Lecture Notes in Computer Science 2057
Edited by G. Goos, J. Hartmanis and J. van Leeuwen

Springer
Berlin
Heidelberg
New York
Barcelona
Hong Kong
London
Milan
Paris
Singapore
Tokyo

Matthew Dwyer (Ed.)

Model Checking Software

8th International SPIN Workshop
Toronto, Canada, May 19–20, 2001
Proceedings

Springer

Series Editors

Gerhard Goos, Karlsruhe University, Germany
Juris Hartmanis, Cornell University, NY, USA
Jan van Leeuwen, Utrecht University, The Netherlands

Volume Editor

Matthew Dwyer
Kansas State University, Department of Computing and Information Sciences
234 Nichols Hall, Manhattan, KS 66506-2302, USA
E-mail: dwyer@cis.ksu.edu

Cataloging-in-Publication Data applied for

Die Deutsche Bibliothek - CIP-Einheitsaufnahme

Model checking software : proceedings / 8th International SPIN Workshop, Toronto, Canada, May 19 - 20, 2001. Matthew Dwyer (ed.). - Berlin ; Heidelberg ; New York ; Barcelona ; Hong Kong ; London ; Milan ; Paris ; Singapore ; Tokyo : Springer, 2001
(Lecture notes in computer science ; Vol. 2057)
ISBN 3-540-42124-6

CR Subject Classification (1998): F.3, D.2.4, D.3.1

ISSN 0302-9743
ISBN 3-540-42124-6 Springer-Verlag Berlin Heidelberg New York

Springer-Verlag Berlin Heidelberg New York
a member of BertelsmannSpringer Science+Business Media GmbH

http://www.springer.de

Printed in Germany

Typesetting: Camera-ready by author, data conversion by PTP-Berlin, Stefan Sossna
Printed on acid-free paper SPIN: 10781569 06/3142 5 4 3 2 1 0

Preface

Research on model checking has matured from a purely theoretical topic to encompass tool development and applications, in addition to more foundational topics. This diversity of model checking research is driving the area onward as foundational developments enable automation, development of robust tool support enables increasingly sophisticated applications, and feedback from applications spurs further work on the underlying theory and tools. The program of the eighth SPIN workshop reflected this diversity; it included three contributions on foundational topics, eight contributions on model checking tools, and eight contributions describing applications of model checking.

Continuing a trend begun in the seventh SPIN workshop, the eighth SPIN workshop emphasized the connections between model checking and program analysis. Research on static program analysis has a long history in both the compiler and software engineering communities. In an effort to establish a dialog between researchers in model checking and software analysis, this year's workshop was co-located with the 23rd International Conference on Software Engineering in Toronto. The workshop program contained several contributions that were clearly targeted at analyzing programs. Three contributions addressed tools for model checking of program source code, implemented in C and Java, and one contribution described model checking of a popular software component architecture.

The workshop featured 13 refereed technical papers selected from 26 submissions and two refereed descriptions of model checking tools selected from four submissions. Each submitted paper was reviewed by at least three members of the program committee; additional reviewers were used for several papers. The program committee discussed the merits of the submitted papers to arrive at the final 15 refereed contributions. In addition to refereed contributions, two leading experts in model checking technology and three groups that are applying model checking techniques in an industrial setting were invited to give presentations. The invited presentations were given by: Doron Peled (Bell Laboratories), Rob Gerth (Intel Corporation), Leszek Holenderski (Philips Research), Erik Engstrom (Honeywell Laboratories), and Bernhard Steffen (Metaframe Technologies). This proceedings issue contains four contributions detailing the content of the invited presentations. A panel session on the topic of "Prospects for and impediments to practical model checking" was organized to generate a dialog between those working on applying model checking and researchers working on extending model checking technologies.

Historically, the SPIN workshop has served as a forum for researchers interested in the subject of automata-based, explicit-state model checking technologies for the analysis and verification of asynchronous concurrent and distributed systems. In recent years, the scope of the workshop has broadened to encompass applications of model checking to software analysis. The workshop is named af-

ter the SPIN model checker, developed by Gerard Holzmann, which is one of the best known and most widely used model checking tools. The first SPIN workshop was held in October 1995 in Montréal. Subsequent workshops were held in New Brunswick (August 1996), Enschede (April 1997), Paris (November 1998), Trento (July 1999), Toulouse (September 1999), and at Stanford University (August 2000).

Acknowledgments. The editor wishes to thank the program committee members, and the referees, for their help in in organizing the workshop. Committee members volunteered many valuable suggestions as well as a significant amount of time for refereeing and discussing papers. The workshop organizers wish to thank the ICSE'2001 organizing committee for facilitating the co-location of the SPIN workshop and ACM SIGSOFT, Lucent Technologies, Microsoft Research, and the Office of Naval Research for their sponsorship and support of the SPIN workshop.

March 2001 Matthew B. Dwyer

Organization

Organizing Committee

General Chair: Moshe Y. Vardi (Rice University, USA)
Program Chair: Matthew B. Dwyer (Kansas State University, USA)
Local Arrangements Chair: Marsha Chechik (University of Toronto, Canada)

Program Committee

George Avrunin (University of Massachusetts, USA)
Thomas Ball (Microsoft Research, USA)
Ed Brinksma (University of Twente, The Netherlands)
Marsha Chechik (University of Toronto, Canada)
Dennis R. Dams (Eindhoven University, The Netherlands)
Klaus Havelund (QSS/Recom at NASA Ames Research Center, USA)
Connie Heitmeyer (Naval Research Laboratory, USA)
Gerard J. Holzmann (Bell Laboratories, USA)
Fabio Somenzi (University of Colorado, USA)
Willem Visser (RIACS at NASA Ames Research Center, USA)
Pierre Wolper (Université de Liege, Belgium)

Referees

R. Bloem
D. Bosnacki
J. Geldenhuys
D. Giannakopoulou
H. Hermanns
L. Holenderski
J. Katoen
R. Langerak
F. Lerda
S. Park
C. Pecheur
G. Rosu
T. Ruys
R. de Vries
B. Wolter

Sponsoring Organizations

The eighth SPIN Workshop was sponsored by the ACM SIGSOFT (Special Interest Group on Software Engineering). Additional support was provided by: Bell Laboratories, Lucent Technologies, USA, Microsoft Research, Microsoft Inc., USA, and The Office of Naval Research, USA.

Table of Contents

Invited Keynotes

Technical Papers and Tool Reports

Invited Project Summaries

From Model Checking to a Temporal Proof

Doron Peled[1] and Lenore Zuck[2]

[1] Bell Laboratories, 600 Mountain Ave., Murray Hill, NJ 07974
doron@research.bell-labs.com
[2] Department of Computer Science, New York University
zuck@cs.nyu.edu

Abstract. Model checking is used to automatically verify temporal properties of finite state systems. It is usually considered to be 'successful', when an error, in the form of a counterexample to the checked property, is found. We present the dual approach, where, in the presence of no counterexample, we automatically generate a proof that the checked property is satisfied by the given system. Such a proof can be used to obtain intuition about the verified system. This approach can be added as a simple extension to existing model checking tools.

1 Introduction

The automatic verification of finite state systems, often called *model checking* [1, 4], is highly successful in detecting bugs during software and hardware development. It involves modeling the inspected system, specifying its properties using some logical formalism, and using some graph algorithms to systematically detect whether there are executions of the model of the system that violate the specification. If such executions exist, (at least) one of them is reported as a counterexample.

If the search for counterexamples fails, we can conclude that the model of the system satisfies the specification. This is often considered as a failure of the model checking attempt. Some even go as far as to say that the goal of model checking is not 'verification', but 'falsification'. One reason for this is that the modeling process is itself prone to errors. The model of the checked system is often an oversimplification of the original system since model checking often requires that the model has finitely many states.

An alternative to model checking is the *deductive verification* approach, one of whose goals is to formally explain *why* the system satisfies the checked property. This approach often calls for creativity of the prover. Deductive verification is usually manual and time consuming.

In this paper, we emphasize the point of view that when no counterexample is found, model checking can also be used to *justify* why the verified system satisfies the checked property. We show that a failed systematic search for counterexamples can be used to generate a deductive proof that the model satisfies

M.B. Dwyer (Ed.): SPIN 2001, LNCS 2057, pp. 1–14, 2001.

the checked property, by exploiting the information in the graph that is generated during the search. We apply the automata theoretic view [7,10]. Specifically, we start with an LTL (linear temporal logic) specification, and transform it into an automaton. In fact, it is the negation of the specification formula that is translated. The checked system is also represented as an automaton. We then construct the intersection of these two automata. If the intersection is nonempty, it contains a counterexample that can be reported. Otherwise, the model of the system satisfies the specification. We show how, in the latter case, the intersection graph can be used to produce a proof.

The main challenge is how to represent the proof, which is implicit in the intersection graph. We would like to present the proof in a way that would explain to the user why the property holds for the checked system.

2 Preliminaries

We sketch the model checking procedure for Temporal Logic formulas over finite state formulas. For more details, see [3].

A *Generalized Büchi automaton* A is a 6-tuple $(S, S_0, \delta, \mathcal{F}, L, \Sigma)$, where S is a finite set of *states*, $S_0 \subseteq S$ is a set of *initial states*, $\delta \subseteq S \times S$ is a (nondeterministic) *transition relation*, $\mathcal{F} \subseteq 2^{S \times S}$ is the set of *acceptance sets*, Σ is a set of labels and $L: S \to \Sigma$ is a labeling function of the states.[1] A *run* of the automaton A is an infinite sequence of S-states $\alpha = s_0, s_1, \ldots$ such that $s_0 \in S_0$, and for every $i \geq 0$, $(s_i, s_{i+1}) \in \delta$. A run is accepting if for every $F \in \mathcal{F}$, $(s_i, s_{i+1}) \in F$ for infinitely many i's. The *language accepted by an automaton* A, denoted by $\mathcal{L}(A)$, is the set of (labeled) sequences that are accepted by the automaton.

Let Π be a set of propositions. We consider here linear time propositional temporal logic (LTL) formulas over Π, using the Boolean connectives $\vee$ and $\neg$, and the temporal operators *nexttime* $\bigcirc$ and *until* $\mathcal{U}$. Temporal logic formulas are interpreted over infinite sequences over 2^{Π} (see, e.g., [8]). Let $\sigma = x_0 x_1 \ldots$ be a sequence of states. Denote its suffix $x_i x_{i+1} \ldots$ by σ^i. We denote the fact that a sequence σ satisfies a temporal formula φ by $\sigma \models \varphi$. For a propositional formula $\sigma^i \models \varphi$ if φ holds in the state x_i. $\sigma^i \models \bigcirc\varphi$ if $\sigma^{i+1} \models \varphi$. $\sigma^i \models \varphi \mathcal{U} \psi$ holds if there exists $j \geq i$ such that $\sigma^j \models \psi$, and for each $i \leq k < j$, $\sigma^k \models \varphi$. The other Boolean connectives and Temporal operators ($\Box$, $\Diamond$, $\mathcal{V}$, etc.) can be defined using the above operators, namely $\Diamond\varphi = \mathit{true}\mathcal{U}\varphi$, $\Box\varphi = \neg\Diamond\neg\varphi$ and $\varphi \mathcal{V} \psi = \neg((\neg\varphi)\mathcal{U}(\neg\psi))$.

We assume that all the temporal formulas are given in the *negation normal form*, i.e., with negation appearing only on propositions. This can be easily achieved by pushing negation inwards, using the equivalences $\neg\neg\varphi = \varphi$, $\neg \bigcirc \varphi = \bigcirc\neg\varphi$, $\neg(\varphi \vee \psi) = (\neg\varphi) \wedge (\neg\psi)$, $\neg(\varphi \wedge \psi) = (\neg\varphi)\vee(\neg\psi)$, $\neg(\varphi\mathcal{U}\psi) =$

[1] The definition of automata here is a variant of the standard definition for reasons that will become clear later. In particular, the labeling here is on the states, and each acceptance set is a set of transitions.

$(\neg\varphi)\,\mathcal{V}\,(\neg\psi)$ and $\neg(\varphi\,\mathcal{V}\,\psi) = (\neg\varphi)\mathcal{U}(\neg\psi)$. The *language* accepted by a temporal logic formula φ, denoted by $\mathcal{L}(\varphi)$, is the set of infinite sequences that satisfy φ. Given a temporal formula φ, we sketch how to construct an automaton $A_\varphi = (X, X_0, \delta^A, \mathcal{F}^A, L^A, 2^{2^{\Pi}})$ such that $\mathcal{L}(A_\varphi) = \mathcal{L}(\varphi)$. The construction here is essentially the one in [5]. Each state of A_φ is labeled with a propositional formula over the variables Π, thus $\Sigma = 2^{2^{\Pi}}$.

With each state x of A_φ, we associate a formula $\eta(x)$, such that for every accepting run $\sigma = x_0, x_1, \ldots$ of A_φ, $\sigma^i \models \eta(x_i)$. The formula $\eta(x)$ is of the form

$$\Big(\bigwedge_{i=1,\ldots,m_x} \nu_i^x\Big) \wedge \Big(\bigwedge_{j=1,\ldots,n_x} \bigcirc\psi_j^x\Big). \tag{1}$$

Note that each temporal formula can be trivially brought into the form of Equation (1), when $m_x = 1$ and $n_x = 0$. We denote by $present(x)$ the set of ν_i^s formulas, and by $next(x)$ the set of ψ_i^x formulas. In addition to $\eta(x)$, we associate with each state a list of *incoming* edges from predecessor states. Nodes (states) are *refined* and *split*, according to the formulas in $present(x)$.

We start the construction with a node x that initially contains the formula to be translated in $present(x)$. The set $next(x)$ is empty, and there is one incoming edge to x marked with *init*. This is a dummy edge, pointing to x, but without any predecessor node. We repeatedly apply to the nodes *refinement* and *splitting*, as follows:

Refinement: If $\chi_1 \wedge \chi_2 \in present(x)$, we add χ_1, χ_2 to $present(x)$. Similarly, if $\chi_1\,\mathcal{U}\,\chi_2 \in present(x)$, we add to $present(x)$ the formula $\chi_2 \vee (\chi_1 \wedge \bigcirc(\chi_1\,\mathcal{U}\,\chi_2))$. If $\chi_1\,\mathcal{V}\,\chi_2 \in present(x)$, we add to $present(x)$ the formula $\chi_2 \wedge (\chi_1 \vee \bigcirc(\chi_1\,\mathcal{V}\,\chi_2))$. Finally, if $\bigcirc\chi \in present(x)$, we add χ to $next(x)$.

Splitting: given a formula of the form $\chi_1 \vee \chi_2$ in $present(x)$, which was not used before for splitting, we split the current node into two nodes x_1 and x_2. Then we set $present(x_1)$ to be $present(x) \cup \{\chi_1\}$ and the value of $present(x_2)$ to be $present(x) \cup \{\chi_2\}$. The set of subformulas $next(x)$ are copied to $next(x_1)$ and $next(x_2)$. We also copy the list of incoming edges from x to x_1 and to x_2.

We stop refining and splitting a node x when all the formulas in $present(x)$ were used. We then add node x to the list X of automaton nodes if there exist no node x' with $present(x) = present(x')$ and $next(x) = next(x')$. Otherwise, i.e., if there exists such a node x', we only update the list of incoming edges in x' by adding the incoming edges of the new node x. After a new node is added to X, we generate a successor node x', and set $present(x')$ to be the $next(x)$ formulas, and $next(x')$ to be empty.

The acceptance conditions $\mathcal{F}^A$ of A_φ guarantees that each $\mathcal{U}$ subformula of φ is *fulfilled*. Thus, for every such subformula $\chi_1\,\mathcal{U}\,\chi_2$ of φ there is an acceptance condition in $\mathcal{F}^A$ that includes all the outgoing edges from nodes x such that either $\chi_2 \in present(x)$, or that $\chi_1\,\mathcal{U}\,\chi_2 \notin present(x)$. The initial states X_0 of A_φ consist of the states $x \in X$ has the incoming edge *init*. The label $L^A(x)$ of a node x is the propositional formula $prop(x)$, defined as the conjunction of

the propositions and negated propositions that appear in $present(x)$ after the construction of x.

Theorem 1 *For every temporal formula φ and automaton A_φ constructed as above, $\mathcal{L}(A_\varphi) = \mathcal{L}(\varphi)$.*

For every state x of A_φ, we define $\mu(x) = \neg\eta(x)$, i.e.,

$$\mu(x) \;=\; (\bigvee_{i=1,\ldots,m_x} \neg\nu_i^x) \vee (\bigvee_{i=1,\ldots,n_x} \bigcirc\neg\psi_i^x)$$

The following lemma follows immediately from the construction:

Lemma 1 *If a node x in the constructed Büchi automaton has n immediate successors, $x_1 \ldots x_n$, then $\eta(x) \rightarrow \bigvee_{i=1,n} \bigcirc\eta(x_i)$. Equivalently, $\bigwedge_{i=1,n} \bigcirc\mu(x_i) \rightarrow \mu(x)$.*

In order to simplify the automatically generated proof, presented in the sequel, we can remove from $\eta(x)$ every formula that causes splitting (e.g., remove $\chi_1 \vee \chi_2$, once χ_1 or χ_2 is added), or refinement. The formula $\mu(x)$ is changed accordingly.

Example. Consider the case where we want to verify the property $\varphi = \Box\Diamond p$, i.e., p happens infinitely often. Then, we translate $\neg\varphi = \Diamond\Box\neg p$. We can rewrite this formula with the $\mathcal{U}$ and $\mathcal{V}$ operators as $true\,\mathcal{U}(false\,\mathcal{V}\,\neg p)$ or change the translation algorithm to deal directly with the operator $\Box$ and $\Diamond$. The automaton obtained in this way appears in Figure 1. In this figure, we translated, for simplicity, the formulas back to the form with $\Box$ and $\Diamond$. We included in each node x the formula $\mu(x)$. There is one accepting set for this automaton, which includes a single transition (x_2, x_2).

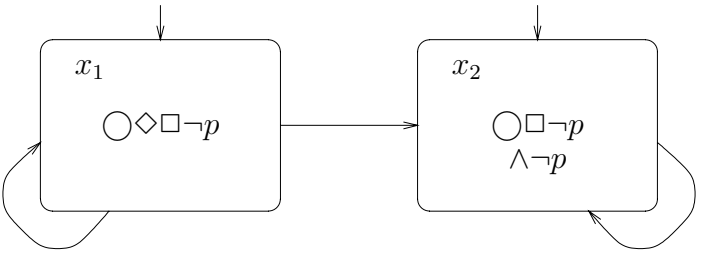


Fig. 1. An automaton for $\neg\varphi = \Diamond\Box\neg p$

We consider finite state systems $P = \langle V, \Theta, \mathcal{T}, \mathcal{F}^P \rangle$ consisting of:

V **-** A set of *system variables*. A *state* of the system provides an interpretation of the system variables V. For a state s and a variable v, we denote by $s[v]$ the value assigned to v by the state s.

Θ - The *initial condition*, which is a propositional formula over the variables V characterizing the initial states.

$\mathcal{T}$ - A set of *transitions formulas* of the form $\tau(V, V')$, relating the values V of the system's variables in state $s \in S$ to the values V' in a τ-successor s'.

$Fair^P$ - A set of *fair conditions*, each element $G \in Fair^P$ is a formula over V and V'.

A computation of the system P is an infinite sequence of states $\sigma = s_0, s_1, \ldots$ satisfying:

- s_0 is initial, i.e., $s_0 \models \Theta$
- For each $i \geq 0$, the state s_{i+1} is the τ-successor of s_i for some $\tau \in \mathcal{T}$. That is, $(s_i, s_{i+1}) \models \tau(V, V')$ where for each $v \in V$, we interpret v as $s_i[v]$ and v' as $s_{i+1}[v]$.
- For every $G \in Fair^P$, $(s_i, s_{i+1}) \models G$ for infinitely many i's.

It is easy to represent weakly fair transitions system as automata: The set of interpretations for the system variables is the set of states S^P. The initial states S_0^P are all the interpretations satisfying Θ. The transition relation δ^P is generated by the formulas in $\mathcal{T}$. Each acceptance condition $F \in \mathcal{F}^P$ corresponds to the transitions that satisfy a fairness condition $G \in Fair^P$. The labeling functions L^P maps each state with an assignment over the set of propositions Π. Note that in some cases we have that $V = \Pi$. A weakly fair transitions system is thus represented by an automaton $(S^P, S_0^P, \delta^P, \mathcal{F}^P, L^P, 2^\Pi)$. Let $succ(s) = \{s' | (s, s') \in \delta^P\}$, i.e., the set of successors of node s. Denote by $s \longrightarrow \{s_1, \ldots, s_n\}$ the fact that s has, according to the automaton P, exactly n successors, $s_1, \ldots, s_n$.

For a finite state system P and a temporal formula, we say that φ *is valid over* P, or that φ *is* P*-valid*, denoted by $P \models \varphi$, if for every fair computation σ of P, $\sigma \models \varphi$.

We are mainly interested in concurrent systems over a set processes $\mathcal{P}$. Each edge in δ^P corresponds to a transition executed by one or more processes in P. Under weak process fairness [8], each acceptance condition F in $\mathcal{F}^P$ corresponds to a process. It contains all the edges that exist from any state in which that process is disabled and all the edges that correspond to the execution of an atomic transition by this process.

Example. Consider a system with two processes, competing on getting to a critical section. Each process in $\mathcal{P} = \{T_1, T_2\}$ consists of three transitions. There is one Boolean variable *turn*, which arbitrates among the processes, to resolve the case where both want to enter the critical section. The program counter of each process pc_i can be in one of the following labels:

nt_i The process T_i is currently not trying to enter its critical section.
tr_i The process T_i is trying to enter its critical section.
cr_i The process T_i is in its critical section.

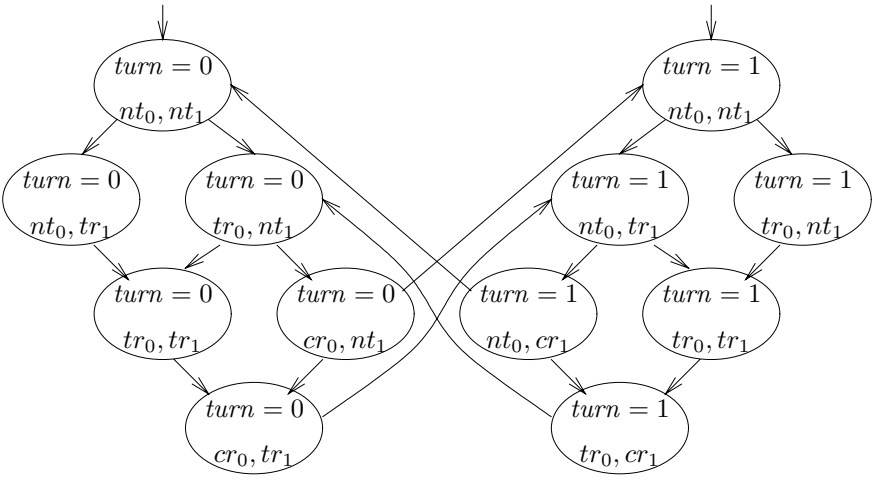


Fig. 2. A mutual exclusion system

There are three transitions for each process T_i:

$$t1_i = pc_i = nt_i \wedge {pc_i}' = tr_i \wedge {pc_{1-i}}' = pc_{1-i} \wedge turn' = turn$$
$$t2_i = pc_i = tr_i \wedge {pc_i}' = cr_i \wedge turn = i \wedge pc'_{1-i} = pc_{1-i} \wedge turn' = turn$$
$$t3_i = pc_i = cr_i \wedge {pc_i}' = nt_i \wedge pc'_{1-i} = pc_{1-i} \wedge turn' = 1 - turn$$

The initial condition is $\Theta : pc_0 = nt_0 \wedge pc_1 = nt_1$. The state space of this system is presented in Figure 2. For simplicity of the presentation, we do not impose any fairness constraint on this system (in fact, in this case, fairness would not make any difference). For a state $s \in S$ and a temporal formula φ, denote by $(P, s) \hookrightarrow \eta$, the fact that there exists a suffix σ of a sequence accepted by the system automaton P, which starts with the state s, such that $\sigma \models \eta$. Denote by $(P, s) \models \mu$ the fact that for every suffix σ of a sequence in P, which starts with the state s, $\sigma \models \mu$. We often omit P, when it is known from the context, and write $s \models \mu$ instead of $(P, s) \models \mu$. Note that $\hookrightarrow$ and $\models$ are dual relations, since $(P, s) \models \mu$ iff $(P, s) \not\hookrightarrow \neg\mu$.

3 Checking the Validity of a Formula over a Program

Our approach can be used to establish a proof that $\sigma \models \varphi$ for every sequence σ accepted by the automaton P. We denote that by $P \models \varphi$.

In order to verify that φ is P-valid, we build an *intersection automaton* that accepts $\mathcal{L}(\neg\varphi) \cap \mathcal{L}(P)$, and show it to be empty. Let $A_{\neg\varphi}$ be the automaton $(X, X_0, \delta^A, \mathcal{F}^A, L^A, 2^{2^\Pi})$, which accepts $\mathcal{L}(\neg\varphi)$. Let P be the system automaton $(S^P, S_0^P, \delta^P, \mathcal{F}^P, L^P, 2^\Pi)$. The *product* automaton, that accepts $\mathcal{L}(P) \cap \mathcal{L}(\neg\varphi)$, is $A^P_{\neg\varphi} = (S, S_0, \delta, \mathcal{F}, L, 2^\Pi)$, where

1. $S = X \times S^P$.
   In the intersection graph of $A^P_{\neg\varphi}$, we distinguish two kinds of nodes: *success nodes* of the form (x, s), where $L^P(s) \in L^A(x)$, i.e., the propositional assignment of s satisfies the propositional formula $prop(x)$, and *failed nodes*, otherwise.
2. $S_0 = X_0 \times S_0$.
3. $((x, s), (x', s')) \in \delta$ iff $(x, x') \in \delta^A$, $(s, s') \in \delta^P$ and $L^P(s) \in L^A(x)$. That is, the transition relation agrees with transition relations of both the system and the property automata. Moreover, there is a transition from a state $(x, s) \in S$ only if it is a success node.
4. The accepting condition consists of the acceptance sets of both automata. Formally, we define an operator $\bowtie$ such that for every sets A, B, $C \subseteq A \times A$, and $D \subseteq B \times B$,
$$C \bowtie D \;=\; \bigcup_{(a,a') \in C; (b,b') \in D} \big((a, b), (a', b')\big).$$
   Then, $\mathcal{F} \;=\; \{(X \times X) \bowtie F | F \in \mathcal{F}^P\} \cup \{F \bowtie (S^P \times S^P) | F \in \mathcal{F}^A\}$.
5. $L(x, s) = L^P(s)$; i.e., the labeling of each state is its labeling in the system automaton.

