansart J. *Sedos Software Environment for the Design of Open Distributed Systems.* The Formal Description Technique LOTOS. North-Holland. 1989

[5] Dotan Y., Arazi B. *Using Flat Concurrent Prolog in System Modeling.* IEEE Transactions on Software Engineering, Vol. 17, Nº 6, June 1991.

[6] GISUM (Software Engineering Group). Universidad de Malaga. *EVP: Un Entorno Declarativo Distribuido para Especificacion y Validacion de Protocolos. Manual de Usuario.* Proyecto Tema. December 1994 (Spanish).

[7] GISUM (Software Engineering Group). Universidad de Malaga. *EVP: Un Entorno Declarativo Distribuido para Especificacion y Validacion de Protocolos. Tecnicas de Implementación*. Proyecto Tema. December 1994 (Spanish).

[8] ITU-T Study Group 8, Draft Recommendation T.125. *Multipoint Communication Service Protocol Specification*. November 1993.

[9] Reed R., Bouma W., Evans J.D., Dauphin M., Michel M. eds. The SPECS Consortium. *Specification and Programming Environment for Communication Software*. North-Holland. 1993.

[10] Ruado F., Benoliel L. *El Protocolo AV*. Telefónica I+D. October 1993.(Spanish)

[11] Sidhu D.P., *Protocol Verification via Executable Logic Specifications*. Proc. III IFIP Symposium on Protocol Specification, Testing, and Verification, North-Holland, 1983.

PVS: Combining Specification, Proof Checking, and Model Checking*

S. Owre, S. Rajan, J. M. Rushby, N. Shankar, and M. Srivas

Computer Science Laboratory, SRI International, Menlo Park CA 94025 USA
{owre, sree, rushby, shankar, srivas}@csl.sri.com
URL: http://www.csl.sri.com/pvs.html
Phone: +1 (415) 859-5272 Fax: +1 (415) 859-2844

PVS (Prototype Verification System) is an environment for constructing clear and precise specifications and for developing readable proofs that have been mechanically verified. It is designed to exploit the synergies between language and deduction, automation and interaction, and theorem proving and model checking. For example, the type system of PVS requires the use of theorem proving to establish type correctness, and conversely, type information is used extensively during a proof. Similarly, decision procedures are heavily used in order to simplify the tedious and obvious steps in a proof leaving the user to interactively supply the high-level steps in a verification. Model checking is one such decision procedure that is used to discharge temporal properties of specific finite-state systems.

A variety of examples from functional programming, fault tolerance, and real time computing have been verified using PVS [7]. The most substantial use of PVS has been in the verification of the microcode for selected instructions of a commercial-scale microprocessor called AAMP5 designed by Rockwell-Collins and containing about 500,000 transistors [5]. Most recently, PVS has been applied to the verification of the design of an SRT divider [9]. The key elements of the PVS design are described below in greater detail below.

1 Combining Theorem Proving and Typechecking

The PVS specification language is based on classical, simply typed higher-order logic, but the type system has been augmented with subtypes and dependent types. Though typechecking is undecidable for the PVS type system, the PVS typechecker automatically checks for simple type correctness and generates proof obligations corresponding to predicate subtypes. These proof obligations can be discharged through the use of the PVS proof checker. PVS also has parametric theories so that it is possible to capture, say, the notion of sorting with respect to arbitrary sizes, types, and ordering relations. By exploiting subtyping, dependent typing, and parametric theories, researchers at NASA Langley Research Center and SRI have developed a very general bit-vector library. Paul Miner at NASA

* The development of PVS was funded by SRI International through IR&D funds. Various applications and customizations have been funded by NSF Grant CCR-930044, NASA, ARPA contract A721, and NRL contract N00015-92-C-2177.

has developed a specification of portions of the IEEE 854 floating-point standard in PVS [6].

In PVS, the injective function space `injection` can be defined as a higher-order predicate subtype using the higher-order predicate `injective?` as shown below. The notation `(injective?)` is an abbreviation for {f | injective?(f)} which is the subtype of functions from D to R for which the predicate `injective?` holds.

```
functions [D, R: TYPE]: THEORY
 BEGIN
  f, g: VAR [D -> R]
  x, x1, x2: VAR D

  injective?(f): bool = (FORALL x1, x2: (f(x1) = f(x2) => (x1 = x2)))

  injection: TYPE = (injective?)

 END functions
```

We can also define the subtype `even` of even numbers and declare a function `double` as an injective function from the type of natural numbers `nat` to the subtype `even`.

```
even: TYPE = {i : nat | EXISTS (j : nat): i = 2 * j}

double : injection[nat, even] = (LAMBDA (i : nat): 2 * i)
```

When the declaration of `double` is typechecked, the typechecker generates two proof obligations or type correctness conditions (TCCS). The first TCC checks that the result computed by `double` is an even number. The second TCC checks that the definition of `double` is injective. Both TCCs are proved quickly and automatically using the default TCC strategy employed by the PVS proof checker. Proofs of more complicated TCCs can be constructed interactively.

The PVS specification language has a number of other features that exploit the interaction between theorem proving and typechecking. Conversely, type information is used heavily within a PVS proof so that predicate subtype constraints are automatically asserted to the decision procedures, and quantifier instantiations are typechecked and can generate TCC subgoals during a proof attempt. The practical experience with PVS has been that the type system does rapidly detect a lot of common specification errors.

2 Combining Decision Procedures with Interactive Proof

Decision Procedures. PVS employs decision procedures include the congruence closure algorithm for equality reasoning along with various decision procedures for various theories such as linear arithmetic, arrays, and tuples, in the presence of uninterpreted function symbols [10]. PVS does not merely make use of decision procedures to prove theorems but also to record type constraints and to

simplify subterms in a formula using any assumptions that *govern* the occurrence of the subterm. These governing assumptions can either be the test parts of surrounding conditional (IF-THEN-ELSE) expressions or type constraints on governing bound variables. Such simplifications typically ensure that formulas do not become too large in the course of a proof. Also important, is the fact that automatic rewriting is closely coupled with the use of decision procedures, since many of the conditions and type correctness conditions that must be discharged in applying a rewrite rule succumb rather easily to the decision procedures.

Strategies. The PVS proof checker provides powerful primitive inference steps that make heavy use of decision procedures, but proof construction solely in terms of even these inference steps can be quite tedious. PVS therefore provides a language for defining high-level inference strategies (which are similar to *tactics* in LCF [3]). This language includes recursion, a `let` binding construct, a backtracking `try` strategy construction, and a conditional `if` strategy construction. Typical strategies include those for heuristic instantiation of quantifiers, repeated skolemization, simplification, rewriting, and quantifer instantiation, and induction followed by simplification and rewriting. There are about a hundred strategies currently in PVS but only about thirty of these are commonly used. The others are used as intermediate steps in defining more powerful strategies. The use of powerful primitive inference steps makes it possible to define a small number of robust and flexible strategies that usually suffice for productive proof construction.

3 Integrating Model Checking and Theorem Proving

In the theorem proving approach to program verification, one verifies a property P of a program M by proving $M \supset P$. The model checking approach verifies the same program by showing that the state machine for M is a satisfying model of P, namely $M \models P$. For control-intensive approaches over small finite states, model checking is very effective since a more traditional Hoare logic style proof involves discovering a sufficiently strong invariant. These two approaches have traditionally been seen as incompatible ways of viewing the verification problem. In recent work [8], we were able to unify the two views and incorporate a model checker as decision procedure for a well-defined fragment of PVS.

This integration uses the mu-calculus as a medium for communicating between PVS and a model checker for the propositional mu-calculus. We have used this integration to verify a complicated communication protocol by means of abstraction and model checking [?], and also to prove the correctness of an N-process mutual exclusion protocol in such a way that the induction step used the correctness of the 2-process version of the protocol as verified by the model checker.

The general mu-calculus over a given *state* type essentially provides operators for defining least and greatest fixpoints of monotone predicate transformers. In Park's mu-calculus, the state type is restricted to n-tuple of booleans and extends quantified boolean formulas (i.e., propositional logic with boolean quantification)

to include the application of n-ary boolean predicates to n argument formulas. The relational terms can constructed by means of lambda-abstraction, or by taking the least fixpoint $\mu Q.F[Q]$ where Q is an n-ary predicate variable and F is a monotone predicate transformer. The greatest fixpoint operation can be written as $\nu Q.F[Q]$ and defined as $\neg\mu Q.\neg F[\neg Q]$. The mu-calculus can be easily defined in PVS. The temporal operators of the branching time temporal logic CTL can be defined using the mu-calculus [1]. An efficient model checking algorithm for the propositional mu-calculus was presented by Emerson and Lei [2], and the symbolic variant employing BDDs was presented by Burch, *et al* [1]. PVS employs a BDD-based mu-calculus validity checker due to Janssen [4].

When the state type is finite, i.e., constructed inductively from the booleans and scalar types using records, tuples, or arrays over subranges, the mu-calculus over such finite types (and the corresponding CTL) can be translated into the Boolean mu-calculus and model checking can be used as a decision procedure for this fragment. We do not discuss the details of this encoding here (see [8]).

References

1. J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, June 1992.
2. E.A. Emerson and C.L Lei. Efficient model checking in fragments of the propositional mu-calculus. In *Proceedings of the 10th Symposium on Principles of Programming Languages*, pages 84–96, New Orleans, LA, January 1985. Association for Computing Machinery.
3. M. Gordon, R. Milner, and C. Wadsworth. *Edinburgh LCF: A Mechanized Logic of Computation*, volume 78 of *Lecture Notes in Computer Science*. Springer-Verlag, 1979.
4. G. L. J. M. Janssen. *ROBDD Software*. Department of Electrical Engineering, Eindhoven University of Technology, October 1993.
5. Steven P. Miller and Mandayam Srivas. Formal verification of the AAMP5 microprocessor: A case study in the industrial use of formal methods. In *WIFT '95: Workshop on Industrial-Strength Formal Specification Techniques*, pages 2–16, Boca Raton, FL, 1995. IEEE Computer Society.
6. Paul S. Miner. Defining the IEEE-854 floating-point standard in PVS. Technical Memorandum 110167, NASA Langley Research Center, 1995.
7. Sam Owre, John Rushby, Natarajan Shankar, and Friedrich von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, February 1995.
8. S. Rajan, N. Shankar, and M.K. Srivas. An integration of model-checking with automated proof checking. In Pierre Wolper, editor, *Computer-Aided Verification, CAV '95*, volume 939 of *Lecture Notes in Computer Science*, pages 84–97, Liege, Belgium, June 1995. Springer-Verlag.
9. H. Ruess, M. K. Srivas, and N. Shankar. Modular verification of SRT division. In Rajeev Alur and Tom Henzinger, editors, *Computer-Aided Verification, CAV '96*, Lecture Notes in Computer Science, New Brunswick, NJ, July 1996. Springer-Verlag. To appear.
10. Robert E. Shostak. Deciding combinations of theories. *Journal of the ACM*, 31(1):1–12, January 1984.

STeP: Deductive-Algorithmic Verification of Reactive and Real-Time Systems*

Nikolaj Bjørner, Anca Browne, Eddie Chang, Michael Colón,
Arjun Kapur, Zohar Manna, Henny B. Sipma, and Tomás E. Uribe

Computer Science Department, Stanford University
Stanford, CA 94305

Abstract. The Stanford Temporal Prover, STeP, combines deductive methods with algorithmic techniques to verify linear-time temporal logic specifications of reactive and real-time systems. STeP uses verification rules, verification diagrams, automatically generated invariants, model checking, and a collection of decision procedures to verify finite- and infinite-state systems.

System Description: The Stanford Temporal Prover, STeP, supports the computer-aided formal verification of reactive, real-time (and, in particular, concurrent) systems based on temporal specifications. Reactive systems maintain an ongoing interaction with their environment; their specifications are typically expressed as constraints on their behavior over time. STeP is not restricted to finite-state systems, but combines algorithmic and deductive methods to allow the verification of a broad class of systems, including parameterized (N-component) circuit designs, parameterized (N-process) programs, and programs with infinite data domains.

The deductive methods of STeP verify temporal properties of systems by means of verification rules and verification diagrams. *Verification rules* are used to reduce temporal properties of systems to first-order verification conditions [8]. *Verification diagrams* [7, 3] provide a visual language for guiding, organizing, and displaying proofs. Verification diagrams allow the user to construct proofs hierarchically, starting from a high-level, intuitive proof sketch and proceeding incrementally, as necessary, through layers of greater detail.

Deductive verification almost always relies on finding, for a given program and specification, suitably strong auxiliary invariants and intermediate assertions. STeP implements a variety of techniques for automatic *invariant generation*. These methods include *local*, *linear* and *polyhedral* invariant generation, which perform an approximate, abstract propagation through the system [2]. Verification conditions can then be established using the automatically generated auxiliary invariants as background properties.

* This research was supported in part by the National Science Foundation under grant CCR-92-23226, the Advanced Research Projects Agency under NASA grant NAG2-892, the United States Air Force Office of Scientific Research under grant F49620-93-1-0139, the Department of the Army under grant DAAH04-95-1-0317, and a gift from Intel Corporation.

STeP also provides an integrated suite of simplification and decision procedures for automatically checking the validity of a large class of first-order and temporal formulas. This degree of automated deduction is intended to efficiently handle most verification conditions that arise in deductive verification. An interactive Gentzen-style theorem prover and a resolution-based prover are available to establish the verification conditions that are not proved automatically.

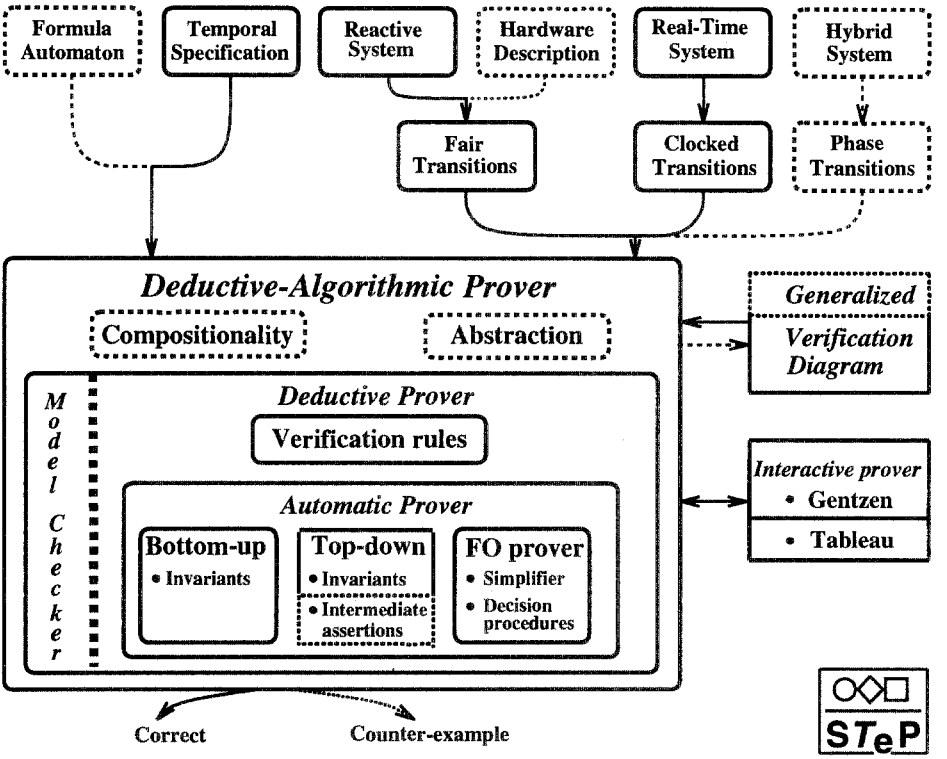


Fig. 1. An overview of the STeP system

System Structure: Fig. 1 presents an overview of STeP. Dotted lines indicate work in progress. The basic inputs are a reactive system (which can be a hardware or software description), expressed as a transition system, and a system property to be proved, represented by a temporal logic formula. Verification can be performed by the model checker or by deductive means. User guidance can be provided as intermediate assertions or verification diagrams. In either case, the system is responsible for generating and proving all of the required verification conditions. Tactics are available to automate parts of the high-level proof search by encoding long or repetitive sequences of proof commands. For a more extensive description of STeP and examples of verified programs, see [1, 6].

Interacting with STeP: STeP has three main interface components: the *Top-level Prover*, from which verification sessions are managed and verification rules are invoked; the *Interactive Prover*, used to prove the validity of first-order and temporal-logic formulas that are not proved automatically; and the *Verification Diagram Editor*, for the creation of Verification Diagrams. Fig. 2 shows these three interfaces, with a version of the Bakery algorithm loaded, together with a tree representing the ongoing proof process.

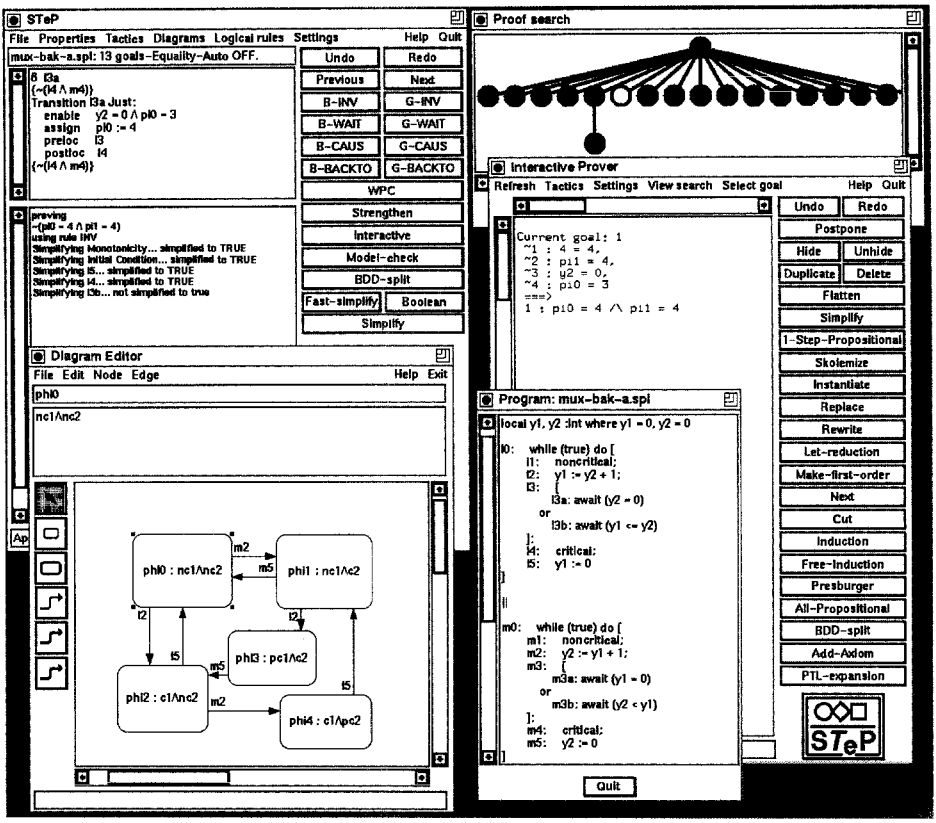


Fig. 2. Overview of STeP's interfaces

Real-time systems: STeP was recently extended to support the verification of safety properties of real-time systems, based on the computational model of *clocked transition systems* [9]. Systems described by timed transition systems or timed automata can be readily translated into this formalism. The specification language of linear-time temporal logic was extended, in turn, with real-valued clocks measuring progress of time.