In order to check that $\mathcal{L}(A^P_{\neg\varphi}) = \emptyset$, it suffices to check that, in the graph defined by $A^P_{\neg\varphi}$, there is no path leading from S_0 to a strongly connected component (SCC) that intersects each of the sets in $\mathcal{F}$. An immediate implication of the construction is:

Theorem 2 *Assume $\mathcal{L}(A^P_{\neg\varphi}) = \emptyset$. Then for every initial state (x_0, s_0) of $A^P_{\neg\varphi}$, $(P, s_0) \models \neg\varphi$. Thus, $P \models \bigwedge_{(x_0,s_0)\in S_0} \mu(x_0)$.*

Example. Consider the system in Figure 2. We want to prove for it the property $\Box\Diamond(cr_0 \vee cr_1)$. The property automaton construction is similar to the one in Figure 1, except that we replace p with $cr_0 \vee cr_1$ (hence, we replace $\neg p$ with $\neg cr_0 \wedge \neg cr_1$). The intersection of the property automaton from Figure 1 with the state space in Figure 2 is shown in Figure 3.

4 Constructing a Temporal Logic Proof

In this approach, we transform $A^P_{\neg\varphi}$ into a temporal proof formula. As a preparatory step, we perform Tarjan's algorithm on the the intersection graph for $A^P_{\neg\varphi}$, obtaining the strongly connected components.

There is a naturally induced partial order $\prec$ between the strongly connected components such that $C \prec C'$ if there is an edge from some node in C to some node in C'. In the proof, we need to complete the proof related to all the components C' such that $C \prec C'$, before we start dealing with C.

In this sound and complete proof system there are four kinds of correctness assertions:

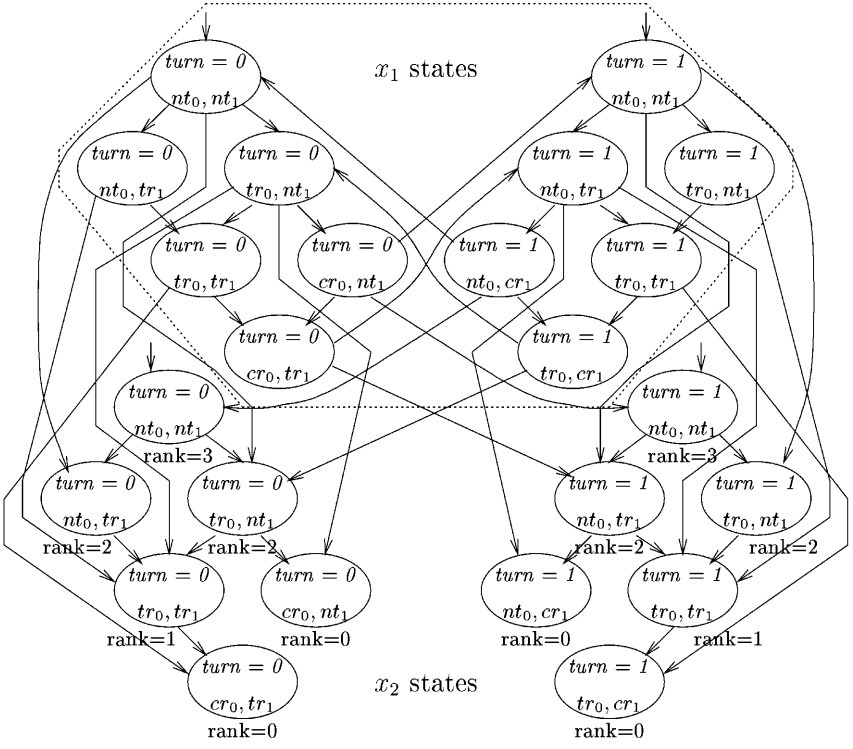


Fig. 3. The intersection of the property automaton in Figure 1 and state space in Figure 2

Failure axiom FAIL**.** Let (x, s) be a failed node. Then we can conclude that $s \models \mu(x)$.

The justification for this axiom is simple: the node has failed because we have checked the assignment of the state s against the propositional claim in x, and the propositional claim has failed to hold. Thus, $s \models \neg prop(x)$. But, note that $\neg prop(x) \rightarrow \mu(x)$.

Successors rule SUCC**.** Let (x, s) be a successful node, such that x has n successors $x_1, \ldots, x_n$, and s has m successors $s_1, \ldots, s_m$. Then we have

$$\frac{s \longrightarrow \{s_1, \ldots, s_m\} \quad \text{For each } 1 \leq i \leq m,\ s_i \models \bigwedge_{j=1,n} \mu(x_j)}{s \models \mu(x)}$$

The validity of this proof rule (see Also [2]) stems from the correctness of the construction. In particular, Lemma 1. Note that in fact, the failure axiom can be seen as a special case of the successors rule, with no premises.

Induction IND**.** Let C be a strongly connected component in $A^P_{\neg\varphi}$. Let $Exit(C)$ be the set of nodes not in C, with an incoming arrow from a node in C. Assume first that the SCC C does not satisfy at least one acceptance condition that stems from the property automaton. That is, there exists at least one acceptance set $F \in \mathcal{F}^P$ such that none of the edges in F is in C.

$$\frac{\begin{array}{c}\text{For each } (x,s) \in Exit(C),\ s \models \mu(x) \\ \text{For each } (x,s) \in C,\ s \longrightarrow succ(s)\end{array}}{\text{For each } (x,s) \in C,\ s \models \mu(x)}$$

In case that the SCC C satisfies all the acceptance conditions that stem from the property automaton, it must not satisfy at least one condition that stems from the system fairness (otherwise, the intersection would not be empty). In this case, we need to add a premise of the following form:

$$\bigwedge_{F \in \mathcal{F}^P} \Box\Diamond \Gamma_F$$

where Γ_F is a formula describing the pairs of adjacent states that belong to F.

Conjunction rule CONJ**.** This rule allows conjoining any pair of conclusions made on a given state, and making temporal logic interferences (we assume for the third premise a given sound and complete propositional temporal logic).

$$\frac{s \models \varphi_1,\ s \models \varphi_2,\ (\varphi_1 \wedge \varphi_2) \rightarrow \varphi}{s \models \varphi}$$

We can now obtain the formal temporal proof, showing that if $\mathcal{L}(A^P_{\neg\varphi}) = \emptyset$, then each initial state $s \in S^P_0$ satisfies $s \models \varphi$, i.e., $P \models \varphi$. This is given according to the following steps:

1. Translate $\neg\varphi$ into an automaton $\mathcal{A}_{\neg\varphi}$, according to the above algorithm. Construct the intersection graph $\mathcal{A}^P_{\neg\varphi}$.
2. Apply Tarjan's DFS to $\mathcal{A}^P_{\neg\varphi}$. Find the SCCs.
3. If all the SCCs are discarded, goto Step 9.
4. Select a strongly connected component C that is not yet discarded, such that all the SCCs C' such that $C \prec C'$ were discarded.
5. If C consists of a single node (x,s) that has no successor (a leaf), then it must be a failure node. Apply the rule FAIL.
6. If C is a trivial SCC, i.e., contains a single node (x,s) without a self loop, but with successors, apply $Succ(x,s)$.
7. If C is a nontrivial SCC, apply IND. Note that some successors of the nodes of C are outside of the SCC, namely in $Exit(C)$. These nodes were handled previously.
8. Discard C and goto Step 3.

9. Let s be an initial state of P, and $x_1, \ldots, x_n$ be all the states such that (x_i, s) is a node in the intersection. Then apply $n-1$ times the rule CONJ to obtain that $s \models \varphi$.

Example. Consider again the property automaton in Figure 1. We have $\mu(x_1) = \bigcirc\Box\Diamond p$ and $\mu(x_2) = p \vee \bigcirc\Diamond p$. Consider the simple system P shown in Figure 4.

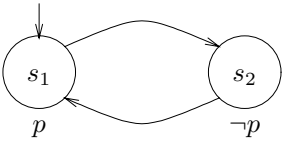


Fig. 4. A simple system

The intersection of the two automata appears in Figure 5. There are three strongly connected components:

1. $\{(x_1, s_1), (x_1, s_2)\}$.
2. $\{(x_2, s_2)\}$.
3. $\{(x_2, s_1)\}$.

The latter two components are trivial. Moreover, the last one consists of a failed node. We obtain the following proof:

1. Using the FAIL axiom on the failed node (x_2, s_1), we obtain
   $s_1 \models p \vee \bigcirc\Diamond p$.
2. Applying *Succ* to the node (x_2, s_2), we obtain
   $s_2 \longrightarrow \{s_1\}$
   $\underline{s_1 \models p \vee \bigcirc\Diamond p}$
   $s_2 \models p \vee \bigcirc\Diamond p$
3. Applying line 1 as a premise to line 2, we obtain
   $s_2 \models p \vee \bigcirc\Diamond p$.
4. We apply the rule IND to the only strongly connected component in the graph, $C = \{(x_1, s_1), (x_1, s_2)\}$, where $Exit(C) = \{(x_2, s_2), (x_2, s_1)\}$. We obtain
   $s_1 \models p \vee \bigcirc\Diamond p$
   $s_2 \models p \vee \bigcirc\Diamond p$
   $s_1 \longrightarrow \{s_2\}$
   $\underline{s_2 \longrightarrow \{s_1\}}$
   $s_1 \models \bigcirc\Box\Diamond p$
   $s_2 \models \bigcirc\Box\Diamond p$
5. Applying lines 1, 3 as premises to line 4, we obtain
   $s_1 \models \bigcirc\Box\Diamond p$ and $s_2 \models \bigcirc\Box\Diamond p$

6. Using the rule Conj, we obtain
   $s_1 \models p \vee \bigcirc\Diamond p$
   $s_1 \models \bigcirc\Box\Diamond p$
   $\underline{((p \vee \bigcirc\Diamond p) \wedge \bigcirc\Box\Diamond p) \rightarrow \Box\Diamond p}$
   $s_1 \models \Box\Diamond p$
7. Applying lines 1, 5 as premises to line 6, we obtain
   $s_1 \models \Box\Diamond p$.

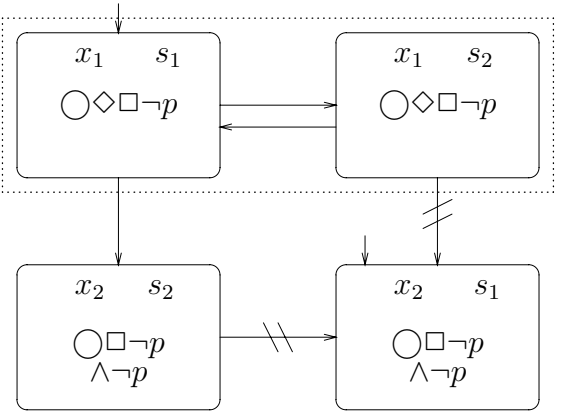


Fig. 5. The intersection graph $\mathcal{A}^P_{\neg\varphi}$

Consider the mutual exclusion system from Figure 2, and the checked property $\Box\Diamond(cr_0 \vee cr_1)$. The intersection graph appears in Figure 3. It includes a nontrivial SCC, whose states are encapsulated with a dotted line. These are the system states that are paired with the property automaton state x_1. The rest of the states are paired with x_2. There are four failed nodes, which have no successors.

The proof shows how the system progresses towards satisfying $\Diamond(cr_0 \vee cr_1)$. Consider first the nodes outside the nontrivial SCC. From each such node, we progress into a failed node where either cr_0 or cr_1 hold. The proof proceeds from the failed nodes backwards, showing that for each node that contains an x_2 component, we have $s \models \Diamond(cr_0 \vee cr_1)$. The nontrivial SCC provides an induction over the execution of the system, hence obtaining $P \models \Box\Diamond(cr_0 \vee cr_1)$. Note that we need only one application of the rule Ind here, since there is only one nontrivial SCC.

5 An Automatic Ranking Function

Together with the proof, we can construct a ranking function ρ that maps each strongly connected component in the intersection graph into a natural number. The ranking of an SCC measures the distance of the component from the bottom of the induced directed acyclic graph of SCCs. The ranking function provides additional intuition about the way the checked system progresses towards satisfying the checked property. In term of the constructed proof, it shows how the proof progresses towards the failed nodes. We denote by $\rho(C)$ the rank of an SCC C, and further define $\rho(t) = \rho(C)$ if $t \in C$.

The ranking function ρ must satisfy the following condition: if $C \prec C'$ then $\rho(C) > \rho(C')$. In particular, if C is a trivial SCC consisting of a failed node, it is often convenient to set $\rho(C) = 0$.

Let m be the number of bits in the binary representation of the largest ranking constructed for a state in $A^P_{\neg\varphi}$. For any integer $r \leq 2^m - 1$, let $bit_i(r)$ be the i^{th} bit in the m-bit binary representation of r. Let V_{aug} be the set V of system variables, augmented with a new variable whose value ranges over X, the set of $A_{\neg\varphi}$-states. This augmentation is needed since the same system state may have a different rank when combined with a different property automaton state.

For every rank r that is constructed, let $states_r$ be a (propositional) formula over the state variables V_{aug} that describes the set of *all* (joint) states (x, s) such that $\rho((x, s)) = r$. Note that $states_r$ can be obtained automatically from the above ranking construction.

The following formula,$\gamma(V_{aug}, r_1, \ldots, r_m)$ represents the connection between states and their ranking.

$$\gamma = \bigvee_{r=1,\ldots,2^m\}} (states_r \wedge \bigwedge_{i=1}^{m} (r_i = bit_i(r)))$$

Now, we can express each one of the bits in the binary representation of the ranking as a formula of the joint automata state.

$$\beta_i = \exists r_1, r_2, \ldots, r_m \, (\gamma \wedge r_i)$$

Thus, β_i is true only in joint states for which the ith bit of the ranking function is 1. Finally, we can automatically obtain a formula $\hat{\rho}$ as the ranking function, mapping a state (x, s) (or more precisely, the corresponding assignments to V_{aug}) to its ranking $\rho((x, s))$. We assume that we can use in such a formula Boolean expressions that return the value 0 for *false* and 1 for *true*. A ranking formula $\hat{\rho}(V_{aug})$ can be obtained as follows:

$$\hat{\rho} = \sum_{i=1,\ldots,m} 2^{i-1} \times \beta_i.$$

Note that the formulas β_i can be simplified, e.g., using the help of a BDD package.

Example. Consider for example a ranking for the intersection in Figure 3.

State formula	*rank r*	$bit_1(r)$	$bit_0(r)$
$cr_0 \vee cr_1$	0	0	0
$tr_0 \wedge tr_1$	1	0	1
$(tr_0 \wedge nt_1) \vee (nt_0 \wedge tr_1)$	2	1	0
$nt_0 \wedge nt_1$	3	1	1

It is easy to see that the least significant bit of the binary representation, β_0 can be expressed as $(tr_0 \wedge tr_1) \vee (nt_0 \wedge nt_1)$. The most significant bit β_1 can be expressed as $(nt_1 \wedge tr_0) \vee (nt_0 \wedge tr_1) \vee (nt_0 \wedge nt_1)$.

The ranking function formula $\hat{\rho}$ requires considerable simplification, before presenting the resulted expression to the user. For that, we can exploit the following options:

- We can use some additional conditions, regarding the relationship between the system variables. For example, in the mutual exclusion system, we can add the following conditions: $(nt_0 \vee tr_0 \vee cr_0) \wedge (nt_1 \vee tr_1 \vee cr_1) \wedge \neg(nt_0 \wedge tr_0) \wedge \neg(nt_0 \wedge cr_0) \wedge \neg(tr_0 \wedge cr_0) \wedge \neg(nt_1 \wedge tr1) \wedge \neg(nt_1 \wedge cr_1) \wedge \neg(tr_1 \wedge cr_1)$.
- The definition of the ranking function gives us some freedom in assigning the actual ranks to SCCs (and their nodes). For example, in the mutual exclusion system, we decided to give a ranking of 0 to all the failure nodes. However, we could have decided to give a ranking of 0 to the nodes where $(cr_0 \wedge tr_1) \vee (tr_0 \wedge cr_1)$, and a ranking of 1 to the nodes where $(cr_0 \wedge nt_1) \vee (nt_0 \wedge cr_1)$.
- After finding the expressions for the different bits, we can attempt to collect together similar terms in order to simplify the ranking formula $\hat{\rho} = \sum_{i=1,\ldots,m} 2^{i-1} \times \beta_i$. Terms that appear in different β_i's should be grouped together, with their multiplication constants added together.

6 Conclusions

Model checking is mostly identified with finding errors. We presented an algorithm for the automatic construction of a proof that the checked property holds in the verified system. The proof is automatically obtained directly from a graph (of an automaton) that is generated by model checking. Such a proof may help in gaining more intuition about the verified system. The ability to automatically form such a proof can be further exploited to conclude new properties of the verified system.

The algorithm presented here can be added to model checking systems that are based on automata theory. In particular, the SPIN system [6] contains an implementation of the LTL translation algorithm in [5].

Acknowledgements. We would like to thank Elsa Gunter and Amir Pnueli for inspiring discussions about this subject. David Long has provided help with his OBDD package. It was brought to our attention that, in parallel with our work, Kedar Namjoshi has developed an algorithm for the automatic generation of proofs for the μ-calculus [9].

References

1. E. M. Clarke, E. A. Emerson, Design and synthesis of synchronization skeletons using branching time temporal logic. Workshop on Logic of Programs, Yorktown Heights, NY, 1981, LNCS 131, Springer-Verlag.
2. G. Bhat, R. Cleaveland, O. Grumberg, Efficient on-the-fly model checking for CTL*. Logic in Computer Science, 1995, San Diego, CA, 388-397
3. E. M. Clarke, O. Grumberg, D. Peled, *Model Checking*, MIT Press, 2000.
4. E. A. Emerson, E. M. Clarke, Characterizing correctness properties of parallel programs using fixpoints, LNCS 85, Springer Verlag, Automata, Languages and Programming, July 1980, 169–181.
5. R. Gerth, D. Peled, M. Y. Vardi,, P. Wolper, Simple on-the-fly automatic verification of linear temporal logic, Protocol Specification Testing and Verification, 3–18, Warsaw, Poland, 1995. Chapman & Hall.
6. G. J. Holzmann, *Design and Validation of Computer Protocols*, Prentice-Hall Software Series, 1992.
7. R.P. Kurshan. *Computer-Aided Verification of Coordinating Processes: The Automata-Theoretic Approach*. Princeton University Press, Princeton, New Jersey, 1994.
8. Z. Manna, A. Pnueli, *The Temporal Logic of Reactive and Concurrent Systems: Specification*, Springer, 1991.
9. K. Namjoshi, Certifying model checkers, Submitted to CAV 2001.
10. M. Y. Vardi, P. Wolper, An automata-theoretic approach to automatic program verification. Proc. 1st Annual Symposium on Logic in Computer Science IEEE, 1986.

Model Checking if Your Life Depends on It: A View from Intel's Trenches

Rob Gerth

Strategic CAD Laboratories (SCL), Intel corp., USA
rob.t.gerth@intel.com

Abstract. Hardware design is considered one of the traditional areas for formal (property) verification (FPV); in particular for symbolic model checking. Indeed, Intel, SUN, Motorola and IBM all develop and deploy model checking tools to ensure design correctness. On the other hand, hardware design is hostile territory because of the huge effort companies traditionally invest in classical testing and validation techniques which tend to be much more automated, require less sophistication from users and which will, in fact, discover many errors when deployed on such a scale. For these reasons FPV will never fully supplant traditional validation.

The real challenge lies in convincing processor design teams that diverting some of their validation resources to FPV leads to provably higher design quality. Complicating factors include the relatively high quality of traditional validation—at least within Intel—which raises the bar for FPV; and the fact that in high-performance processors design large parts of the RTL tends to remain unstable enough throughout the design to make it very hard to verify suitable micro-architectural abstractions. For these reasons, FPV within Intel traditionally targets the same RTL from which the schematics is derived and on which all traditional validation is performed.

Arguably the biggest (public) FPV success story within Intel is that of floating point hardware verification and it is illustrative to see how FPV is deployed in this area and how tool and methodology development is influenced.

The major challenge that we are now facing is to move formal verification upstream in the design flow. Not only because ever increasing micro-architectural complexity creates a strong demand for early verification on an algorithmic level, but also because the ever shortening design cycle forces formal verification to start much earlier in the design. It is here that hardware FPV and software verification start to merge and there are lessons to be learned for either side.

M.B. Dwyer (Ed.): SPIN 2001, LNCS 2057, p. 15, 2001.

Model-Checking Infinite State-Space Systems with Fine-Grained Abstractions Using SPIN

Marsha Chechik, Benet Devereux, and Arie Gurfinkel

Department of Computer Science, University of Toronto,
Toronto, ON M5S 3G4, Canada.
{chechik,benet,arie}@cs.toronto.edu

Abstract. In analyzing infinite-state systems, it is often useful to define multiple-valued predicates. Such predicates can determine the (finite) levels of desirability of the current system state and transitions between them. We can capture multiple-valued predicates as elements of a logic defined over finite total orders (FTOs). In this paper we extend automata-theoretic LTL model-checking to reasoning about a class of multiple-valued logics. We also show that model-checking over FTOs is reducible to classical model-checking, and thus can be implemented in SPIN.

1 Introduction

Currently, model-checking is essentially limited to reasoning about medium-sized finite-state models. Reasoning about large models, especially if these are not finite-state, is typically done using abstraction [CGL94]. Abstraction techniques, such as *abstract interpretation* [CC77], require the user to supply the mapping between concrete and abstract data types in their models. *Predicate abstraction*, introduced by Graf and Saidi [GS97], is a form of abstraction specified as a number of predicates over the concrete data. For example, if we are interested in checking whether x is always positive, we can define predicates $x > 0$ and $x \leq 0$, and use them to compute the abstract system. A number of researchers, e.g., [CU98,VPP00,BDL96,DDP99,SS99], explored the use of predicate abstraction.

However, boolean predicates often do not give the desired precision. For example, consider reasoning about a leader-election protocol, parameterized by N – the number of processes engaged in it. We can either set N to be a (small) constant, and define predicates on the exact number of processes that have agreed on the elected leader; or leave N as is, and define predicates such as "everyone agreed on the leader", "no one agreed on the leader", etc. In this situation we cannot ask questions about the likelihood of the agreement, whereas such questions may be desirable.

As an alternative, we propose modeling such systems using multiple-valued predicates, where their values form a linear order. In the above situation, we can assign different values to the level of agreement on the leader: "everyone

M.B. Dwyer (Ed.): SPIN 2001, LNCS 2057, pp. 16–36, 2001.

agreed", "the agreement is likely", "no information is available", "the agreement is unlikely", "no one agreed", obtaining a linear order on the level of agreement. Furthermore, if we do not limit ourselves to classical logic, our model-checking procedure will distinguish between different values of agreement, e.g., between cases where no agreement has been reached and where complete agreement has not been reached, but the majority have agreed. Taking this reasoning one step further, we can assign values to transitions. Intuitively, a transition value is the *possibility* that it will be taken. Thus, we can potentially distinguish between paths that can always be taken, paths that can likely be taken, etc.

In fact, giving predicates values from a linear order can be useful in a variety of situations: (a) consensus-building, where the abstraction is over counting (e.g., the leader-election protocol mentioned above); (b) explicitly distinguishing between "regular" and "faulty" behaviors, where we may be interested in properties that hold always, and those that hold "most of the time", i.e., over "regular" behaviors; (c) rechecking a partial system after a change to it has been made, where we are interested in differentiating between possible effects of the change; (d) any situation where we want to assign "desirability" to a transition. This can happen in cases where we have varying tolerances, e.g., in analyzing families of SCR specifications [HJL96].

Note that using linear order-valued predicates does not increase the expressive power of our modeling language, since they can be encoded using a number of boolean predicates. However, such encoding results in cluttering the models with lots of auxiliary variables that bear no natural meaning, and, more importantly, greatly increases the sizes of the models, making model-checking less feasible [HK93].

Multiple-valued reasoning has been explored in a variety of domains. For example, a nine-valued logic is prescribed as a standard [IEE93] for VLSI design, where the interpretation of values is in terms of voltage thresholds. Other examples include databases [Gai79], knowledge representation [Gin87], and machine learning [Mic77]. However, most of the work concentrated on the 3-valued reasoning, with values "True", "Maybe" and "False". Melvin Fitting [Fit91,Fit92] has done seminal work in studying 3-valued modal logic, and our work on logic in this paper is somewhat similar to his. Three-valued logic has also been shown to be useful for analyzing programs using abstract interpretation [CD00,SRW99], and for analyzing partial models [BG99,BG00]. Bruns and Godefroid also proved that automata-theoretic model-checking on 3-valued predicates reduces to classical model-checking.

In this paper we give semantics to automata-theoretic model-checking over arbitrary finite linear orders. We define multiple-valued Büchi automata and multiple-valued LTL and show that such model-checking reduces to a classical problem, and thus can be implemented on top of SPIN. The rest of this paper is organized as follows: we review the definition of linear orders and define multiple-valued sets and relations over them in Section 2. χLTL, a multiple-valued extension of LTL, is defined in Section 3. Section 4 defines multiple-valued languages and Büchi automata. In Section 5 we show how to represent χLTL logic formulas

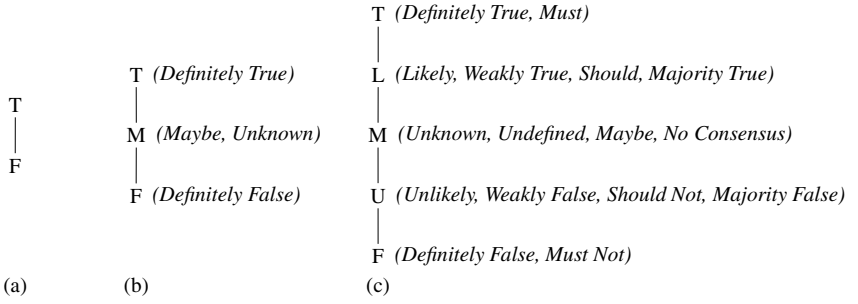


Fig. 1. (a) **2**, the classical logic FTO; (b) **3**, a three-valued logic FTO; (c) **5**, a five-valued logic FTO and possible interpretations of its values.

as multiple-valued Büchi automata. Section 6 defines the model-checking problem on multiple-valued Büchi automata and shows that it reduces to a number of queries to a classical model-checker, such as SPIN. We conclude the paper in Section 7.

2 Preliminaries

In this section we review the definition of logics based on total orders. We also define multiple-valued sets and relations over them.

2.1 Finite Total Orders

A *partial order* is a relation which is reflexive, symmetric, and transitive. A *partially ordered set*, usually abbreviated *poset*, is a pair $\mathcal{L} = (\mathcal{O}, \sqsubseteq)$ where $\mathcal{O}$ is a set and $\sqsubseteq$ a partial order defined on it. If, for all $a, b \in \mathcal{O}$, either $a \sqsubseteq b$ or $b \sqsubseteq a$, then $\sqsubseteq$ is a *total order* or *linear order*. We consider, for the purposes of this paper, only finite totally ordered sets, which we refer to as FTOs.

The operations of maximum and minimum are defined on FTOs as follows:

$$\begin{array}{llll} a \sqcap b = a \Leftrightarrow a \sqsubseteq b & \text{(minimum)} & a \sqcup b = b \Leftrightarrow a \sqsubseteq b & \text{(maximum)} \\ b \sqcap a = a \Leftrightarrow a \sqsubseteq b & \text{(minimum)} & b \sqcup a = b \Leftrightarrow a \sqsubseteq b & \text{(maximum)} \end{array}$$

Lemma 1. *Let $(\mathcal{O}, \sqsubseteq)$ be an FTO. Then for all $a, b, c \in \mathcal{O}$,*

$$\begin{array}{ll} c \sqsubseteq a \sqcap b \Leftrightarrow (c \sqsubseteq a) \wedge (c \sqsubseteq b) & (\texttt{min-}\wedge) \\ c \sqsubseteq a \sqcup b \Leftrightarrow (c \sqsubseteq a) \vee (c \sqsubseteq b) & (\texttt{max-}\vee) \end{array}$$

We further define $\bot = \bigsqcap \mathcal{O}$ and $\top = \bigsqcup \mathcal{O}$.

Any FTO of height n is isomorphic to the integers from 0 to $(n-1)$ with the ordinary ordering. We call this isomorphism the *canonical isomorphism for the FTO* and denote it by $\zeta_{\mathcal{L}}$. The *difference* between two elements of an FTO is their absolute difference:

$$a \ominus b = |\zeta_{\mathcal{L}}(a) - \zeta_{\mathcal{L}}(b)|$$

and *negation* in the FTO can be defined in terms of difference:

$$\neg a \triangleq \top \ominus a \qquad \text{(def. of negation)}$$

FTOs with this definition of negation satisfy the following properties:

$$\begin{array}{rlrl}
\neg(a \sqcap b) = \neg a \sqcup \neg b & \text{(De Morgan)} & \neg\neg a = a & (\neg \text{ involution}) \\
\neg(a \sqcup b) = \neg a \sqcap \neg b & & a = b \Leftrightarrow \neg a = \neg b & (\neg \text{ bijective}) \\
\neg\bot = \top & (\bot \text{ negation}) & \neg\top = \bot & (\top \text{ negation}) \\
a \sqsubseteq b \Leftrightarrow \neg a \sqsupseteq \neg b & (\neg \text{ antimonotonic}) & &
\end{array}$$

In this paper we use multiple-valued logics whose truth values form an FTO. Conjunction and disjunction of the logic are defined as $\sqcap$ and $\sqcup$ (meet and join) operations of $(\mathcal{O}, \sqsubseteq)$, respectively, and negation is defined as the $\neg$ operator of $(\mathcal{O}, \sqsubseteq)$. In fact, we will not distinguish between an FTO and a logic it defines, referring to both as $\mathcal{L}$. We also note that most of the usual laws of logic are obtained in $\mathcal{L}$, with the exception of the laws of Universality $(a \sqcap \neg a = \bot)$ and Excluded Middle $(a \sqcup \neg a = \top)$.

Figure 1 presents several commonly-used FTOs: classical logic, a three-valued logic with uncertainty, and a five-valued logic with more degrees of uncertainty.

2.2 Multiple-Valued Sets and Relations

Let $\mathcal{L} = (\mathcal{O}, \sqsubseteq)$ be an FTO and D be some (finite) domain. We consider $\mathcal{O}^D$, the set of all total functions from D into $\mathcal{O}$, and refer to elements of $\mathcal{O}^D$ as *multiple-valued subsets of* D, and, when D is clear from the context, just *multiple-valued sets* or MV-sets. We introduced this notion in [CDE01a], and briefly review it below.

Definition 1. *Given multiple-valued sets* $A, B \in \mathcal{O}^D$,

$$\begin{array}{rll}
x \in_{\mathcal{L}} A \triangleq A(x) & & \text{(MV-set membership)} \\
x \in_{\mathcal{L}} A \cup_{\mathcal{L}} B \triangleq A(x) \sqcup B(x) & & \text{(MV-set union)} \\
x \in_{\mathcal{L}} A \cap_{\mathcal{L}} B \triangleq A(x) \sqcap B(x) & & \text{(MV-set intersection)} \\
x \in_{\mathcal{L}} \overline{A} \triangleq \neg A(x) & & \text{(MV-set complement)}
\end{array}$$

Consider the multiple-valued set of Figure 2. In this example, we use the three-valued FTO **3** to model ambiguity about whether 0 is a positive integer. The MV-set is $\mathbb{Z}^{+?} \in \mathbf{3}^{\mathbb{Z}}$, where for all $n \geq 1$, $(n \in_{\mathcal{L}} \mathbb{Z}^{+?}) = \top$; for all $n \leq -1$, $(n \in_{\mathcal{L}} \mathbb{Z}^{+?}) = \bot$; and $(0 \in_{\mathcal{L}} \mathbb{Z}^{+?}) = \mathrm{M}$.

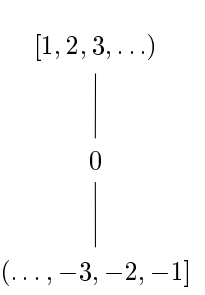


Fig. 2. An example of a multiple-valued set over **3**.

Theorem 1. *[Gol99] Let D be a finite set, and $(\mathcal{O}, \sqsubseteq)$ be an FTO. Define $\sqsubseteq_{\mathcal{O}^D}$ as follows for any $f, g \in \mathcal{O}^D$:*

$$f \sqsubseteq_{\mathcal{O}^D} g \Leftrightarrow \forall d \in D \cdot f(d) \sqsubseteq g(d).$$

Then $(\mathcal{O}^D, \sqsubseteq_{\mathcal{O}^D})$ is an FTO, with MV-union and MV-intersection defined as join and meet, respectively.

The practical use of this result is that all of the properties defined for FTOs, such as the De Morgan rules and distributivity, carry over to MV-sets.

Given two sets P, Q, we can define a *multiple-valued relation* [CDE01a] on them as a multiple-valued subset of $P \times Q$, or an element of $\mathcal{O}^{P \times Q}$.

This work is, to our knowledge, the first use of valued subsets in formal verification; however, such theories are developed elsewhere [Eil78,Gol99].

3 χLTL

In this section we extend the semantics of LTL to allow reasoning over a given FTO $\mathcal{L} = (\mathcal{O}, \sqsubseteq)$, representing our multiple-valued logic. We refer to the resulting language as χLTL. Just like in classical propositional LTL, formulas in χLTL are built from a set *Prop* of values of atomic propositions and are closed under the application of propositional operators, the unary temporal connective $\circ$ ("next") and the binary temporal connective $\mathcal{U}$ ("until"). χLTL is interpreted over multiple-valued computations. A computation is a function $\pi : \mathbb{N} \to \mathcal{O}^{Prop}$ which assigns values from the logic $\mathcal{L}$ to the elements of *Prop* at each time instant (natural number). For a computation π and a point $i \in \mathbb{N}$, we have:

$$\begin{aligned}
&\pi, i \models_{\mathcal{L}} p && \triangleq p \in_{\mathcal{L}} \pi(i) \\
&\pi, i \models_{\mathcal{L}} \neg\varphi && \triangleq \neg(\pi, i \models_{\mathcal{L}} \varphi) \\
&\pi, i \models_{\mathcal{L}} \varphi \wedge \psi && \triangleq \pi, i \models_{\mathcal{L}} \varphi \sqcap \pi, i \models_{\mathcal{L}} \psi \\
&\pi, i \models_{\mathcal{L}} \varphi \vee \psi && \triangleq \pi, i \models_{\mathcal{L}} \varphi \sqcup \pi, i \models_{\mathcal{L}} \psi
\end{aligned}$$

$$\begin{aligned}
\varphi \,\mathcal{U}\, \psi &= \psi \vee (\varphi \wedge \circ(\varphi \,\mathcal{U}\, \psi)) \\
\Diamond\psi &= \psi \vee \circ(\Diamond\psi) \\
\Box\psi &= \psi \wedge \circ(\Box\psi) \\
\varphi \,\mathcal{R}\, \psi &= \psi \wedge (\varphi \vee \circ(\varphi \,\mathcal{R}\, \psi))
\end{aligned}$$

Fig. 3. Properties of χLTL operators.

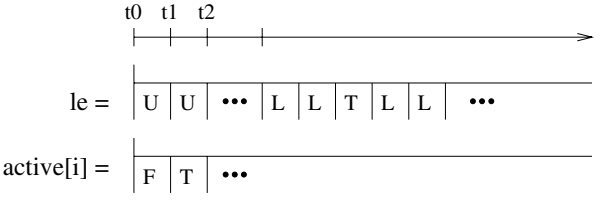


Fig. 4. A partial execution of the Leader Election protocol.

Now we define the temporal operators:

$$\begin{aligned}
\pi, i \models_{\mathcal{L}} \circ\varphi &\triangleq \pi, i+1 \models_{\mathcal{L}} \varphi \\
\pi, i \models_{\mathcal{L}} \varphi \,\mathcal{U}\, \psi &\triangleq \bigsqcup_{j \geq i} \big((\pi, j \models_{\mathcal{L}} \psi) \sqcap (\bigsqcap_{i \leq k < j} \pi, k \models_{\mathcal{L}} \varphi) \big)
\end{aligned}$$

The value of a property on a run is the value that it has in the 0th state of the run:

$$\pi \models_{\mathcal{L}} \varphi \;\triangleq\; \pi, 0 \models_{\mathcal{L}} \varphi$$

As usual, $\Diamond\varphi = \top \,\mathcal{U}\, \varphi$, $\Box\varphi = \neg\Diamond\neg\varphi$, and $\varphi \,\mathcal{R}\, \psi = \neg(\neg\varphi \,\mathcal{U}\, \neg\psi)$. χLTL operators satisfy the expected LTL properties, for example, the fixpoint properties in Figure 3.

Consider the example in Figure 4. This figure presents partial execution of the Leader Election protocol specified using the five-valued logic **5**. Let N be the number of processes (which we assume to be an even number), and K be the number that have agreed on the leader. We abstract K using the 5-valued predicate `le` ("leader elected") which is *true* when $K = N$, *weakly true* when $(N/2) < K < N$, *undecided* when $K = N/2$, *weakly false* when $0 < K < (N/2)$, and *false* when $K = 0$. Let `active`$[i]$ indicate that the ith process is currently active. In this system, $\pi, 0 \models_{\mathcal{L}}$ `le` is U, $\pi, 0 \models_{\mathcal{L}}$ `le` $\wedge$ `active`[`i`] is F ($\bot$), and $\pi, 0 \models_{\mathcal{L}} \neg$`active`[`i`] is T ($\top$). The value for `le` indicates that originally there was no consensus on the leader (U), then consensus started forming (L) and was reached (T). However, in the next state one of the processes changed its mind, and thus the consensus went back to L. For this run, the value of $\Diamond$`le` is T, but the value of $\Diamond\Box$`le` is L. Note that we get this value without the need to re-annotate our model under a different level of abstraction and rerun the check.

4 Multiple-Valued Languages and Automata

In the task of using multiple-valued logic for system specification and verification, it is natural to consider *multiple-valued formal languages* and *multiple-valued automata*. We introduce them in this section.

4.1 Multiple-Valued Languages

Let Σ be a finite alphabet, Σ^* be the set of all *finite* words over Σ, Σ^ω be the set of all *infinite* words, and $\Sigma^{\leq\omega} = \Sigma^* \cup \Sigma^\omega$. We can catenate any two finite words, and consider the *empty string* λ as the identity for catenation: $w\lambda = \lambda w = w$. The empty string is contained in Σ^*, but not in Σ^ω.

Definition 2. *A* multiple-valued language *over an alphabet Σ is a multiple-valued subset of Σ^*, or an element in $\mathcal{O}^{\Sigma^*}$; a* multiple-valued ω-language *is an element in $\mathcal{O}^{\Sigma^\omega}$. A multiple-valued language X is* proper *if $(\lambda \in_{\mathcal{L}} X) = \top$.*

We shall use the term "MV-language" to refer, indiscriminately, to any multiple-valued language or ω-language, wherever the distinction is not important. An MV-language is an assignment of values to words. If $\mathcal{O} = \mathbf{2}$, then an MV-language is an ordinary formal language, where every word that is assigned value $\top$ is considered to be in the language. The properness criterion assures that λ is contained in the language as the identity for catenation.

MV-languages are just MV-sets of words, so union, intersection, and complement are already defined on them. The standard language operation of *catenation* can be extended to the multiple-valued case, as given below.

Definition 3. *Given $X, Y \in \mathcal{O}^{\Sigma^*}$ and $w \in \Sigma^*$,*

$$w \in_{\mathcal{L}} XY \triangleq \bigsqcup_{\{u,v \mid w=uv\}} (u \in_{\mathcal{L}} X) \sqcap (v \in_{\mathcal{L}} Y) \text{ (MV-language catenation)}$$

Transitive closure (Kleene star) and infinite closure (ω) can be defined in terms of multiple-valued catenation.

Consider the two multiple-valued languages $X = \{a \to T, ab \to L\}$ and $Y = \{bc \to M, c \to U\}$, defined on the logic **5**. We are interested in the value that abc has in XY. It can be formed either by catenating a and bc, with value $T \sqcap M = M$, or by catenating ab and c, with value $L \sqcap U = U$. By the definition, we take the maximum of those two values, making the value of $abc \in_{\mathcal{L}} XY$ to be M.

4.2 Multiple-Valued Automata

A multiple-valued finite automaton A takes any word $w \in \Sigma^{\leq\omega}$ and computes its membership degree, a value in $\mathcal{O}$. Thus, an automaton corresponds to a

multiple-valued language $L(A)$. Details about multiple-valued automata on *finite* words (in the more general case, of semiring-valued languages) can be found elsewhere [Eil78]; our treatment of multiple-valued infinite words and their automata is, so far as we know, new, but it is a natural extension.

A multiple-valued Büchi automaton has transitions between states that take on some value ranging between $\top$ or $\bot$ of an FTO. This value, intuitively, is a *possibility* that a transition will be taken. Thus, we can assign possibilities to individual transitions and to infinite strings that the automaton receives.

Definition 4. *A* multiple-valued Büchi automaton*, or* χ*Büchi automaton, is a tuple* $(\mathcal{L}, Q, q_0, \Sigma, \Delta, F)$ *where:*

- $\mathcal{L} = (\mathcal{O}, \sqsubseteq)$ *is an FTO;*
- Q *is a finite set of states;*
- q_0 *is the unique initial state;*
- Σ *is a finite alphabet;*
- $\Delta \in \mathcal{O}^{Q \times \Sigma \times Q}$ *is the multiple-valued transition relation.* $\Delta(q, \alpha, q')$ *gives the value of the transition from* q *to* q' *on symbol* α*;*
- F *is a set of accepting states.*

The *runs* of the automaton are infinite sequences of states, always beginning with q_0. We define a projection of Q onto F as

$$\pi_F(q) = \begin{cases} q \text{ if } q \in F \\ \lambda \text{ otherwise} \end{cases}$$

which we extend to Q^ω, and define the *accepting runs* $\mathcal{AR}$ of the automaton to be the elements of

$$\{\sigma \mid \pi_F(\sigma) \in F^\omega\}.$$

Intuitively, $\mathcal{AR}$ is the set of all runs in which some accepting state occurs infinitely often.

For a χBüchi automaton A, $L(A) \in \mathcal{O}^{\Sigma^\omega}$ is the multiple-valued subset of Σ^ω defined by the automaton. The value assigned by the automaton to a word $w = w_0 w_1 w_2 \ldots$ in Σ^ω is given in terms of the accepting runs:

$$\big(w \in_{\mathcal{L}} L(A)\big) = \bigsqcup_{\sigma \in \mathcal{AR}} \prod_{i \in \mathbb{N}} \Delta(\sigma_i, w_i, \sigma_{i+1})$$

Consider the χBüchi automaton in Figure 5. This automaton assigns values from **5** to its inputs. In the input sequence $abbcd^\omega$, the prefix abb takes the automaton only through $\top$-valued transitions. Then, c follows an L-transition to an accepting state; after this occurs, the value of the whole sequence *cannot* exceed L. The automaton loops through the accepting state on the infinite sequence of d's, so this word is accepted with value L.

χBüchi automata are similar in spirit to *Markov chains* [Fel68]. Markov chains also assign values, representing probabilities, to nonterminating finite-state computations, and have been used [VW86] to check probabilistic system specifications. Our approach is more *possibilistic*, motivated by the problem of

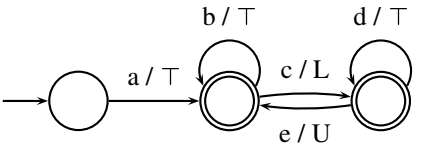


Fig. 5. An example χBüchi automaton.

requirements analysis. Given two independent events, the *probability* of the occurrence of at least one is the sum of their individual probabilities; but the *possibility* or *necessity* of at least one event occuring is the *maximum* of their individual possibilities.

4.3 Composition

Our definitions of parallel composition, synchronous and asynchronous, are extensions of the standard construction [Tho90].

We start by defining synchronous parallel composition, or MV-intersection of languages. Let L_1 and L_2 be multiple-valued ω-languages and $A_1 = (\mathcal{L}, Q_1, q_0^1, \Sigma, \Delta_1, F_1)$ and $A_2 = (\mathcal{L}, Q_2, q_0^2, \Sigma, \Delta_2, F_2)$ be χBüchi automata for L_1 and L_2, respectively. We construct two classical automata, $\hat{A}_i$ (for $i = 1, 2$), where $\hat{\Delta}_i(q, \alpha, q')$ is true exactly if $\Delta_i(q, \alpha, q') \neq \bot$. Then we intersect the two classical automata, creating $\hat{A}_{12} = (Q_1 \times Q_2 \times \{0, 1, 2\}, (q_0^1, q_0^2, 0), \Sigma, \hat{\Delta}_{12}, F_1 \times F_2 \times \{2\})$. Finally, we create the *multiple-valued intersection* of the two χBüchi automata by transforming $\hat{A}_{12}$ into a χBüchi automaton A_{12} with the new multiple-valued transition relation:

$$\Delta_{12}\big((q, r, j), \alpha, (q', r', j')\big) = \begin{cases} \Delta_1(q, \alpha, q') \sqcap \Delta_2(r, \alpha, r') \\ \text{if } \hat{\Delta}_{12}\big((q, r, j), \alpha, (q', r', j')\big) \\ \bot \text{otherwise} \end{cases}$$

for all $j \in \{0, 1, 2\}$.

Theorem 2. *The value that A_{12} gives to a word w is the same as its value in $L_1 \cap_{\mathcal{L}} L_2$.*

Figure 6 illustrates the intersection construction. The first automaton gives the value $\top$ to ac^ω and L to ba^ω; the second gives value $\top$ to ba^ω. Every other word evaluates to $\bot$. In the intersection, ac^ω becomes $\bot$, and ba^ω evaluates to the minimum of L and $\top$, namely L. Note that (q_2, r_1) is labelled with 2, making it an accepting state in the intersection automaton, because it is a final state in *both* A_1 and A_2.

We proceed to define asynchronous composition on two χBüchi automata with (possibly different) alphabets and the same logic.

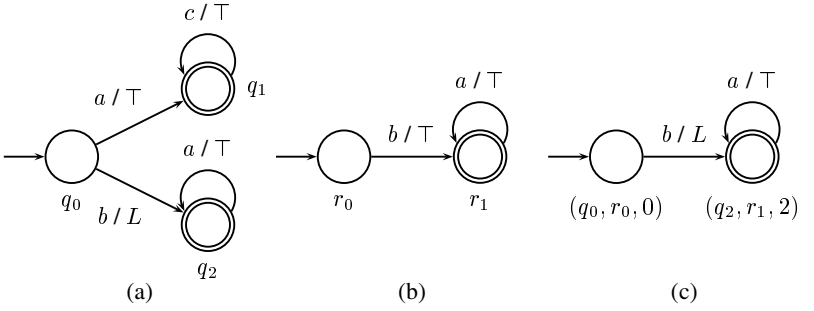


Fig. 6. Intersection of χBüchi automata. (c) shows the intersection of automata in (a) and (b).

Definition 5. *Let* $A_1 = (\mathcal{O}, Q_1, q_0^1, \Sigma_1, \Delta_1, F_1)$, $A_2 = (\mathcal{O}, Q_2, q_0^2, \Sigma_2, \Delta_2, F_2)$ *be two χBüchi automata. The* asynchronous composition $A_1 \,||\, A_2 = (\mathcal{O}, Q_1 \times Q_2, (q_0^1, q_0^2), \Sigma_1 \cup \Sigma_2, \Delta, F)$ *of the two automata has the following transition relation:*

$$\begin{array}{ll} \Delta\big((q_1,q_2),\alpha,(q_1',q_2)\big) = \Delta_1(q_1,\alpha,q_1') & \text{if } q_1 \neq q_1' \\ \Delta\big((q_1,q_2),\alpha,(q_1,q_2')\big) = \Delta_2(q_2,\alpha,q_2') & \text{if } q_2 \neq q_2' \\ \Delta\big((q_1,q_2),\alpha,(q_1,q_2)\big) = \Delta_1(q_1,\alpha,q_1) \sqcup \Delta_2(q_2,\alpha,q_2) & \text{otherwise} \end{array}$$

A state (q_1, q_2) *of the asynchronous composition is considered final if either* q_1 *or* q_2 *are final, so*

$$F = (F_1 \times Q_2) \cup (Q_1 \times F_2)$$

5 Conversion between χLTL and χBüchi Automata

In this section we describe how to convert between χLTL formulas, defined in Section 3 and χBüchi automata. Our algorithm is based on the classical LTL to Büchi automata conversion algorithm presented in [GPVW95]. As in [GPVW95], we start by defining *Generalized χBüchi Automata* and *Labeled Generalized χBüchi Automata (LGχBA).*

Definition 6. *A* Generalized χBüchi automaton *(GχBA) is a tuple* $(\mathcal{L}, Q, q_0, \Sigma, \Delta, \mathcal{F})$ *where* $\mathcal{L}, Q, q_0, \Sigma$ *and* Δ *are as in ordinary χBüchi automata, but* $\mathcal{F} = \{F_1, F_2, \ldots, F_k\}$ *is a set of* k *sets of accepting states. Each set* F_i *has the projection* π_{F_i} *defined for it, and the accepting runs are those where at least one element from* each F_i *appears infinitely often:*

$$\mathcal{AR} = \{\sigma \mid \sigma \in q_0 Q^\omega \wedge \forall i \leq k \cdot \pi_{F_i}(\sigma) \in F_i^\omega\}$$

Definition 7. *A* Labeled Generalized χBüchi Automaton (LGχBA) *is a tuple* $(\mathcal{L}, Q, q_0, \Sigma, \Delta, \mathcal{F}, Lab)$ *where:*

- $\mathcal{L} = (\mathcal{O}, \sqsubseteq)$ *is an FTO;*
- Q *is a finite set of states;*
- q_0 *is the unique initial state;*
- $\Sigma = \mathcal{O}^{Prop}$ *is an alphabet consisting of all multiple-valued sets over the set* $Prop$ *of propositional symbols;*
- $\Delta \in \mathcal{O}^{Q \times Q}$ *is a multiple-valued transition relation;*
- $\mathcal{F} = \{F_1, F_2, \ldots F_n\}$ *is a set of sets of accepting states;*
- $Lab : Q \to 2^{Prop \cup \neg Prop}$ *is a labeling function that assigns a subset of* $Prop \cup \neg Prop$ *to every state.*

The set of accepting runs ($\mathcal{AR}$) for a LGχBA is defined the same as for a Generalized χBüchi automaton given in Section 4.

Notice that each element $\alpha \in \Sigma$ is a total function from $Prop$ to $\mathcal{O}$. We extend this function to elements of $\neg Prop$ by defining $\alpha(\neg p) \triangleq \neg\alpha(p)$, $\forall p \in Prop$. Let $\hat{\alpha} : 2^{Prop \cup \neg Prop} \to \mathcal{O}$ be a set-wise extension of α, defined as

$$\hat{\alpha}(D) \triangleq \bigsqcap_{d \in D} \alpha(d) \qquad \text{(set-wise extension)}$$

For a Labeled Generalized χBüchi automaton A, $L(A) \in \mathcal{O}^{\Sigma^\omega}$ is the multiple-valued subset of Σ^ω defined by the automaton. The value assigned by the automaton to a word $w = w_0 w_1 w_2 \ldots$ in Σ^ω is given in terms of the accepting runs:

$$w \in_{\mathcal{L}} L(A) = \bigsqcup_{\sigma \in \mathcal{AR}} \bigsqcap_{i \in \mathbb{N}} \Delta(\sigma_i, \sigma_{i+1}) \sqcap \hat{w}_i\big(Lab(\sigma_{i+1})\big)$$

where $\hat{w}_i$ is the set-wise extension of w_i.