Applications: STeP has been used to analyze a diverse number of systems, including: an infinite-state demarcation protocol used in distributed databases, a pipelined four-stage multiplication circuit, Ricart and Agrawala's mutual exclusion protocol, several (N-component) ring arbiters, Szymanski's N-process mutual-exclusion algorithm, and an industrial split-transaction bus protocol to coordinate access for six processors. Real-time systems analyzed include Fisher's mutual-exclusion protocol and a (parameterized) railroad gate controller [5].

STeP is being extended to support *deductive model checking* as described in [10], as well as modular verification diagrams [4].

Educational Version: An educational version of the system, which accompanies the textbook [8], is available. The distribution includes a comprehensive user manual [1] and a tutorial, as well as 40 example programs and their specifications, from the textbook, ready to be loaded. For many programs, ready-to-load verification diagrams are included as well.

STeP is implemented in Standard ML of New Jersey, using CML and eXene for its X-windows user interface. For information on obtaining the system, send e-mail to `step-request@cs.stanford.edu`.

References

1. BJØRNER, N., BROWNE, A., CHANG, E., COLÓN, M., KAPUR, A., MANNA, Z., SIPMA, H., AND URIBE, T. STeP: The Stanford Temporal Prover, User's Manual. Tech. Rep. STAN-CS-TR-95-1562, Computer Science Department, Stanford University, Nov. 1995.
2. BJØRNER, N., BROWNE, A., AND MANNA, Z. Automatic generation of invariants and intermediate assertions. In 1^{st} *Intl. Conf. on Principles and Practice of Constraint Programming* (Sept. 1995), vol. 976 of *LNCS*, Springer-Verlag, pp. 589–623.
3. BROWNE, A., MANNA, Z., AND SIPMA, H. Generalized verification diagrams. In *15th Conference on the Foundations of Software Technology and Theoretical Computer Science* (Dec. 1995), vol. 1026 of *LNCS*, pp. 484–498.
4. BROWNE, A., MANNA, Z., AND SIPMA, H. Modular verification diagrams. Tech. rep., Computer Science Department, Stanford University, 1996.
5. HEITMEYER, C., AND LYNCH, N. The generalized railroad crossing: A case study in formal verification of real-time systems. In *Proc. ICCC Real-Time Systems Symposium* (1994), IEEE Press, pp. 120–131.
6. MANNA, Z., ANUCHITANUKUL, A., BJØRNER, N., BROWNE, A., CHANG, E., COLÓN, M., DE ALFARO, L., DEVARAJAN, H., SIPMA, H., AND URIBE, T. STeP: The Stanford temporal prover. Tech. Rep. STAN-CS-TR-94-1518, Computer Science Department, Stanford University, July 1994.
7. MANNA, Z., AND PNUELI, A. Temporal verification diagrams. In *Proc. Int. Symp. on Theoretical Aspects of Computer Software* (1994), vol. 789 of *LNCS*, Springer-Verlag, pp. 726–765.
8. MANNA, Z., AND PNUELI, A. *Temporal Verification of Reactive Systems: Safety.* Springer-Verlag, New York, 1995.
9. MANNA, Z., AND PNUELI, A. Clocked transition systems. Tech. Rep. STAN-CS-TR-96-1566, Department of Computer Science, Stanford University, Apr. 1996.
10. SIPMA, H., URIBE, T., AND MANNA, Z. Deductive model checking. In *Proc.* 8^{th} *Intl. Conference on Computer Aided Verification* (July 1996), Springer-Verlag.

Symbolic Model Checking*

E. Clarke[1], K. McMillan[2], S. Campos[1] and V. Hartonas-Garmhausen[1]

[1] Carnegie Mellon University School of Computer Science, Pittsburgh, USA.
[2] Cadence Labs, Berkeley, USA

1 Introduction

Extensive simulation is currently the most widely used verification technique. However, simulation does not check all possible behaviors of a computing system. Exhaustive simulation is too expensive, and non-exhaustive simulation can miss important events, especially if the number of states in the system being verified is large. Other approaches for verification include theorem provers, term rewriting systems and proof checkers. These techniques, however, are usually very time consuming and require significant user intervention. Such characteristics limit the size of the systems they can verify in practice.

Temporal logic model checking [6, 7] is an alternative approach that has achieved significant results recently. Efficient algorithms are able to verify properties of realistic complex systems. In this technique, specifications are written as formulas in a propositional temporal logic and computer systems are represented by state-transition graphs. Verification is accomplished by an efficient breadth first search procedure that views the transition system as a model for the logic, and determines if the specifications are satisfied by that model.

Recent model checkers use symbolic algorithms, which allow the verification of extremely large state-spaces. In this approach the transition relation is represented implicitly by boolean formulas, and implemented by *binary decision diagrams* [1]. This usually results in a much smaller representation for the transition relation [16], allowing the size of the models being verified to increase up to more than 10^{20} states [2].

There are several other advantages to this approach. An important one is that the procedure is completely automatic. The model checker accepts a model description, specifications to be verified and determines, without user intervention, if the formulas are true or not for that model. Another advantage is that, if the formula is not true, the model checker will provide a counterexample. The counterexample is an execution trace that shows why the formula is not true. This is an extremely useful feature because it can help locate the source of the error and speed up the debugging process. Another advantage is the ability to verify partially specified systems. If a component hasn't been fully specified, some of its outputs can be assigned nondeterministic values. The set of

* This research was sponsored in part by the National Science Foundation under grant no. CCR-8722633, by the Semiconductor Research Corporation under contract 92-DJ-294, and by The Defense Advanced Research Projects Agency, Information Science and Technology Office, under the title "Research on Parallel Computing", ARPA Order No. 7330, issued by DARPA/CMO under Contract MDA972-90-C-0035.

behaviors modeled this way is a superset of the actual behaviors of the component. Useful information about the correctness of the system can be gathered before all the details have been determined. The abstracted model is then refined when more information about the component becomes available. This allows the verification of a system to proceed concurrently with its design. Consequently verification can provide valuable hints that will help designers eliminate errors earlier and define better systems.

The model checker used in this work is *Symbolic Model Verifier* (SMV) [16]. It has been applied successfully in the verification of several industrial systems. Examples include the Futurebus+ cache coherence protocol [9], the PCI Local Bus [3], a railway signalling system [14], an aircraft controller [5], a manufacturing system [13], and a medical monitoring system [4]. A survey about the technique can be found in [11].

2 Describing the System

The system being verified is described in the SMV language. We can specify synchronous or asynchronous, detailed deterministic or abstract nondeterministic finite state machines. The language provides modular hierarchical descriptions, reuse of components, and parameterization so that multiple instances of a module can use different data values. Within every module, local variables may be declared. The type of a variable may be boolean, an enumeration type or an integer subrange. For example:

```
VAR state0: {noncritical, trying, critical};
```

The value of the variables in each state are defined using `init` and `next`:

```
init(state0) := noncritical;
next(state0) :=
case
  (state0 = noncritical) : {trying,noncritical};
  (state0 = trying) & (state1 = noncritical): critical;
  (state0 = trying) & (state1 = trying) & (turn = turn0):
                          critical;
  (state0 = critical) :    {critical,noncritical};
  1: state0;
esac;
```

An SMV program can be viewed as a system of simultaneous equations whose solution determines the next state. When describing communication protocols, asychronous circuits, or other systems whose actions are not sychronized, we can define a set of parallel processes whose actions are interleaved arbitrarily in the execution of the program.

Fairness constraints can also be specified in SMV. A fairness constraint is an arbitrary set of states in the model, described by a temporal logic formula. A path in the model is considered fair with respect to a set of fairness constraints if each constraint is true infinitely often along the path (i.e. some state in the fair set of states is visited infinitely often). In SMV verification can be restricted to fair paths.

3 Verifying the System

Computation tree logic, CTL, is the logic used by SMV to express properties that will be verified. Formulas in CTL are built from atomic propositions, where each proposition corresponds to a variable in the model, boolean conectives $\neg$ and $\wedge$, and *temporal operators*. Each operator consists of two parts: a path quantifier followed by a temporal operator. Path quantifiers indicate that the property should be true of *all* paths from a given state (**A**), or *some* path from a given state (**E**). The temporal modality describes how events should be ordered with respect to time for a path specified by the path quantifier. They have the following informal meanings:

- **F** p — p holds sometime in the future.
- **G** p — p holds globally on the path.
- **X** p — p holds in the next state.
- p **U** q — q holds in the future, and p holds in all states until the state in which $q$holds.

The most common CTL operators are: **AG** p — p is globally true in all paths from the current state, i.e., p is *invariant*; **AF** p — p holds sometime in the future in all paths, i.e., p is *inevitable*; **EF** p — p holds sometime in the future for some path, i.e., p is *reachable*. Some examples of CTL formulas are given below to illustrate the expressiveness of the logic.

- **AG**$(req \rightarrow$ **AF** $ack)$: It is always the case that if the signal req is high, then eventually ack will also be high.
- **EF**$(started \wedge \neg ready)$: It is possible to get to a state where $started$ holds but $ready$ does not hold.
- **AG EF** $restart$: From any state it is possible to get to the $restart$ state.
- **AG**$(send \rightarrow$ **A**$[send$ **U** $recv])$: It is always the case that if $send$ occurs, then eventually $recv$ is true, and until that time, $send$ must remain true.

4 Conclusions

Symbolic model checking is a powerful formal specification and verification method that has been applied successfully in several industrial designs. Using symbolic model checking techniques it is possible to verify industrial-size finite state systems. State spaces with up to 10^{30} states can be exhaustively searched in minutes. Models with more than 10^{120} states have been verified using special techniques.

Several extensions to the original technique have been developed, making it even more powerful. Timing properties can be verified by performing a quantitative timing analysis [3, 5]. The designer can then analyze the performance of a system and gain insight in how well a system works early in the design process. Word-level model checking allows the verification of datapaths in addition to control [12]. Symmetry [8], abstraction [10, 15] and compositional reasoning [15] techniques significantly extend the power of model checking by exploiting the hierarchical structure of complex circuit designs and protocols.

More information about SMV, as well as the source code for the model checker can be found at: `http://www.cs.cmu.edu/~modelcheck`

References

1. R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8), 1986.
2. J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *Symposium on Logic in Computer Science*, 1990.
3. S. Campos, E. Clarke, W. Marrero, and M. Minea. Verifying the performance of the PCI local bus using symbolic techniques. In *International Conference on Computer Design*, 1995.
4. S. V. Campos, E. M. Clarke, W. Marrero, and M. Minea. Timing analysis of industrial real-time systems. In *Workshop on Industrial-strength Formal specification Techniques*, 1995.
5. S. V. Campos, E. M. Clarke, W. Marrero, M. Minea, and H. Hiraishi. Computing quantitative characteristics of finite-state real-time systems. In *IEEE Real-Time Systems Symposium*, 1994.
6. E. M. Clarke and E. A. Emerson. Synthesis of synchronization skeletons for branching time temporal logic. In *Logic of Programs: Workshop, Yorktown Heights, NY, May 1981*. Springer-Verlag, 1981. *Lecture Notes in Computer Science*, volume 131.
7. E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, 1986.
8. E. M. Clarke, T. Filkorn, and S. Jha. Exploiting symmetry in temporal logic model checking. In *Proceedings of the Fifth Workshop on Computer-Aided Verification*, June 1994.
9. E. M. Clarke, O. Grumberg, H. Hiraishi, S. Jha, D. E. Long, K. L. McMillan, and L. A. Ness. Verification of the Futurebus+ cache coherence protocol. In L. Claesen, editor, *International Symposium on Computer Hardware Description Languages and their Applications*. North-Holland, April 1993.
10. E. M. Clarke, O. Grumberg, and D. E. Long. Model checking and abstraction. In *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Programming Languages*, January 1992.
11. E. M. Clarke, O. Grumberg, and D. E. Long. Verification tools for finite-state concurrent systems. In *A Decade of Concurrency — Reflections and Perspectives*, 1994. Springer Lecture Notes in Computer Science, 803.
12. E. M. Clarke, M. Khaira, and X. Zhao. Word level model checking — avoiding the pentium FDIV error. In *Design Automation Conference*, June 1996.
13. V. Hartonas-Garmhausen, E.M. Clarke, and S. Campos. Deadlock prevention in flexible manufacturing systems using symbolic model checking. In *International Conference on Robotics and Automation*, 1996.
14. V. Hartonas-Garmhausen, T. Kurfess, E.M. Clarke, and D. Long. Automatic verification of industrial designs. In *Workshop on Industrial-strength Formal specification Techniques*, 1995.
15. D. E. Long. *Model checking, abstraction and compositional reasoning*. PhD thesis, SCS, Carnegie Mellon University, 1993.
16. K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

COSPAN

R. H. Hardin[1], Z. Har'El[2] and R. P. Kurshan[1]

[1] Bell Laboratories, Murray Hill, New Jersey 07974
[2] Dept. of Mathematics, Technion, Haifa, Israel

Abstract. COSPAN (*Co*ordination *Sp*ecification *An*alizer [AKS83]) is an algorithmic computer-aided verification system. It's semantic model [Ku94] is founded on ω-automata: for a *process* P (modelling a system to be verified) and a *task* T which P is intended to perform, *verification* consists of the automata language containment test

$$\mathcal{L}(P) \subset \mathcal{L}(T) \ .$$

If the test fails, COSPAN presents an error track which illustrates the error. Typically, P is not monolithic, but is represented as a ("synchronous") parallel composition $P = P_1 \otimes \cdots \otimes P_k$ of component processes (all modelled as ω-automata). Asynchronous coordination of component processes may be modelled through nondeterminism in the components. The process model can be set either to Mealy or Moore machines.
COSPAN has been used on commercial applications for over a decade, both for software and hardware design verification [HK90]. Recently, it has been implemented as the "verification engine" in the commercial hardware verification tool FormalCheck^{TM}, which is supported for hardware verification by the Bell Labs Design Automation center.
The COSPAN application domains and utilities are enumerated.

1 Methodology

COSPAN supports top-down design development through *successive refinements*, in which one may start by verifying an abstract prototype design, and then successively refine the prototype, verifying new properties in each respective refinement, in such a way that each successive refinement inherits all the properties verified in all previous steps. This is possible because ω-automata define *linear-time* behaviors (in contrast with *branching-time* behaviors as defined by logics such as CTL [CE82]). From the final design model, COSPAN will generate C code as well as input to the Lucent Technologies synthesis tool $BESTMAP^{TM}$ in order to implement automatically its models as software or hardware.

Its analysis algorithms include automated *localization reduction*, *symmetry reduction* and the more general user-defined *homomorphic reduction* [Ku94], used to limit the computational complexity of verification (which in general is intractable in the number of coordinating components k). Localization is an (automatically) generated homomorphic reduction, which reduces the model P relative to the property T which is to be verified. This reduction conservatively

discards parts of the design irrelevant to T, and resizes retained portions relative to the other retained parts. If the design changes, already-verified properties need not be re-verified if the changes lie outside of their respective localizations. Whether this is the case can be checked through a computationally simple CRC map of the localized parse trees before and after the change, for each respective property ("regression verification" [HKRS96]). In homomorphic verification, the user produces two models (possibly with completely different sets of events and system variables) and a language map which relates the two; COSPAN checks that the map in fact defines a (behavior-preserving) homomorphism between the two models. This check may be done component-wise, which is important for circumventing the possibly intractable complexity of performing the check on the entire system at once.

COSPAN supports *real-time* verification, implemented with Rajeev Alur in terms of successive approximations [AIKY93] in order to limit its computational complexity.

In its core routines, COSPAN can use either symbolic- (BDD-based) or explicit-state enumeration algorithms. Both algorithms are "on the fly" in the sense that errors may be detected before exploring the entire state space. The error track that COSPAN produces optionally contains line-number/source-file information for each variable to support *back-referencing*, indicating where in the source the given variable was assigned the value indicated in the error track (at the point where the other variables were assigned their designated respective values). The BDD-based algorithms (based on a new BDD package implemented by David Long) use partitioning, dynamic reordering and a version of the Emerson-Lei algorithm [EL86] for the language-containment test. The explicit-state algorithms optionally invoke several caching and hashing options, a generalized Hopcroft state minimization algorithm [Gr73], and use the Tarjan strongly connected components algorithm [Ta72] for the language-containment test.

2 Language

The *state* of a computer program at an instant of time is the simultaneous value of all its respective variables (assuming that is well-defined). As space complexity is a principal bottleneck in algorithmic verification, the dimension of this state vector is an important factor in the over-all complexity of the verification [Ku94]. If the values of some variables are functions (or relations) of the values of other variables, then the effective dimension of the state vector, as a factor in computing space complexity, is the actual dimension less the number of dependent variables.

If a programming language syntactically draws a distinction between such *state* variables, and the remaining *combinational* variables whose [possible] values at each instant are a function [relation] of the values of the state variables, then this distinction can be exploited in algorithmic verification. Programming languages designed to describe integrated circuits ("hardware description languages" or HDL's) often distinguish between such state variables, used to desig-

nate *latches* or computer memory and combinational variables, used to designate *logic*, for the reason that the performance and fabrication cost of hardware are governed by the same factors as algorithmic verification. For hardware synthesized automatically from HDL code, the syntactic distinction between state and logic in the HDL is reflected in a corresponding physical distinction in the hardware.

An even more fundamental distinction between hardware and software, as it pertains to algorithmic verification, is the use of recursion and pointers, less common in HDL's and more problematic for algorithmic verification.

All of these issues have been significant factors in the generally greater acceptance of algorithmic verification in the commercial hardware design process than in the commercial software design process, and are reflected in the commercial languages for which interfaces to COSPAN have been built. Notwithstanding this, the principles of algorithmic verification and their implementation in COSPAN are applicable equally to hardware and software.

To date, interfaces to COSPAN have been written for the commercial HDL's Verilog and VHDL, the CCITT-standard protocol specification language SDL, and the non-commercial languages HSIS [HSIS94] from UC Berkeley, Holzmann's Promela [Ho91] and Lamport's TLA [La94]. However, it would be feasible to make interfaces to any other languages for which the above language issues pertaining to verification can be adequately addressed.

Conversely, the COSPAN parser can translate its native S/R language into input to the verification tools SMV [Mc93] and SPIN [Ho91] and the automated theorem-provers HOL [Go88] and Larch [GG91] (via TLA).

COSPAN 's native language S/R distinguishes between state variables and combinational variables. The language supports nondeterministic, conditional (*i.e.*, if-then-else) variable assignments; variables of type bounded integer, enumerated, Boolean and pointer (to structures); arrays and records; and integer and bit-vector arithmetic. Modular hierarchy, scoping, parallel and sequential execution, homomorphism declaration and general ω-automaton fairness (acceptance) are supported as well. Property specification also is supported through a library of parameterized automata designed to facilitate the definition of any ω-regular property. Thus, the user need not deal with automata acceptance conditions directly, but may confine coding to conventional programming plus the definition of the properties to be verified and system constraints (if any) specified by assigning propositional expressions to parameters of the library automata. The commercial FormalCheck version of COSPAN provides a graphical interface for this purpose.

3 Utilities

As already described above, the principal COSPAN algorithms include:

- explicit- and symbolic-state enumeration language containment tests
- localization reduction

- homomorphic verification
- timing verification
- state space minimization
- symmetry reduction
- regression verification
- C code generation (model implementation)
- back-referencing from an error track to the source

Additionally, COSPAN supports the following options:

- data-path profiling; manual variable ordering (for BDD's)
- Wolper's [WL93] k-bit generalization of Holzmann's state-vector bit-hashing
- over-writing of states outside of the search path (explicit enumeration)
- randomized property checking
- automated sub-model extraction
- stability checking (for asynchronous models)
- deadlock detection; CTL AGEFp (*NOT* preserved by refinement)
- embedding of C code into S/R (for implementation and test generation)
- listing of the transition graph
- suspension, termination, polling and restarting verification runs
- interactive "check-pointing" (single path exploration)

4 To Obtain

A version of COSPAN is available to universities for research and educational purposes, at no charge: inquire to k@research.bell-labs.com .

References

[AKS83] S. Aggarwal, R. P. Kurshan, D. Sharma, A Language for the Specification and Analysis of Protocols, PSTV III, North-Holland (1983) 35–50.

[AIKY93] R. Alur, A. Itai, R. P. Kurshan, M. Yannakakis, Timing Verification by Successive Approximation, LNCS **663** (1993) 137–150.

[CE82] E. M. Clarke, E. A. Emerson, Design and Synthesis of Synchronization Skeletons for Branching Time Temporal Logic, LNCS **131** (1982) 52–71.

[EL86] E. A. Emerson, C. L. Lei, Efficient Model Checking in Fragments of the Propositional Mu-Calculus, LICS 1986, 267–278.

[GG91] S. J. Garland, J. V. Guttag, A Guide to LP: the Larch Prover, Technical Report 82, Digital Equipment Corporation, 1991.

[Go88] M. Gordon, A Proof-Generating System for Higher-Order Logic, Kluwer SECS **35**, 1988, 73–128.

[Gr73] D. Gries, Describing an Algorithm by Hopcroft, Acta Inf. **2** (1973) 97–109.

[HKRS96] R. H. Hardin, R. P. Kurshan, J. A. Reeds, N. J. A. Sloane, Regression Verification, Bell Laboratories TM11272-960502-14 (1996).

[HK90] Z. Har'El, R. P. Kurshan, Software for Analytical Development of Communications Protocols, *AT&T Tech. J.* **69** (1990) 45–59.

[Ho91] G. J. Holzmann, *Design and Validation of Computer Protocols*, Prentice Hall, 1991.

[HSIS94] A. Aziz, *et. al*, HSIS: A BDD-Based Environment for Formal Verification, Proc. 31st Design Automation Conf., 1994.

[Ku94] R. P. Kurshan, *Computer-aided Verification of Coordinating Processes – The Automata-Theoretic Approach*, Princeton Univ. Press, 1994.

[La94] L. Lamport, The temporal logic of actions, TOPLAS **16** (1994) 872–923.

[Mc93] K. L. McMillan, *Symbolic Model Checking*, Kluwer, 1993.

[Ta72] R. E. Tarjan, Depth-First Search and Linear Graph Algorithms, SIAM J. Comput. **1** (1972) 146–160.

[WL93] P. Wolper, D. Leroy, Reliable Hashing Without Collision Detection, Lecture Notes in Computer Science (LNCS) **697** (1993) 59–70.

VIS : A System for Verification and Synthesis

Robert K. Brayton* Gary D. Hachtel** Alberto Sangiovanni-Vincentelli*
Fabio Somenzi** Adnan Aziz* Szu-Tsung Cheng* Stephen Edwards*
Sunil Khatri* Yuji Kukimoto* Abelardo Pardo** Shaz Qadeer*
Rajeev K. Ranjan* Shaker Sarwary*** Thomas R. Shiple* Gitanjali Swamy*
Tiziano Villa*

1 Introduction

VIS (Verification Interacting with Synthesis) is a tool that integrates the verification, simulation, and synthesis of finite-state hardware systems. It uses a Verilog front end and supports fair CTL model checking, language emptiness checking, combinational and sequential equivalence checking, cycle-based simulation, and hierarchical synthesis.

We designed VIS to maximize performance by using state-of-the-art algorithms, and to provide a solid platform for future research in formal verification. VIS improves upon existing verification tools by:

1. providing a better programming environment,
2. providing new capabilities, and
3. improving performance.

We have incorporated software engineering methods into the design of VIS. In particular, we provide extensive documentation that is automatically extracted from the source files for browsing on the World Wide Web.

We describe the major capabilities of VIS in Section 2, and give a brief description of the underlying algorithms in Section 3. We discuss the VIS programming environment in Section 4, and conclude with future work in Section 5.

2 Capabilities of VIS

We briefly describe the salient features of VIS. VIS has both an interactive command interface and a batch mode. For a detailed description of the full functionality of VIS, with examples of usage, refer to the VIS Manual [2].

Verilog front end VIS operates on an intermediate format called BLIF-MV, which is an extension of BLIF, the intermediate format for logic synthesis accepted by SIS [8]. VIS includes a stand-alone compiler from Verilog to BLIF-MV, called VL2MV [3], which supports a synthesizable subset of Verilog. VL2MV extracts a set of interacting finite state machines that preserves the behavior of the source Verilog program defined in terms of simulated results. Two new features have been added to Verilog:

1. Nondeterminism. A nondeterministic construct, $ND, has been added to specify nondeterminism on wire variables; this is the only legal way to introduce nondeterminism in VIS.

* Department of EECS, University of California, Berkeley, CA 94720
** Department of Electrical and Computer Engineering, University of Colorado, Boulder, CO 80309
*** Lattice Semiconductor, Milpitas, CA 95035

2. Symbolic variables. We allow the symbolic specification of multi-valued variables, rather than insisting on explicit binary encodings for each value. VL2MV extends Verilog to allow symbolic variables using an enumerated type mechanism similar to the one available in the C programming language.

It would be easy to provide a translator from another HDL language, like VHDL or Esterel, to BLIF-MV.

Hierarchy and initialization When a BLIF-MV description is read into VIS, it is stored hierarchically as a tree of modules, which in turn consist of sub-modules. The variables of these modules may be related using a symbolic "table". This hierarchy can be traversed in a manner similar to traversing directories in UNIX. The hierarchy, or portions of it, may be flattenned to a netlist of (symbolic) logic gates (network). The network is the starting point for the verification and simulation algorithms; simulation and verification operations can be performed at any subtree of the hierarchy. It is possible to replace the subhierarchy rooted at the current node with a new hierarchy specified by a new BLIF-MV file, which might be a synthesized module or a manually abstracted module. VIS can also output the hierarchy below the current node to a BLIF-MV file.

Interaction with synthesis VIS can interact with SIS to optimize the existing logic by reading and writing the BLIF format, which SIS recognizes. Synthesis can be performed on any node of the hierarchy.

Abstraction Manual abstraction can be performed by giving a file containing the names of variables to abstract. For each variable appearing in the file, a new primary input node is created to drive all the nodes that were previously driven by the variable. Abstracting a net effectively allows it to take any value in its range, at every clock cycle.

Fair CTL model checking and language emptiness check VIS performs fair CTL model checking under Büchi fairness constraints. In addition, VIS can perform language emptiness checking by model checking the formula $EG\ true$. The language of a design is given by sequences over the set of reachable states that do not violate the fairness constraint. The language emptiness check can be used to perform language containment by expressing the set of bad behaviors as another component of the system. If model checking or language emptiness fail, VIS reports the failure with a counterexample, (i.e., behavior seen in the system that does not satisfy the property - for model checking, or valid behavior seen in the system - for language emptiness). This is called the "debug" trace.

Equivalence checking VIS provides the capability to check the combinational equivalence of two designs. An important usage of combinational equivalence is to provide a sanity check when re-synthesizing portions of a network. VIS also provides the capability to test the sequential equivalence of two designs. Sequential verification is done by building the product finite state machine, and checking whether a state where the values of two corresponding outputs differ, can be reached from the set of initial states of the product machine. If this happens, a debug trace is provided. Both combinational and sequential verification are implemented using BDD-based routines.

Simulation VIS also provides traditional design verification in the form of a cycle-based simulator that uses BDD techniques. Since VIS performs both formal verification and simulation using the same data structures, consistency between them is ensured. VIS can generate random input patterns or accept user-specified input patterns. Any subtree of the specified hierarchy may be simulated.

3 Algorithms

This section briefly discusses the significant algorithms of VIS. The fundamental data structure for these algorithms is a multi-level network of latches and combinational gates that is created by flattening the hierarchy. It is assumed that there are no combinational cycles in the network. The primary inputs and latch outputs are referred to as *combinational inputs* and the primary outputs and latch inputs are referred to as *combinational outputs*. The variables of a network are multi-valued, and logic functions over these variables are represented by multi-valued decision diagrams (MDDs) which are an extension of BDDs.

MDD variable ordering In order to construct the MDD's, the combinational input and next state variables must be ordered. The combinational input variables are ordered by doing a depth-first traversal of the logic that generates the combinational outputs. The order in which the output logic cones are visited is determined using the algorithm of Aziz *et al.* [1]. This algorithm orders the latches to decrease a communication complexity bound (where backward edges are more expensive than forward edges) on the latch communication graph. The traversal of an output logic cone is done in such a way that the combinational inputs farthest from the outputs appear earlier in the ordering. We use the merging technique of Fujii *et al.* to handle those variables that appear in multiple cones of logic [6]. Next state variables are inserted into the variable ordering immediately after the corresponding present state variables. We have found that forcing corresponding present state and next state variables to remain adjacent to each other is usually beneficial.

If the user has some knowledge of a good ordering, then a partial or total ordering on the variables can be read in. In addition, dynamic variable ordering is supported.

Partitioning the network Once the description of a system has been read in and the ordering of the variables assigned, an abstracted view of the system is created in which the functions of the network are stored as MDDs. This abstracted view, called a "partition", is the input to model checking and reachability. It can be created in several ways. At one extreme, combinational output functions are defined directly in terms of combinational inputs (monolithic transition functions). On the other extreme, there is an MDD corresponding to each node in the network representing the functionality of the node in terms of its fanins, i.e., a variable is introduced for each node in the network. In general, intermediate variables can be introduced to represent the functionality of a cluster of nodes in the original network. This flexibility allows very large designs to be represented and manipulated.

Image/Pre-image computation Our image/pre-image computation technique is based on an early variable quantification heuristic [7]. The initialization process consists of creating a bit-level relation for the next state function of each latch in the network. These bit-level relations are then ordered to optimally exploit early quantification. Next, the

relations of several bits are grouped together, making a cluster whenever the MDD size of the group reaches a threshold. Next, each cluster is simplified by quantifying out the primary inputs local to that cluster. Finally, the orders of the clusters for image and pre-image are calculated and stored. Also stored is the schedule of variables for early quantification.

Reachability analysis Reachability analysis uses image computation. In addition to the improved image computation described, the performance of reachability analysis is also improved by exploiting three sets of don't cares (in the following $R_k(\mathbf{x})$ represents the set of states reached from the initial states in k or fewer steps):

1. Selection of the frontier set for computing $R_{k+1}(\mathbf{x})$, given $R_k(\mathbf{x})$. The frontier set $F(\mathbf{x})$ can be any set satisfying the following inequality: $R_k(\mathbf{x})\overline{R_{k-1}(\mathbf{x})} \subseteq F(\mathbf{x}) \subseteq R_k(\mathbf{x})$.
2. Simplification of the transition relation $T(\mathbf{x}, \mathbf{u}, \mathbf{y})$, by taking the generalized cofactor with respect to $F(\mathbf{x})$ (we care only about the transitions originating from the frontier states).
3. Simplification of the transition relation $T(\mathbf{x}, \mathbf{u}, \mathbf{y})$, by taking the generalized cofactor with respect to $\overline{R_k(\mathbf{y})}$ (we care only about the transitions to the set of states not reached thus far).

Model checking and debugging We use the algorithms presented in [4] as the basis for fair CTL model checking and debugging. In addition, a special algorithm has been implemented to improve the efficiency of checking invariants. Also, a structural pruning technique is used to eliminate those parts of the network that cannot affect the formula being checked. This is particularly useful in conjunction with the abstraction mechanism mentioned in Section 2. Finally, don't cares arising from the unreachable states, and from the fixed point computations, are used to simplify intermediate MDDs.

4 Programming Environment

One of the key goals of VIS is to serve as a platform for developing new verification algorithms. We have used object-oriented programming style of SIS as our paradigm. VIS is composed of 18 packages; each exports a set of routines for manipulating a particular data structure, or for performing a set of related functions (e.g., there are packages for model checking, variable ordering, and manipulating the network data structure). New packages can be added easily. This wealth of exported functions can be used by future programmers to quickly assemble new algorithms. All functions adhere to a common naming convention so that it is easy to find functions in the documentation.

Particular attention was paid to the design of the interfaces to packages that are still the subject of ongoing research (e.g., MDD variable ordering, image computation, and partitioning). This makes it easy for other researchers to plug in their algorithms for performing a particular task, and then evaluate their algorithm within the context of VIS.

Extensive user and programmer documentation exists for VIS. The creation of this documentation was aided by the tool `ext` [5], which extracts documentation embedded in the source code. For each function, the programmer provides a synopsis and a complete description, and `ext` automatically extracts this information, along with the function name and argument types, into an HTML file that can be viewed on the World Wide Web. Documentation for user commands is extracted in a similar fashion.

5 Conclusions and Future Work

We have described the verification and synthesis tool VIS, which offers a better programming environment, new capabilities, and improved performance over existing verification tools. We have implemented VIS using the C programming language, and it has been ported to many different operating systems and architectures. The capabilities of VIS have been tested on the sequential circuits from the ISCAS benchmark set and some industrial designs.

As part of future work, we intend to explore and support explicit methods for state enumeration, verification of asynchronous systems, hierarchical synthesis, partitioning schemes, language containment, and incremental techniques for synthesis and verification. In particular, we want to explore the synergy between verification and synthesis.

For more information about VIS, to demo VIS, or to get a copy, visit the VIS home page [9].

Acknowledgments

We would like to thank Adrian Isles, Sriram Rajamani, and Serdar Tasiran for their assistance in developing VIS.

References

1. A. Aziz, S. Tasiran, and R. K. Brayton. BDD Variable Ordering for Interacting Finite State Machines. In *Proc. of the Design Automation Conf.*, pages 283–288, San Diago, CA, June 1994.
2. R. K. Brayton, G. D. Hachtel, A. Sangiovanni-Vincentelli, F. Somenzi, A. Aziz, S. T. Cheng, S. Edwards, S. Khatri, Y. Kukimoto, A. Pardo, S. Qadeer, R. K. Ranjan, S. Sarwary, T. R. Shiple, G. Swamy, and T. Villa. VIS: A System for Verification and Synthesis. Technical Report UCB/ERL M95, Electronics Research Lab, Univ. of California, Berkeley, CA 94720, Dec. 1995.
3. S. T. Cheng. Compiling Verilog into Automata. Technical Report UCB/ERL M94/37, Electronics Research Lab, Univ. of California, Berkeley, CA 94720, May 1994.
4. E. M. Clarke, O. Grumberg, K. L. McMillan, and X. Zhao. Efficient generation of counterexamples and witnesses in symbolic model checking. In *Proc. 32nd Design Automat. Conf.*, pages 427–432, June 1995.
5. S. Edwards. *The Ext System*, 1995. `http://www.eecs.berkeley.edu/~sedwards/ext`.
6. H. Fujii, G. Ootomo, and C. Hori. Interleaving based variable ordering methods for ordered binary decision diagrams. In *Proc. Intl. Conf. on Computer-Aided Design*, pages 38–41, Nov. 1993.
7. R. K. Ranjan, A. Aziz, B. Plessier, C. Pixley, and R. K. Brayton. Efficient Formal Design Verification: Data Structure + Algorithms. Technical Report UCB/ERL M94/100, Electronics Research Lab, Univ. of California, Berkeley, CA 94720, Oct. 1994.
8. E. M. Sentovich, K. J. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P. R. Stephan, R. K. Brayton, and A. L. Sangiovanni-Vincentelli. SIS: A System for Sequential Circuit Synthesis. Technical Report UCB/ERL M92/41, Electronics Research Lab, Univ. of California, Berkeley, CA 94720, May 1992.
9. The VIS Group. *VIS: Verification Interacting with Synthesis*, 1995. `http://www-cad.eecs.berkeley.edu/Respep/Research/vis`.

MDG Tools for the Verification of RTL Designs

K.D. Anon[1] N. Boulerice[1] E. Cerny[1] F. Corella[2]
M. Langevin[3] X. Song[1] S. Tahar[1] Y. Xu[1] Z. Zhou[1]

[1] D'IRO, Université de Montréal, Canada, {cerny,song,zhouz}@iro.umontreal.ca
[2] Hewlett-Packard Company, USA, fcorella@hprpcd.rose.hp.com
[3] GMD-SET, Germany, langevin@gmd.de

1 Introduction

Although ROBDDs [1, 2] have proved to be a powerful tool for automated hardware verification, they require a Boolean representation of the circuit. Since the size of an ROBDD grows, sometimes exponentially, with the number of Boolean variables, ROBDD-based verification cannot be directly applied to circuits with complex datapaths.

We have recently proposed a new class of decision graphs, called *Multiway Decision Graphs* (MDGs) [3], that comprises, but is much broader than, the class of ROBDDs. The underlying logic of MDGs is a subset of many-sorted first-order logic with a distinction between *concrete* and *abstract* sorts. A concrete sort has an enumeration while an abstract sort does not. Hence a data value can be represented by a single variable of abstract sort, rather than by a vector of Boolean variables, and a data operation can be viewed as a black box and represented by an *uninterpreted* function symbol. MDGs are thus much more compact than ROBDDs for designs containing a datapath, and this greatly increases the range of applications.

We have developed a collection of MDG tools that include implementations of the basic MDG operators and verification procedures for RTL designs.

2 The Structure of MDG Tools

The current MDG tools (V1.0) are logically organized into five modules in three layers as shown in Figure 1.

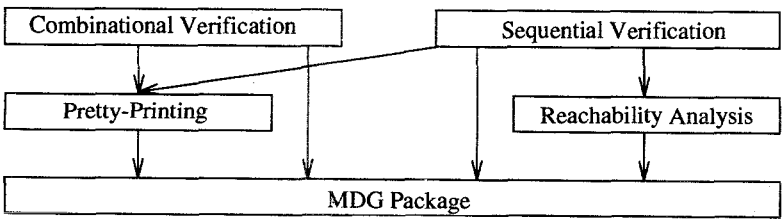


Fig. 1. The structure of MDG tools

The lowest layer is the *MDG package* module. It provides a set of MDG operators and a number of utilities, which are the basic building blocks for MDG-based applications.

In the middle layer, there are two modules: (i) the *printing module* provides various pretty-printing procedures; (ii) the *reachability analysis module* implements the reachability analysis algorithm for abstract state machines (ASMs) [4]. It also contains a counterexample facility which generates explanations when the invariant being verified is violated.

Currently, there are two modules in the application layer. The *combinational verification module* provides the equivalence checking of combinational circuits. The *sequential verification module* have two applications: (i) invariant checking and (ii) equivalence checking of two state machines.

As a prototype, the MDG tools have been implemented in Prolog and currently run under Quintus Prolog Version 3.2.

3 The MDG Package

Like an ROBDD package, the MDG package contains the basic procedures for developing MDG-based applications. Here we briefly review the MDG operators and the utilities. For more details see [3, 6].

- *Disjunction* (Disj): performs disjunction for a set of MDGs.
- *Relational product* (RelP): performs conjunction, abstraction by existential quantification and renaming operations in one traversal of graphs for a set of MDGs. It is used for image computation.
- *Pruning-by-subsumption* (PbyS): takes as input two MDGs P and Q, and produces an MDG that approximates $P \wedge \neg Q$ by removing zero or more paths in MDG P which are subsumed by MDG Q. When P and Q represents sets of states, if all paths in P are removed, then $P \subseteq Q$. PbyS is used for subsumption checking and frontier-set simplification.
- *Rewriting* (Rew): rewrites the first-order terms appearing in an MDG as edges and node labels according to a conditional term rewriting system.

Computed-tables, implemented as hash tables, are used to reduce the complexity of the algorithms by exploiting the structure sharing of MDGs.