Given an LTL property φ, the algorithm in [GPVW95] constructs a Labeled Generalized Büchi automaton in two major steps. In the first step, it uses the syntactic structure of the formula to construct a graph $G = (V, E)$ together with three labeling functions, *New*, *Old*, and *Next*, that assign a subset from a *closure* of φ to each node of G. In the second step, the algorithm constructs an automaton, using G to define its basic structure, and the labeling functions to define its accepting states and state labels. The resulting Generalized Labeled Büchi automaton accepts a word if and only if the word satisfies φ. This automaton can be easily converted into a Büchi automaton with a polynomial blowout in its size.

Since χLTL is syntactically equivalent to LTL, we reuse the graph construction part of the algorithm in [GPVW95]. Thus, given a χLTL property φ, our algorithm starts by constructing a graph $G = (V, E)$ and the node labeling functions *New*, *Old*, and *Next* using the procedure in [GPVW95]. However, we modify this procedure to ensure the correct handling of $p \wedge \neg p$ (not necessarily $\bot$) and $p \vee \neg p$ (not necessarily $\top$), where p is any propositional formula. The algorithm

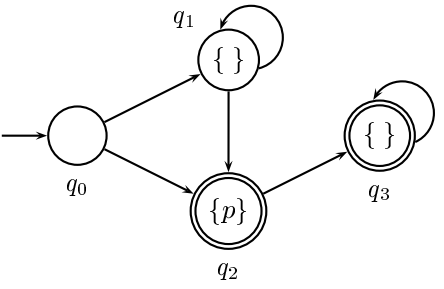


Fig. 7. A LGχBA corresponding to $\diamond p$.

then proceeds to construct a LGχBA $A = (\mathcal{L}, Q, q_0, \Sigma, \Delta, \mathcal{F}, Lab)$ by letting the set of states Q of the automaton be the nodes of G, with the root node of G being the initial state q_0. The accepting set $\mathcal{F}$ is constructed as in the original algorithm. The transition relation Δ is constructed from the edges of the graph G such that $\Delta(q, q') = \top$ if the edge (q, q') is in G, and $\Delta(q, q') = \bot$ otherwise. Finally, the labeling function Lab is constructed as a restriction of the labeling function Old to the set of all positive and negative propositional symbols of φ; that is, for a given state q, $Lab(q) = Old(q) \cap (Prop \cup \neg Prop)$. It is easy to show that the resulting LGχBA can be transformed into a χBüchi automaton via an extended version of the transformation used in the classical case.

For example, consider the automaton in Figure 7 which corresponds to a χLTL property $\diamond p$. In this figure we show only $\top$ transitions. Every accepting run of this automaton must pass through the state q_2. Therefore, the value that the automaton assigns to a given word w is

$$\bigsqcup_{i \in \mathbb{N}} (p \in_{\mathcal{L}} w_i)$$

which corresponds to the definition of $w \models_{\mathcal{L}} \diamond p$ from Section 3.

Theorem 3. *The automaton A constructed for a property φ assigns a value ℓ to an infinite sequence w over $\mathcal{O}^{Prop}$ if and only if $\ell = (w \models_{\mathcal{L}} \varphi)$.*

Proof. The proof is a straightforward extension of the proof of correctness of the algorithm in [GPVW95], and is omitted here. □

The immediate consequence of Theorem 3 is that if $\mathcal{L}$ is **2**, the automaton constructed by our algorithm is equivalent to the Labeled Generalized Büchi automaton produced by the original algorithm in [GPVW95].

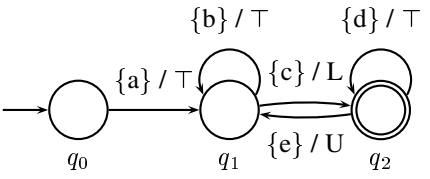


Fig. 8. An example Büchi automaton 2.

6 χLTL Model-Checking

In this section we define automata-theoretic multiple-valued model-checking and describe a decision procedure for it.

6.1 The Model-Checking Problem

Automata-theoretic model-checking procedure can be viewed as a function that receives a program P and property φ and returns a value from the logic $\mathcal{L}$ indicating the possibility that (the degree to which) P satisfies φ. For example, in the classical case $MC(P, \varphi) = \top$ if and only if every computation of P satisfies φ. In the remainder of the paper we use MC to indicate the classical model-checking function. MC is formally defined as

$$MC(P, \varphi) \triangleq \forall w \in \Sigma^\omega \cdot w \in L(A_P) \rightarrow w \in L(A_\varphi) \qquad (MC\text{-definition})$$

where A_P and A_φ are the Büchi automata corresponding to the program P and property φ, respectively.

We extend this definition to the multiple-valued case and define a multiple-valued model-checking function χMC as follows:

Definition 8. *Let P be a multiple-valued program, φ a χLTL property, and A_P, A_φ the corresponding χBüchi automata. Then, the multiple-valued model-checking function χMC is defined as*

$$\begin{aligned} \chi MC(P, \varphi) &\triangleq \prod_{w \in \Sigma^\omega} \big(w \in_{\mathcal{L}} L(A_P) \rightarrow w \in_{\mathcal{L}} L(A_\varphi)\big) && (\chi MC\text{-definition 1}) \\ &\triangleq \neg \bigsqcup_{w \in \Sigma^\omega} \big(w \in_{\mathcal{L}} L(A_P) \sqcap w \in_{\mathcal{L}} L(A_{\neg\varphi})\big) && (\chi MC\text{-definition 2}) \end{aligned}$$

Intuitively, the possibility of a program satisfying a property is inversely proportional to the possibility that the program can produce a computation violating the property. For example, consider a χBüchi automaton in Figure 8, corresponding to some program P. The set of propositional symbols of this

automaton is $\{a, b, c, d, e\}$, and as each transition is taken, exactly one of these symbols becomes $\top$ and the rest become $\bot$. Thus all transitions are labeled with singleton sets. For example, a transition between q_1 and q_2 is labeled with $\{c\}/\top$ to indicate that the transition is taken with possibility L when c becomes $\top$ (and a, b, d, e become $\bot$). Any non-$\bot$ computation w of this automaton contains a w_i such that $(d \in_{\mathcal{L}} w_i) = \top$; therefore, the result of $\chi MC(P, \Diamond d)$ is $\top$. That is, the program satisfies the property $\Diamond d$ with the value $\top$. On the other hand, the value of $\chi MC(P, \Diamond\Box d) = L$ since there exists a computation $w = \{a\}(\{c\}\{d\}\{e\})^\omega$, s.t. $\big(w \in_{\mathcal{L}} L(A)\big) = U$ and $\big(w \in_{\mathcal{L}} L(\neg\Diamond\Box d)\big) = \top$.

To establish correctness of our definition we show that it is equivalent to the classical definition when the logic used is **2**, and that it preserves the expected relationships between programs and χLTL properties.

Theorem 4. *Let P be a program, φ be a (χ)LTL property, and A_P, A_φ be the corresponding (χ)Büchi automata. Then, if the logic $\mathcal{L}$ used to define the χBüchi automata is* **2***, then*

$$MC(P, \varphi) = \chi MC(P, \varphi)$$

Proof. Follows directly from the definitions of MC and χMC. □

Intuitively, the degree to which a program P satisfies a conjunction of two properties cannot exceed the degree to which it satisfies each of these properties individually. Similarly, the degree to which a program P satisfies a disjunction of two properties is higher then the degree to which it satisfies each of the properties individually. Finally, in the classical case, if two programs satisfy a property, then so does their independent composition. This implies that in the multiple-valued case the degree to which a program $P_1 + P_2$ satisfies a given property φ must equal the smallest degree to which each program satisfies the property individually.

Theorem 5. *Let P_1 and P_2 be programs, and φ and ψ be χLTL properties. Then,*

(1) $\chi MC(P, \varphi \wedge \psi) = \chi MC(P, \varphi) \sqcap \chi MC(P, \psi)$ (property intersection)
(2) $\chi MC(P, \varphi) \sqcup \chi MC(P, \psi) \sqsubseteq \chi MC(P, \varphi \vee \psi)$ (property union)
(3) $\chi MC(P_1 + P_2, \varphi) = \chi MC(P_1, \varphi) \sqcap \chi MC(P_2, \varphi)$ (program composition)

Proof. The proof of (1) and (2) is based on the fact that the language of a χBüchi automaton $A_{\varphi\wedge\psi}$ ($A_{\varphi\vee\psi}$) is the multiple-valued intersection (union) of the languages $L(A_\varphi)$ and $L(A_\psi)$ corresponding to the properties φ and ψ, respectively. The proof of (3) is based on the fact that the language of a χBüchi automaton $A_{P_1+P_2}$ is the multiple-valued union of the languages $L(A_{P_1})$ and $L(A_{P_2})$ corresponding to programs P_1 and P_2, respectively. □

6.2 Decision Procedure for χLTL Model-Checking

In this section we show that a single χLTL model-checking problem, with an FTO of size $|\mathcal{O}|$, can be transformed into $(|\mathcal{O}| - 1)$ classical model-checking problems.

Recall the definition of $MC(P, \varphi)$. The formal definition is equivalent to the problem of *language containment*; we must check that $L(A_P) \subseteq L(A_\varphi)$. In practice, this is done via checking for emptiness of $L(A_P) \cap \overline{L(A_\varphi)}$, where $\overline{L(A_\varphi)} = L(A_{\neg\varphi})$ [VW86]. A classical ω-language, viewed as an element $Z \in \mathbf{2}^{\Sigma^\omega}$, is *nonempty* if and only if there exists a $w \in \Sigma^\omega$ such that $(w \in_{\mathcal{L}} Z) = \top$; that is,

$$Nonempty(Z) \triangleq \top \sqsubseteq \big(\bigvee_{w \in \Sigma^\omega} (w \in Z) \big) \quad \text{(non-emptiness)}$$

We wish to restate $\chi MC\ (\mathcal{P}, \varphi)$ in terms of language intersection and emptiness as well, so we start by generalizing the above definition to MV-languages.

Definition 9. *Let Z be an MV-language, $\mathcal{L} = (\mathcal{O}, \sqsubseteq)$ be an FTO, and $\ell \in (\mathcal{O} \setminus \{\bot\})$. Then*

$$Nonempty(Z, \ell) \triangleq \ell \sqsubseteq \big(\bigsqcup_{w \in \Sigma^\omega} (w \in_{\mathcal{L}} Z) \big) \quad (\ell\text{-non-emptiness})$$

If $\mathcal{O} = \mathbf{2}$ and $\ell = \top$, this definition reduces to the classical definition of emptiness. In the multiple-valued case, however, we can have degrees of emptiness, and this is captured by the generalized definition. For instance, if the maximal value of any word in an MV-language is M, then it is M-nonempty, but not L-nonempty or $\top$-nonempty.

We now define a reduction on MV-automata w.r.t. a logic value ℓ, known as an ℓ-cut [CDE+01b].

Definition 10. *Let $\mathcal{L} = (\mathcal{O}, \sqsubseteq)$ be an FTO. Then for any χBüchi automaton $A = (\mathcal{L}, Q, q_0, \Sigma, \Delta, F)$ and $\ell \in \mathcal{O}$, an* ℓ-cut *of A, denoted A^ℓ, is an automaton $(Q, q_0, \Sigma, \Delta^\ell, F)$ where:*

$$\Delta^\ell(q, \alpha, q') = \begin{cases} \top \text{ if } \ell \sqsubseteq \Delta(q, \alpha, q') \\ \bot \text{ otherwise} \end{cases} \quad \text{(definition of } \Delta^\ell\text{)}$$

The conversion from any A to A^ℓ can be done in $O(|Q|^2)$ time. Now we establish a few properties of ℓ-cuts.

Theorem 6. *Let A_1 and A_2 be arbitrary χBüchi automata. Then*

$$L\big((A_1 \cap_{\mathcal{L}} A_2)^\ell\big) = L(A_1^\ell) \cap L(A_2^\ell) \quad (\ell\text{-cut of language intersection})$$

Proof.

$$
\begin{aligned}
& w \in L\big((A_1 \cap_{\mathcal{L}} A_2)^{\ell}\big) \\
\Leftrightarrow\ & \text{Definition of cut} \\
& \ell \sqsubseteq (w \in_{\mathcal{L}} L\big((A_1 \cap_{\mathcal{L}} A_2)\big) \\
\Leftrightarrow\ & \text{Theorem 2} \\
& \ell \sqsubseteq \big(w \in_{\mathcal{L}} (L(A_1) \cap_{\mathcal{L}} L(A_2))\big) \\
\Leftrightarrow\ & \text{MV-intersection} \\
& \ell \sqsubseteq \big((w \in_{\mathcal{L}} L(A_1)) \sqcap (w \in_{\mathcal{L}} L(A_2))\big) \\
\Leftrightarrow\ & \text{min-}\wedge \text{ rule} \\
& \big(\ell \sqsubseteq (w \in_{\mathcal{L}} L(A_1))\big) \wedge \big(\ell \sqsubseteq (w \in_{\mathcal{L}} L(A_2))\big) \\
\Leftrightarrow\ & \text{Definition of cut} \\
& \big(w \in L(A_1^{\ell})\big) \wedge \big(w \in L(A_2^{\ell})\big) \\
\Leftrightarrow\ & \text{Intersection} \\
& w \in L(A_1^{\ell}) \cap L(A_2^{\ell})\big)
\end{aligned}
$$

□

Theorem 7. *Let A_1 and A_2 be arbitrary χBüchi automata. Then*

$$L\big((A_1 \,||\, A_2)^{\ell}\big) = L\big(A_1^{\ell} \,||\, A_2^{\ell}\big) \quad (\ell\text{-cut of parallel composition})$$

Proof. It is obvious that all transitions which are *not* self-loops will be in the ℓ-cut of the composition if and only if they are in the ℓ-cut of the process which moves on the transition. Let Δ be the transition relation of $(A_1 \,||\, A_2)^{\ell}$, and Δ' be the transition relation of $(A_1^{\ell} \,||\, A_2^{\ell})$. We show that the existence of a self-loop in Δ is equivalent to the existence of a self-loop in Δ':

$$
\begin{aligned}
& \Delta\big((q_1, q_2), \alpha, (q_1, q_2)\big) \\
\Leftrightarrow\ & \text{cut of parallel composition} \\
& \ell \sqsubseteq \Delta_1(q_1, \alpha, q_1) \sqcup \Delta_2(q_2, \alpha, q_2) \\
\Leftrightarrow\ & \text{max-}\vee \\
& (\ell \sqsubseteq \Delta_1(q_1, \alpha, q_1)) \vee (\ell \sqsubseteq \Delta_2(q_2, \alpha, q_2)) \\
\Leftrightarrow\ & \text{definition of cut} \\
& \Delta_1^{\ell}(q_1, \alpha, q_1)) \vee \Delta_2^{\ell}(q_2, \alpha, q_2) \\
\Leftrightarrow\ & \text{classical parallel composition} \\
& \Delta'\big((q_1, q_2), \alpha, (q_1, q_2)\big)
\end{aligned}
$$

□

Cuts can also be used to define the decision procedure for MV-language emptiness.

Theorem 8. *Let $\mathcal{L} = (\mathcal{O}, \sqsubseteq)$ be an FTO. Then for any χBüchi automaton $A = (\mathcal{L}, Q, q_0, \Sigma, \Delta, F)$ and $\ell \in \mathcal{O}$, the ℓ-nonemptiness of A is decidable.*

Proof. Construct A^ℓ, the ℓ-cut of A. $L(A^\ell)$ is nonempty if and only if there is some word w, for which there is an accepting run $\sigma \in \mathcal{AR}$ with only $\top$-valued transitions. That is:

$$
\begin{aligned}
& w \in L(A^\ell) \\
\Leftrightarrow\ & \text{Büchi acceptance} \\
& \exists \sigma \in \mathcal{AR} \cdot \forall i \in \mathbb{N} \cdot \Delta^\ell(\sigma_i, w_i, \sigma_{i+1}) \\
\Leftrightarrow\ & \text{definition of } \Delta^\ell \\
& \exists \sigma \in \mathcal{AR} \cdot \forall i \in \mathbb{N} \cdot \ell \sqsubseteq \Delta(\sigma_i, w_i, \sigma_{i+1}) \\
\Leftrightarrow\ & \text{min-}\wedge \text{ rule} \\
& \exists \sigma \in \mathcal{AR} \cdot \ell \sqsubseteq \bigsqcap_{i \in \mathbb{N}} \Delta(\sigma_i, w_i, \sigma_{i+1}) \\
\Leftrightarrow\ & \text{max-}\vee \text{ rule} \\
& \ell \sqsubseteq \bigsqcup_{\sigma \in \mathcal{AR}} \bigsqcap_{i \in \mathbb{N}} \Delta(\sigma_i, w_i, \sigma_{i+1}) \\
\Leftrightarrow\ & \chi\text{Büchi acceptance} \\
& \ell \sqsubseteq \big(w \in L(A)\big)
\end{aligned}
$$

In other words, if $L(A^\ell)$ is nonempty, then there is some word w such that $\ell \sqsubseteq (w \in_{\mathcal{L}} L(A))$, and $L(A)$ is ℓ-nonempty. Since A^ℓ is a classical Büchi automaton, its nonemptiness is decidable [Tho90]. □

We now have an effective decision procedure for finding the ℓ-nonemptiness of $L(A_P) \cap_{\mathcal{L}} L(A_{\neg\varphi})$ for any $\ell \in \mathcal{O}$. We can iterate this procedure to find the *maximal* ℓ for which this intersection is non-empty. The *complement* of this maximal ℓ can be returned as the value of property φ in system P. Figure 9 describes the model-checking procedure in detail. In order to gain some intuition for this result, first consider the classical case, where we simply need to check that the intersection of the system with the negated property automaton is $\top$-nonempty: if it is $\top$-nonempty, there are $\bot$-valued counterexamples to the property.

6.3 χLTL Model-Checking in SPIN

In this section we show how to implement a multi-valued automata-theoretic model-checker, which we call MV-SPIN, using SPIN as a black box. In Section 6.2 we established that model-checking of a property φ over a system P reduces to computing a series of ℓ-cuts over $P \cap_{\mathcal{L}} A_{\neg\varphi}$. By Theorem 6, we can perform ℓ-cuts of the property and the system automaton individually. We also note that the system is usually not a monolithic Promela model, corresponding to one

Given a system P, and a χLTL property φ:

1. Convert $\neg\varphi$ to a χBüchi automaton $A_{\neg\varphi}$ using the method of Section 5.
2. Compute $C = P \cap_{\mathcal{L}} A_{\neg\varphi}$ according to the construction of Section 4.3.
3. For each $\ell \in \mathcal{O}$, construct the cut C^{ℓ} and check it for nonemptiness.
4. Let ℓ_{max} be the maximal ℓ for which C^{ℓ} is nonempty.
5. Return $\neg\ell_{max}$.

Fig. 9. Decision procedure for multi-valued model-checking.

```
procedure MV-SPIN (P, φ)
    A_φ = B2Prom(χ2B(φ))
    for ℓ = ⊤ downto ⊥
        P' = Cut(P, ℓ)
        A'_φ = Cut(A_φ, ℓ)
        ce = SPIN (P', A'_φ)
        if (ce ≠ ∅)
            return ¬ℓ as answer and
                   ce, if present, as the counter-example
```

Fig. 10. Algorithm for MV-SPIN.

Büchi automaton, but a collection of processes which are run under asynchronous parallelism. Furthermore, SPIN does not compute the entire automaton of the model; instead, it performs model-checking on-the-fly [GPVW95]. Thus, our goal is to specify multiple-valued models in some Promela-like language, extended with MV-semantics and then generate Promela without building the complete Büchi automata.

Extending Promela with multiple-valued guard commands is not difficult, as indicated by the work on probabilistic GCL [HSM97]. Asynchronous parallel composition of χBüchi automata was given in Definition 5. By Theorem 7, ℓ-cuts of the entire model are equal to ℓ-cuts of each individual process. Assume that this operation is done by function `Cut` which takes a model in extended Promela and a logic value ℓ and converts it into "regular" Promela while performing the reduction ℓ-cut.

The algorithm for MV-SPIN is given in Figure 10. Functions`χ2B` and `B2Prom` are the modifications of existing LTL to Büchi automata and Büchi automata to Promela algorithms, respectively, enriched to handle χBüchi automata. The result of SPIN is stored in `ce`. If `ce` is empty, the classical model-checking procedure succeeded; else, `ce` is returned as the counter-example.

Note that the performance penalty of MV-SPIN w.r.t. SPIN manifests itself in a $O(|\mathcal{O}|)$ expansion in the size of the Büchi automaton constructed from the χLTL property, in executing SPIN up to $|\mathcal{O}|$ times and in executing up to $2 \times |\mathcal{O}|$ cuts. Cuts are performed on individual Promela processes and are

proportional to the number of lines in respective text files. Thus, we get an overall $O(|\mathcal{O}|^2)$ performance penalty. However, the sizes of resulting models are smaller than they would have been if we replaced multiple-valued variables by a collection of boolean variables. In addition, FTOs allow to compactly represent incompleteness and uncertainty in the system; such situations can be modeled in classical logic by using additional variables and thus leading to the exponential growth in the size of the state space [CDE01a].

7 Conclusion

In this paper we extended classical automata-theoretic model-checking to reasoning over multiple-valued logics, whose values form total linear orders. We gave semantics to a multiple-valued extension of LTL, called χLTL, described notions of multiple-valued languages and automata, and defined a general model-checking problem. We also showed that the multiple-valued model-checking problem reduces to a set of queries to a classical model-checking procedure, and thus can be easily implemented on top of SPIN.

We further note that FTOs are a subclass of *quasi-boolean logics* – logics based on lattices with specially-defined negation. We used quasi-boolean logics in our previous work [CDE01a,CDE$^+$01b]. In fact, our definitions of χLTL, χBüchi automata and multiple-valued model-checking can be used verbatim if we replace FTOs by quasi-boolean logics. Furthermore, Theorem 8 also holds for all join-irreducible [CDE$^+$01b] elements of the lattices. However, we do not yet have an effective decision procedure for other elements of the logic.

Acknowledgments. We thank members of the University of Toronto formal methods reading group, and in particular Steve Easterbrook and Albert Lai, for many useful discussions. This work was financially supported by NSERC and CITO.

References

[BDL96] C. Barret, D. Dill, and K. Levitt. "Validity Checking for Combinations of Theories with Equality". In *Formal Methods in Computer-Aided Design*, volume 1166 of *LNCS*, pages 187–201, November 1996.

[BG99] G. Bruns and P. Godefroid. "Model Checking Partial State Spaces with 3-Valued Temporal Logics". In *Proceedings of CAV'99*, volume 1633 of *LNCS*, pages 274–287, 1999.

[BG00] G. Bruns and P. Godefroid. "Generalized Model Checking: Reasoning about Partial State Spaces". In *Proceedings of CONCUR'00*, volume 877 of *LNCS*, pages 168–182, August 2000.

[CC77] P. Cousot and R. Cousot. "Static Determination of Dynamic Properties of Generalized Type Unions". *SIGPLAN Notices*, 12(3), March 1977.

[CD00] M. Chechik and W. Ding. "Lightweight Reasoning about Program Correctness". CSRG Technical Report 396, University of Toronto, March 2000.

[CDE01a] M. Chechik, B. Devereux, and S. Easterbrook. "Implementing a Multi-Valued Symbolic Model-Checker". In *Proceedings of TACAS'01*, April 2001.

[CDE+01b] M. Chechik, B. Devereux, S. Easterbrook, A. Lai, and V. Petrovykh. "Efficient Multiple-Valued Model-Checking Using Lattice Representations". Submitted for publication, January 2001.

[CGL94] E.M. Clarke, O. Grumberg, and D.E. Long. "Model Checking and Abstraction". *IEEE Transactions on Programming Languages and Systems*, 19(2), 1994.

[CU98] M. Colon and T. Uribe. "Generating Finite-State Abstractions of Reactive Systems using Decision Procedures". In *Proceedings of the 10th Conference on Computer-Aided Verification*, volume 1427 of *LNCS*. Springer-Verlag, July 1998.

[DDP99] S. Das, D. Dill, and S. Park. "Experience with Predicate Abstraction". In *Proceedings of the 11th International Conference on Computer-Aided Verification*, volume 1633 of *LNCS*, pages 160–171. Springer-Verlag, 1999.

[Eil78] S. Eilenberg. *Automata, Languages and Machines*, volume A. Academic Press, New York, 1978.

[Fel68] W. Feller. *An Introduction to Probability Theory and its Applications*, volume I. John Wiley and Sons, New York, 1968.

[Fit91] M. Fitting. "Many-Valued Modal Logics". *Fundamenta Informaticae*, 15(3–4):335–350, 1991.

[Fit92] M. Fitting. "Many-Valued Modal Logics II". *Fundamenta Informaticae*, 17:55–73, 1992.

[Gai79] Brian R. Gaines. "Logical Foundations for Database Systems". *International Journal of Man-Machine Studies*, 11(4):481–500, 1979.

[Gin87] M. Ginsberg. "Multi-valued logic". In M. Ginsberg, editor, *Readings in Nonmonotonic Reasoning*, pages 251–255. Morgan-Kaufmann Pub., 1987.

[Gol99] J. S. Golan. *Power Algebras over Semirings*. Kluwer Academic, 1999.

[GPVW95] R. Gerth, D. Peled, M. Vardi, and P. Wolper. "Simple On-the-fly Automatic Verification of Linear Temporal Logic". In *In Proceedings of 15th Workshop on Protocol Specification, Testing, and Verification*, Warsaw, North-Holland, June 1995.