The major utilities provided with the package are term assembly and graph assembly, which construct terms and graphs using reduction tables, also implemented as hash tables.

4 Applications to Hardware Verification

Abstract State Machines. The MDG tools are intended for Abstract State Machines (ASM) verification [4, 3] rather than Finite State Machine (FSM) verification. They can be used for FSMs as well, but they are less efficient than ROBDDs for this purpose, due in part to the space requirements of our current Prolog implementation.

An *abstract description* of a state machine, called *abstract state machine* (ASM) [4], is obtained by letting some data input, state or output variables be of an abstract sort, and the datapath operations be uninterpreted function symbols. Just as ROBDDs for encoding FSMs, MDGs are used to compactly represent sets of (abstract) states and transition/output relations for ASMs.

The MDG tools accept as hardware description a Prolog-style HDL, MDG–HDL, which allows the use of abstract variables for representing data signals. The MDG–HDL description is then compiled into the ASM model in internal MDG data structures.

MDG–HDL supports structural descriptions, behavioral ASM descriptions, or a mixture of structural and behavioral descriptions. A structural description is usually a netlist of components (predefined in MDG–HDL) connected by signals. A behavioral description is given by a *tabular* representation of the transition/output relation or truth table. Currently, we are implementing a translator for a subset of VHDL to MDG–HDL.

Like ROBDDs, MDGs require a fixed order of node labels along all paths. This order is supplied by the user. Unlike ROBDDs where all variables are Boolean, in MDGs every variable/signal must be assigned an appropriate sort, and a type definition must be provided for all functions. If needed, rewrite rules may be used to partially interpret the otherwise uninterpreted function symbols.

Reachability Analysis. The reachability analysis for an ASM is based on a technique called *abstract implicit enumeration* [3] which is analogous to the implicit enumeration [2] used for FSMs. It verifies whether an invariant holds in all reachable states of the ASM.

The image computation is based on the RelP operator. It uses transition relation partitioning and early quantification heuristics. The special operator PbyS is used for multiple purposes: frontier-set simplification, detection of termination and invariant checking.

When the invariant is violated at some stage of the reachability analysis, a counterexample facility gives a sequence of input-state pairs leading from the initial state to the faulty behavior. This is very helpful for identifying design errors.

Applications. Invariant checking is the direct application of reachability analysis. A variation is the equivalence checking of two ASMs. We make a product machine for two ASMs by putting them together and feeding them the same inputs, then perform reachability analysis for the product machine and check an invariant stating the equivalence of the corresponding outputs.

In addition, we have a procedure for checking the combinational equivalence of circuits having the same input, output and state variables (if any). For each circuit, we derive MDGs relating each output and state variable to the inputs and state variables. These MDGs collectively represent the output and transition relations. Then, using the canonicity property of MDGs, we simply check that corresponding MDGs for the two circuits have the same MDG identifier.

Due to the use of many-sorted logic and uninterpreted function symbols, the specification and the implementation to be compared must be couched in terms of the same set of uninterpreted function symbols and sort assignments. This restriction, however, does not rule out using rewrite rules to exploit partial meaning of the uninterpreted function symbols.

The reachability analysis procedure may not terminate in general. However, for a class of interesting problems, the non-termination problem can be avoided by state generalization [3] and/or by using the term rewriting facility.

5 Conclusions and Future Work

We presented MDG tools which can reason at the abstract level and are thus suitable for RTL design verification. The contribution of MDGs as a representation and computation tool, beyond the use of abstract types, is that they open the way to the development of new techniques for the verification of circuit and system designs at higher levels of abstraction, making it possible to lift some of the ROBDD techniques that have been successful at the Boolean level.

We are currently in the course of developing a model checking algorithm for a restricted first-order temporal logic. This additional feature would allow us to perform the verification for temporal properties.

The MDG home page [6] contains a complete list of MDG references, including algorithms, case studies and the most recent work on the verification of an ATM switch fabric [5].

References

1. Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, August 1986.
2. J. R. Burch, E. M. Clarke, D. E. Long, K. L. McMillan, and D. L. Dill. Symbolic model checking for sequential circuit verification. *IEEE Transactions on Computer-Aided Design*, 13(4):401–424, April 1994.
3. F. Corella, Z. Zhou, X. Song, M. Langevin, and E. Cerny. Multiway decision graphs for automated hardware verification. To appear in the journal *Formal Methods in System Design*. Available as IBM technical report RC19676.
4. F. Corella, M. Langevin, E. Cerny, Z. Zhou and X. Song. State enumeration with abstract descriptions of state machines. In *Proc. of IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods (Charme'95)*. October, 1995, Frankfurt, Germany.
5. S. Tahar, Z. Zhou, X. Song, E. Cerny and M. Langevin. Formal Verification of an ATM Switch Fabric using Multiway Decision Graphs. In *IEEE Proc. of Sixth Great Lakes Symposium on VLSI*. Iowa, USA, March, 1996.
6. The home page of the multi-site MDG verification group: http://www.iro.umontreal.ca/labs/lasso/research/mdgverif/mdgverif_eng.html.

CADP
A Protocol Validation and Verification Toolbox*

Jean-Claude Fernandez, Hubert Garavel, Alain Kerbrat,
Laurent Mounier, Radu Mateescu, and Mihaela Sighireanu

Inria/Vérimag, Miniparc Zirst, rue Lavoisier, 38330 Montbonnot St-Martin, France

1 Introduction

CADP(CÆSAR/ALDÉBARAN Development Package) is a toolbox for protocol engineering. It offers a wide range of functionalities, ranging from interactive simulation to the most recent formal verification techniques. A first presentation of CADP can be found in [FGM+92]. Work on CADP started in 1985 and the first version of the toolbox (version A) was released in 1990. The latest official version (version Y) was released in May 1994. An improved version (version Z) is in preparation, for which beta-releases are available.

The CADP toolbox contains several closely interconnected components: ALDÉBARAN, BCG, CÆSAR, CÆSAR.ADT, OPEN/CÆSAR and XTL. All these components are accessible through a unified graphical user-interface developed in the EUCALYPTUS project. We first present the overall functionalities of the toolbox, followed by individual presentations of each component.

More recently, a prototype, named TGV (Test Generation using Verification techniques) [FJJV96], for the automatic generation of test suites has been developed within the CADP toolbox.

The CADP toolbox has been installed in 130 sites[2] and used for a number of case studies, e.g. [KB95, GM96], including several industrial applications, such as the verification of the bus arbiter of Bull's POWERSCALE™ architecture.

2 Description languages and compilers

The CADP toolbox accepts three different input formalisms:

- It accepts high-level protocol descriptions written in the ISO language LOTOS [International Standard 8807]. The toolbox contains two compilers CÆSAR and CÆSAR.ADT. They translate LOTOS descriptions into C code which can be used for simulation, verification and testing purpose.

* This work has been supported in part by the European Commission, under project ISC-CAN-65 "EUCALYPTUS-2: A European/Canadian LOTOS Protocol Tool Set".

[2] The toolbox is distributed free of charge to universities and academic research centers (under a license agreement). E-mail: caesar@imag.fr

- It accepts low-level protocol descriptions specified as Labelled Transition Systems (LTS, for short), i.e., finite state machines with transitions labelled by action names.
- As an intermediate step, the CADP toolbox accepts networks of communicating automata, i.e., finite state machines running in parallel and connected together using LOTOS parallel composition and hiding operators.

The latest releases of the CADP toolbox devote a growing importance to the concept of intermediate formats and programming interfaces, which allow the CADP tools to be applied to protocol description written in other languages than LOTOS (e.g., SDL with the GEODE compiler, etc.).

3 Validation and verification functionalities

The CADP toolbox allows to cover most of the development cycle of a protocol by offering an integrated set of functionalities. These functionalities (and tools) are interactive or random simulation (OPEN/CÆSAR), partial and exhaustive deadlock detection (OPEN/CÆSAR and ALDÉBARAN), test sequences generation (TGV), verification of behavioural specifications with respect to a bisimulation relation (ALDÉBARAN), verification of branching-time temporal logic specifications (EVALUATOR and XTL).

All the validation and verification tools are based on a same principle consisting in the exploration of an LTS describing the exhaustive behaviour of the protocol under analysis. This LTS can be accessed through several representations: *The explicit representation* consists in the exhaustive list of the states and transitions of the LTS. A compact format (BCG) is available to encode explicit representations efficiently. *The implicit representation* consists in a C library providing a set of functions allowing a dynamic exploration of the LTS. It is well adapted to perform "on the fly" verification, avoiding the generation of the whole LTS. *The symbolic representation* consists in a set of Binary Decision Diagrams (BDD) encoding the transition relation of the LTS. It can be built from program's description of higher level than the LTS level, thus allowing to take advantage of the BDD structure sharing capabilities.

4 Presentations of the toolbox components

1. ALDÉBARAN [FKM93] allows the comparison and the reduction of LTSs modulo various equivalence relations (such as strong bisimulation, observational equivalence, delay bisimulation, τ^*a bisimulation, branching bisimulation, and safety equivalence) and preorder relations (such as simulation preorder and safety preorder). The verification algorithms used in ALDÉBARAN are based either on the Paige-Tarjan algorithm for computing the relational coarsest partition, or on the "on-the-fly" techniques proposed by Fernandez-Mounier, or on symbolic LTS representation using Binary Decision Diagrams (BDDs). ALDÉBARAN has diagnosis capabilities that provide the user with explanations when two LTSs are found not to be related.

2. BCG (Binary-Coded Graphs) is both a format for the representation of explicit LTSs and a collection of libraries and programs dealing with this format. Compared to ASCII-based formats for LTSs, the BCG format uses a binary representation with compression techniques resulting in much smaller (up to 20 times) files. BCG is independent from any source language but keeps track of the objects (types, functions, variables) defined in the source programs. The following tools are currently available for this format: BCG_IO performs conversions between the BCG format and a dozen of other formats; BCG_OPEN establishes a gateway between the BCG format and the OPEN/CÆSAR environment; BCG_DRAW provides a 2-dimension graphical representation of BCG graphs with an automatic layout of states and transitions; BCG_EDIT is an interactive editor which allows to modify manually the display generated by BCG_DRAW.
3. CÆSAR [GS90] is a compiler which translates LOTOS descriptions into LTSs. CÆSAR proceeds in several steps, first translating the LOTOS description to compile into an intermediate Petri Net model, which provides a compact representation of the control and data flows. Then, the LTS is produced by performing reachability analysis on this Petri net. CÆSAR only handles LOTOS specifications with static control features, which is usually sufficient for most applications. The current version of CÆSAR allows the generation of large LTSs (some million states) within a reasonable lapse of time. The efficient compiling algorithms of CÆSAR can also be exploited in the framework of the OPEN/CÆSAR environment.
4. CÆSAR.ADT [Gar89] is a compiler that translates the data part of LOTOS descriptions into libraries of C types and functions. Each LOTOS sort or operation is translated into an equivalent C type or function. One must indicate to CÆSAR.ADT which LOTOS operations are "constructors" and which are not (fairly obvious, in practice). CÆSAR.ADT does not allow non-free constructors ("equations between constructors"). Translation of large programs (several hundreds of lines) is usually achieved in a few seconds. CÆSAR.ADT can be used in conjunction with CÆSAR, but it can also be used separately to compile and execute efficiently large abstract data types descriptions.
5. OPEN/CÆSAR is an extensible programming environment for the design of applications working with the implicit representation of LTSs. Currently, several languages/compilers are connected to the OPEN/CÆSAR environment, including: the CÆSAR and CÆSAR.ADT compilers, the BCG_OPEN gateway for explicit graphs, the EXP.OPEN gateway for networks of communicating automata, etc. Various application programs have already been written in the OPEN/CÆSAR framework, including two interactive simulators (with shell-like and X-window interfaces), a random execution tool, a deadlock detection tool based on G. Holzmann's technique, a reachability analysis tool (with τ^*a on-the-fly reduction), a sequence-searching tool,.an on-the-fly evaluator for branching-time μ-calculus, etc.

6. XTL (eXecutable Temporal Language) is a functional-like programming language designed to allow an easy, compact implementation of various temporal logic operators. These operators are evaluated over an LTS encoded in the BCG format. Besides the usual predefined types (booleans, integers, etc.) The XTL language defines special types, such as sets of states, transitions, and labels of the LTS. It offers primitives to access the informations contained in states and labels, to obtain the initial state, and to compute the successors and predecessors of states and transitions. The temporal operators can be easily implemented using these functions together with recursive user-defined functions working with sets of states and/or transitions of the LTS. A prototype compiler for XTL has been developed, and several temporal logics like HML, CTL, ACTL and LTAC have been easily implemented in XTL.

References

[FGM+92] Jean-Claude Fernandez, Hubert Garavel, Laurent Mounier, Anne Rasse, Carlos Rodríguez, and Joseph Sifakis. A Toolbox for the Verification of LOTOS Programs. In Lori A. Clarke, editor, *Proceedings of the 14th International Conference on Software Engineering ICSE'14 (Melbourne, Australia)*, pages 246–259, New-York, May 1992. ACM.

[FJJV96] J.Cl. Fernandez, C. Jard, T. Jéron, and C. Viho. Using on-the-fly verification techniques for the generation of test suites. In *this book*, 1996.

[FKM93] Jean-Claude Fernandez, Alain Kerbrat, and Laurent Mounier. Symbolic Equivalence Checking. In C. Courcoubetis, editor, *Proceedings of the 5th Workshop on Computer-Aided Verification (Heraklion, Greece)*, volume 697 of *Lecture Notes in Computer Science*, Berlin, June 1993. Springer Verlag.

[Gar89] Hubert Garavel. Compilation of LOTOS Abstract Data Types. In Son T. Vuong, editor, *Proceedings of the 2nd International Conference on Formal Description Techniques FORTE'89 (Vancouver B.C., Canada)*, pages 147–162, Amsterdam, December 1989. North-Holland.

[GM96] Hubert Garavel and Laurent Mounier. Specification and Verification of various Distributed Leader Election Algorithms for Unidirectional Ring Networks. *Science of Computer Programming*, 1996. To appear.

[GS90] Hubert Garavel and Joseph Sifakis. Compilation and Verification of LOTOS Specifications. In L. Logrippo, R. L. Probert, and H. Ural, editors, *Proceedings of the 10th International Symposium on Protocol Specification, Testing and Verification (Ottawa, Canada)*, pages 379–394, Amsterdam, June 1990. IFIP, North-Holland.

[KB95] Alain Kerbrat and Slim Ben Atallah. Formal Specification of a Framework for Groupware Development. In G. v. Bochmann, R. Dssouli, and O. Rafiq, editors, *Proceedings of the 8th International Conference on Formal Description Techniques for Distributed Systems and Communication Protocol FORTE'95 (Montreal, Quebec, Canada)*, October 1995. Short paper.

The FC2TOOLS Set

Amar Bouali[1] and Annie Ressouche[2] and Valérie Roy[1] and Robert de Simone[2]

[1] CMA/Ecole des Mines de Paris, B.P. 207, F-06904 Sophia-Antipolis cedex
[2] INRIA , B.P. 93, F-06902 Sophia-Antipolis cedex

1 Presentation

The AUTO/GRAPH toolset [4] developed in our group was one of the pioneering softwares in the field of analysis and verification of networks of communicating processes. We describe here the next-generation AUTO/GRAPH, consisting of a modular tool suite interfaced around a common file description format named FC2. The format allows representation of single reactive automata as well as combining networks. This format was developed in the scope of Esprit BRA project 7166:CONCUR2 [2].

Of uttermost interest in the new implementation is that most analysis functions are implemented with redundancy using both *explicit* classical representation of automata, and also *implicit* state space symbolic representation using *Binary Decision Diagrams*. The two alternative techniques are shown to offer drastically different performances in different cases, with low predictibility. Then offering both kinds of implementation in a unified framework is a valuable thing in our view.

Both FC2EXPLICIT and FC2IMPLICIT commands perform synchronised product and reachable state space search. They can minimize results w.r.t. *strong, weak, branching* bisimulation notions, and produce the result as an FC2 automaton. They can also *abstract* the system with a notion of "abstract actions", each synthesizing a set of sequences of concrete behaviours (in this sense behavioural abstraction can be seen as reverse from refinement). In addition FC2IMPLICIT has a fast checker for *deadlocks, livelock* or *divergent* states, for which it produces counterexample paths in case of existence, while FC2EXPLICIT allows *compositional* reduction techniques, mostly in case of "observational" bisimulation minimisations.

We are currently extending these features of FC2IMPLICIT so that labeled predicates on states, hiding of behaviours irrelevant to specific analysis, and use of side observer automata would allow to check in practice for much wider types of properties, while keeping with the same algorithmic kernel, and with the renewed aim of *not* introducing an heterogeneous formalism for expression of correctness properties, like temporal logics or μ-calculus.

The tool suite is completed by the graphical editor AUTOGRAPH, which allows for graphical depiction of automata and networks as well as source recollection of counterexample paths back up to the original graphical network; the FC2LINK preprocessor, which merge multifile descriptions of hierarchical networks into a single file for later analysis and verification; the FC2VIEW postprocessor for

source recollection and display of counterexample paths, this time back up to the distributed FC2 files.

Further information on the toolset and its availability can be obtained from the WWW page `http://cma.cma.fr/Verification/verif-eng.html`.

2 The Tool Set

2.1 AUTOGRAPH

Renamed ATG for its new C++ implementation, AUTOGRAPH is a graphical display system for both labeled transition graphs and networks of communicating systems, in the tradition of process algebra graphical depiction. Objects in AUTOGRAPH can also be extensively annotated as allowed by the FC2 format standards. Figure 1 provides a (trivial) example of 3 dining philosophers drawn in AUTOGRAPH.

AUTOGRAPH can be used for edition, but also to visualise automata that were produced elsewhere, typically as an output of verification. Human guidance is then required for lay-out.

2.2 FC2EXPLICIT

This tool performs the following functions, on explicit representations of automata:

Global Automaton Generation. Straightforward.

Compositional Reductions. FC2EXPLICIT can perform automata minimisation with respect to *strong*, *weak* or *branching* bisimulation. When invoked on a network, the hierarchical model construction can be alternated with such reduction steps at intermediate stages. Traditional *Relational Coarsest Partitioning Algorithm* [3] is used to refine a state partition until fix-point.

Comparison with automata specifications. The equivalence checking problem is solved on the disjoint union of the two state spaces, by partitioning them as a whole. The main algorithmic improvement is that negative answer is produced as soon as a class contains no further state from one of the automata, which happens usually quite soon (if ever), long before reaching actual fix-point. Then a path leading to an arbitrary state without match from the other network is provided as counterexample. It can be visualised using AUTOGRAPH or FC2VIEW.

Model Abstraction. *Abstract Actions* allow us to define the atomicity level at which we want to observe an automaton. The idea is to consider terminated sequences of concrete behaviours as atomic and to call such a set abstract action. Reducing a global system w.r.t. a set of abstract actions results in a system conceptually simpler, where meaningful activities have been extracted. As a simple subcase, a single abstract action indicating improper (rational) behaviour can be presented for refutation; this implements language inclusion (in the complement language of the abstract action), and is noticed in FC2EXPLICIT by the

absence of transition in the resulting automaton.Example: `Abstract_Wrong = tau*.enter1.tau*.enter2` can specify lack of mutual exclusion property between two processes.

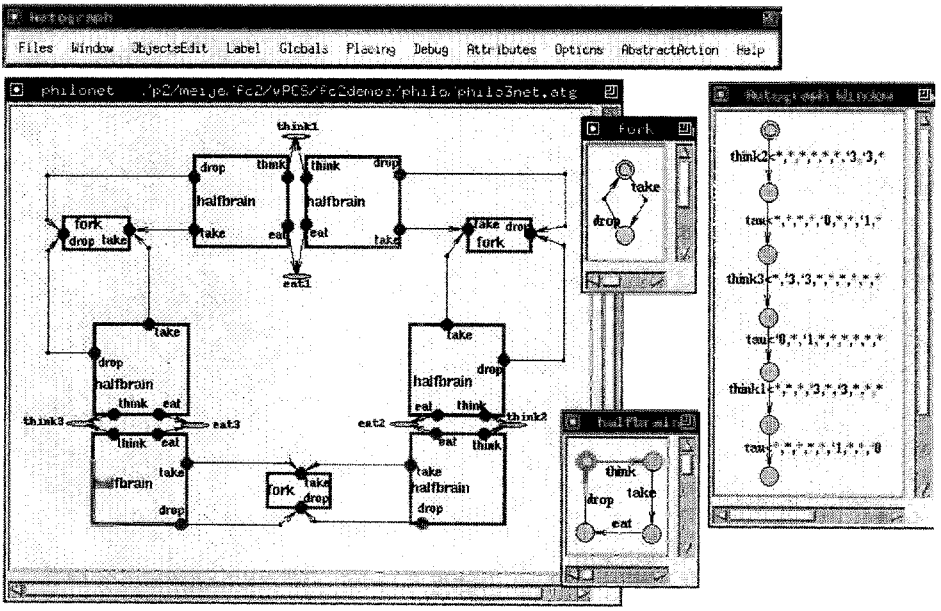


Fig. 1. The 3 dining philosophers specification

2.3 FC2IMPLICIT

performs the following functions, on BDD/implicit representation of states, using the TIGER BDD library:

Global Automata Generation. The reachable state space is of course evaluated in a breadth-first search strategy, applying event synchronisation vectors iteratively until fix-point, staring from initial state. Computation of reachable state computation can be refined to allow for on-line deadlock detection, and followed by livelock or divergent states detection on the result (a divergent state may perform infinite sequences of hidden "tau" actions, a livelock state can exhibit *only* such behaviour). When such an undesirable state is found, a counterexample path can be produced, and mapped by AUTOGRAPH or FC2VIEW back to the original network description. Figure 1 displays a deadlock path for the 3 philosophers (but *not* its distributed mapping).

Bisimulation Minimisation. Symbolic computation of bisimulation classes can also be applied from this BDD description of reachable states. following results from [1]. The (explicit) resulting minimal automaton can be produced in FC2 format on demand.

Bisimulation Checking. First a synchronised product of the two distinct networks is built. Then the same bisimulation partition as before is used, with the algorithmic improvement that it is aborted whenever a class contains no state from one side. Then a counterexample path can be produced, and mapped by AUTOGRAPH or FC2VIEW back to the original network description.

Observers and Annotated Global States.. A great deal of practical verification is usually conducted by compiling the property to establish into an automaton-like structure to act on the side of the observed process, with possibly additional annotations on states and transitions of various sorts (*success*, *failure* or *recur* states, *don't care* transitions,...). Verification then starts by constructing a synchronised product of the (usually large) network state space with the (usually smaller) state space of the observer structure. Such observers can actually be represented without additional theory in the same FC2 format as processes, and particular sets of states and transitions are just used to restrict (or introduce new) relations in algorithms. We are still working on "easy" description of such sets, so as to "hide' as much as possible intricate temporal logic formulation from the non-expert user.

3 Example

We just illustrate the basic verification features on our simple *dining philosophers* problem from figure 1.

We now suppose these three parts (the fork, halfbrain automata and the network) have been independently translated (by ATG) into distinct FC2 files, and then linked together by FC2LINK into file `philo3.fc2`.

We use symbolic methods based on BDDs for an easy evaluation of global state spaces and deadlock checking.

```
0-duick$ fc2implicit -dead -fc2 philo3.fc2 > deadpath.fc2
--- fc2implicit: Making reachable state space
--- fc2implicit: State space depth: 13
--- fc2implicit: First deadlock(s) detected at depth 7
--- fc2implicit: Reachable states: <<214>> -- BDD nodes: <<85>>
--- fc2implicit: Global automaton has 2 DEADLOCKS state(s) -- BDD nodes: <<27>>
0-duick$
```

The first detected deadlocks were found at depth 7. With the option `-fc2`, a diagnostic path was extracted into `deadpath.fc2`. It is displayed in AUTOGRAPH on the left of figure 1. Now clicking appropriately on states or transitions there would highlight contributing locations on the original graphical network.

The following session shows time figures for larger state spaces These were computed on a Sparc10 Workstation with 64 Mb memory. Timing include synthesis of diagnostic paths or production of automata onto files. The full-size automaton for 6 philosophers would be 46654 states, while compositional reductions deals with intermediate constructs of at most 1512 states.

```
philo$ time fc2implicit -dead -fc2 philo8 > deadpath8.fc2
--- fc2implicit: Reachable states: <<1679614>> -- BDD nodes: <<245>>
--- fc2implicit: Global automaton has 2 DEADLOCKS state(s) -- BDD nodes: <<77>>
--- fc2implicit: First deadlock detected at depth 16
real    1m52.21s user   1m39.08s sys    0m0.46s

philo$ time fc2explicit -comp -w obs_philo6_rec > opr6_eW.fc2
--- fc2min: Automaton has 728 states
real    3m4.85s user    2m2.30s sys     0m11.55s
```

Larger experiments were conducted, showing the possibilities of the tools. For the next future we are concentrating in replacing sequential automata components by *synchronous reactive processes* (such as produced by the ESTEREL language for instance), to be able to deal with asynchronous networks of synchronous processes. We have promising initial results in this direction, mainly from the fact that both domains allow partition of the transition relation for simpler symbolic application on implicit state spaces.

References

1. A. Bouali and R. de Simone. Symbolic bisimulation minimisation. In *Fourth Workshop on Computer-Aided Verification*, volume 663 of *LNCS*, pages 96–108, Montreal, 1992. Springer-Verlag.
2. A. Bouali, A. Ressouche, V. Roy, and R. de Simone. The FCTOOLS user manual. In *final ESPRIT-BRA CONCUR2 deliverables*, Amsterdam, October 1995. Available by ftp from `cma.cma.fr:pub/verif` as file *fc2userman.ps*.
3. P.C. Kanellakis and S.A. Smolka. CCS expressions, finite state processes, and three problems of equivalence. *Information and Computation*, 86:43–68, 1990.
4. V. Roy and R. de Simone. AUTO and autograph. In R. Kurshan, editor, *proceedings of Workshop on Computer Aided Verification*, New-Brunswick, June 1990. AMS-DIMACS.

The Real-Time Graphical Interval Logic Toolset

L. E. Moser, P. M. Melliar-Smith, Y. S. Ramakrishna, G. Kutty, L. K. Dillon

Department of Electrical and Computer Engineering
Department of Computer Science
University of California, Santa Barbara 93106

1 Introduction

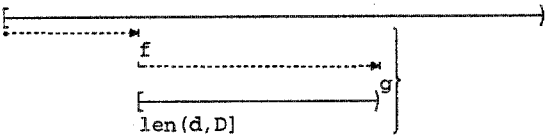
The tools that we have developed for Real-Time Graphical Interval Logic (RTGIL) are intended for specifying and reasoning about time-bounded safety and liveness properties of concurrent real-time systems. These tools include a syntax-directed editor that enables the user to construct graphical formulas on a workstation display, a theorem prover based on a decision procedure that checks the validity of attempted proofs and produces a counterexample if an attempted proof is invalid, and a proof management and database system that tracks proof dependencies and allows graphical formulas to be stored and retrieved.

2 Real-Time Graphical Interval Logic

RTGIL is a linear-time temporal logic in which formulas are interpreted on traces of states indexed by the non-negative real numbers. To exclude the occurrence of instantaneous states and Zeno runs, these traces are required to be right continuous and finitely variable. Right continuity requires that each primitive proposition holds its value for a non-zero duration, while finite variability ensures that there are only a finite number of state changes in any finite duration.

The key construct of RTGIL is the interval, which provides a context within which properties are asserted to hold. An interval is defined by two search patterns, which locate its left and right endpoints. A search pattern is a sequence of one or more searches. Each search locates the first state at which its target formula holds. The state located by one search is the state at which the next search begins. An interval is half-open in that it begins with the state located by the first of its two search patterns and extends up to but does not include the state located by the second search pattern. Once an interval is defined, properties can be asserted to hold on the interval, including initial, henceforth, and eventuality properties. Most importantly, real-time bounds on the duration of an interval can be specified.

For example, in the following RTGIL formula,



the interval begins with the first state at which the formula f holds and ends just prior to the next state at which the formula g holds. The duration of that interval is asserted to be greater than d time units and less than or equal to D time units.

This research was supported in part by NSF/ARPA grant CCR-9014382.

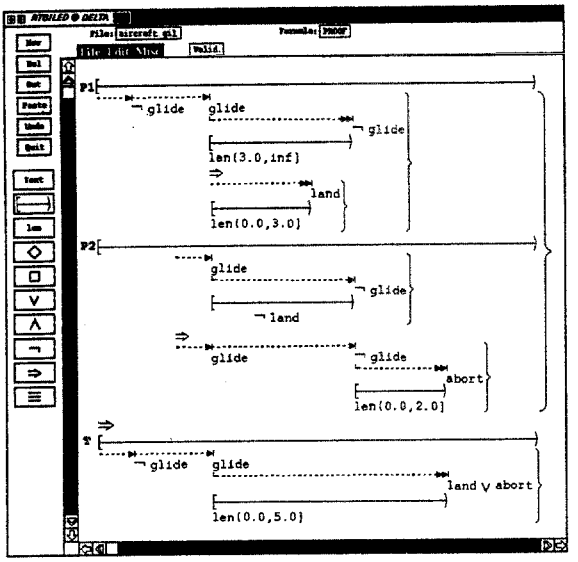


Fig. 1. The graphical user interface with a validated proof. The premises $P1$ and $P2$ are conjoined, represented by vertical composition, and imply the theorem T. Premise $P1$ requires that if the aircraft is in the glide path for more than 3.0 secs, then it will land within 3.0 secs of entering the glide path. Premise $P2$ requires that if the aircraft does not land before leaving the glide path, then it will abort within 2.0 secs of leaving the glide path. Theorem T states that within 5.0 secs of entering the glide path, the aircraft will either land or abort.

Formulas in RTGIL are read from top to bottom and from left to right, starting with the topmost interval. Formulas can be combined using standard logical infix operators laid out vertically. In vertical layout, a conjunction is indicated by stacking the formulas one below the other without the conjunction operator. Braces are used to disambiguate formulas.

3 The Graphical Editor

The graphical user interface to the RTGIL editor is shown in Fig. 1. The editor provides high-level editing operations and supplies templates containing boxes for formulas that enable the user to construct graphical formulas incrementally. The mouse enables the user to select a box or formula on the display and to highlight it.

The pull-down menus (File, Edit, Misc) at the top of the display contain commands for storing and retrieving formulas, for overriding the default layout of formulas, and for invoking the theorem prover. The buttons on the upper left (New, Del, Cut, Paste, etc) provide editing operations that enable the user to create a new formula, delete a selected formula, store a selected formula in a buffer, and subsequently insert that formula in a selected box. The buttons on the lower left (Text, $\vdash\!\!\!\!-\!\!\!\!-$), len, etc) enable the user to select an appropriate RTGIL construct to apply to the currently highlighted subformula. Scroll bars allow the user to view large formulas.

The editor provides capabilities for automatically replacing formulas with other formulas, resizing formulas to suit the context length, etc. If a formula does not fit into the allotted space, an error is indicated by highlighting the formula. The user can then resize the context length or the search arrows to allow the formula to be drawn correctly. All subformulas of the formula are automatically resized to scale.

The editor also enables the user to align corresponding points in the formulas that comprise a proof. The user can thus see how states in different formulas are ordered relative to one another, how intervals are aligned relative to each other, and how durations of intervals are related to satisfy real-time constraints. Alignment is helpful in constructing proofs and in debugging attempted proofs that are invalid.

4 The Theorem Prover

The RTGIL theorem prover is a satisfiability checker based on a decision procedure, rather than a Gentzen-style theorem prover based on inference rules. The decision procedure for RTGIL is given as an automata-theoretic method in [4]. The implementation, however, is a tableau-theoretic method that achieves better time and space efficiency, on average, than the automata-theoretic method. It employs the notion of timed tableau, the analogue of the timed automaton of Alur and Dill [1].

The user, working in the theory defined by his specifications and the underlying logic, creates theorems and proofs and submits the proofs to the decision procedure for validation. To prove a theorem T, the user selects a subset of the axioms and previously proved lemmas and theorems as the premises $P1, \ldots, PN$ of the proof. The editor displays the proof represented by the formula $P1 \wedge \ldots \wedge PN \Rightarrow T$ in its graphical form. The graphical representation is converted into a Lisp S-expression, is negated, and is then submitted to the decision procedure.

The decision procedure checks the satisfiability of the negated implication by building a tableau for that formula and checking the emptiness of the tableau. The procedure first constructs an untimed tableau for the formula and performs the standard eventuality-based pruning of the tableau. Using the duration formulas in the nodes of the remaining tableau, it then constructs a timed tableau by adding timing constraints to the edges of the untimed tableau. Timing consistency of the timed tableau is checked using Dill's algorithm [2]. This step may eliminate some possible traces from the original tableau because of timing restrictions and, consequently, a further round of eventuality-based pruning is required. If, at any stage, the tableau becomes empty or the initial node is eliminated, the negated implication is unsatisfiable and the attempted proof is valid. Otherwise, there exists a timing consistent trace through the final tableau that constitutes a counterexample to the attempted proof.

If the decision procedure determines that an attempted proof is invalid, the user can invoke the theorem prover to produce a counterexample by extracting a satisfying model for the negated implication from the tableau. The counterexample is displayed in an accompanying window, shown in Fig. 2, as a sequence of states and, additionally, as a timing diagram if the user selects that option. By associating the targets of the searches in the formulas of the proof with the states in the sequence at which the predicates become true or false, or the points in the timing diagram at which the signals rise and fall, the user can more readily discover the fallacy in the attempted proof and correct it.

The worst-case time complexity of the decision procedure is $2^{O(n^{2k} \cdot k \cdot \log n + t \cdot \log t)}$, where n is the number of logical connectives, k is the depth of interval nesting, and t is the size of the binary encoding of the largest duration constant in the formula.

5 The Proof Management and Database System

RTGIL formulas saved to disk are stored in a simple database consisting of Unix files. Several formulas can be stored in the same file by associating a unique name with each of them. The user can invoke the editor to display the names of the formulas in a file and also to load, add or delete a formula to or from a file. For each formula in a file, the user can invoke the proof manager to determine if a proof already exists, and to list the premises of an existing proof.

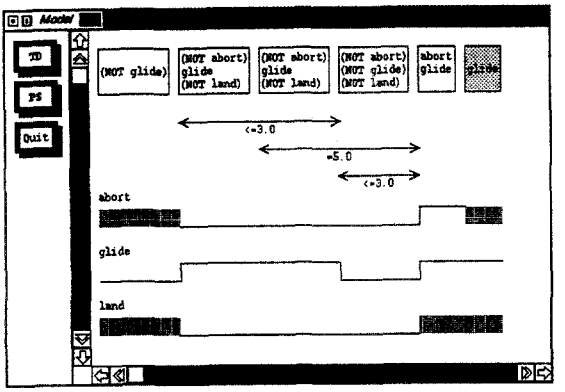


Fig. 2. The graphical user interface with a counterexample model. The proof in Fig. 1 is modified so that the upper bound of 2.0 secs in premise $P2$ is replaced by 3.0 secs. The attempted proof is invalid, and this counterexample is generated. Note that the interval from *glide* to $\neg$*glide* without an intervening *land* is at most 3.0 secs and the interval from $\neg$*glide* to *abort* is at most 3.0 secs. Thus, the interval from *glide* to *land* $\vee$ *abort* is at most 6.0 secs, which is greater than the 5.0 secs in theorem T.

If an attempted proof of a theorem is valid, the proof dependency file is updated with information about the premises of the proof and the time at which the proof was performed. To confirm that a proof is up-to-date, the proof manager checks that neither the theorem nor any of the premises has been modified since the time of the proof. It also detects circularities in a proof and ensures that the proof dependency graph is acyclic.

6 Conclusion

Our experience in using the RTGIL tools has shown that these tools and the graphical representation of the logic are very helpful for specifying and verifying properties of concurrent real-time systems. In addition to the aircraft example, we have used these tools to specify and verify properties of a railroad crossing system, a robot, an alarm system, and a four-phase handshaking protocol.

The RTGIL tools are implemented in Lucid Common Lisp and also in Franz Allegro Common Lisp, and require at least 32 MBytes of main memory and 64 Mbytes of swap space. The graphical editor was implemented using the Garnet graphics toolkit [3], which runs within the X window system. The RTGIL tools and related papers are publicly available, and can be obtained by anonymous ftp from alpha.ece.ucsb.edu in directory /pub/RTGIL.

References

1. R. Alur and D. Dill, "Automata for modelling real-time systems," *Proceedings of 17th International Conference on Automata Languages and Programming*, Warwick University, England (July 1990), LNCS 443, Springer-Verlag, pp. 322-335.
2. D. L. Dill, "Timing assumptions and verification of finite-state concurrent systems," *Proceedings of International Workshop on Automatic Verification Methods for Finite State Systems*, Grenoble, France (June 1989), LNCS 407, Springer-Verlag, pp. 196-212.
3. B. A. Myers, D. A. Giuse, R. B. Danneberg, B. VanderZanden, D. S. Kosbie, E. Pervin, A. Mickish and P. Marchal, "Garnet: Comprehensive support for graphical, highly interactive user interfaces," *IEEE Computer* (November 1990), pp. 71-85.
4. Y. S. Ramakrishna, L. K. Dillon, L. E. Moser, P. M. Melliar-Smith and G. Kutty, "A real-time interval logic and its decision procedure," *Proceedings of Thirteenth Conference on Foundations of Software Technology and Theoretical Computer Science*, Bombay, India (December 1993), LNCS 761, Springer-Verlag, pp. 173-192.

The METAFrame'95 Environment

Bernhard Steffen, Tiziana Margaria, Andreas Claßen, Volker Braun

Lehrstuhl für Programmiersysteme
Universität Passau
D-94030 Passau (Germany)
{steffen,tiziana,classen,deepthou}@fmi.uni-passau.de

1 The METAFrameEnvironment

METAFrame is a *meta-level framework* designed to offer a sophisticated support for the systematic and structured computer aided generation of application-specific complex objects from collections of reusable components. Figure 1 shows its overall organization. Special care has been taken in the design of an adequate, almost natural-language specification language, of a user-friendly graphical interface, of a hypertext based navigation tool, and a of semi-automatic synthesis process and repository management. This application-independent core is complemented by application-specific libraries of components, which constitute the objects of the synthesis. The principle of separating the component implementation from its description is systematically enforced: for each application we have a distinct Meta-Data repository containing a logic view of the components. The tools themselves and their documentation are available in a different repository. This organization offers a maximum of flexibility since the synthesis core is independent of the direct physical availability of the tools (except for the execution, which is a different matter).

METAFrame constitutes a sophisticated programming environment for large to huge grain programs whose implementation is supported by the automatic, library-based synthesis of linear compositions of modules. More complex control structures glueing the linear portions together must be programmed by hand. Still, being able to synthesize linear program fragments drastically improves over other methods where only single components can be retrieved from the underlying repository. This is already true in cases where one is only interested in the functionality of single components, because our synthesis algorithm will automatically determine the required interfacing modules. Thus METAFrame supports the rapid and reliable realization of efficient application specific complex systems without sophisticated user interaction, making it an ideal means for a systematic investigation and construction of adequate implementations in a problem specific scenario.