[GS97] S. Graf and H. Saidi. "Construction of Abstract State Graphs with PVS". In *Proceedings of the 9th International Conference on Computer-Aided Verification*, volume 1254 of *LNCS*. Springer-Verlag, 1997.

[HJL96] C. L. Heitmeyer, R. D. Jeffords, and B. G. Labaw. "Automated Consistency Checking of Requirements Specifications". *ACM Transactions on Software Engineering and Methodology*, 5(3):231–261, July 1996.

[HK93] R. Hähnle and W. Kernig. Verification of switch-level designs with many-valued logic. In *International Conference LPAR '93*, volume 698. Springer-Verlag, 1993.

[HSM97] J. He, K. Seidel, and A. McIver. Probabilistic models for the guarded command language. *Science of Computer Programming*, 28(2–3):171–192, April 1997.

[IEE93] IEEE Standard 1164–1993. 1993.

[Mic77] R. S. Michalski. "Variable-Valued Logic and its Applications to Pattern Recognition and Machine Learning". In D. C. Rine, editor, *Computer Science and Multiple-Valued Logic: Theory and Applications*, pages 506–534. North-Holland, Amsterdam, 1977.

[SRW99] M. Sagiv, T. Reps, and R. Wilhelm. "Parametric Shape Analysis via 3-Valued Logic". In *Proceedings of 26th Annual ACM Symposium on Principles of Programming Languages*, 1999.

[SS99] H. Saidi and N. Shankar. "Abstract and Model Check while you Prove". In *Proceedings of the 11th Conference on Computer-Aided Verification*, volume 1633 of *LNCS*, pages 443–454, July 1999.

[Tho90] W. Thomas. Automata on infinite objects. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, chapter 4, pages 133–191. Elsevier Science Publishers B. V., 1990.

[VPP00] W. Visser, S. Park, and J. Penix. "Applying Predicate Abstraction to Model Check Object-Oriented Programs". In *Proceedings of 4th International Workshop on Formal Methods in Software Practice*, August 2000.

[VW86] M. Y. Vardi and P. Wolper. "An Automata-Theoretic Approach to Automatic Program Verification". In *Proceedings of 1st Symposium on Logic in Computer Science*, pages 322–331, Cambridge MA, 1986.

Implementing LTL Model Checking with Net Unfoldings*

Javier Esparza[1] and Keijo Heljanko[2]

[1] Institut für Informatik, Technische Universität München, Germany
esparza@in.tum.de
[2] Lab. for Theoretical Computer Science, Helsinki University of Technology, Finland
Keijo.Heljanko@hut.fi

Abstract. We report on an implementation of the unfolding approach to model-checking LTL-X recently presented by the authors. Contrary to that work, we consider an state-based version of LTL-X, which is more used in practice. We improve on the checking algorithm; the new version allows to reuse code much more efficiently. We present results on a set of case studies.

1 Introduction

Unfoldings [14,6,5] are a partial-order approach to the automatic verification of concurrent and distributed systems, in which partial-order semantics is used to generate a compact representation of the state space. For systems exhibiting a high degree of concurrency, this representation can be exponentially more succinct than the explicit enumeration of all states or the symbolic representation in terms of a BDD, thus providing a very good solution to the state-explosion problem. Unfolding-based model-checking techniques for LTL without the next operator (called LTL-X in the sequel) were first proposed in [22]. A new algorithm with better complexity bounds was introduced in [3], in the shape of a tableau system. The approach is based on the automata-theoretic approach to model-checking (see for instance [20]), consisting of the following well-known three steps: (1) translate the negation of the formula to be checked into a Büchi automaton; (2) synchronize the system and the Büchi automaton in an adequate way to yield a composed system, and (3) check emptiness of the language of the composed system, where language is again defined in a suitable way.

In [3] we used an action-based version of LTL-X having an operator $\phi_1 \mathcal{U}^a \phi_2$ for each action a; $\phi_1 \mathcal{U}^a \phi_2$ holds if ϕ_1 holds until action a occurs, and immediately after ϕ_2 holds. Step (2) is very simple for this logic, which allowed us to concentrate on step (3), the most novel contribution of [3]. However, the state-based version of LTL-X is more used in practice. The first contribution of this paper is a solution to step (2) for this case, which turns out to be quite delicate.

* Work partially supported by the Teilprojekt A3 SAM of the Sonderforschungsbereich 342 "Werkzeuge und Methoden für die Nutzung paralleler Rechnerarchitekturen", the Academy of Finland (Projects 47754 and 43963), and the Emil Aaltonen Foundation.

M.B. Dwyer (Ed.): SPIN 2001, LNCS 2057, pp. 37–56, 2001.

The second contribution of this paper concerns step (3). In [3] we presented a two-phase solution; the first phase requires to construct one tableau, while the second phase requires to construct a possibly large set of tableaux. We propose here a more elegant solution which, loosely speaking, allows to merge all the tableaux of [3] into one while keeping the rules for the tableau construction simple and easy to implement.

The third contribution is an implementation using the `smodels` NP-solver [18], and a report on a set of case studies.

The paper is structured as follows. Section 2 contains basic definitions on Petri nets, which we use as system model. Section 3 describes step (2) above for the state-based version of LTL-X. Readers wishing to skip this section need only read (and believe the proof of) Theorem 1. Section 4 presents some basic definitions about the unfolding method. Section 5 describes the new tableau system for (3), and shows its correctness. Section 6 discusses the tableau generation together with some optimizations. Section 7 reports on the implementation and case studies, and Section 8 contains conclusions.

2 Petri Nets

A *net* is a triple (P, T, F), where P and T are disjoint sets of *places* and *transitions*, respectively, and F is a function $(P \times T) \cup (T \times P) \to \{0, 1\}$. Places and transitions are generically called *nodes*. If $F(x, y) = 1$ then we say that there is an *arc* from x to y. The *preset* of a node x, denoted by ${}^\bullet x$, is the set $\{y \in P \cup T \mid F(y, x) = 1\}$. The *postset* of x, denoted by $x^\bullet$, is the set $\{y \in P \cup T \mid F(x, y) = 1\}$. In this paper we consider only nets in which every transition has a nonempty preset *and* a nonempty postset. A *marking* of a net (P, T, F) is a mapping $P \to \mathbb{N}$ (where $\mathbb{N}$ denotes the natural numbers including 0). We identify a marking M with the multiset containing $M(p)$ copies of p for every $p \in P$. For instance, if $P = \{p_1, p_2\}$ and $M(p_1) = 1$, $M(p_2) = 2$, we write $M = \{p_1, p_2, p_2\}$.

A marking M *enables* a transition t if it marks each place $p \in {}^\bullet t$ with a token, i.e. if $M(p) > 0$ for each $p \in {}^\bullet t$. If t is enabled at M, then it can *fire* or *occur*, and its occurrence *leads to* a new marking M', obtained by removing a token from each place in the preset of t, and adding a token to each place in its postset; formally, $M'(p) = M(p) - F(p, t) + F(t, p)$ for every place p. For each transition t the relation $\xrightarrow{t}$ is defined as follows: $M \xrightarrow{t} M'$ if t is enabled at M and its occurrence leads to M'.

A 4-tuple $\Sigma = (P, T, F, M_0)$ is a *net system* if (P, T, F) is a net and M_0 is a marking of (P, T, F) (called the *initial marking* of Σ). A sequence of transitions $\sigma = t_1 t_2 \dots t_n$ is an *occurrence sequence* if there exist markings M_1, M_2, ... , M_n such that

$$M_0 \xrightarrow{t_1} M_1 \xrightarrow{t_2} \dots M_{n-1} \xrightarrow{t_n} M_n$$

M_n is the marking reached by the occurrence of σ, which is also denoted by $M_0 \xrightarrow{\sigma} M_n$. A marking M is a *reachable marking* if there exists an occurrence sequence σ such that $M_0 \xrightarrow{\sigma} M$. An *execution* is an infinite occurrence sequence starting from the initial marking. The *reachability graph* of a net system

Σ is the labelled graph having the reachable markings of Σ as nodes, and the $\xrightarrow{t}$ relations (more precisely, their restriction to the set of reachable markings) as edges. In this work we only consider net systems with finite reachability graphs.

A marking M of a net is *n-safe* if $M(p) \leq n$ for every place p. A net system Σ is n-safe if all its reachable markings are n-safe. Fig. 1 shows a 1-safe net system.

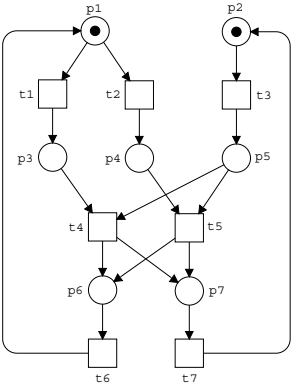


Fig. 1. The net system Σ

Labelled nets. Let $\mathcal{L}$ be an alphabet. A *labelled net* is a pair (N, l) (also represented as a 4-tuple (P, T, F, l)), where N is a net and $l\colon P \cup T \to \mathcal{L}$ is a labelling function. Notice that different nodes of the net can carry the same label. We extend l to multisets of $P \cup T$ in the obvious way.

For each label $a \in \mathcal{L}$ we define the relation $\xrightarrow{a}$ between markings as follows: $M \xrightarrow{a} M'$ if $M \xrightarrow{t} M'$ for some transition t such that $l(t) = a$. For a finite sequence $w = a_1 a_2 \ldots a_n \in \mathcal{L}^*$, $M \xrightarrow{w} M'$ denotes that for some reachable markings $M_1, M_2, \ldots, M_{n-1}$ the relation $M \xrightarrow{a_1} M_1 \xrightarrow{a_2} M_2 \ldots M_{n-1} \xrightarrow{a_n} M'$ holds. For an infinite sequence $w = a_1 a_2 \ldots \in \mathcal{L}^\omega$, $M \xrightarrow{w}$ denotes that $M \xrightarrow{a_1} M_1 \xrightarrow{a_2} M_2 \ldots$ holds for some reachable markings $M_1, M_2, \ldots$.

The reachability graph of a labelled net system (N, l, M_0) is obtained by applying l to the reachability graph of (N, M_0). In other words, its nodes are the set

$$\{l(M) \mid M \text{ is a reachable marking}\}$$

and its edges are the set

$$\{l(M_1) \xrightarrow{l(t)} l(M_2) \mid M_1 \text{ is reachable and } M_1 \xrightarrow{t} M_2\}\,.$$

3 Automata Theoretic Approach to Model Checking LTL

We show how to modify the automata theoretic approach to model checking LTL [20] to best suit the net unfolding method. We restrict the logic LTL by removing the next time operator X. We call this stuttering invariant fragment LTL-X. Given a finite set Π of atomic propositions, the abstract syntax of LTL-X is given by:

$$\varphi ::= \pi \in \Pi \mid \neg\varphi_1 \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \mathcal{U} \varphi_2$$

The semantics is a set of ω-words over the alphabet 2^{Π}, defined as usual.

Given a 1-safe net system Σ with initial marking M_0, we identify the atomic propositions Π with a subset $Obs \subseteq P$ of *observable places* of the net system, while the rest of the places are called *hidden*. Each marking M determines a valuation of $\Pi = Obs$ in the following way: $p \in Obs$ is true at M if M puts a token in p. Now, an execution $M_0 \xrightarrow{t_1} M_1 \xrightarrow{t_2} \ldots$ of Σ satisfies φ iff the ω-word $M_0 M_1 \ldots$ satisfies φ. The net system Σ satisfies φ, denoted $\Sigma \models \varphi$, if every execution of Σ satisfies φ.

The approach. Let φ be a formula of LTL-X. Using well-known algorithms (see e.g. [8]) we construct a Büchi automaton $\mathcal{A}_{\neg\varphi}$ over the alphabet $2^{\Pi} = 2^{Obs}$ which accepts a word w iff $w \not\models \varphi$.

We define a 1-safe product net system $\Sigma_{\neg\varphi}$ from Σ and $\mathcal{A}_{\neg\varphi}$. $\Sigma_{\neg\varphi}$ can be seen as the result of placing Σ in a suitable environment, i.e., $\Sigma_{\neg\varphi}$ is constructed by connecting Σ to an environment net system through new arcs.

It is easy to construct a product net system with a distinguished set of transitions I such that Σ violates φ iff some execution of the product fires some transition of I infinitely often. We call such an execution an *illegal ω-trace*. However, this product synchronizes $\mathcal{A}_{\neg\varphi}$ with Σ *on all transitions*, which effectively disables all concurrency present in Σ. Since the unfolding approach exploits the concurrency of Σ in order to generate a compact representation of the state space, this product is not suitable, and so we propose a new one.

We define the set V of *visible transitions* of Σ as the set of transitions which change the marking of some observable place of Σ. Only these transitions will synchronize with the automaton. So, for instance, in order to check a property of the form $\Box(p \rightarrow \Diamond q)$, where p and q are places, we will only synchronize with the transitions removing or adding tokens to p and q. This approach is similar but not identical to Valmari's tester approach described in [19]. (In fact, a subtle point in Valmari's construction makes its direct implementation unsuitable for checking state based LTL-X.)

The price to pay for this nicer synchronization is the need to check not only for illegal ω-traces, but also for so-called illegal livelocks. The new product contains a new distinguished set of transitions L (for livelock). An *illegal livelock* is an execution of the form $\sigma_1 t \sigma_2$ such that $t \in L$ and σ_2 does not contain any visible transition. For convenience we use the notation $M_0 \xrightarrow{\sigma} M \xrightarrow{\tau}$ to denote this, and implicitly require that $\sigma = \sigma_1 t$ with $t \in L$ and that τ is an infinite sequence which only contains invisible transitions.

In the rest of the section we define $\Sigma_{\neg\varphi}$. Readers only interested in the definition of the tableau system for LTL model-checking can safely skip it. Only the following theorem, which is proved hand in hand with the definition, is necessary for it. Property (b) is what we win by our new approach: The environment only interferes with the visible transitions of Σ.

Theorem 1. *Let Σ be a 1-safe net system whose reachable markings are pairwise incomparable with respect to set inclusion.*[1] *Let φ be an LTL-X formula over the observable places of Σ. It is possible to construct a net system $\Sigma_{\neg\varphi}$ satisfying the following properties:*

(a) $\Sigma \models \varphi$ iff $\Sigma_{\neg\varphi}$ has neither illegal ω-traces nor illegal livelocks.
(b) The input and output places of the invisible transitions are the same in Σ and $\Sigma_{\neg\varphi}$.

Construction of $\Sigma_{\neg\varphi}$ We describe the synchronization $\Sigma_{\neg\varphi}$ of Σ and $\mathcal{A}_{\neg\varphi}$ in a semiformal but hopefully precise way. Let us start with two preliminaries. First, we identify the Büchi automaton $\mathcal{A}_{\neg\varphi}$ with a net system having a place for each state q, with only the initial state q^0 having a token, and a net transition for each transition (q, x, q'); the input and output places of the transition are q and q', respectively; we keep $\mathcal{A}_{\neg\varphi}$, q and (q, x, q') as names for the net representation, the place and the transition. Second, we split the executions of Σ that violate φ into two classes: *executions of type I*, which contain infinitely many occurrences of visible transitions, and *executions of type II*, which only contain finitely many. We will deal with these two types separately.

$\Sigma_{\neg\varphi}$ is constructed in several steps:

(1) Put Σ and (the net representation of) $\mathcal{A}_{\neg\varphi}$ side by side.
(2) For each observable place p add a *complementary place* (see [17]) $\overline{p}$ to Σ. $\overline{p}$ is marked iff p is not, and so checking that proposition p does not hold is equivalent to checking that the place $\overline{p}$ has a token. A set $x \subseteq \Pi$ can now be seen as a conjunction of literals, where $\overline{p} \in x$ is used to denote $p \in (\Pi \setminus x)$.
(3) Add new arcs to each transition (q, x, q') of $\mathcal{A}_{\neg\varphi}$ so that it "observes" the places in x.
This means that for each literal p ($\overline{p}$) in x we add an arc from p ($\overline{p}$) to (q, x, q') and an arc from (q, x, q') to p ($\overline{p}$). The transition (q, x, q') can only be enabled by markings of Σ satisfying all literals in x.
(4) Add a *scheduler* guaranteeing that:
  – Initially $\mathcal{A}_{\neg\varphi}$ can make a move, and all *visible* moves (i.e., the firings of visible transitions) of Σ are disabled.
  – After a move of $\mathcal{A}_{\neg\varphi}$, only Σ can make a move.
  – After Σ makes a visible move, $\mathcal{A}_{\neg\varphi}$ can make a move and until that happens all visible moves of Σ are disabled.

[1] This condition is purely technical. Any 1-safe net system can be easily transformed into an equivalent one satisfying it by adding some extra places and arcs; moreover, the condition can be removed at the price of a less nice theory.

This is achieved by introducing two *scheduler places* s_f and s_s [22]. The intuition behind these places is that when s_f (s_s) has a token it is the turn of the Büchi automaton (the system Σ) to make a move. In particular, visible transitions transfer a token from s_s to s_f, and Büchi transitions from s_f to s_s. Because the Büchi automaton needs to observe the initial marking of Σ, we initially put one token in s_f and no tokens on s_s.

(5) Let I be a subset of transitions defined as follows. A transition belongs to I iff its postset contains a final state of $\mathcal{A}_{\neg\varphi}$.

Observe that since only moves of $\mathcal{A}_{\neg\varphi}$ and visible moves of Σ are scheduled, invisible moves can still be concurrently executed.

Let $\Sigma'_{\neg\varphi}$ be the net system we have constructed so far. The following is an immediate consequence of the definitions:

Σ has an execution of type I if and only if $\Sigma'_{\neg\varphi}$ has an illegal ω-trace.

We now extend the construction in order to deal with executions of type II. Let σ be a type II execution of Σ. Take the sequence of markings reached along the execution of σ, and project it onto the observable places. Since σ only contains finitely many occurrences of visible transitions, the result is a sequence of the form $O_0^0 O_0^1 \dots O_0^j O_1^0 O_1^1 \dots O_1^k O_2^0 \dots O_n^0 (O_n)^\omega$. (The moves from O_i to O_{i+1} are caused by the firing of visible transitions.)

We can split σ into two parts: a finite prefix σ_1 ending with the last occurrence of a visible transition (σ_1 is empty if there are no visible transitions), and an infinite suffix σ_2 containing only invisible transitions. Clearly, the projection onto the observable places of the marking reached by the execution of σ_1 is O_n

Since LTL-X is closed under stuttering, $\mathcal{A}_{\neg\varphi}$ has an accepting run

$$r = q_0 \xrightarrow{O_0} q_1 \xrightarrow{O_1} \dots \xrightarrow{O_{n-1}} q_n \xrightarrow{O_n} q_{n+1} \xrightarrow{O_n} q_{n+2} \dots$$

where the notation $q \xrightarrow{O} q'$ means that a transition (q, x, q') is taken such that the literals of x are true at the valuation given by O. We split this run into two parts: a finite prefix $r_1 = q_0 \xrightarrow{O_0} q_1 \dots q_{n-1} \xrightarrow{O_{n-1}} q_n$ and an infinite suffix $r_2 = q_n \xrightarrow{O_n} q_{n+1} \xrightarrow{O_n} q_{n+2} \dots$.

In the net system representation of $\mathcal{A}_{\neg\varphi}$, r_1 and r_2 correspond to occurrence sequences. By construction, the "interleaving" of r_1 and σ_1 yields an occurrence sequence τ_1 of $\Sigma'_{\neg\varphi}$.

Observe that reachable markings of $\Sigma'_{\neg\varphi}$ are of the form (q, s, O, H), meaning that they consist of a token on a state q of $\mathcal{A}_{\neg\varphi}$, a token on one of the places of the scheduler (i.e., $s \in \{s_s, s_f\}$), a marking O of the observable places, and a marking H of the hidden places. Let (q_n, s_f, O_n, H) be the marking of $\Sigma'_{\neg\varphi}$ reached after executing τ_1. (We have $s = s_f$ because the last transition of σ_1 is visible.) The following property holds: With q_n as initial state, the Büchi automaton $\mathcal{A}_{\neg\varphi}$ accepts the sequence O_n^ω. We call any pair (q, O) satisfying this property a *checkpoint* and define $\Sigma_{\neg\varphi}$ as follows:

(6) For each checkpoint (q, O) and for each reachable marking (q, s_f, O, H) of $\Sigma'_{\neg\varphi}$, add a new transition having all the places marked at (q, s_f, O, H) as preset, and all the places marked at O and H as postset. Let L (for *livelocks*) be this set of transitions.

The reader has possibly observed that the set L can be very large, because there can be many hidden markings H for a given marking O (exponentially many in the size of Σ). Apparently, this makes $\Sigma_{\neg\varphi}$ unsuitable for model-checking. In Sect. 6 we show that this is not the case, because $\Sigma_{\neg\varphi}$ need not be explicitly constructed.

Observe that after firing a L-transition no visible transition can occur anymore, because all visible transitions need a token on s_s for firing. We prove:

Σ has an execution of type II if and only if $\Sigma_{\neg\varphi}$ has an *illegal livelock*.

For the only if direction, assume first that σ is a type II execution of Σ. Let τ_1 be the occurrence sequence of $\Sigma_{\neg\varphi}$ defined above (as the "interleaving" of the prefix σ_1 of σ and the prefix r_1 of r). Further, let (q_n, s_f, O_n, H) be the marking reached after the execution of τ_1, and let t be the transition added in (6) for this marking. Define $\rho_1 = \tau_1$ and $\rho_2 = \sigma_2$. It is easy to show that $\rho_1 t \rho_2$ is an execution of $\Sigma_{\neg\varphi}$ and so an illegal livelock. For the if direction, let $\rho_1 t \rho_2$ be an illegal livelock of $\Sigma_{\neg\varphi}$, where t is an L-transition. After the firing of t there are no tokens in the places of the scheduler, and so no visible transition can occur again; hence, no visible transition of Σ occurs in ρ_2. Let σ_1 and σ_2 be the projections of ρ_1 and ρ_2 onto the transitions of Σ. It is easy to see that $\sigma = \sigma_1\sigma_2$ is an execution of Σ. Since σ_2 does not contain any visible transition, σ is an execution of type II.

4 Basic Definitions on Unfoldings

In this section we briefly introduce the definitions we needed to describe the unfolding approach. More details can be found in [6].

Occurrence nets. Given two nodes x and y of a net, we say that x is *causally related* to y, denoted by $x \leq y$, if there is a (possibly empty) path of arrows from x to y. We say that x and y are in *conflict*, denoted by $x\#y$, if there is a place z, different from x and y, from which one can reach x and y, exiting z by different arrows. Finally, we say that x and y are *concurrent*, denoted by $x \mathbin{co} y$, if neither $x < y$ nor $y < x$ nor $x\#y$ hold. A *co-set* is a set of nodes X such that $x \mathbin{co} y$ for every $x, y \in X$. *Occurrence nets* are those satisfying the following three properties: the net, seen as a directed graph, has no cycles; every place has at most one input transition; and, no node is in self-conflict, i.e., $x\#x$ holds for no x. A place of an occurrence net is *minimal* if it has no input transitions. The net of Fig. 2 is an infinite occurrence net with minimal places a, b. The *default initial marking* of an occurrence net puts one token on each minimal place an none in the rest.

Branching processes. We associate to Σ a set of *labelled* occurrence nets, called the *branching processes* of Σ. To avoid confusions, we call the places and transitions of branching processes *conditions* and *events*, respectively. The conditions and events of branching processes are labelled with places and transitions of Σ, respectively. The conditions and events of the branching processes are subsets from two sets $\mathcal{B}$ and $\mathcal{E}$, inductively defined as the smallest sets satisfying the following conditions:

- $\perp \in \mathcal{E}$, where $\perp$ is an special symbol;
- if $e \in \mathcal{E}$, then $(p, e) \in \mathcal{B}$ for every $p \in P$;
- if $\emptyset \subset X \subseteq \mathcal{B}$, then $(t, X) \in \mathcal{E}$ for every $t \in T$.

In our definitions of branching process (see below) we make consistent use of these names: The label of a condition (p, e) is p, and its unique input event is e. Conditions $(p, \perp)$ have no input event, i.e., the special symbol $\perp$ is used for the minimal places of the occurrence net. Similarly, the label of an event (t, X) is t, and its set of input conditions is X. The advantage of this scheme is that a branching process is completely determined by its sets of conditions and events. We make use of this and represent a branching process as a pair (B, E).

Definition 1. *The set of* finite branching processes *of a net system Σ with the initial marking $M_0 = \{p_1, \ldots, p_n\}$ is inductively defined as follows:*

- $(\{(p_1, \perp), \ldots, (p_n, \perp)\}, \emptyset)$ *is a branching process of Σ.*
- *If (B, E) is a branching process of Σ, $t \in T$, and $X \subseteq B$ is a co-set labelled by ${}^\bullet t$, then $(B \cup \{(p, e) \mid p \in t^\bullet\} , E \cup \{e\})$ is also a branching process of Σ, where $e = (t, X)$. If $e \notin E$, then e is called a* possible extension *of (B, E).*

The set of branching processes of Σ is obtained by declaring that the union of any finite or infinite set of branching processes is also a branching process, where union of branching processes is defined componentwise on conditions and events. Since branching processes are closed under union, there is a unique maximal branching process, called the *unfolding* of Σ. The unfolding of our running example is an infinite occurrence net. Figure 2 shows an initial part. Events and conditions have been assigned identificators that will be used in the examples. For instance, the event $(t_1, \{(p_1, \perp)\})$ is assigned the identificator 1.

We take as partial order semantics of Σ its unfolding. This is justified, because it can be easily shown the reachability graphs of Σ and of its unfolding coincide. (Notice that the unfolding of Σ is a *labelled* net system, and so its reachability graph is defined as the *image* under the labelling function of the reachability graph of the *unlabelled* system.)