The METAFrame approach abstracts from implementational details by allowing designers a *high-level-development* of the tools. *Specifications* express constraints in a *temporal logic* that uniformly and elegantly captures an abstract view of the repository. *Implementations* are in a *high-level language* tailored to express the combination of reusable analysis, verification and transformation components stored in the repository, which are considered as atomic on this level.

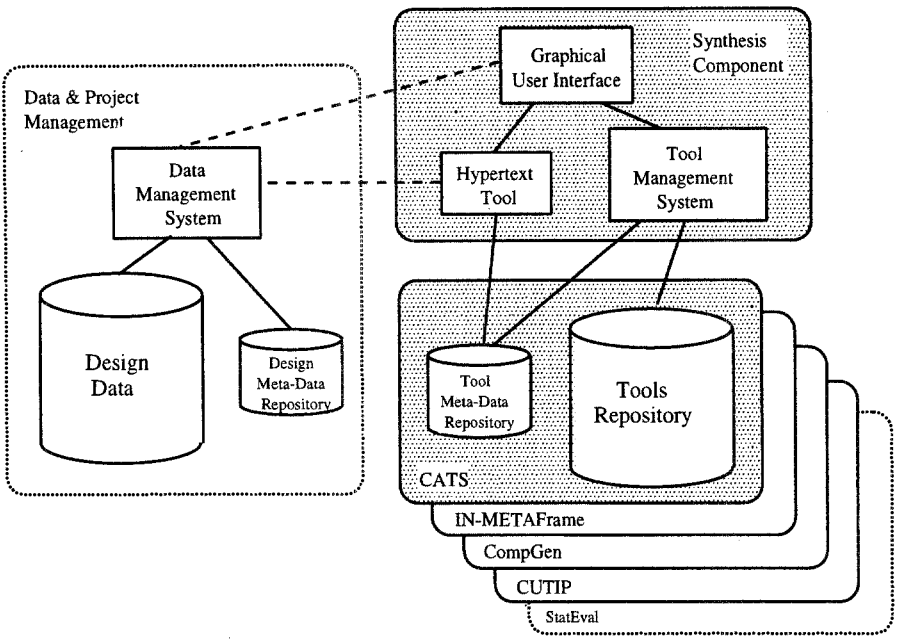


Fig. 1. The METAFrame Global Architecture

Synthesized systems are hierarchically structured. Since only the meta-structure, combining such components, is automatically generated from the specifications, the efficiency of the resulting system depends on the efficiency of the single components. In particular, different components may be written in different application languages (C, ML, C++) of different programming paradigms (imperative, functional, object-oriented).

METAFrame is currently available for UNIX systems, e.g. the Siemens Nixdorf RM line, and for PCs under LINUX. Its graphical interface as well as the hypertext browser are built on top of the Tcl/Tk graphics library [5]. Target language is the HLL, whose interpreter is implemented in C++.

2 Applications

The current experience has shown the practicability and flexibility of the METAFrame approach in several areas of application, which we briefly sketched in the following.

CATS. This application concerns the **C***omputer-***A***ided* **T***ool* **S***ynthesis* via automatic composition of heterogeneous verification algorithms for hardware and software systems from basic transformation and analysis components [4, 10]. It

supports the automatic generation of complex tools from a repository of components including equivalence checkers, model checkers, theorem provers, and a variety of decision procedures for application-specific problems. The number of basic components and infrastructure modules is steadily growing, as e.g. most of the tools come in many variants coping with infinite state-systems, values, time (discrete or dense), priorities, probabilities and combinations thereof, which all require their specific extensions of the 'basic' data structures.

IN-METAFrame. This project concerns the semi-automatic programming of *Intelligent Network Services* in cooperation with Siemens Nixdorf, Munich [7, 8]. IN-METAFrame is a development (creation) environment for reliable custom configurations of telephone services from a library of existing service-independent modules. The creation process is structured as follows: Initially, an existing service of similar application profile is loaded from the service library, or a completely new design from scratch is done (under model checking control). Alternatively, initial prototypes could automatically be generated from the set of underlying consistency conditions and constraints, a feature, which is not part of the current version of our service creation environment. These prototypes serve as a starting point for modifications that eventually lead to the intended service. Modification is guided by abstract views, and controlled by on-line verification of certain consistency constraints, guaranteeing the executability and testability of the current prototype.

CompGen. One of the major problems in data flow analysis or program optimization is the determination of the correct ordering of the individual analyses and optimizing transformations. Using METAFrame we can automatically syntesize complex heterogeneous optimization tools that respect certain important ordering constraints, guaranteeing correctness and often optimality of the overall algorithms [2, 3]. Moreover, in cases where there is not yet consensus about sensible orderings, METAFrame supports the corresponding investigation through its rapid prototyping facility. One should note here that the number of individual algorithms, many of which can be automatically generated (see[9]), may, like in the CATS-case, grow very large.

3 Evaluation and Perspectives

Our approach exactly meets the demands recently expressed by Goguen and Luqi in [1] for the emerging paradigm of *Domain Specific Formal Methods*: the essence of their proposal is to use formal methods on a large or huge grain level rather than on elementary statements, thus to support the programming with whole subroutines and modules as the elementary building blocks. This is precisely what METAFrame is designed for (see [6]).

Moreover, the possibility of natural language-like specification, the hypertext support of repository navigation, the specific profiles of the stored tools, and the user friendly graphical interface encourage successful experimentation without

requiring much expertise. In fact, one can use METAFrame also as an educational tool to train newcomers in an application field[1].

In the meantime, applications of METAFrame have been presented at various international fairs, like the CeBit'96 in Hannover, and, more interestingly, by Siemens Nixdorf with success at the TELECOM'95 in Geneva. GA*IN* (Global Advanced Intelligent Network), the IN solution jointly developed by Siemens and Siemens Nixdorf which includes IN-METAFrame, reached the market early this year and it has already been delivered to a number of customers, like e.g. Deutsche Telekom.

References

1. J.A. Goguen, Luqi: "*Formal Methods and Social Context in Software Development*," (inv. talk) TAPSOFT'95, Aarhus (DK), May 1995, LNCS N.915, pp.62-81.
2. M. Klein, J. Knoop, D. Koschützki, B. Steffen: "DFA&OPT-METAFrame*: A Tool Kit for Program Analysis and Optimization*" (Tool description) - Proc. TACAS'96, Int. Workshop on Tools and Algorithms for the Construction and Analysis of Systems, Passau, March 1996, LNCS 1055, Springer Verlag, pp. 422-26.
3. J. Knoop, O. Rüthing, B. Steffen: '*The power of assignment motion*," Proc. PLDI'95, ACM SIGPLAN, La Jolla, CA, June'95, SIGPLAN Notices 30, 6 (1995), pp.233-245.
4. T. Margaria, A. Claßen, B. Steffen: *"Computer Aided Tool Synthesis in the* METAFrame *"*, 3. GI/ITG Workshop on "Anwendung formaler Methoden beim Entwurf von Hardwaresystemen", Passau (D), March 1995, pp. 11-20, Shaker Verlag.
5. John K. Ousterhout: "*Tcl and the Tk Toolkit*," Addison–Wesley, April 1994.
6. B. Steffen, T. Margaria, A. Claßen, V. Braun: "*Incremental Formalization: a Key to Industrial Success* ", to appear in "SOFTWARE: Concepts and Tools", Springer Verlag, 1996.
7. B. Steffen, T. Margaria, A. Claßen, V. Braun, M. Reitenspieß: "*An Environment for the Creation of Intelligent Network Services*", invited contribution to the book "Intelligent Networks: IN/AIN Technologies, Operations, Services, and Applications – A Comprehensive Report" Int. Engineering Consortium, Chicago IL, 1996, pp. 287-300 – also invited to the *Annual Review of Communications*, IEC, 1996.
8. B. Steffen, T. Margaria, A. Claßen, V. Braun, V. Kriete, M. Reitenspieß: "*A Constraint-Oriented Service Creation Environment*" (Tool description) - Proc. TACAS'96, Int. Workshop on Tools and Algorithms for the Construction and Analysis of Systems, Passau, March 1996, LNCS 1055, Springer Verlag, pp. 418-421.
9. B. Steffen: "*Generating Data Flow Analysis Algorithms from Modal Specifications*," Science of Computer Programming N.21, 1993, pp.115-139.
10. B. Steffen, T. Margaria, A. Claßen: "*Heterogeneous Analysis and Verification for Distributed Systems*," "SOFTWARE: Concepts and Tools" N. 17, pp. 13-25, March 1996, Springer Verlag.

[1] At the University of Aarhus (Denmark) as well as in Passau it is used as a support in regular courses covering software techniques and formal methods in system development.

Verification Support Environment

Frank Koob, Markus Ullmann, Stefan Wittmann

Bundesamt fuer Sicherheit in der Informationstechnik
Godesberger Allee 183
D-53133 Bonn, Germany
Tel.: x49-228-9582154
Fax.: x49-228-9582455
Email: vse@bsi.de

Abstract. Formal methods are recognized as the most promising way to produce high assurance software systems. In reality this fact is not enough to convince industry to use them. Formal methods must be applicable and usable in several areas (security, safety), engineers have to accept a change in software development work but should not be asked to give up the environment they are used to and bosses must realize that higher effort during the design phase can save money and time later. This paper describes the recently completed formal specification and verification tool Verification Support Environment (VSE). An advantage of the design of the VSE tool is the possibility of using formal and semiformal development methods combined in a unique working environment. After official release of the VSE-system March 1995 several pilot projects were carried out with industry. The paper gives an overview of the VSE-system and describes the results of the pilot applications.

1 General

This paper briefly describes the functionality of the Verification Support Environmet System (VSE), its integration into the high assurance development process and an outline of the result of an industrial pilot project recently carried out.

In 1991, the Bundesamt fuer Sicherheit in der Informationstechnik (German Information Security Agency) initiated a CASE tool project with emphasis on formal specification and verification. Now, after four years of work, the Verification Support Environment (VSE) Tool has attained a status that permits industrial application. This tool combines the two traditions of semiformal and formal methods within a unique framework. Therefore the benefit of the VSE tool is both to support migration to formal methods where needed and, where it is sufficient, to rely on semiformal methods ([1]).

2 Components of VSE

1. specification language VSE-SL
   - formal language describing abstract data types and abstract state transiton machines, using terms of First-Order Predicate Logic with Equality and Dynamic Logic
   - syntax controlled editor, consistency and type checking
2. graphical interactive development presentation
3. import/export functions
4. dual interactive verifier subsystems
   - Dynamic Logic: KIV (Karlsruhe Interactive Verifier)
   - Predicate Logic: INKA (INduction prover KArlsruhe)
   - predifined proof strategies, tactics, heuristics
5. verification management system and database (multi-use capability)
6. standard design interface (ODS) to conventional CASE-tools

3 Functionality

The VSE-system supports the software development process from analysis to code generation. During analysis it has to be determined which parts of the future system are security (safety) critical. The non-critical parts can be developed conventionally within the regular production environment. For the critical parts VSE provides a specification language (VSE-SL) to structure them in a way to support later proof activities. The top level specification formally describes the functionality on an abstract level. The security (safety) model defines characteristics of the objects that have to be fulfilled. With means of refinement the top level specification is modified stepwise to abstract programs and, using the code generator, ADA sourcecode.

The VSE prover (verifier) subsystem automatically generates proof obligations out of the specification and the refinement process.

- The top level specification fulfills the security (safety) model.
- The entire refinement process guarantees that the generated code has the same functionality as the top level specification.

All proof obligations have to be verified in order to be able to call the source code correct in accordance to specification and security (safety) model. The functionality of the deduction subsystems includes the following features ([2] and [3]):

- semi-automatic switch between Dynamic Logic (KIV) and Predicate Logic (INKA)

- provision of proof tactics and empirically predifined heuristics running automatically
- provision of pre-selected deduction rules applicable to the current proof goal in an interactive mode
- proof protocol (text and graphics), replay mode, restart at any proof step

4 Applications

Two major case studies (disposition control and system and nuclear power plant access control system) were carried out within the VSE-project to show that VSE is applicable even on large projects (4100 verified lines of code, 20000 proof steps, average automation rate 80 was used by eight industry companies and government institutions from Germany and Italy for pilot applications dealing with traffic control (railway), space flight, smart cards, hospital administration and secure message handling. The pilot application presented is a drug administration component as a part of a hospital administration system. This part is based on SEMA.

The product Health System 2000 (HS2000) was developed by SEMA Group Ismaning, Germany. It is a hospital information system to administrate patient files, medical means and accounting of a hospital. The subject of the pilot project was an additional part to the existing HS2000 to control and check access to and applicability of drugs in a hospital pharmacy. The security-critical kernel was to be developed with the VSE-system. From April 20, 1995 until June 1, 1995 one expert from the University of Ulm and one company representative worked on the project. The task of the VSE-expert was to give a basic training in the VSE development method and the VSE-tool and to support the company representative only as much as needed.

5 Conclusion

The pilot projects reached almost all the goals that had been defined before the start. Concerning the size of the problems and time/personal constraints the results were surprising. This pilot project showed that formal methods are applicable in industrial environments and they significantly improve the quality of software systems. Facts like less misinterpretations and incompatibilities in the requirements and the reusability of verified components strengthen the trustworthiness of the systems developed with the formal VSE method. Besides that re-specifications helped to detect errors in existing systems and showed the limitations of conventional software development.

Nevertheless six weeks of work with formal methods and the VSE-tool are not enough to be prepared for independent formal development. There is still a need for intensive training and support of experts to handle the specification language and the verifier subsystems. But the cooperation of software engineers and VSE-specialists turned out to be a very promising way to introduce formal

methods to industry and to transfer the new technology. New fields of industrial services like formal design or proof engineering do not belong to science fiction.

All partners realized that formal methods solve a lot of problems in the area of critical software development. A mathematically based engineering including verification, all supported by a tool that minimizes the practical work showes the way to a new dimension of software quality. But especially non-governmental partners cannot ignore the rules of the market. Formal development, even using VSE, takes considerably more effort. The result might be the better product but it also might be too late and significantly more expensive than conventionally developed products. The solution to this problem still has to be worked out. Software applications that have to fulfill high security/safety standards (e.g. avionic systems with the safety level 'catastrophe') already need an enormous amount of quality assurance measures like tests, code inspections and simulation). Both additional efforts for formal methods and conventional QA-activities should be evaluated and compared. There was no time to do that during the pilot projects but the results might be interesting. Not only the producers of high assurance software have to be convinced that formal methods are applicable, improve the quality and save money on testing and warranty acticities; potential clients have to realize that it is worth to spend more money on a product developed with means that guarantee high reliability.

VSE Version 1 is the first successful step to open the market for formal methods in software development. Nevertheless the pilot projects showed that VSE has to be modified and improved: better means of structurizing, batch mode for proofs, better integration of the verifiers, applicability on embedded, reactive systems, code generation C(++), interfaces to Z and model checkers, extension of the interface to the CASE-tool TEAMWORK, etc. Major modifications and improvements will be realized in a follow-on project VSE-II starting summer 1996. The Bundesamt fuer Sicherheit in der Informationstechnik will continue to keep the public informed about changes, experiences and new developments.

References

1. Koob, F., Ullmann, M., Wittmann, S.: The Formal VSE Development Method - A Way to Engineer High-Assurance Software Systems. Eleventh Annual of the COMPUTER SECURITY APPLICATIONS Conference (1995) 196–204
2. Reif, W., Schellhorn, G., Stenzel, K.: Interactive Correctness Proofs for Software Modules Using KIV. Proceedings of the Tenth Annual Conference on Computer Assurance (1995) 151–162
3. Hutter, D. et al: Deduction in the Verification Support Environment (VSE). Springer LNCS **1051** (1996) 268–286

Marrella: A Tool for Simulation and Verification

Dominique AMBROISE
Université de Caen et URA 1526 CNRS
G.R.E.Y.C. - Esplanade de la Paix, 14 032 Caen cedex
e_mail Dominique.Ambroise@info.unicaen.fr

Brigitte ROZOY
Université de Paris XI et URA 410 CNRS[1]
L.R.I., Bât. 490, 91 405 Orsay cedex, France
Tel (33 1) 69 41 66 09, e_mail rozoy@lri.fr

Abstract : This paper presents the structure of our tool Marrella the construction of which has been motivated by practical problems in the context of simulation and parallel program debugging, where the correct evaluation of global properties requires a careful analysis of the causal structure of the execution. The underlying model is based on prime event structures that are considered as exhibiting all the behaviors of distributed programs : the tool gives the possibilities of generating one, some or all of their executions. On one hand, a careful implementation spares memory ; on the other hand, precise and neat algorithms benefit from the trace properties of prime event structures and thus gain in avoiding the enumerations of equivalent interleavings.

1 - Introduction

It is now well known and recognized that one execution of a distributed program may be symbolized by a partial order. In the same spirit, we claim that prime event structures can be used to exhibit in a single object all possible executions. Thus their use may become of a great help for both formal specifications, simulations and verifications of such programs. Inherently, our method is based on prime event structures and exploits technics that unfold the behaviors of programs into acyclic nets. In fact and although not formulated in that terms, this idea has already been used for verification purposes in [MacM. 92] where unfoldings of occurrence Petri Nets are constructed. In case of distributed programs and as it partially avoids the famous state explosion problem, we think that this model is well adapted for efficient simulations, even exhaustive, thus for verifications. Substantial efficiencies are obtained as the enumeration of all possible interleaving are avoided.
Therefore our tool Marrella has heavily been based on these prime event structures.

2 - The Levels in Marrella

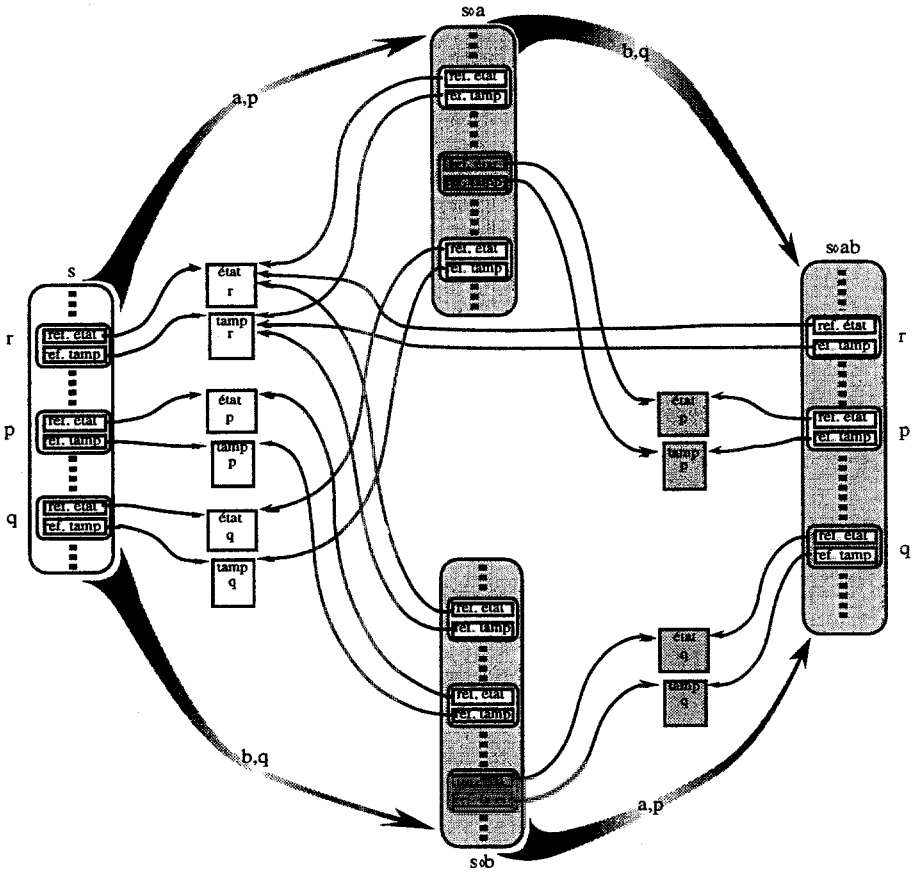
The description of the system is made at three distinct levels. Starting with a concrete distributed asynchronous algorithm, the tool Marrella implements an object $\mathcal{R}$ that describes both the network and the algorithm. It allows to construct $\mathcal{G}(\mathcal{R})$, the graph of states of the system, actually implemented either totally or piece by piece in case it is too big ; this graph is shown to be isomorphic to the graph of configurations of some prime event structure $\mathcal{S}$: $\mathcal{G}(\mathcal{R}) \approx \mathcal{C}onf(\mathcal{S})$. This prime event structure $\mathcal{S}$ constitutes our abstract level, not implemented but used to derive properties of $\mathcal{G}(\mathcal{R})$, therefore to build efficient algorithms.
This is to put side by side to the partial order associated with any distributed computation that is used to verify the associated lattice of ideals.

The real implemented object : a distributed program $\mathcal{R}$

First of all, the really implemented object $\mathcal{R}$ is some kind of network the nodes of which are automaton equipped with buffers and communicating by channels ; this specification is very closed to an implementation using an Estelle like language [Amb. 96].

[1] This work has partially been supported by the inter PRC project "Modèles et Preuves du Parallélisme" of the french MESR.

The static description involves a network of communicating asynchronous processes together with a collection of in-coming and out-coming buffers : a set of processes and for any process p a set Act(p) of firable actions, a set State(p) of (local) states, at last sets Mes(p) of messages and Buff(p) of buffers.
The dynamic part expresses the ability for any process to fire actions, to change states and to send messages. As a basic hypothesis is that the system is asynchronous and without shared memory, the resulting functions are locally testable. For every process p, firts the boolean function g_p : State(p) $\times$ Buff(p) $\times$ Act(p) $\rightarrow$ Boolean allows to compute the enable actions. Second, if an action "a" is possible at state "s" by process "p" with buffers "b", then $f_p(s,b,a)$ and $h_p(s,b,a)$ express the resulting modifications where f_p and h_p are local functions, f_p : State(p) $\times$ Buff(p) $\times$ Act(p) $\rightarrow$ State(p) $\times$ Buff(p) and h_p : State(p) $\times$ Buff(p) $\times$ Act(p) $\rightarrow$ Buff(q_1) $\times \dots \times$ Buff(q_v).



Independent computations of the modifications due to independent actions

A global state of such a system is a collection of local states s_k together with the contents of in-coming and out-coming buffers B_k. Given an initial (global) state, *the labeled graph of states of the system* is classically defined as the graph whose nodes are the reachable

states (s_1, ..., s_n, B_1, ..., B_n), expressing the fact that every process P_k has reach a state s_k and buffers B_k ; the arrows are labeled by the (individual) actions used to perform the step. We use another graph $\mathcal{G}(\mathcal{R})$, which is called *the unfolded labeled graph of states of the system* : if at the current stage every process P_k has executed a sequence w_k of actions, reaching state s_k and buffers B_k, then the associated global unfolded state is (s_1, ..., s_n, B_1, ..., B_n, $[w_1]$, ..., $[w_n]$), where $[w_k]$. is the trace equivalence class of the sequence of actions w_n. It is clear that any construction and exploration of the unfolded graph $\mathcal{G}(\mathcal{R})$ allows a similar approach on the graph of states.

The abstract level : a prime event structure $\mathcal{S}$

At an abstract level, the description uses a labeled prime event structure that will not be constructed but is useful both to understand the properties of the graph and to derive an algorithm for constructing it. Whereas a partial order representation stands for one execution of a distributed program, an event structure exhibits in a single object all its possible executions. In that sense event structures are closed to Petri Nets with which they have strong connections [Nie. Plo. Win. 81], [Wins. Niel. 95], while prime event structures are a special case related to traces [Maz. 87].

The semantic naive interpretation of a labeled prime event structure $\mathcal{S}$ = (E, <, #, λ, A, π, P) describes A as the set of actions possible by processes p in P and elements of E as occurrences of actions performed by these processes during the executions : $e \in E$, $\pi(e) = p$ and $\lambda(e) = a$ may be understood as : an event e will be performed by the process p that fires the action a. The relation < is the classical happened before Lamport relation [Lam. 78] whereas the conflict # may be viewed as the impossibility for two events to belong to the same behavior : at some point of the execution a choice has been made between two events and in consequence events that causally follow one of them will never follow the other.

Similarly to PoSets for which the graph is a lattice of ideals[1], event structures theory gives a great place to the notion of configurations which are there an account of global states : a configuration is any past closed and conflict free subset of events. The graph of configurations is classically defined as the transition system $\mathcal{C}onf(\mathcal{S}) = (\mathcal{C}_f(\mathcal{S}), E, \emptyset, \longrightarrow)$ where $\mathcal{C}_f(\mathcal{S})$ is the set of finite configurations and $(C, e, C') \in \longrightarrow$ iff $e \notin C$ and $C' = C \cup \{e\}$.

It is well known that the set of configurations of an event structure satisfies certain important domains properties and that the original structure may be recovered from its set of configurations using prime elements [Nie. Plo. Win. 81]. These properties are not far from those of lattices : these latter are prime event structures with an empty conflict relation. Here the set of finite configurations is no longer a lattice but admits however a formal characterization : it may be called a budding lattice.

The symbolic level : a budding lattice $\mathcal{G}$

The last description of the system consists in $\mathcal{G}(\mathcal{R})$, the unfolded labeled graph of reachable states which arises to be isomorphic to the graph of configurations of a prime event structure.

Theorem

$\mathcal{G}(\mathcal{R})$, the unfolded graph of states associated to the starting real object $\mathcal{R}$ above, is isomorphic to $\mathcal{C}onf(\mathcal{S})$, the PoSet of finite configurations of some labeled event structure.

---

[1] Ideals being also called global sates or consistent cuts.

Therefore, using this result, efficient constructions and traversals of this graph are implemented in our tool Marrella [Amb. Roz. 96].

3 - Conclusion

The first interest of such a construction is that Marrella uses algorithms that benefit by this structure of budding lattice : applied to prime event structures, we combine techniques that are well known on partial orders as on-the-fly model checking [Fer. Mou. Jar. Jér. 92], traces [Kat. Pel. 92], [Pel. 94] and partial order reductions [God. Wol. 91], [Val. 93], [Esp. 93]. This lead to optimal algorithms that construct either a tree $\mathcal{A}$ that covers the graph or the graph $\mathcal{G}$ itself ; note that, depending on the size, they may either be totally constructed or investigated piece by piece. Similarly, these properties are used to derive adequate simulation strategies [Amb. Roz. 96].

The second interest lies in the implementation : as event structures clearly exhibit the notion of event, the construction and the simulation identify events without ambiguity, thus the action of a given event is computed, executed and stored only once, what is a notable profit both in time and space. Moreover, independent actions may be executed independently on distinct parts of the memory, thus distributed simulations are conceivable.

References

[Amb. 96] D. Ambroise, Marrella ou la simulation guidée d'algorithmes répartis sur réseaux, thèse de l'université de Paris 11, en préparation.

[Amb. Roz. 96] D. Ambroise, B. Rozoy, Using Event Structures for the effficient analysis of states graphs, T.R. 96, Paris-Orsay University.

[Esp. 93] J. Esparza, Model checking using net unfoldings, TapSoft'93, Orsay, LNCS n° 668, 613-628.

[Fer. Mou. Jar. Jér. 92] J.C. Fernandez, L. Mounier, C. Jard, T. Jéron, On the fly verification of finite transition systems, Formal Methods in System Design, Kluwer, 251-273.

[God. Wol. 91] P. Godefroid, P. Wolper, Using Partial Orders for the efficient verification of deadlock freedom and safety properties, CAV'91, LNCS n°575, 332-342.

[Kat. Pel. 92] S. Katz, D. Peled, Verification of distributed programs using representative interleaving sequences, Distributed Computing 6, 107-120.

[Lam. 78] L. Lamport, Time, clocks and the ordering of events in a distributed system, Communications of the ACM, 21(7), 558-565, July 1978.

[MacM. 92] K. L. MacMillan, Using Unfoldings to Avoid the State Explosion Problem in the Verification of Asynchronous Circuits, CAV'92, LNCS n°663, p. 164-177.

[Maz. 87] A. Mazurkiewicz, Trace Theory, Advanced Course on Petri Nets, Bad Honnef, Germany LNCS n° 254, 269-324.

[Nie. Plo. Win. 81] M. Nielsen, G. Plotkin, G. Winskel, Petri Nets, Event Structures and Domains, Theor. Comp. Sci. 13 (1981) p. 85-108.

[Pel. 94] D. Peled, Combining partial order reduction with on-the-fly model checking, Proc. 6th Int. Conf. on computer aid verification, Greece, CAV'93, 409-423.

[Val. 93] A. Valmari, On the fly verification with stubborn sets, Proc. 5th Int. Conf. on computer aid verification, Greece, CAV'93, 397-408.

[Wins. Niel. 95] G. Winskel, M. Nielsen, Models for concurrency, in Handbook of Logic in Computer Science (S. Abramsky, DM. Gabbay, TSE. Maibaum eds.).

Verifying the Safety of a Practical Concurrent Garbage Collector

Georges Gonthier

INRIA Rocquencourt
78153 LE CHESNAY CEDEX
FRANCE

Abstract. We describe our experience in the mechanical verification of the safety invariants of an asynchronous garbage-collection algorithm [1], using the TLP system [2]. We only give a cursory overview of the algorithm and its formalisation. Our main focus is on the lessons learned from carrying a sizeable (22,000+ lines) formal proof through an off-the-shelf prover. In particular, we found the TLP style of structured proofs to be particularly effective for organising, writing, and managing proof scripts.

1 Motivation

If there is one kind of algorithm that warrants a mechanical proof, it probably is concurrent garbage collection. One of the main motivations for automatic garbage collection is safety from devastating, hard-to-detect memory management errors. This requires a very high degree of safety of the collection algorithm; however, such a degree is unattainable by simple testing for a concurrent collector, where errors are hard to trigger.

Therefore the prudent practice in concurrent garbage collection has been to use an array of interlocks to limit the asynchrony, thus enforcing a tamer model of concurrency in which a simple, robust algorithm has been proven (e.g., a sequential one). This caution has a price, though: the extra synchronisation is costly in terms of performance and/or of portability.

In [1] we showed that another tradeoff was possible: using rigourous formal methods, one can design a concurrent garbage collection algorithm that will perform efficiently under realistic concurrency assumptions. However, this demonstration was somewhat incomplete, since it rested only on a manual proof involving 2898 cases... Mechanical verification thus seemed the only way of making the tradeoff of algorithmic versus engineering safety worthwhile.

Furthermore, it appeared that this example could be used to exercise theorem provers in a particular manner. Mechanised proofs tend to fall in two categories: "mathematical proofs" and "certifications". In the former the object is a fragment of mathematics or an abstract algorithm (e.g., [4, 7]), the definitions are dense and layered, the proofs are heavily guided, and the main output is a better understanding of the theory at hand. In the latter, the object is a hardware/software system (e.g, [5, 8]), the definitions are very long and "flat",

the proofs are highly repetitive, hence automated, and the main output is certification of the system. Of course both kinds of proofs occur in a large verification, and recently "hybrid" provers have received considerable attention.

Our collector example, however, is by itself a "hybrid" proof. It is "mathematical" in that it pertains to a short (100-line) algorithm, that it seeks to establish a simple result (for garbage collector safety, simple type correctness is convincing enough), that it involves abstract mathematics (reachability in graphs), and that a better understanding of the algorithm and its invariants was an expected output. On the other hand, our collector proof is also a "certification", because the combinatorics of concurrency blow up the proof size, and because validation of the algorithm was also an important output.

In section 2 we present the development cycle that lead us to the TLP proof. In section 3 we summarise the lessons learned during the proof itself. Section 4 discusses the directions in which this effort could be pursued.

2 The development cycle

The precise description of our algorithm and its formalisation can be found in [1]. Here we will only describe how these were developed.

The development of our algorithm can be cast in the standard "waterfall" model: requirements and architecture, then algorithm design and coding, then abstract formalisation, then formal proof, and finally mechanical verification. Note that formal methods appear here as an expansion of the "testing" stage; we did indeed use them as a debugging tool.

The best evidence that this development plan was sound is provided by the error trace. Each stage caused several major revisions of its immediate predecessor, and a few minor revisions of its grandfather, but changes never propagated more than two levels up:

- Writing down a formal model of the algorithm revealed a major synchronisation error in the algorithm design, and helped to clarify and strengthen the requirements.
- Writing down and manually checking the safety invariants revealed many errors in the model, as well a few secondary synchronisation errors in the algorithm.
- The mechanical verification uncovered a serious omission in the main collector invariant, and a few minor errors in the formal model, none of which reflected errors in the actual implementation (the model being more general).

In addition, there were some simple pragmatic facts supporting our plan:

- Since the whole point was to trade simplicity for performance, it would have been ludicrous to do a full formal analysis before implementing to check the efficiency.
- Since the invariants are about as long (100 lines), but much harder to understand than the program itself, it would have been needlessly hard to try to develop the program from the invariants.

- The invariants themselves are based on 33 definitions which in effect create an abstract view of the algorithm. These definitions involve sets, relations, and transitive closures. It is highly unlikely that they would have emerged naturally from the blind interaction with a prover.

The most crucial step was selecting the level of formalisation. The first attempt was too abstract and did not detect the error in the initial algorithm [1]; on the other hand, it was necessary to abstract from control flow and list management details to have a manageable proof. The transition-predicate approach of TLA [6] provided a convenient framework for making these tradeoffs.

3 The verification

Engberg's TLP [2] is a front-end for the LP prover [3]. It provides support for the TLA logic, for Lamport's formula list syntax, and his structured proofs, as well as a modest macro facility. The TLA support was largely irrelevant for us: the safety proof did not involve any significant temporal reasoning, so we only used the "prime" notation for next state variables.

On the other hand, the apparently trivial support for Lamport's structured proofs turned out to be crucial for the success of our effort. A TLP proof script is a sequence of prover commands and *steps*; a step is simply a formula together with its proof script, which may recursively contain substeps. A step may also introduce hypotheses which will be discharged upon exit; TLP also keeps track of any needed skolemisation. Each (sub)step is proved in its context; in particular, all previous steps and hypotheses, as well as any fact derived from them by forward inference prover commands, are available for the proof. A simple depth-based indexing scheme makes references to these local facts short and convenient.

This structure makes TLP proof scripts especially readable and robust, because they consist of a sequence of true statements, interspersed with forward inference commands (the few TLP backward inference commands turned out to be impractical). This is very close to a hand proof, so much so that we were able to write most of our 22,000 line script *off-line*, using the proof-checker only as a (sluggish) debugger!

The robustness stems from the fact that the validity of the substeps is generally independent from the prover deduction strategy and from superficial changes of the problem definition. Hence it is very easy to cut and paste pieces of scripts, or to adapt a script to a new context (a similar case has been made for Nqthm [5]). This feature turned out to be a lifesaver when we discovered a serious omission in the main collector invariant, during the proof of the last of 64 main lemmas, four months into the proof. The invariant and several definitions had to be reworked, but hardly any of the script needed any change; most substeps were still valid (albeit sometimes for different reasons!).

Many advocate the decomposition of a proof in a succession of small lemmas; however very few proof systems provide features for organising the resulting lemmata army. Our proof involves over 3,500 substeps (some of which would

not have been needed with a more powerful prover than TLP, with support for typechecking, arithmetic, etc). It would have been next to impossible to spell out each of these as a separately named lemma with an explicit context. The very simple TLP indexing scheme was invaluable here.

Not everything in TLP was great, though. We had to develop some ingenuity to compensate for TLP's limitations in typechecking, higher-order rewriting, and quantifications. However, ultimately, it was the control provided by the proof structure that was decisive and that enabled us to go through with the proof.

4 Going further

Proving the termination of the collector cycle would appear to be our next logical step. However, this would give us fairly little return – livelock is rarely a problem for a concurrent garbage collector – for a proof that would be just as complex, and probably more given the weakness of LP in arithmetic.

A more ambitious project would be to layer the specification, using the invariant definitions as a refinement mapping from program (concrete) to invariant (abstract) variables. Currently the safety proof for each action consists of three parts: a proof that the action maintains some representation invariants, a proof the the program action (on concrete variables) implements a certain *abstract action* (on abstract variables), and finally a proof that the abstract action satisfies the algorithm's invariants. Layering the specification would yield a clean separation between the three parts, and would open the way for proving the correctness of an even more detailed version of the algorithm. This may become practical if a TLP-like interface is built on more powerful prover.

References

1. Doligez, D., Gonthier, G.: Portable, unobtrusive garbage collection for multiprocessor systems. ACM POPL (1994) 70–83
2. Engberg, U., Gr/onning, P., Lamport, L.: Mechanical verification of concurrent systems with TLP. LNCS 663 (CAV 1992) 44–55
3. Garland, S. J., Guttag, J. V.: An overview of LP, the Larch prover. LNCS 355 (RTA 1989) 137–151
4. Huet, G.: Residual theory in λ-calculus: a formal development. J. Func. Prog. **4** (1994) 371–394
5. Hunt, W. A. Jr., Brock, B.: A formal HDL and its use in the FM9001. Proc. Royal Soc. (1992)
6. Lamport, L.: The temporal logic of actions. ACM TOPLAS **16** (1994) 872–923
7. Lincoln, P., Rushby, J.: Formal verification of an algorithm for interactive consistency under a hybrid fault model. CAV 1993
8. Miller, S. P., Srivas, M.: Formal verification of the AAMP5 microprocessor. IEEE Workshop on Industrial-Strength Formal Spec. Techniques (1995)

Verification by Behaviour Abstraction

—

A Case Study of Service Interaction Detection in Intelligent Telephone Networks

Carla Capellmann[a], Ralph Demant[b], Farhad Fatahi-Vanani[b], Rafael Galvez-Estrada[b], Ulrich Nitsche[b] and Peter Ochsenschläger[b]

[a]Deutsche Telekom, Technologiezentrum Darmstadt
Research Group for Functional Aspects of Networks
Am Kavalleriesand 3, D-64295 Darmstadt, Germany.
Email: carla@fz.telekom.de

[b]GMD – German National Research Centre for Information Technology
Institute for Telecooperation Technology
Rheinstraße 75, D-64295 Darmstadt, Germany.
Email: {demant,fatahi,galvez,nitsche,ochsenschlaeger}@darmstadt.gmd.de

1 Introduction

Modern telephone systems (Intelligent Networks; IN) allow to flexibly introduce new services in the network. There, one is increasingly confronted with undesired interactions of services and service features. The problem of finding service interactions is referred to as *service interaction detection* (or: feature interaction detection). The aim of the project presented in this abstract is the detection of service interactions on the specification level. The core of such a specification is the so called *Basic Call State Model* (BCSM) [Q.1], where the services are coupled to.