Configurations. A *configuration* of an occurrence net is a set of events C satisfying the two following properties: C is causally closed, i.e., if $e \in C$ and $e' < e$ then $e' \in C$, and C is conflict-free, i.e., no two events of C are in conflict. Given an event e, we call $[e] = \{e' \in E \mid e' \leq e\}$ the *local configuration* of e. Let *Min* denote the set of minimal places of the branching process. A configuration C of the branching process is associated with a marking of Σ denoted by $Mark(C) = l((Min \cup C^\bullet) \setminus {}^\bullet C)$. The corresponding set of

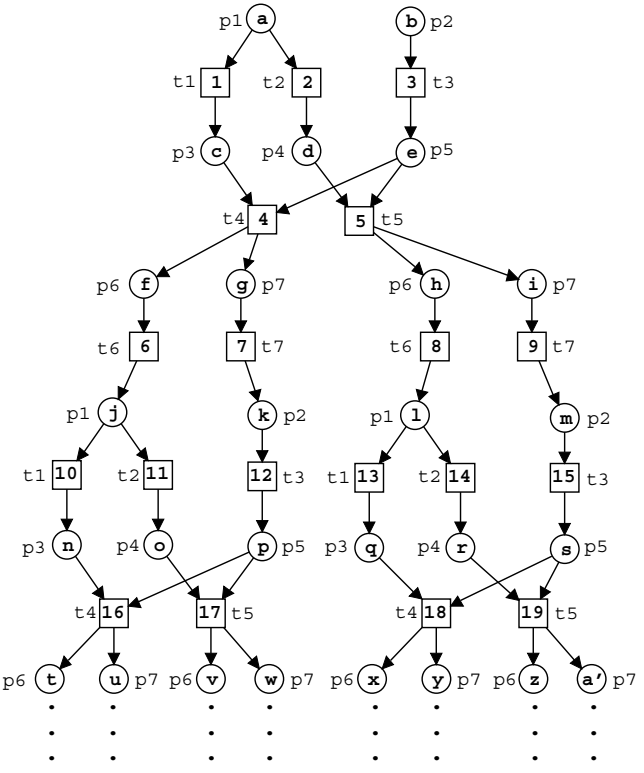


Fig. 2. The unfolding of Σ

conditions associated with a configuration is called a *cut*, and it is defined as $Cut(C) = ((Min \cup C^\bullet) \setminus {}^\bullet C)$.

In Fig. 2, $\{1, 3, 4, 6\}$ is a configuration, and $\{1, 4\}$ (not causally closed) or $\{1, 2\}$ (not conflict-free) are not. A set of events is a configuration if and only if there is one or more firing sequences of the occurrence net (from the default initial marking) containing each event from the set exactly once, and no further events. These firing sequences are called *linearisations*. The configuration $\{1, 3, 4, 6\}$ has two linearisations, namely 1 3 4 6 and 3 1 4 6. All linearisations lead to the same reachable marking. For example, the two sequences above lead to the marking $\{p_1, p_7\}$. By applying the labelling function to a linearisation we obtain a firing sequence of Σ. Abusing of language, we also call this firing sequence a linearisation. In our example we obtain $t_1t_3t_4t_6$ and $t_3t_1t_4t_6$ as linearisations.

Given a configuration C, we denote by $\uparrow C$ the set of events of the unfolding $\{e \mid e \not\in C \ \wedge \ \forall e' \in C : \neg(e \# e')\}$. Intuitively, $\uparrow C$ corresponds to the behavior of Σ from the marking reached after executing any of the linearisations of C. We call $\uparrow C$ the *continuation* after C of the unfolding of Σ. If C_1 and C_2 are two finite configurations leading to the same marking, i.e. $Mark(C_1) = M = Mark(C_2)$, then $\uparrow C_1$ and $\uparrow C_2$ are isomorphic, i.e., there is a bijection between them which

preserves the labelling of events and the causal, conflict, and concurrency relations (see [6]).

Adequate orders. To implement a net unfolding algorithm we need the notion of *adequate order* on configurations [6]. Given a configuration C of the unfolding of Σ, we denote by $C \oplus E$ the set $C \cup E$, under the condition that $C \cup E$ is a configuration satisfying $C \cap E = \emptyset$. We say that $C \oplus E$ is an *extension* of C. Now, let C_1 and C_2 be two finite configurations leading to the same marking. Then $\uparrow C_1$ and $\uparrow C_2$ are isomorphic. This isomorphism, say f, induces a mapping from the extensions of C_1 onto the extensions of C_2; the image of $C_1 \oplus E$ under this mapping is $C_2 \oplus f(E)$.

Definition 2. *A partial order $\prec$ on the finite configurations of the unfolding of a net system is an* adequate order *if:*

- *$\prec$ is well-founded,*
- *$C_1 \subset C_2$ implies $C_1 \prec C_2$, and*
- *$\prec$ is preserved by finite extensions; if $C_1 \prec C_2$ and $Mark(C_1) = Mark(C_2)$, then the isomorphism f from above satisfies $C_1 \oplus E \prec C_2 \oplus f(E)$ for all finite extensions $C_1 \oplus E$ of C_1.*

Total adequate orders for 1-safe Petri nets and for synchronous products of transition systems have been presented in [6,5].

5 Tableau System

We showed in Section 3 that the model checking problem for LTL-X can be solved by checking the existence of illegal ω-traces and illegal livelocks in $\Sigma_{\neg\varphi}$. In [3] these problems are solved using tableau techniques. A branching process can be seen as a "distributed" tableau, in which conditions are "facts" and events represent "inferences". For two conditions b and b', $b \; co \; b'$ models that the facts represented by b and b' can be simultaneously true. A tableau is constructed by adding new events (inferences) one by one following an adequate order; some events are declared as "terminals", and the construction of the tableau terminates when no new event can be added having no terminals among its predecessors. The tableau systems of [3] require to construct a possibly large set of branching processes. Here we present a new tableau system consisting of one single branching process.[2]

An Adequate Order for LTL. We simplify the implementation of the tableau system by selecting a special adequate order. We use $\prec$ to denote the total adequate order defined for 1-safe Petri nets in [6]. We call an event corresponding to an L-transition an *L-event*. We define for a set of events C the function *before L-event* as $BL(C) = \{e \in C \mid [e] \setminus \{e\}$ contains no L-events$\}$. The function *after L-event* is defined correspondingly as $AL(C) = (C \setminus BL(C))$. We can now define our new adequate order.

[2] For the reader familiar with [3]: the L-transitions in the net system $\Sigma_{\neg\varphi}$ act as glue to connect a set of branching processes (the tableau components of [3]) together into one larger tableau.

Definition 3. *Let C_1 and C_2 be two finite configurations of the unfolding of the product net system $\Sigma_{\neg\varphi}$. $C_1 \prec_{LTL} C_2$ holds if*

- $BL(C_1) \prec BL(C_2)$, *or*
- $BL(C_1) = BL(C_2)$ *and* $C_1 \prec C_2$.

The adequate order $\prec_{LTL}$ is application specific in the sense that it is not an adequate order for an arbitrary net system Σ, but needs some special properties of the net system $\Sigma_{\neg\varphi}$. We have the following result.

Theorem 2. *The order $\prec_{LTL}$ is a total adequate order for finite configurations of the unfolding of $\Sigma_{\neg\varphi}$.*

See [4] for the proof.

New Tableau System. We first divide the unfolding of $\Sigma_{\neg\varphi}$ into two disjoint sets of events. Intuitively, the first set is used for the ω-trace detection part, and the second for the illegal livelock detection part. We define *part-I* to be the set of events e such that $[e]$ does not contain an L-event and *part-II* as the set of events which are not in part-I.

Definition 4. *An event e of the unfolding $\Sigma_{\neg\varphi}$ is a* terminal, *if there exists another event e' such that $Mark([e']) = Mark([e])$, $[e'] \prec_{LTL} [e]$, and one of the following two mutually exclusive cases holds:*

(I) $e \in$ *part-I, and either*
  (a) $e' < e$, *or*
  (b) $\neg(e' < e)$ *and* $\#_I[e'] \geq \#_I[e]$, *where* $\#_I C$ *denotes the number of* I*-events in* C.

(II) $e \in$ *part-II, and either*
  (a) $BL([e']) \prec_{LTL} BL([e])$, *or*
  (b) $BL([e']) = BL([e])$ *and* $\neg(e' \# e)$, *or*
  (c) $BL([e']) = BL([e])$, $e' \# e$, *and* $|[e']| \geq |[e]|$.

A tableau $\mathcal{T}$ *is a branching process (B, E) of $\Sigma_{\neg\varphi}$ such that for every possible extension e of (B, E) at least one of the immediate predecessors of e is a terminal. A terminal is* successful *if it is type (I)(a) and $[e] \setminus [e']$ contains an I-event, or it is of type (II)(b). All other terminals are* unsuccessful. *A tableau $\mathcal{T}$ is* successful *if it contains a successful terminal, otherwise it is* unsuccessful.

Loosely speaking, a tableau is a branching process which cannot be extended without adding a causal successor to a terminal.

We have the following result:

Theorem 3. *Let $\mathcal{T}$ be a tableau for $\Sigma_{\neg\varphi}$.*

- *$\Sigma_{\neg\varphi}$ has an illegal ω-trace iff $\mathcal{T}$ has a successful terminal of type I.*
- *$\Sigma_{\neg\varphi}$ has an illegal livelock iff $\mathcal{T}$ has a successful terminal of type II.*
- *$\mathcal{T}$ contains at most K^2 non-terminal events, where K is the number of reachable markings of $\Sigma_{\neg\varphi}$.*

See [4] for the proof.

6 Generating the Tableau

We describe an implementation of the tableau system of Sect. 5. The main goal is to keep the tableau generation as similar as possible to a conventional prefix generation algorithm [6]. In this way any prefix generation algorithm can be easily adapted to also perform LTL model checking. The tableau generation algorithm (Algorithm 1) is almost identical to the main routine of a prefix generation algorithm. The changes are: an additional block of code devoted to generating the L-events dynamically; a different but easy to implement adequate order; a new cut-off detection subroutine. The main feature of the implementation is the efficient handling of L-transitions, which we discuss next.

Generating the L-transitions Dynamically. Recall that in the synchronization $\Sigma_{\neg\varphi}$ we can for each Büchi state q have as many L-transitions as there are reachable markings of the form (q, s_f, O, H) in the net system $\Sigma_{\neg\varphi}$. Clearly we can not explicitly generate them all due to efficiency reasons. Instead we generate a net system $\Sigma^s_{\neg\varphi}$ (s stands for *static*) in which this set of L-transitions (added by step (6) of the synchronization procedure in Section 3) is replaced by:

(6') Add for each Büchi transition $t = (q, x, q')$ in the net system $\Sigma'_{\neg\varphi}$ (i.e., the synchronization after steps (1)-(5) as defined in Sect. 3) a new transition t'. The preset of t' is equivalent to the preset of t and the postset of t' is empty. Let L (for *livelocks*) be this set of transitions.

We can now dynamically generate any of the (enabled) L-transitions of $\Sigma_{\neg\varphi}$. Namely, for a transition t corresponding to a reachable marking $M = (q, s_f, O, H)$ to be enabled in $\Sigma_{\neg\varphi}$, a transition t' (for some (q, x, q')) must be enabled in $\Sigma^s_{\neg\varphi}$ and the Büchi automaton must accept O^ω when q is given as the initial state. Loosely speaking we test the first label of the sequence using the transition t', and if this test succeeds we check whether O can be infinitely stuttered. (Using this construction it is easy to implement "no-care values" for selected atomic propositions by leaving them out of the preset of t'.) Now generating the postset of t from M is trivial.

Optimizations in Dynamic Creation. We can thus dynamically generate L-transitions for each reachable marking M as required. However, we can do better by using the net unfolding method. The main idea is to generate the unfolding of $\Sigma_{\neg\varphi}$ by using $\Sigma^s_{\neg\varphi}$ to find "candidate" L-events. Assume we have found an event e^s corresponding to a transition t' in the unfolding of $\Sigma^s_{\neg\varphi}$ and the stuttering check described above passes for the marking $M = \mathit{Mark}([e^s])$. Then we add an event e into the unfolding of $\Sigma_{\neg\varphi}$ corresponding to the effect of the transition t in the marking M. If we would directly use the construction above we would also add an event e' to the unfolding of $\Sigma_{\neg\varphi}$ for each marking $M' = (q, s_f, O, H')$ which is reachable from M using only invisible transitions. We now show that adding only the event e suffices: Let E be an extension of $[e]$. If there is an illegal livelock starting from $M' = \mathit{Mark}([e] \oplus E)$ then there is also an illegal livelock starting from M. This can be easily seen to be the case because all extensions E contain only invisible events and thus the set of observable places in both M

and M' is O. Algorithm 1 uses the property described above to add the required L-events dynamically. Another optimization used is the fact that only the places in the presets of invisible transitions (denoted *InvisPre*) need to be added to the postset of an L-transition.

Algorithm 2 is the cut-off detection subroutine. It handles events in *part-I* and *part-II* differently. This is one example implementation, and it closely follows the definition of the tableau. It sets the global boolean variable *success* to *true* and calls the counterexample generation subroutine (Algorithm 3) if it finds a counterexample.

The implementation of the check whether $\mathcal{A}^q_{\neg\varphi}$ accepts O^ω in Algorithm 1 can be done in linear time in the size of the automaton $\mathcal{A}_{\neg\varphi}$ as follows. First restrict $\mathcal{A}_{\neg\varphi}$ to transitions satisfying O, and then use a linear time emptiness checking algorithm (see e.g. [2]) to check whether an accepting loop can be reached starting from q in this restricted automaton. Because $\mathcal{A}_{\neg\varphi}$ is usually quite small compared to the size of the model checked system this should not be a limiting factor. Caching of these check results can also be used if necessary.

The adequate order $\prec_{LTL}$ can also be quite efficiently implemented. We can prove that if a configuration C contains an L-event e, then $BL(C) = [e]$. Now it is also the case that each configuration only includes at most one L-event. By using these two facts a simple and efficient implementation can be devised.

Each time our algorithm adds a non-terminal L-event, it first finds out whether a livelock counterexample can be generated from its future. Only if no counterexample is found, it continues to look for illegal ω-traces and further L-events. Thus we use the adequate order $\prec_{LTL}$ to force a search order similar to that used by Valmari in [19] which detects divergence counterexamples in interleaved state spaces. However, our algorithm is "breadth-first style" and it also does illegal ω-trace detection, a part which is not included in [19].

7 Experimental Results

We have implemented a prototype of the LTL model checking procedure called `unfsmodels`. We use the SPIN tool [12] version 3.4.3 to generate the Büchi automaton $\mathcal{A}_{\neg\varphi}$ and a tool by F. Wallner [22] to generate the synchronization $\Sigma'_{\neg\varphi}$ which is given to the prototype tool as input. The `smodels` tool [18] is used to calculate the set of possible extensions of a branching process. It is a NP-solver which uses logic programs with stable model semantics as the input language. Calculating the possible extensions is a quite demanding combinatorial problem. Actually a decision version of the problem can be show to be NP-complete in the general case [10]. However if the maximum preset size of the transitions $|{}^\bullet t|$ is bounded the problem becomes polynomial [7]. (The problem is closely related to the *clique* problem which has a similar characteristic, for a longer discussion see [7].)

We chose to use `smodels` to solve this combinatorial problem instead of implementing a dedicated algorithm. That choice allowed us to concentrate on other parts of the implementation. The translation employs constructs similar to those presented for the submarking reachability problem in [11], however it

Algorithm 1 *The tableau generation algorithm*

input: The product net system $\Sigma^s_{\neg\varphi} = (P, T, F, M_0)$, where $M_0 = \{p_1, \ldots, p_n\}$.
output: *true* if there is a counterexample, *false* otherwise.
global variables: *success*
begin
$Fin := \{(p_1, \bot), \ldots, (p_n, \bot)\}$;
cut-off $:= \emptyset$;
$pe := PE(Fin)$; /* Compute the set of possible extensions */
success := *false*;
while $pe \neq \emptyset$ and *success* = *false* **do**
    choose an event $e = (t, X)$ in pe such that $[e]$ is minimal
    with respect to $\prec_{LTL}$;
    $Y := t^\bullet$; /* Remember the postset of t */

    /* Create the required L-events dynamically */
    if t is a L-transition **then**
        $M := Mark([e] \setminus \{e\})$; /* The marking $M = (q, s_f, O, H)$ */
        $q := M \cap Q$; /* Extract the Büchi state q */
        /* (Büchi emptiness checking algorithm can be used here) */
        if $\mathcal{A}^q_{\neg\varphi} = (\Gamma, Q, q, \rho, F)$ does not accept O^ω **then**
            continue; /* Discard e because (q, O) is not a checkpoint */
        endif
        $X := Cut([e] \setminus \{e\})$; /* Extend the preset to also remove tokens from H */
        $e := (t, X)$; /* Rename e (i.e., add arcs from all preset conditions to e) */
        $Y := (M \cap InvisPre)$; /* Project M on invisible transition presets */
    endif

    if $[e] \cap$ *cut-off* $= \emptyset$ **then**
        append to *Fin* the event e and a condition (p, e)
            for every place $p \in Y$;
        $pe := PE(Fin)$; /* Compute the set of possible extensions */
        if *is_cutoff*(e) **then**
            cut-off := *cut-off* $\cup \{e\}$;
        endif
    else
        $pe := pe \setminus \{e\}$;
    endif
enddo
return *success*;
end

Algorithm 2 *The is_cutoff subroutine*

input: An event e.
output: *true* if e is a terminal of the tableau, *false* otherwise.
begin
foreach e' such that $Mark([e']) = Mark([e])$ **do** /* $[e'] \prec_{LTL} [e]$ holds */
  if $e \in$ *part-I* **then** /* case (I) */
    if $e' < e$ **then**
      if $[e] \setminus [e']$ contains an I-event **then**
        success := *true*; /* Counterexample found! */
        counterexample(e, e');
      endif
      return *true*;
    else if $\#_I[e'] \geq \#_I[e]$ **then**
      return *true*;
    endif
  else /* case (II) */
    if $BL([e']) \prec_{LTL} BL([e])$ **then**
      return *true*;
    else if $\neg(e' \# e)$ **then** /* $BL([e']) = BL([e])$ holds */
      success := *true*; /* Counterexample found! */
      counterexample(e, e');
      return *true*;
    else if $|[e']| \geq |[e]|$ **then** /* $BL([e']) = BL([e])$ holds */
      return *true*;
    endif
  endif
enddo
return *false*;
end

Algorithm 3 *The counterexample subroutine*

input: A successful event e with the corresponding event e'.
begin
$C_1 := [e] \cap [e']$;
$C_2 := [e] \setminus C_1$;
/* C_1 contains the prefix and C_2 the accepting loop */
print_linearisation(C_1);
print_linearisation(C_2);
end

differs in several technical details. The translation is linear in the sizes of both the net and the prefix, however we will not present it here due to space restrictions.

For benchmarks we used a set of LTL model checking examples collected by C. Schröter. The experimental results are collected in Fig. 3. The 1-safe net systems used in the experiments are as follows:

- BRUIJN(2), DIJKST(2), and KNUTH(2): Mutex algorithms modeled by S. Melzer.
- BYZA4_0B and BYZA4_0B: Byzantine agreement algorithm versions modeled by S. Merkel [16].
- RW1W1R, RW1W3R and RW2W1R: Readers and writers synchronization modeled by S. Melzer and S. Römer [15].
- PLATE(5): A production cell example from [13], modeled by M. Heiner and P. Deussen [9].
- EBAHN: A train model by K. Schmidt.
- ELEV(3) and ELEV(4): Elevator models by J. C. Corbett [1], converted to nets by S. Melzer and S. Römer [15].
- RRR(xx): Dining philosophers with xx philosophers, modeled by C. Schröter.

The reported running times only include `unfsmodels 0.9` running times, as the Büchi automata generation and the synchronization with the original net system took insignificant amount of time. All the running times are reported as the sum of system and user times as reported by the `/usr/bin/time` command when run on a PC with an AMD Athlon 1GHz processor, 512MB RAM, using `gcc 2.95.2` and Linux 2.2.17. The times are all averaged over 5 runs.

The `unfsmodels` tool in an on-the-fly tool in the sense that it stops the prefix (tableau) generation if it finds a counterexample during the unfolding. The reported prefix sizes in this case are the partial prefix at the time the counterexample was found. The tool can also be instructed to generate a *conventional prefix* using the prefix generation algorithm described in [6] for comparison.

Problem	B_{LTL}	E_{LTL}	$\#c_{LTL}$	Cex	B_{Fin}	E_{Fin}	$\#c_{Fin}$	States	Sec_{LTL}	Sec_{Fin}
BRUIJN(2)	2874	1336	327	N	2676	1269	318	5183	13.1	11.0
DIJKST(2)	1856	968	230	N	1700	921	228	2724	4.8	3.8
KNUTH(2)	2234	1044	251	N	2117	1009	251	4483	7.1	6.1
BYZA4_0B	1642	590	82	N	1630	587	82	>2000000	7.0	6.9
BYZA4_2A	401	125	4	N	396	124	4	>2500000	0.3	0.3
RW1W1R	568	296	32	N	563	295	32	2118	0.5	0.5
RW1W3R	28143	15402	5210	N	28138	15401	5210	165272	1863.4	1862.2
RW2W1R	18280	9242	1334	N	18275	9241	1334	127132	1109.6	1108.2
PLATE(5)	1803	810	12	N	1619	768	12	1657242	14.0	11.8
EBAHN	151	62	21	Y	1419	673	383	7776	0.0	0.7
ELEV(3)	124	64	10	Y	7398	3895	1629	7276	0.1	91.7
ELEV(4)	154	80	13	Y	32354	16935	7337	48217	0.1	1706.2
RRR(10)	88	42	5	Y	85	45	19	14985	0.0	0.0
RRR(20)	167	81	8	Y	161	81	32	>10000000	0.1	0.0
RRR(30)	240	114	9	Y	230	110	41	>10000000	0.2	0.1
RRR(50)	407	201	18	Y	388	188	70	>10000000	0.7	0.5

Fig. 3. Experimental results.

In Fig. 3 the columns of the table have the following meanings:

- Problem: The name of the problem with the size of the instance.
- B_{LTL}, E_{LTL}, and $\#c_{LTL}$: The number of conditions, events, and the number of events which are terminals in the LTL prefix, respectively.
- Cex: N - There was no counterexample, the formula holds. Y - There was a counterexample, the formula does not hold.
- B_{Fin}, E_{Fin}, and $\#c_{Fin}$: The size of different parts of the finite complete prefix as above but for the original net system Σ using the conventional prefix generation algorithm described in [6].
- States: The number of states n in the reachability graph of the original net system Σ obtained using the `PROD` tool [21], or a lower bound $> n$.
- Sec_{LTL}: The time used by `unfsmodels` in seconds needed to find a counterexample or to show that there is none.
- Sec_{Fin}: The time used by `unfsmodels` in seconds needed to generate a finite complete prefix of the original net system Σ.

At this point there are a couple of observations to be made. First of all, on this set of example nets and formulas, the speed of computing a LTL prefix is almost identical to the speed of computing a conventional prefix (of comparable size). The main reason for this is that the time needed to compute the possible extensions dominates the computation time in our prototype. Thus the (slightly) more complicated algorithm needed for the cut-off detection do not contribute in a major way to the running time of the tool. Secondly, on all of the experiments, the size of the LTL prefix is of the same order of magnitude as the conventional prefix. Thus in this set of examples the quadratic worst-case blow-up (possible according to Theorem 3) does not materialize. We expect this to be the case also in other examples when the used LTL formulas are short and the properties to be checked are local, in the sense that the product net system preserves most of the concurrency present in the original net system.

In Fig. 4 a detailed breakdown of the different components of the LTL prefix is given. The subscripts I and II denote the part of the prefix used for ω-trace and livelock checking, respectively (i.e., events in *part-I* and *part-II*). Column *Cpt* contains the number of checkpoints, i.e. how many of the L-events are checkpoints. Finally *Formula type* gives the type of the formula being checked.

In Fig. 4 we can also see that in the cases a counterexample was found it was found after only a small amount of the prefix was generated. Actually in all the experiments the counterexample was a livelock counterexample, and the livelock was found from the first checkpoint found during the prefix generation. This allowed the LTL model checking procedure to terminate quite early with a counterexample in many case, see e.g. the ELEV(4) example.

The net systems used in experiments and `unfsmodels 0.9` are available at ⟨`http://www.tcs.hut.fi/~kepa/experiments/spin2001/`⟩.

8 Conclusions

We have presented an implementation of the tableau system of [3]. We have been able to merge the possibly large set of tableaux of [3] into a single one. In this way, the algorithm for model checking LTL with unfoldings remains

Problem	B_I	E_I	$\#c_I$	B_{II}	E_{II}	$\#c_{II}$	Cpt	Formula type
BRUIJN(2)	2874	1336	327	0	0	0	0	$\Box\neg(p_1 \wedge p_2)$
DIJKST(2)	1856	968	230	0	0	0	0	$\Box\neg(p_1 \wedge p_2)$
KNUTH(2)	2234	1044	251	0	0	0	0	$\Box\neg(p_1 \wedge p_2)$
BYZA4_0B	1642	590	82	0	0	0	0	$\Box(p_1 \rightarrow \Diamond p_2)$
BYZA4_2A	401	125	4	0	0	0	0	$\Box(p_1 \rightarrow \Diamond p_2)$
RW1W1R	568	296	32	0	0	0	0	$\Box(p_1 \rightarrow \Diamond p_2)$
RW1W3R	28143	15402	5210	0	0	0	0	$\Box(p_1 \rightarrow \Diamond p_2)$
RW2W1R	18280	9242	1334	0	0	0	0	$\Box(p_1 \rightarrow \Diamond p_2)$
PLATE(5)	1803	810	12	0	0	0	0	$\Box((p_1 \wedge \neg p_2 \wedge \neg p_3) \vee (\neg p_1 \wedge p_2 \wedge \neg p_3) \vee (\neg p_1 \wedge \neg p_2 \wedge p_3))$
EBAHN	113	48	20	38	14	1	1	$\Box\neg(p_1 \wedge p_2)$
ELEV(3)	22	10	0	102	54	10	1	$\Box(p_1 \rightarrow \Diamond p_2)$
ELEV(4)	25	12	0	129	68	13	1	$\Box(p_1 \rightarrow \Diamond p_2)$
RRR(10)	40	14	0	48	28	5	1	$\Box(p_1 \rightarrow \Diamond p_2)$
RRR(20)	73	27	0	94	54	8	1	$\Box(p_1 \rightarrow \Diamond p_2)$
RRR(30)	104	38	0	136	76	9	1	$\Box(p_1 \rightarrow \Diamond p_2)$
RRR(50)	173	67	0	234	134	18	1	$\Box(p_1 \rightarrow \Diamond p_2)$

Fig. 4. Detailed LTL tableau statistics.

conceptually similar to the algorithms used to generate prefixes of the unfolding containing all reachable states [6,5]: We just need more sophisticated adequate orders and cut-off events. The division of the tableau into *part-I* and *part-II* events is the price to pay for a partial-order approach to model checking. Other partial-order techniques, like the one introduced by Valmari [19], also require a special treatment of divergences or livelocks. [3] We have shown that the conditions for checking if *part-I* or *part-II* events are terminals remain very simple.