We specify services and the BCSM using Product Nets [OP95] (high level Petri Nets). By a complete reachability analysis we obtain an automata representation of the complete behaviour of the services. To avoid state-space explosion, a compositional analysis technique can be used [Och95]. To check for service interactions we check temporal properties of the behaviour. In general, the behaviour of the specified services, including the BCSM, is too complex to efficiently check temporal logic formulae on the automata representation of the behaviour (the reachability graph) [CMR93]. Therefore, we first compute an abstraction of the behaviour that has a sufficiently small automata representation to check temporal properties efficiently. When calculating the abstract behaviour we have to be careful not to hide important behaviour with respect to possible interactions of the specified services. As abstraction technique, we use language homomorphisms applied to the language accepted by the automaton representing the behaviour of the combined services. Because it is very important for a telephone system to be able to make progress (we always want to be able to make a call eventually), an important class of properties we have to check in regard to service interaction detection is the class of liveness properties. It is

especially this class of properties that is very delicate to handle with respect to abstractions.

In the following section we present two example services, Call Forwarding Unconditional (CFU) and Selective Call Rejection (SCR). By discussing these two services we give a brief introduction to the methods we use to detect service interactions. Finally we summarize the current state of our project and give an outlook on the topics we address next.

2 CFU and SCR

As a small example to illustrate the basic idea of our approach towards interaction detection, we have selected two simple services, Call Forwarding Unconditional (CFU) and Selective Call Rejection (SCR). CFU is a service that, when active, forwards an incoming call to a selected recepient. Selective Call Rejection (SCR) prohibits an incoming call to be signalled; i.e. an incoming call is rejected without notifying the intended receiver.

To keep a first model of CFU and SCR simple, we describe both services on a high abstraction level, assuming a very simplified basic call process where the activation of a service simply depends on its parameters: The parameter that activates CFU is the telephone number to which an incoming call has to be forwarded. SCR's parameter is a list of telephone numbers such that calls by a caller whose number is in the list are rejected.

3 Interaction of CFU and SCR

As already mentioned, we have specified CFU and SCR logically as a Product Net. Here, "logically" means that we did not involve the underlying model of the telephone system, the Basic Call State Model (BCSM) as defined in [Q.1]. This first step was intended to show that the methods we propose for service interaction detection are indeed suitable for this task.

We analyzed our model of CFU and SCR for all combinations of CFU and SCR being active and being not active involving 3 subscribers (users). The complete reachability analysis of each active/not active combination of CFU and SCR led to automata representations having at most 780 states. All the automata were deadlock free. This is an important property for the specified services; formally this is a safety property. As said in the introduction, the properties that guarantee progress are the liveness properties. The liveness property we want to check is whether it is always possible to eventually get a connection to an intended receiver. If *connect* is the event that describes the establishment of a desired connection, then in temporal logic we want to check if the formulae $\Box\Diamond$*connect* ("always eventually connect") is a liveness property with respect to the behaviour of the specified services. We can check this property on an abstraction of the behaviour that keeps the *connect* event visible and hides all other events. Formally, such an abstraction is defined by a language homomorphism.

To be able to check liveness properties on the abstraction[Nit94a, Nit94b, NO95], the homomorphisms have to satisfy a special condition called *simplicity* of language homomorphisms [Och94].

Applying a suitable *simple* homomorphism to the automata representation of the behaviour of the specification of CFU and SCR leads to an empty automaton, if CFU and SCR are active such that the number of the caller is in the rejection list of the subscriber to whom the call is forwarded. That means, for this constellation, $\Box\Diamond$*connect* is not a property of the specification, even if the originally called subscriber did not want to reject that call. Therefore, CFU and SCR interact by SCR interfering CFU.

4 Verification of the BCSM

So far, we only have checked the interaction of services on a logical level. Nevertheless, for detecting service interactions in a more realistic specification, the underlying telephone system (BCSM) has to be specified as well, to embed the services in it. So we have already specified the BCSM as a Product Net and have verified its correctness. More precisely, using *simple* homomorphisms, we were able to prove that our specification shows exactly the behaviour as it is described by the automata in [Q.1].

5 Conclusion

As shown above, verification of temporal properties on behaviour abstractions is suitable for detecting service interactions. In the example that we briefly explained the abstraction step was not really necessary, for the example itself is small enough to check temporal properties directly on the behaviour.

A complete analysis of the BCSM model already leads to a behaviour representation having several thousand states. Including services, we expect several tenthousand states. This reaches an order of magnitude of the number of states that makes abstraction techniques absolutely necessary. Here, even the analysis of the specification can become difficult with respect to complexity. Hence, a compositional analysis technique of specifications that allows to compute a representation of an abstract behaviour without exhaustive construction of the complete state space is currently developed [Och95]. We expect that our approach will allow us to check several combinations of services for undesired interactions where any direct verification approach without abstractions would be intractable.

References

[CMR93] Pierre Combes, Max Michel, and Béatrice Renard. Formal verification of telecommunication service interactions using SDL methods and tools. In O. Faergemand and A. Sarma, editors, *SDL'93: Using Objects*, pages 441–452. Elsevier, 1993.

[Nit94a] Ulrich Nitsche. Propositional linear temporal logic and language homomorphisms. In Anil Nerode and Yuri V. Matiyasevich, editors, *Logical Foundations of Computer Science '94—Logic at St. Petersburg*, volume 813 of *Lecture Notes in Computer Science*, pages 265–277. Springer Verlag, 1994.

[Nit94b] Ulrich Nitsche. A verification method based on homomorphic model abstraction. In *Proceedings of the 13th Annual ACM Symposium on Principles of Distributed Computing*, page 393, Los Angeles, 1994. ACM Press.

[NO95] Ulrich Nitsche and Peter Ochsenschläger. Approximately satisfied properties of systems and simple language homomorphisms. Arbeitspapiere der GMD 965, Gesellschaft für Mathematik und Datenverarbeitung (GMD), Darmstadt, December 1995.

[Och94] Peter Ochsenschläger. Verification of cooperating systems by simple homomorphisms using the product net machine. In Jörg Desel, Andreas Oberweis, and Wolfgang Reisig, editors, *Workshop: Algorithmen und Werkzeuge für Petrinetze*, pages 48–53. Humboldt Universität Berlin, 1994.

[Och95] Peter Ochsenschläger. Compositional verification of cooperating systems using simple homomorphisms. In Jörg Desel, Hans Fleischhack, Andreas Oberweis, and Sonnenschein Michael, editors, *Workshop: Algorithmen und Werkzeuge für Petrinetze*, pages 8–13. Universität Oldenburg, 1995.

[OP95] Peter Ochsenschläger and Rainer Prinoth. *Modellierung verteilter Systeme – Konzeption, Formale Spezifikation und Verifikation mit Produktnetzen.* Vieweg, Wiesbaden, 1995.

[Q.1] Draft Revised ITU-T Recommendation Q.1214: *Distributed Functional Plane for Intelligent Network CS-1.* March 1995.

List of Authors

Lecture Notes in Computer Science

For information about Vols. 1–1029

please contact your bookseller or Springer-Verlag

Vol. 1030: F. Pichler, R. Moreno-Díaz, R. Albrecht (Eds.), Computer Aided Systems Theory - EUROCAST '95. Proceedings, 1995. XII, 539 pages. 1996.

Vol.1031: M. Toussaint (Ed.), Ada in Europe. Proceedings, 1995. XI, 455 pages. 1996.

Vol. 1032: P. Godefroid, Partial-Order Methods for the Verification of Concurrent Systems. IV, 143 pages. 1996.

Vol. 1033: C.-H. Huang, P. Sadayappan, U. Banerjee, D. Gelernter, A. Nicolau, D. Padua (Eds.), Languages and Compilers for Parallel Computing. Proceedings, 1995. XIII, 597 pages. 1996.

Vol. 1034: G. Kuper, M. Wallace (Eds.), Constraint Databases and Applications. Proceedings, 1995. VII, 185 pages. 1996.

Vol. 1035: S.Z. Li, D.P. Mital, E.K. Teoh, H. Wang (Eds.), Recent Developments in Computer Vision. Proceedings, 1995. XI, 604 pages. 1996.

Vol. 1036: G. Adorni, M. Zock (Eds.), Trends in Natural Language Generation - An Artificial Intelligence Perspective. Proceedings, 1993. IX, 382 pages. 1996. (Subseries LNAI).

Vol. 1037: M. Wooldridge, J.P. Müller, M. Tambe (Eds.), Intelligent Agents II. Proceedings, 1995. XVI, 437 pages. 1996. (Subseries LNAI).

Vol. 1038: W: Van de Velde, J.W. Perram (Eds.), Agents Breaking Away. Proceedings, 1996. XIV, 232 pages. 1996. (Subseries LNAI).

Vol. 1039: D. Gollmann (Ed.), Fast Software Encryption. Proceedings, 1996. X, 219 pages. 1996.

Vol. 1040: S. Wermter, E. Riloff, G. Scheler (Eds.), Connectionist, Statistical, and Symbolic Approaches to Learning for Natural Language Processing. IX, 468 pages. 1996. (Subseries LNAI).

Vol. 1041: J. Dongarra, K. Madsen, J. Waśniewski (Eds.), Applied Parallel Computing. Proceedings, 1995. XII, 562 pages. 1996.

Vol. 1042: G. Weiß, S. Sen (Eds.), Adaption and Learning in Multi-Agent Systems. Proceedings, 1995. X, 238 pages. 1996. (Subseries LNAI).

Vol. 1043: F. Moller, G. Birtwistle (Eds.), Logics for Concurrency. XI, 266 pages. 1996.

Vol. 1044: B. Plattner (Ed.), Broadband Communications. Proceedings, 1996. XIV, 359 pages. 1996.

Vol. 1045: B. Butscher, E. Moeller, H. Pusch (Eds.), Interactive Distributed Multimedia Systems and Services. Proceedings, 1996. XI, 333 pages. 1996.

Vol. 1046: C. Puech, R. Reischuk (Eds.), STACS 96. Proceedings, 1996. XII, 690 pages. 1996.

Vol. 1047: E. Hajnicz, Time Structures. IX, 244 pages. 1996. (Subseries LNAI).

Vol. 1048: M. Proietti (Ed.), Logic Program Syynthesis and Transformation. Proceedings, 1995. X, 267 pages. 1996.

Vol. 1049: K. Futatsugi, S. Matsuoka (Eds.), Object Technologies for Advanced Software. Proceedings, 1996. X, 309 pages. 1996.

Vol. 1050: R. Dyckhoff, H. Herre, P. Schroeder-Heister (Eds.), Extensions of Logic Programming. Proceedings, 1996. VII, 318 pages. 1996. (Subseries LNAI).

Vol. 1051: M.-C. Gaudel, J. Woodcock (Eds.), FME'96: Industrial Benefit and Advances in Formal Methods. Proceedings, 1996. XII, 704 pages. 1996.

Vol. 1052: D. Hutchison, H. Christiansen, G. Coulson, A. Danthine (Eds.), Teleservices and Multimedia Communications. Proceedings, 1995. XII, 277 pages. 1996.

Vol. 1053: P. Graf, Term Indexing. XVI, 284 pages. 1996. (Subseries LNAI).

Vol. 1054: A. Ferreira, P. Pardalos (Eds.), Solving Combinatorial Optimization Problems in Parallel. VII, 274 pages. 1996.

Vol. 1055: T. Margaria, B. Steffen (Eds.), Tools and Algorithms for the Construction and Analysis of Systems. Proceedings, 1996. XI, 435 pages. 1996.

Vol. 1056: A. Haddadi, Communication and Cooperation in Agent Systems. XIII, 148 pages. 1996. (Subseries LNAI).

Vol. 1057: P. Apers, M. Bouzeghoub, G. Gardarin (Eds.), Advances in Database Technology — EDBT '96. Proceedings, 1996. XII, 636 pages. 1996.

Vol. 1058: H. R. Nielson (Ed.), Programming Languages and Systems – ESOP '96. Proceedings, 1996. X, 405 pages. 1996.

Vol. 1059: H. Kirchner (Ed.), Trees in Algebra and Programming – CAAP '96. Proceedings, 1996. VIII, 331 pages. 1996.

Vol. 1060: T. Gyimóthy (Ed.), Compiler Construction. Proceedings, 1996. X, 355 pages. 1996.

Vol. 1061: P. Ciancarini, C. Hankin (Eds.), Coordination Languages and Models. Proceedings, 1996. XI, 443 pages. 1996.

Vol. 1062: E. Sanchez, M. Tomassini (Eds.), Towards Evolvable Hardware. IX, 265 pages. 1996.

Vol. 1063: J.-M. Alliot, E. Lutton, E. Ronald, M. Schoenauer, D. Snyers (Eds.), Artificial Evolution. Proceedings, 1995. XIII, 396 pages. 1996.

Vol. 1064: B. Buxton, R. Cipolla (Eds.), Computer Vision – ECCV '96. Volume I. Proceedings, 1996. XXI, 725 pages. 1996.

Vol. 1065: B. Buxton, R. Cipolla (Eds.), Computer Vision – ECCV '96. Volume II. Proceedings, 1996. XXI, 723 pages. 1996.

Vol. 1066: R. Alur, T.A. Henzinger, E.D. Sontag (Eds.), Hybrid Systems III. IX, 618 pages. 1996.

Vol. 1067: H. Liddell, A. Colbrook, B. Hertzberger, P. Sloot (Eds.), High-Performance Computing and Networking. Proceedings, 1996. XXV, 1040 pages. 1996.

Vol. 1068: T. Ito, R.H. Halstead, Jr., C. Queinnec (Eds.), Parallel Symbolic Languages and Systems. Proceedings, 1995. X, 363 pages. 1996.

Vol. 1069: J.W. Perram, J.-P. Müller (Eds.), Distributed Software Agents and Applications. Proceedings, 1994. VIII, 219 pages. 1996. (Subseries LNAI).

Vol. 1070: U. Maurer (Ed.), Advances in Cryptology – EUROCRYPT '96. Proceedings, 1996. XII, 417 pages. 1996.

Vol. 1071: P. Miglioli, U. Moscato, D. Mundici, M. Ornaghi (Eds.), Theorem Proving with Analytic Tableaux and Related Methods. Proceedings, 1996. X, 330 pages. 1996. (Subseries LNAI).

Vol. 1072: R. Kasturi, K. Tombre (Eds.), Graphics Recognition. Proceedings, 1995. X, 308 pages. 1996.

Vol. 1073: J. Cuny, H. Ehrig, G. Engels, G. Rozenberg (Eds.), Graph Grammars and Their Application to Computer Science. Proceedings, 1994. X, 565 pages. 1996.

Vol. 1074: G. Dowek, J. Heering, K. Meinke, B. Möller (Eds.), Higher-Order Algebra, Logic, and Term Rewriting. Proceedings, 1995. VII, 287 pages. 1996.

Vol. 1075: D. Hirschberg, G. Myers (Eds.), Combinatorial Pattern Matching. Proceedings, 1996. VIII, 392 pages. 1996.

Vol. 1076: N. Shadbolt, K. O'Hara, G. Schreiber (Eds.), Advances in Knowledge Acquisition. Proceedings, 1996. XII, 371 pages. 1996. (Subseries LNAI).

Vol. 1077: P. Brusilovsky, P. Kommers, N. Streitz (Eds.), Mulimedia, Hypermedia, and Virtual Reality. Proceedings, 1994. IX, 311 pages. 1996.

Vol. 1078: D.A. Lamb (Ed.), Studies of Software Design. Proceedings, 1993. VI, 188 pages. 1996.

Vol. 1079: Z.W. Raś, M. Michalewicz (Eds.), Foundations of Intelligent Systems. Proceedings, 1996. XI, 664 pages. 1996. (Subseries LNAI).

Vol. 1080: P. Constantopoulos, J. Mylopoulos, Y. Vassiliou (Eds.), Advanced Information Systems Engineering. Proceedings, 1996. XI, 582 pages. 1996.

Vol. 1081: G. McCalla (Ed.), Advances in Artificial Intelligence. Proceedings, 1996. XII, 459 pages. 1996. (Subseries LNAI).

Vol. 1082: N.R. Adam, B.K. Bhargava, M. Halem, Y. Yesha (Eds.), Digital Libraries. Proceedings, 1995. Approx. 310 pages. 1996.

Vol. 1083: K. Sparck Jones, J.R. Galliers, Evaluating Natural Language Processing Systems. XV, 228 pages. 1996. (Subseries LNAI).

Vol. 1084: W.H. Cunningham, S.T. McCormick, M. Queyranne (Eds.), Integer Programming and Combinatorial Optimization. Proceedings, 1996. X, 505 pages. 1996.

Vol. 1085: D.M. Gabbay, H.J. Ohlbach (Eds.), Practical Reasoning. Proceedings, 1996. XV, 721 pages. 1996. (Subseries LNAI).

Vol. 1086: C. Frasson, G. Gauthier, A. Lesgold (Eds.), Intelligent Tutoring Systems. Proceedings, 1996. XVII, 688 pages. 1996.

Vol. 1087: C. Zhang, D. Lukose (Eds.), Distributed Artificial Intelliegence. Proceedings, 1995. VIII, 232 pages. 1996. (Subseries LNAI).

Vol. 1088: A. Strohmeier (Ed.), Reliable Software Technologies – Ada-Europe '96. Proceedings, 1996. XI, 513 pages. 1996.

Vol. 1089: G. Ramalingam, Bounded Incremental Computation. XI, 190 pages. 1996.

Vol. 1090: J.-Y. Cai, C.K. Wong (Eds.), Computing and Combinatorics. Proceedings, 1996. X, 421 pages. 1996.

Vol. 1091: J. Billington, W. Reisig (Eds.), Application and Theory of Petri Nets 1996. Proceedings, 1996. VIII, 549 pages. 1996.

Vol. 1092: H. Kleine Büning (Ed.), Computer Science Logic. Proceedings, 1995. VIII, 487 pages. 1996.

Vol. 1093: L. Dorst, M. van Lambalgen, F. Voorbraak (Eds.), Reasoning with Uncertainty in Robotics. Proceedings, 1995. VIII, 387 pages. 1996. (Subseries LNAI).

Vol. 1094: R. Morrison, J. Kennedy (Eds.), Advances in Databases. Proceedings, 1996. XI, 234 pages. 1996.

Vol. 1095: W. McCune, R. Padmanabhan, Automated Deduction in Equational Logic and Cubic Curves. X, 231 pages. 1996. (Subseries LNAI).

Vol. 1096: T. Schäl, Workflow Management Systems for Process Organisations. XII, 200 pages. 1996.

Vol. 1097: R. Karlsson, A. Lingas (Eds.), Algorithm Theory – SWAT '96. Proceedings, 1996. IX, 453 pages. 1996.

Vol. 1098: P. Cointe (Ed.), ECOOP '96 – Object-Oriented Programming. Proceedings, 1996. XI, 502 pages. 1996.

Vol. 1099: F. Meyer auf der Heide, B. Monien (Eds.), Automata, Languages and Programming. Proceedings, 1996. XII, 681 pages. 1996.

Vol. 1101: M. Wirsing, M. Nivat (Eds.), Algebraic Methodology and Software Technology. Proceedings, 1996. XII, 641 pages. 1996.

Vol. 1102: R. Alur, T.A. Henzinger (Eds.), Computer Aided Verification. Proceedings, 1996. XII, 472 pages. 1996.

Vol. 1103: H. Ganzinger (Ed.), Rewriting Techniques and Applications. Proceedings, 1996. XI, 437 pages. 1996.

Vol. 1105: T.I. Ören, G.J. Klir (Eds.), Computer Aided Systems Theory – CAST '94. Proceedings, 1994. IX, 439 pages. 1996.

Vol. 1106: M. Jampel, E. Freuder, M. Maher (Eds.), Over-Constrained Systems. X, 309 pages. 1996.