In our tableau system the size of a tableau may grow quadratically in the number of reachable states of the system. We have not been able to construct an example showing that this bound can be reached, although it probably exists. In all experiments conducted so far the number of events of the tableau is always smaller than the number of reachable states. In examples with a high degree of concurrency we obtain exponential compression factors.

The prototype implementation was created mainly for investigating the sizes of the generated tableau. Implementing this procedure in a high performance prefix generator such as the one described in [5] is left for further work.

Acknowledgements. We would like to thank Claus Schröter for collecting the set of LTL model checking benchmarks used in this work.

[3] The idea of dynamically checking which L-transitions are checkpoints could also be used with the approach of [19] to implement state based LTL-X model checking.

References

1. J. C. Corbett. Evaluating deadlock detection methods for concurrent software. Technical report, Department of Information and Computer Science, University of Hawaii at Manoa, 1995.
2. 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.
3. J. Esparza and K. Heljanko. A new unfolding approach to LTL model checking. In *Proceedings of 27th International Colloquium on Automata, Languages and Programming (ICALP'2000)*, pages 475–486, July 2000. LNCS 1853.
4. J. Esparza and K. Heljanko. Implementing LTL model checking with net unfoldings. Research Report A68, Helsinki University of Technology, Laboratory for Theoretical Computer Science, Espoo, Finland, March 2001. Available at ⟨http://www.tcs.hut.fi/Publications/reports/A68abstract.html⟩.
5. J. Esparza and S. Römer. An unfolding algorithm for synchronous products of transition systems. In *Proceedings of the 10th International Conference on Concurrency Theory (Concur'99)*, pages 2–20, 1999. LNCS 1664.
6. J. Esparza, S. Römer, and W. Vogler. An improvement of McMillan's unfolding algorithm. In *Proceedings of 2nd International Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'96)*, pages 87–106, 1996. LNCS 1055.
7. J. Esparza and C. Schröter. Reachability analysis using net unfoldings. In *Proceeding of the Workshop Concurrency, Specification & Programming 2000, volume II of Informatik-Bericht 140*, pages 255–270. Humboldt-Universität zu Berlin, 2000.
8. R. Gerth, D. Peled, M. Y. Vardi, and P. Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *Proceedings of 15th Workshop Protocol Specification, Testing, and Verification*, pages 3–18, 1995.
9. M. Heiner and P. Deussen. Petri net based qualitative analysis - A case study. Technical Report Technical Report I-08/1995, Brandenburg Technische Universität Cottbus, Cottbus, Germany, December 1995.
10. K. Heljanko. Deadlock and reachability checking with finite complete prefixes. Research Report A56, Helsinki University of Technology, Laboratory for Theoretical Computer Science, Espoo, Finland, December 1999. Licentiate's Thesis. Available at ⟨http://www.tcs.hut.fi/Publications/reports/A56abstract.html⟩.
11. K. Heljanko. Using logic programs with stable model semantics to solve deadlock and reachability problems for 1-safe Petri nets. *Fundamenta Informaticae*, 37(3):247–268, 1999.
12. G. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997.
13. C. Lewerentz and T. Lindner. *Formal Development of Reactive Systems: Case Study Production Cell.* Springer-Verlag, 1995. LNCS 891.
14. K. L. McMillan. *Symbolic Model Checking.* Kluwer Academic Publishers, 1993.
15. S. Melzer and S. Römer. Deadlock checking using net unfoldings. In *Proceedings of 9th International Conference on Computer-Aided Verification (CAV '97)*, pages 352–363, 1997. LNCS 1254.
16. S. Merkel. Verification of fault tolerant algorithms using PEP. Technical Report TUM-19734, SFB-Bericht Nr. 342/23/97 A, Technische Universität München, München, Germany, 1997.
17. W. Reisig. *Petri Nets, An Introduction.* Springer-Verlag, 1985.

18. P. Simons. *Extending and Implementing the Stable Model Semantics*. PhD thesis, Helsinki University of Technology, Laboratory for Theoretical Computer Science, April 2000. Also available on the Internet at ⟨http://www.tcs.hut.fi/Publications/reports/A58abstract.html⟩.
19. A. Valmari. On-the-fly verification with stubborn sets. In *Proceeding of 5th International Conference on Computer Aided Verification (CAV'93)*, pages 397–408, 1993. LNCS 697.
20. M. Y. Vardi. An automata-theoretic approach to linear temporal logic. In *Logics for Concurrency: Structure versus Automata*, pages 238–265, 1996. LNCS 1043.
21. K. Varpaaniemi, K. Heljanko, and J. Lilius. PROD 3.2 - An advanced tool for efficient reachability analysis. In *Proceedings of the 9th International Conference on Computer Aided Verification (CAV'97)*, pages 472–475, 1997. LNCS 1254.
22. F. Wallner. Model checking LTL using net unfoldings. In *Proceeding of 10th International Conference on Computer Aided Verification (CAV'98)*, pages 207–218, 1998. LNCS 1427.

Directed Explicit Model Checking with HSF-SPIN

Stefan Edelkamp, Alberto Lluch Lafuente, and Stefan Leue

Institut für Informatik
Albert-Ludwigs-Universität
Georges-Köhler-Allee
D-79110 Freiburg
{edelkamp,lafuente,leue}@informatik.uni-freiburg.de

Abstract. We present the explicit state model checker HSF-SPIN which is based on the model checker SPIN and its Promela modeling language. HSF-SPIN incorporates directed search algorithms for checking safety and a large class of LTL-specified liveness properties. We start off from the A* algorithm and define heuristics to accelerate the search into the direction of a specified failure situation. Next we propose an improved nested depth-first search algorithm that exploits the structure of Promela Never-Claims. As a result of both improvements, counterexamples will be shorter and the explored part of the state space will be smaller than with classical approaches, allowing to analyze larger state spaces. We evaluate the impact of the new heuristics and algorithms on a set of protocol models, some of which are real-world industrial protocols.

1 Introduction

Model Checking [3] is a formal analysis technique that has been developed to automatically validate functional properties for software or hardware systems. The properties are usually specified using some sort of a temporal logic or using automata. There are two primary approaches to model checking. First, *Symbolic* Model Checking [21] uses binary decision diagrams to represent the state set. The second formalization uses an *explicit* representation of the system's global state graph. An explicit state model checker evaluates the validity of the temporal properties over the model by interpreting its global state transition graph as a Kripke structure. In this paper we focus on explicit state model checking and its application to the validation of communication protocols. The protocol model we consider is that of collections of extended communicating finite state machines as described, for instance, in [2] and [12]. Communication between two processes is either realized via synchronous or asynchronous message passing on communication channels (queues) or via global variables. Sending or receiving a message is an event that causes a state transition. The system's global state space is generated by the asynchronous cross product of the individual communicating finite state machines (CFSMs). For the description of the state machine model

M.B. Dwyer (Ed.): SPIN 2001, LNCS 2057, pp. 57–79, 2001.

we use the language Promela [17], and for the validation of Promela models we use the model checker SPIN[1] [16].

The use of model checking in system design has the great advantage over the use of deductive formal verification techniques that once the requirements are specified and the model has been programmed, model checking validation can be implemented as a push-button process that either yields a positive result, or returns an error trail. Two primary strategies for the use of model checking in the system design process can be observed.

- *Complete validation* is used to certify the quality of the product or design model by establishing its absolute correctness. However, due to the large size of the search space for realistic systems it is hardly ever possible to explore the full state space in order to decide about the correctness of the system. In these cases, it either takes too long to explore all states in order to give an answer within a useful time span, or the size of the state space is too large to store it within the bounds of available main memory.
- The second strategy, which also appears to the more commonly one used, is to employ the model checker as a *debugging aid* to find residual design and code faults. In this setting, one uses the model checker as a search tool for finding violations of desired properties. Since complete validation is not intended, it suffices to use hashing-based partial exploration methods that allow for covering a much larger portion of the system's state space than if complete exploration is needed.

When pursuing debugging, there are some more objectives that need to be addressed. First, it is desirable to make sure that the length of a search until a property violation is found is short, so that error trails are easy to interpret. Second, it is desirable to guide the search process to quickly find a property violation so that the number of explored states is small, which means that larger systems can be debugged this way. To support these objectives we present an approach to *Directed Model Checking* in our paper.

Our model-checker HSF-SPIN extends the SPIN framework with various heuristic search algorithms to support directed model checking. Experimental results show that in many cases the number of expanded nodes and the length of the counter-examples are significantly reduced. HSF-SPIN has been applied to the detection of deadlocks, invariant and assertion violations, and to the validation of LTL properties. In most instances the estimates used in the search are derived from the properties to be validated, but HSF-SPIN also allows some designer intervention so that targets for the state space search can be specified explicitly in the Promela code.

We propose an improvement of the depth-first search algorithm that exploits the structure of never claims. For a broad subset of the specification patterns described in [8], such as *Response* and *Absence*, the proposed algorithm performs less transitions during state space search and finds shorter counterexamples compared to classical nested-depth first search. Given a Promela *Never Claim* A the algorithm automatically computes a partitioning of A in linear time with respect

[1] Available from `netlib.bell-labs.com/netlib/spin`.

to the number of states in A. The obtained partitioning into non-, fully and partially accepting strongly connected components will be exploited during state space exploration. We improve the heuristic estimate by taking the structure of the temporal property into account.

Related Work. In earlier work on the use of directed search in model checking the authors apply best-first exploration to protocol validation [20]. They are interested in typical safety properties of protocols, namely unspecified reception, absence of deadlock and absence of channel overflow. In the heuristics they therefore use an estimate determined by identifying *send* and *receive* operations. In the analysis of the X.21 protocol they obtained savings in the number of expansion steps of about a factor of 30 in comparison with a typical depth first search strategy. We have incorporated this strategy in HSF-SPIN. While the approach in [20] is limited to the detection of deadlocks, channel overflows and unspecified reception in protocols with asynchronous communication, the approach in this paper is more general and handles a larger range of errors and communication types. While the labelings used in [20] are merely stochastic measures that will not yield optimal solutions, the heuristics we propose are lower bound estimators and hence allow us to find optimal solutions.

The authors of [30] use BDD-based symbolic search within the Murϕ validation tool. The best first search procedure they propose incorporates symbolic information based on the Hamming distance of two states. This approach has been improved in [26], where a BDD-based version of the A* algorithm [11] for the μcke model checker [1] is presented. The algorithm outperforms symbolic breadth-first search exploration for two scalable hardware circuits. The heuristic is determined in a static analysis prior to the search taking the actual circuit layout and the failure formula into account. The approach to symbolic guided search in CTL model checking documented in [25] applies 'hints' to avoid sections of the search space that are difficult to represent for BDDs. This permits splitting the fix-point iteration process used in symbolic exploration into two parts yielding under- and overapproximation of the transition relation, respectively. Benefits of this approach are simplification of the transition relation, avoidance of BDD blowup and a reduced amount of exploration for complicated systems. However, in contrast to our approach providing 'hints' requires user intervention. Also, this approach is not directly applicable to explicit state model checking, which is our focus.

Exploiting structural properties of the Büchi Automaton in explicit state mode checking has been considered in the literature in the context of weak alternating automata (WAA) [5]. WAA were invented to reason about temporal logics, generalize the transition function with boolean expressions of the successor set, and partition the automaton structure. The classification of the states of a WAA differs from ours, since the partitioning into disjoint sets of states that are either all accepting or all rejecting does not imply our partitioning.

The simplification of Büchi automata proposed in [27] is inferred from an LTL property, whereas we work on the basis of Büchi automata. This work also considers a partitioning according to WAA-type weakness conditions and hence differs from the approach taken in our paper.

The approach taken in [29] addresses explicit CTL* model checking in SPIN using hesistant alternating automata (HAAs). The paper shows that the performance of the proposed 'LTL nonemptiness game' is in fact a reformulation and improvement of nested depth-first search. Both the partitioning and the context of HAA model checking are significanty different from our setting.

In our paper we will use a number of protocols as benchmarks. These include Lynch's protocol, the alternating bit protocol, Barlett's protocol, an erroneous solution for mutual exclusion (mutex)[2], the optical telegraph protocol [17], an elevator model[3], a deadlock solution to Dijkstra's dining philosopher problem, and a model of a concurrent program that solves the stable marriage problem [22]. Real-World examples that we use include the Basic Call processing protocol [23], a model of a relay circuit [28], the Group Address Registration Protocol GARP [24], the CORBA GIOP protocol [18], and the telephony model POTS [19][4].

Precursory Work. The precursor [10] to this paper considers safety property analysis for simple protocols. In the current paper we extend on this work by refining the safety heuristics, by providing an approach to validating LTL-specified safety properties, and by experimenting with a larger set of protocols.

Structure of Paper. In Section 2 we review automata-based model checking. Section 3 discusses the analysis of safety properties in directed model checking and describes the use of the A* algorithm for this purpose. In Section 4 we discuss liveness property analysis. We present approaches to improve search strategies for validation of LTL properties. In Section 5 we discuss how to devise informative heuristic estimates in communication protocols. The new protocol validator HSF-SPIN is presented in Section 6. Experimental results for various protocols are discussed in Section 7. We conclude in Section 8.

2 Automata-Based Model Checking

In this Section we review the automata theoretic framework for explicit state model checking. Since we model infinite behaviors the appropriate formalization for words on the alphabet of transitions sequences are Büchi-Automata. They inherit the structure of finite state automata but with a different acceptance condition. A run (infinite path) in a Büchi-Automaton is accepting if the set of states that appear infinitely often in the run has a non-empty intersection with the set of accepting states. The language $L(\mathcal{A})$ of a Büchi-Automaton $\mathcal{A}$ consists of all accepting runs. The expressiveness of Büchi-Automata includes LTL.

Formally, LTL specification $F(M)$ according to a Kripke Model M are defined as follows: All predicates a are in $F(M)$ and if f and g are in $F(M)$, so are

[2] Available from `netlib.bell-labs.com/netlib/spin`

[3] Available from `www.inf.ethz.ch/personal/biere/teaching/mctools/elsim.html`

[4] The Promela sources and further information about these models can be obtained from `www.informatik.uni-freiburg.de/~lafuente/models/models.html`

$\neg f, f \vee g, f \wedge g, X\ f, F\ f, G\ f$, and $f\ U\ g$. In LTL, temporal modalities are expressed through the operators $\Box$ for *globally* (G) and $\Diamond$ for *eventually* (F).

In automata-based Model Checking we construct the Büchi-Automaton $\mathcal{A}$ and the automaton $\mathcal{B}$ that represents the system M. $\mathcal{A}$ is sometimes obtained by translating an LTL formula into a Büchi Automaton. While this translation is exponential in the size of the formula, typical property specifications result in small LTL formulae so that this complexity is not a practical problem. The system $\mathcal{B}$ satisfies $\mathcal{A}$ when $L(\mathcal{B}) \subseteq L(\mathcal{A})$. This is equivalent to $L(\mathcal{B}) \cap \overline{L(\mathcal{A})} = \emptyset$, where $\overline{L(\mathcal{A})}$ denotes the complement of $L(\mathcal{A})$. Note that Büchi-Automata are closed under complementation. In practice, $\overline{L(\mathcal{A})}$ can be computed more efficiently by deriving a Büchi-Automaton from the negated formula. Therefore, in the SPIN validation tool LTL formulae are first negated, and then translated into a *Never Claim* (automaton) that represent the negated formula. As an example we consider the commonly used *response* property which states that whenever a certain request event p occurs a response event q will eventually occur. Response properties are specified in LTL as $\Box(p \rightarrow \Diamond q)$ and the negation is $\Diamond(p \wedge\ \Box\neg\ q)$. The Büchi-Automaton and the corresponding Promela Never-Claim for the negated response property are illustrated in Figure 1.

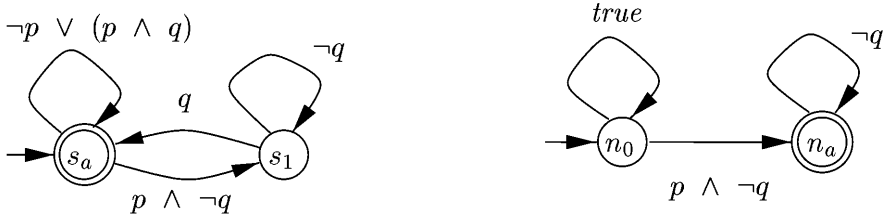


Fig. 1. Büchi-Automaton for response property (left) and for its negation (right).

The emptiness of $L(\mathcal{B}) \cap \overline{L(\mathcal{A})}$ is determined using an on-the-fly algorithm based on the synchronous product of $\mathcal{A}$ and $\mathcal{B}$: Assume that $\mathcal{A}$ is in state s and $\mathcal{B}$ is in state t. $\mathcal{B}$ can perform a transition out of t if $\mathcal{A}$ has a successor state s' of s such that the label of the edge from s to s' represents a proposition satisfied in t. A run of the synchronous product is accepting if it contains a cycle through at least one accepting state of $\mathcal{A}$. $L(\mathcal{B}) \cap \overline{L(\mathcal{A})}$ is empty if the synchronous product does not have an accepting run. We use the standard distinction of safety and liveness properties. Safety properties refer to states, whereas liveness properties refer to paths in the state transition diagram. Safety properties can be validated through a simple depth-first search on the system's state space, while liveness properties require a two-fold nested depth-first search. When property violations are detected, the model checker will return a witness (counterexample) which consists of a trace of events or states encountered.

3 Searching for Safety Property Violations

The detection of a safety error consists of finding a state in which some property is violated. Typically, the algorithms used for this purpose are depth-first and breadth-first searches. Depth-first search is memory efficient, but not very fast in finding target states. We describe how heuristic search algorithms can be used instead in order to accelerate the exploration.

Heuristic search algorithms take additional search information in form of a evaluation function into account that returns a number purporting to describe the desirability of expanding a node. When the nodes are ordered so that the one with the best evaluation is expanded first and if the evaluation function estimates the cost of the cheapest path from the current state to a desired one, the resulting greedy best-first search (BF) often finds solutions fast. However, it may suffers from the same defects as depth-first search – it is not optimal and may be stuck in dead-ends or local minima.

Breadth-first search (BFS), on the other hand, is complete and optimal but very inefficient. Therefore, A* [13] combines both approaches for a new evaluation function by summing the generating path length $g(u)$ and the estimated cost of the cheapest path $h(u)$ to the goal yielding the estimated cost $f(u) = g(u) + h(u)$ of the cheapest solution through u. If $h(u)$ is a lower bound then A* is optimal. Table 1 depicts the implementation of A* to search safety violations, where $g(u)$ is the length of the traversed path to u and $h(u)$ is the estimate from u to a failure state.

Table 1. The A* Algorithm Searching for Violations of Safety Properties.

```
A*(s)
  Open ← {(s, h(s))}; Closed← {}
  while (Open ≠ ∅)
    u ← Deletemin(Open); Insert(Closed,u)
    if (failure(u)) exit Safety Property Violated
    for all v in Γ(u)
      f'(v) ← f(u) + 1 + h(v) − h(u)
      if (Search(Open, v))
        if (f'(v) < f(v))
          DecreaseKey(Open, (v, f'(v))
      else if (Search(Closed, v))
        if (f'(v) < f(v))
          Delete(Closed, v); Insert(Open, (v, f'(v))
      else Insert(Open, (v, f'(v))
```

Similar to Dijkstra's single source shortest path exploration [7], starting with the initial state, A* extracts states from the priority queue *Open* until a failure

state is found. In a uniform-cost graph with integral lower-bound estimate the f-values are integer and bounded by a constant, such that the states can be kept in doubly-linked lists stored in buckets according to their priorities [6]. Therefore, given a node reference *Insert* and *Close* can be executed in constant time while the operation *DeleteMin* increases the bucket index for the next node to be expanded. If the differences of the priorities of successive nodes are bounded by a constant, *DeleteMin* runs in $O(1)$. Nodes that have already been expanded might be encountered on a shorter path. Contrary to Dijkstra's algorithm, A* deals with them by possibly re-inserting nodes from the set of already expanded nodes into the set of priority queue nodes (re-opening).

Figure 2 depicts the impact of heuristic search in a grid graph with all edge costs being 1. If $h \equiv 0$, A* reduces to Dijkstra's algorithm, which in case of uniform graphs further collapses to BFS. Therefore, starting with s all nodes shown are added to the (priority) queue until the goal node t is expanded. If we use $h(u)$ as the Euclidian distance $\|u-t\|_2$ to state t, then only the nodes in the hatched region are ever removed from the priority queue.

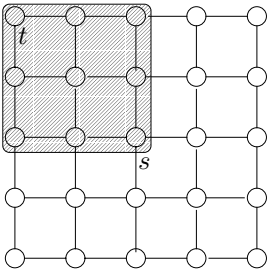


Fig. 2. The Effect of Heuristic Search in a Grid Graph.

Weightening scales the influence of the heuristic estimate such that the combined merit function f of the generating path length g and the heuristic estimate h is given by $f(u) = \alpha g(u) + (1-\alpha)h(u)$ for all states u and $\alpha \in [0,1]$. In case $\alpha < 0.5$, optimality of the search algorithms is affected, for $\alpha = 0$ we exibit BF, and for $\alpha = 1$ we simulate BFS.

4 Searching for Liveness Property Violations

Liveness properties refer to paths of the state transition graph and the detection of liveness property violations entails searching for cycles in the state graph. This is typically achieved by a nested depth-first search (Nested-DFS) that can be implemented with two stacks as shown in Figure 3 (cf. [3]).

One feature of this algorithm is that a state, once *flagged* will not be considered further on. For the correctness of the algorithm the post-order traversal of

```
Nested-DFS(s)
    hash(s)
    for all successors s' of s do
      if s' not in the hash table then Nested-DFS(s')
    if accept(s) then Detect-Cycle(s)

Detect-Cycle(s)
    flag(s)
    for all successors s' of s do
      if s' on Nested-DFS-Stack then exit LTL-Property violated
      else if s' not flagged then Detect-Cycle(s')
```

Fig. 3. Nested-Depth-First-Search.

the search tree is crucial, such that the secondary depth-first traversal only encounters nodes that have already been visited in the main search routine. Therefore in the application of heuristic methods for the first traversal of Nested-DFS, we are restricted to move ordering techniques: using a heuristic function for establishing the order in which the successors of a state will be explored. However, the second search can be improved by directed cycle detection search. Since we are aiming for those states in the first stack we can use heuristics to perform a directed search for the cycle-closing states. The disadvantage of a pre-ordered nested search approach (search the acceptance state in the Never-Claim and, once encountered, search for a cycle) is its quadratic worst-case time and linear memory overhead, since the second search has to be invoked with a newly initialized visited list. To address this drawback we developed a single pass DFS algorithm applicable to a large set of practical property specifications.

4.1 Classification of Never Claims

Strongly connected components (SCC) partition a directed graph into groups such that there is no cycle combining two components. A subset of nodes in a directed graph is strongly connected if for all nodes u and v there is a path from u to v and a path from v to u. SCCs are maximal in this sense and can be computed in linear time [4]. In the Never-Claim of the example in Figure 1 we find two strongly connected components: the first is formed by n_0 and the second by n_a. Furthermore, there is no path from the second SCC to the first. Therefore, accepting cycles in the Never-Claim exist only in the second SCC. Accepting cycles in the synchronous product automaton are composed of states in which the Never-Claim is always in state n_a (second SCC). A cycle is found if a state is encounterd on the stack. Moreover, if the local state of the never claim in the found global state belongs to the first SCC, the established cycle is not accepting, and if it belongs to the second SCC it is an accepting one.

In order to generalize the observation suppose that we have pre-computed all SCCs of a given Never-Claim. Due to the synchronicity of the product of

the model automaton and the Never-Claim a cycle in the synchronous product is generated by a cycle in exactly one SCC. Moreover, if the cycle is accepting, so is the corresponding cycle in the SCC of the never claim. Suppose that each SCC is either composed only of non-accepting states or only of accepting states. Then global accepting cycles only contain accepting states, while non-accepting cycles only contain non-accepting states. Therefore, a single depth-first search can be used to detect accepting cycles: if a state s is found in the stack, then the established cycle is accepting if and only if s itself is accepting.

The restriction on the SCC partitioning given by the above rules can be relaxed according to the following classification of the SCCs.

- We call an SCC *accepting* if at least one of its states is accepting, and *non-accepting* (N-SCC) otherwise.
- We call an accepting SCC *fully accepting* (F-SCC) if all of its cycles contain at least one accepting state.
- We call an accepting SCC *partially accepting* (P-SCC) if there is at least one cycle that does not contain an accepting state.

If the Never-Claim contains no partially accepting SCC, then acceptance cycle detection for the global state space can be performed by a single depth-first search: if a state is found in the stack, then it is accepting, if the never state belong to an accepting SCC. A special case occurs if the never claim has an endstate. If this state is reached the never claim is said to be violated; a *bad* sequence is found. We indicate the presence of endstates with the letter S. Bad sequences are tackled similarily to safety properties by standard heuristic search.

The classification of patterns in property specifications [8] reveals that a database of 555 LTL properties partitions into *Absence* (85/555), *Universality* (119/555), *Existence* (27/555), *Response* (245/555), *Precedence(26/555)*, and *Others (53/555)*. Using this pattern scheme and the modifiers *Globally, Before, After, Between*, and *Until* we obtain a partitioning into SSCs according to Table 2.

Table 2. SCC Classification for LTL-Specification Patterns. S indicates the presence of endstates in the never claim, while N, P, F indicate the presenc of at least one N-SCC, P-SCC and F-SCC respectively.

Pattern	Globally	Before	After	Between	Until
Absence	S+N	S+N	S+N	S+N	S+N+P
Universality	S+N	S+N	S+N	S+N+P+F	S+N+P
Existence	F	S+P+N	N+F	S+N+P	S+N+F
Response	N+F	S+N+P+F	N+F	S+N+P+F	S+N+P+F
Precedence	S+N+P	S+N	N+P	S+N	S+N+P

4.2 Improved Nested Depth-First-Search

In this section we present an improvement of the Nested-DFS algorithm called *Improved-Nested-DFS*. It finds acceptance cycles without nested search for all problems which partition into N- or F-components. The algorithm reduces the number of transitions required for full validation of liveness properties. Except for P-SCCs it avoids the post-order traversal. For P-SCCs we guarantee that the second cycle detection traversal is restricted to the strongly connected component of the seed. The Improved-Nested-DFS algorithm is given in Fig. 5. In this Figure, SCC(s) is the SCC of state s, F-SCC(s) determines if the SCC of state s is of type F (fully accepting), P-SCC(s) determines if the SCC of the state is of type P (partially accepting) and *neverstate*(s) denotes the local state of the Never Claim in the global state s. The algorithms considers the successors of a node in depth-first manner and marks all visited nodes with the label *hash*. If a successor s' is already contained in the stack, a cycle C is found. If C corresponds to a cycle in a F-SCC of the *neverstate* of s', it is an accepting one. Cycles for the P-SCCs parts in the never claim are found as in Nested-DFS, with the exception that the successors of a node are pruned which *neverstates* are outside the component. If a endstate in the Never Claim is reached the algorithm terminates inmediately. Figure 4 depicts the different cases of cycles detected in the search. The correctness of Improved-Nested-DFS follows from the fact that all cycles in the state-transition graphs correspond to cycles in the Never-Claim. Therefore, if there is no cycle combining two components in the Never-Claim, so there is none in the overall search space.

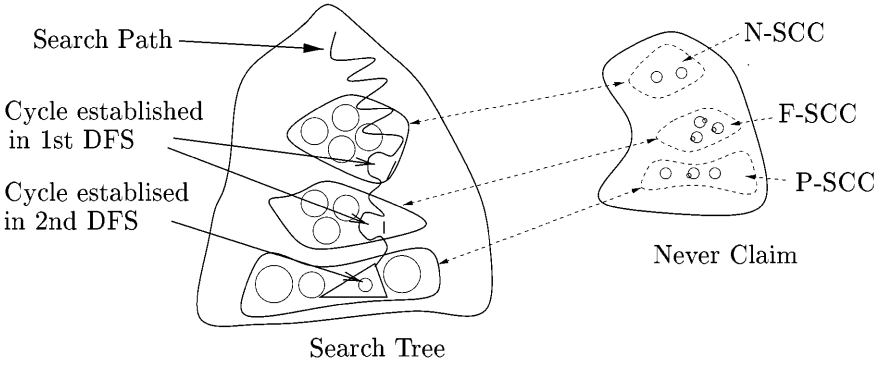


Fig. 4. Visualization of the Different Cases in Improved-Nested-DFS.

As mentioned above, the strongly connected components can be computed in time linear to the size of the Never Claim, a number which is very small in practice. Partitioning the SCCs in *non-accepting*, *partially accepting* and *fully accepting* can also be achieved in linear time by a variant of Nested-DFS in the

```
Improved-Nested-DFS(s)
    hash(s)
    for all successors s' of s do
      if s' in Improved-Nested-DFS-Stack and F-SCC(neverstate(s')) then
        exit LTL-Property violated
      if s' not in the hash table then Nested-DFS(s')
    if accept(s) and P-SCC(neverstate(s)) then Improved-Detect-Cycle(s)

Improved-Detect-Cycle(s)
    flag(s)
    for all successors s' of s do
      if s' on Improved − Nested − DFS-Stack then exit LTL-Property violated
      else if s' not flagged and SCC(neverstate(s)) = SCC(neverstate(s')) then
        Improved-Detect-Cycle(s')
```

Fig. 5. Improved Nested Depth-First Search.

Never Claim. In contrast to the heuristic directed search the improved nested depth-first search algorithm accelerates the search for full validation. The ease of implementation suggests to Improved-Nested-DFS to the SPIN validation tool.

4.3 A* and Improved-Nested-DFS

So far we have not considered heuristic search for Improved-Nested-DFS. Once more, we consider the example of *Response* properties to be validated. In a first phase, states are explored by A*. The evaluation function to focus the search can easily be designed to reach the accepting cycles in the SCCs faster, since all states that we are aiming at are accepting. This approach generalizes to a hybrid algorithm A* and Improved-Nested-DFS, *A*+DFS* for short, that alternates between heuristic search in N-SCCs, single-pass searches in F-SCCs, and Nested-Search in P-SCCs. If a P- or S-component is encounterd, Improved Nested-DFS is invoked and searches for cycles. The heuristic estimate respects the combination of all F-SCCs and P-SCCs, since accepting cycles are present in either of the two components. The nodes at the horizon of a F- and P-component lead to pruning of the sub-searches and are inserted back into the *Open*-List (priority queue) of A*, which contains all horizon nodes with a neverstate in the corresponding N-SCCs. Therefore *A* + Improved-Nested-DFS* continues with directed search, if cycle detection in the F- and P-component components fails. As in the naive approach, cycle detection search itself might be accelerated with an evaluation function heading back to the states where it was started.

Figure 6 visualizes this strategy for our simple example. The Never Claim corresponds to a response property. It has the following SCCs: SCC_0 which is a N-SCC, and SCC_a which is F-SCC. The state space can be seen as divided in two partitions, each one composed of states where the Never Claim is a state belonging to one of the SCCs. In a first phase, A* is used for directing the search

to states of the partition corresponding to SCC_a. Once a goal state is found, the second phase begins, where the search for accepting cycles is performed by Improved-Nested-DFS.

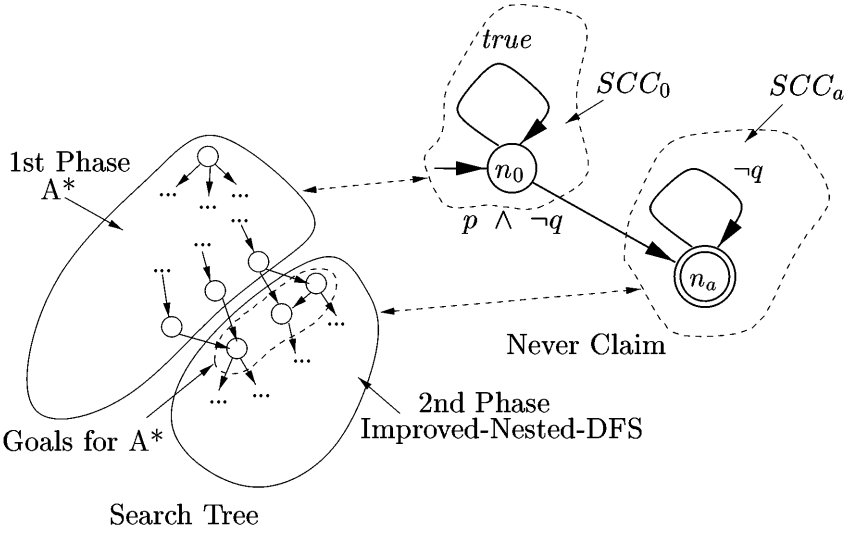


Fig. 6. Visualization of A* and Improved-Nested-DFS for a response property.

5 Heuristics for Errors in Protocols

In this section we introduce search heuristics to be used in the detection of errors in models written in Promela. We start off with precompiling techniques that help to efficiently compute different heuristic estimates.

5.1 Precompiling State Distance Tables

We now discuss how to calculate heuristic estimates through a precompilation step. We assume that a transition system $T = (T_1, \ldots, T_k)$ is given with T_i being the set of transitions within the process P_i. We use S to denote global system states. In S we have a set P of currently active processes $P_1, \ldots, P_k$. We write pc_i to denote the current control state for process P_i. The information we infer is the *Local State Distance Table* D that is defined for each process type. The value $D_i(u, v)$ fixes the minimal number of transitions necessary to reach the local state $u \in S_i$ starting from the local state $v \in S_i$ in the finite state machine

Table 3. The formula-based heuristics: a denotes a Boolean variable and g and h are logical predicates, t is a transition, q a queue. The symbol $\otimes$ represents relational operators $(=, \neq, \leq, <, \geq, >)$.

f	$H_f(S)$
true	0
false	∞
a	if a then 0 else 1
$\neg g$	$\overline{H}_g(S)$
$g \vee h$	$\min\{H_g(S), H_h(S)\}$
$g \wedge h$	$H_g(S) + H_h(S)$
$full(q)$	$capacity(q) - length(q)$
$empty(q)$	$length(q)$
$q?[t]$	minimal prefix of q without t (+1 if q contains no message tagged with t)
$a \otimes b$	if $a \otimes b$ then 0, else 1
$i@s$	$D_i(pc_i, s)$

representation for P_i. The matrix D_i is determined cubic time [4] with respect to the size of the number of states in the finite state representation of P_i.

5.2 The Formula-Based Heuristics

The formula-based heuristics assumes a logical description f of the failure to be searched. Given f and starting from S, $H_f(S)$ is the estimation of the number of transitions necessary until a state S' is reached where $f(S')$ holds. Similarly, $\overline{H}_f(S)$ is the minimum number of transitions that have to be taken until f is violated. Table 3 depicts the distance measure $H_f(S)$ of the failure formula that we used. The estimator $\overline{H}_f(S)$ is defined analogously.

We allow formulae to contain other terms such as relational operators and Boolean functions over queues, since they often appear in failure specifications of safety properties: The function $q?[t]$ is read as *message at head of queue q tagged with t*. Another statement is the $i@s$ predicate which denotes that a process with a process id i of a given proctype is in its local control state s.

In the definition of $H_{g \wedge h}$ and $\overline{H}_{g \vee h}$, we can replace *plus* $(+)$ with *max* if we want a lower bound. In some cases the proposed definition is not optimistic, e.g., when repeated terms appear in g and h. The estimate can be improved based on a refined analysis of the domain. For example suppose that variables are only decremented or incremented, then $H_{a=b}$ can be fixed as $a - b$.

Heuristics for Safety Properties

Invariants. System invariants are state predicates that are required to hold over every reachable system state S. To obtain a heuristics it is necessary to estimate the number of system transitions until a state is reached where the invariant does not hold. Therefore, the formula for the heuristics is derived from invariant.

Assertions. Promela allows to specify logical assertions. Given that an assertion a labels a transition (u, v), with $u, v \in S_i$, then we say a is violated if the formula $f = (i@u) \wedge \neg a$ is satisfied. According to f the estimate H_f for *assertion violation* can now be derived.

Deadlocks. S is a deadlock state if there is no transition starting from S and at least one of the processes of the system is not in a *valid endstate*, i.e., no process has a statement that is executable. In Promela, there are statements that are always executable: assignments, `else` statements, `run` statements (used to start processes), etc. For other statements such as `send` or `receive` operations or statements that involve the evaluation of a guard, executability depends on the current state of the system. For example, a send operation `q!m` is only executable if the queue `q` is not full. A naive approach to the derivation of an estimator function is is to count the number of active (or non-blocked) processes in the current state S. We call this estimator H_{ap}. It turns out that best-first search using this estimator is quite effective in practice. For the formula based heuristics H_f we can devise conditions for executability for a significant portion of Promela statements:

1. Untagged receive operation (`q?x`, with `x` variable) are not executable if the queue is empty. The corresponding formula is $\neg$ *empty*(q).
2. Tagged receive operations (`q?t`, with `t` tag) are not executable if the head of the queue is a message tagged with a different tag than `t` yielding the formula $\neg\ q?[t]$.
3. Send operations (`q!m`) are not executable if `q` is full indicated by the predicate $\neg full(q)$.
4. Conditions (boolean expressions) are not executable if the value of the condition is false corresponding to the term c.

We now turn to the problem of estimating the number of transitions necessary to reach a deadlock state. The deadlock in state S' can be formalized as the conjunct

$$\mathit{deadlock} \equiv \bigwedge_{P_i \in P} \mathit{blocked}(i, pc_i(S'), S')$$

where the predicate $\mathit{blocked}(i, pc_i(S'), S')$ is defined as

$$\mathit{blocked}(i, u, S) \equiv (i@u) \wedge \bigwedge_{t=(u,v) \in T_i} \neg\ \mathit{executable}(t, S).$$

Unfortunately, we do not know the set of states in which the system deadlocks such that we cannot compute the formula at exploration time. A possible solution to this problem is to approximate the deadlock formula. First we determine in which states a process can block and call such states *dangerous*. Therefore, we consider a process P_i to be blocked if $\mathit{blocked}(i, u, S)$ is valid for some $u \in C_i$, with C_i being the set of dangerous states of P_i. We define $\mathit{blocked}(i, S)$ as a predicate for process P_i to be blocked in system state S,i.e., $\mathit{blocked}(i, S) = \bigvee_{u \in C_i} \mathit{blocked}(i, S, u)$ and approximate the deadlock formula with $\mathit{deadlock'} = \bigwedge_{P_i \in P} \mathit{blocked}(i, S)$.

Heuristics for the Violation of Liveness Properties. For the validation of LTL specifications we need a heuristics for accelerating the search into the direction of the accepting state in the Never Claim. This can be achieved by declaring all accepting states as dangerous and use the local distance table to derive an estimate. An alternative is to collect all incoming transition labels for the accepting states and build a formula-based heuristics on the disjunction of that labeling. For the example of the response property we devise the heuristics $H_{p \wedge \neg q}$.

During the second phase of the nested depth-first search we need cycle-detection search algorithms. Since we know which accepting state to search for we can refine $H_f(S)$ for the given state S as

$$f = \bigwedge_{P_i \in P} i@pc_i(S)$$

Designer Devised Heuristics. The designer of the protocol can support the search for failures by devising a more accurate heuristics than the automatically inferred one. In HSF-SPIN, there are several options. First of all, the designer can alter the recursive tabularized definition of the heuristics estimate to improve the inference mechanism. Another possibility is to concretize deadlock occurences in the Promela code. Without designer intervention, all reads, sends and conditions are considered dangerous. Additionally, the designer can explicitly define which states of the processes are dangerous by including Promela labels with prefix `danger` into the protocol specification.

6 The Model Checker HSF-SPIN

We chose SPIN as a basis for HSF-SPIN. It inherits most of the efficiency and functionality of Holzmann's original source of SPIN as well as the sophisticated search capabilities of the Heuristic Search Framework (HSF) [9]. HSF-SPIN uses Promela as its modeling language. We refined the state description of SPIN to incorporate solution length information, transition labels and predecessors for solution extraction. We newly implemented universal hashing, and provided an interface consisting of a node expansion function, initial and goal specification. In order to direct the search, we realized different heuristic estimates. HSF-SPIN also writes trail information to be visualized in the XSPIN interface. As when working with SPIN, the validation of a model with HSF-SPIN is done in two phases: first the generation of an analyzer of the model, and second the validation run. The protocol analyzer is generated with the program `hsf-spin` which is basically a modification of the SPIN analyzer generator. By executing `hsf-spin -a <model>` several `c++` files are generated. These files are part of the source of the model checker for the given model. They have to be compiled and linked with the rest of the implementation, incorporating, for example, data structures, search algorithms, heuristic estimates, statistics and solution generation. HSF-SPIN also supports partial search by implementing *sequential bit-state hashing* [14]. Especially for the IDA* algorithm, bit-state hashing supports the search

for various beams in the search trees. Although the hash function does not disambiguate all synonyms and the length of a witness is often minimal [10].

The result is an model checker that can be invoked with different parameters: kind of error to be detected, property to be validated, algorithm to be applied, heuristic function to be used, weightening of the heuristic estimator. HSF-SPIN allows textual simulation to interactively traverse the state space which greatly facilitates in explaining witnesses that have been found.

7 Experimental Results

All experimental results were produced on a SUN workstation, UltraSPARC-II CPU with 248 Mhz. If nothing else is stated, the parameters while experimenting with SPIN (3.3.10) and HSF-SPIN are a depth bound of 10,000 and a memory limit of 512 MB. Supertrace is not used, but partial order reduction is used in SPIN. We list our experimental results in terms of expanded states and witness path length, i.e., the length of the counterexample. SPIN does not give the number of expanded states. We calculate it as the number of stored states plus one; in SPIN all stored states except the error state are expanded due to the depth first search traversal. Note that we apply SPIN with partial order reduction, while HSF-SPIN does not yet include this feature.

7.1 Experiments on Detecting Deadlocks

This section is dedicated to experiments with protocols that contain deadlocks. Table 4 depicts experimental results with these protocols. For parametrized protocols, we have used the largest configuration that a breadth-first search (BFS) can solve. We experimented with two heuristics for deadlock detection: H_{ap} and $H_f + U$: H_{ap} is the weak heuristics, counting the number of active processes; and $H_f + U$ is the formula based heuristics, where the deadlock formula is inferred from the user designated dangerous states. In A*, $H_f + U$ seem to perform better than H_{ap}. On the other hand, with best-first search the results achieved for both heuristics are similar. Therefore, we give the results with H_{ap} for BF only.

BFS and A* find optimal solutions, while BF finds optimal or near to optimal solutions in most cases. To the contratry, the depth-first search (DFS) traversal in HSF-SPIN and in SPIN generally provide solutions far from the optimum. The most significant cases are the Dining Philosophers and the Snoopy protocol. SPIN finds counterexamples of length larger than 1,000, while the optimal solution is about 30 times smaller. In some cases, A* expands almost as many nodes as BFS, which indicates a less-informed heuristic estimate. This weakness is compensated in best-first searches, in which the number of expanded nodes is smaller than in other search strategies for most cases.

In [10] we analyzed the scalability of the search strategies. Evidently, BFS does not scale. A* and DFS also tend to struggle when the protocols are parametrized with higher values. However, best-first search seems to be very stable: in most cases it scales linearly with the parameter tuned, offering near-to optimal solutions. Table 5 depicts some experimental results with the deadlock

solution to the dining philosophers problem. These results show that directed search can find errors in protocols, where undirected search techniques are not able to find them. In the presented case SPIN fails to find a deadlock for large configurations of the philosophers problem.

Table 4. Detection of Deadlocks in Various Protocols.

	HSF-SPIN					SPIN
GARP	BFS	DFS	A*,H_{ap}	A*,$H_f + U$	Best-First,H_{ap}	DFS
Expanded States	834	62	1,145	53	33	56
Generated States	2,799	70	3,417	194	60	64
Witness Length	16	50	16	18	28	58
Philosophers ($p = 8$)						
Expanded States	1,801	1,365	41	69	249	1,365
Generated States	10,336	1,797	97	69	646	1,797
Witness Length	34	1,362	34	34	66	1,362
Snoopy						
Expanded States	37,191	5,823	32,341	6,872	152	1,243
Generated States	131,475	7,406	110,156	24,766	299	1,646
Witness Length	40	4,676	40	40	40	1,113
Telegraph ($p = 6$)						
Expanded States	75,759	44	38	366	38	44
Generated States	445,434	45	108	1,897	108	45
Witness Length	38	44	38	38	38	44
Marriers ($p = 4$)						
Expanded States	403,311	294,549	333,529	284,856	6,281	36,340
Generated States	1,429,380	1,088,364	1,176,336	996,603	16,595	47,221
Witness Length	62	112	62	62	112	112
GIOP ($u = 1, s = 2$)						
Expanded States	49,679	247	38,834	27,753	315	338
Generated States	168,833	357	126,789	89,491	504	377
Witness Length	61	136	61	61	83	136
Basic Call ($p = 2$)						
Expanded States	80,137	115	4,170	36	57	117
Generated States	199,117	136	8,785	60	89	140
Witness Length	30	96	30	30	42	96

7.2 Experiments on Detecting Violation of System Invariants

This Section is dedicated to experiments of models with system invariants. In the following table we summarize the models and the invariant that they violate. Note that we simplified the denotation of invariant for better understanding.

Model	Invariant
Elevator	$\Box(\neg \mathit{opened} \vee \mathit{stopped})$
POTS	$\neg\Diamond(P_1@s_1 \wedge P_2@s_2 \wedge P_3@s_3 \wedge P_4@s_4)$

The search for the violation is performed with $H_{\neg i}$ as heuristic estimate, where i is the system invariant. Table 6 depicts the results of experiments with two models: an Elevator model, and the model of a Public Old Telephon System (POTS). The latter is not scalable, and the former has been configurated with 3 floors. For the Elevator model, the meaning of the invariant is self explaining. For the POTS model, the invariant describes the fact that not all processes are in a conversation state. As explained in [19], we use this invariant to test whether a given POTS model is capable of establishing a phone conversation at all.

Table 5. Number of expanded states and solution lengths achieved by A* in the dining philosophers protocol (p=number of philosophers).

		HSF-SPIN					SPIN
p		BFS	DFS	A*,H_{ap}	A*,$H_f + U$	Best-First,H_{ap}	DFS
2	Expanded States	10	12	10	10	10	12
	Generated States	12	14	12	12	12	14
	Witness Length	10	10	10	10	10	10
3	Expanded States	18	19	16	14	32	19
	Generated States	30	22	22	19	52	22
	Witness Length	14	14	14	14	14	18
4	Expanded States	33	57	21	21	69	57
	Generated States	77	75	33	27	155	75
	Witness Length	18	54	18	18	26	54
8	Expanded States	1,801	1,365	41	69	249	1,365
	Generated States	10,336	1,797	97	69	646	1,797
	Witness Length	34	1,362	34	34	66	1,362
12	Expanded States	-	-	61	50	539	278,097
	Generated States	-	-	193	127	1,468	46,435
	Witness Length	-	-	50	50	98	9,998
16	Expanded States	-	-	81	66	941	-
	Generated States	-	-	321	201	2,626	-
	Witness Length	-	-	66	66	130	-

Table 6. Detection of Invariant Violations.

	HSF-SPIN				SPIN
Elevator	BFS	DFS	A*	Best-First	DFS
Expanded States	228,479	310	227,868	16,955	305
Generated States	1,046,983	388	1,045,061	53,871	363
Witness Length	205	521	205	493	521
POTS					
Expanded States	49,143	1,465,103	409	68	2,012,345
Generated States	154,874	4,460,586	1,287	185	2,962,232
Witness Length	66	1,055	66	66	872

As the Elevator model violates a very simple invariant, the results show that A* performs like breadth-first search; an optimal solution is found, but the number of expanded nodes are almost the same. SPIN and our depth-first search algorithm (DFS) yield about same results. The number of expanded nodes is small compared to breadth-first search and best-first search expands more nodes than DFS for a better solution quality. However, best-first search does not approximate the solution quality. The cause of these unexpected *bad* performances of the heuristic search algorithms is the restricted range of the heuristic estimate: the integer range [0..2]. The quality of the estimate and the efficiency of the heuristic search procedures for system invariants correlates with the amount of information that can be extracted from the invariant.

The POTS protocol violates a more complicated invariant. The formula f used for the heuristic estimate H_f is the negation of the invariant. The function f is a conjunction of four statements about the local state of four different pro-

cesses. The heuristic estimate exploits the information of the transition graph corresponding to each process. While SPIN has serious problems to find the violation of the invariant, A*'s performance is superior. It finds an optimal solution with a relatively small number of expanded nodes. Best-First search achieves even better results, since it still finds optimal solutions expanding less nodes.

7.3 Experiments on Detecting Assertion Violations

We have a small group of models containing errors such as violation of assertions summarize as follows.

Model	Assertion
Lynch's Protocol	$i = last_i + 1$
Barlett	$mr = (lmr + 1)\%\max$
Mutex	$in = 1$
Relay	$(k_{14_1} = (s_{1_1} \wedge \neg k_{12_1})) \wedge$ $(k_{12_1} = (dienstv \wedge (\neg s_{1_1} \vee k_{12_1}))) \wedge$ $(k_{14_2} = (s_{1_2} \wedge \neg k_{12_2})) \wedge$ $(k\|12_2 = (dienstv \wedge (\neg s_{1_2} \vee k_{12_2}))) \wedge$ $(dienstv = (k\|14_1 \vee k_{14_2}))) <= \neg(k_{14_1} \wedge k_{14_2})$
GARP	$false$

Table 7 depicts experimental results with these protocols. The results show that directed search strategies in HSF-SPIN offer shorter counterexamples for assertion violations than SPIN. For the GARP Protocol the number of expanded states is considerably high, since the heuristic according to the assertion *false* is very weak. In all other cases, the number of expansions for heuristic search is by far smaller smaller than the corresponding number of expanded states in SPIN or exceeds it by at most three times.

7.4 Experiments on Detecting Violation of LTL Properties

In the following table we summarize test cases for the detection of LTL property violations. Note that the error in the GIOP protocol has been seeded by explicit source code annotation.

Model	LTL formula
Alternating Bit	$\Box(p \rightarrow ((\Diamond q) \vee (\Diamond q))$
Elevator	$\Box(p \rightarrow \Diamond(q \wedge r))$
GIOP	$\Box(p \rightarrow \Diamond(q \wedge r))$

The LTL properties of the Elevator and GIOP protocols correspond to the Response (Globally) pattern, the structure of the property in the alternating bit is similar such that the *A*+DFS* algorithm for response properties can be used.

Table 8 shows experimental results on detecting the violation of LTL formulae. We used a variant of the elevator model that includes a controller satisfying the previously discussed invariant but violates a response property. This protocol has been configurated with 4 floors, while the GIOP protocol is configured with 1 server and 3 clients. Comparing the results of the new proposed *Improved-Nested-DFS* with those of the classical Nested-DFS, the new algorithm finds

Table 7. Detection of Assertion Violations in Various Protocols.

	HSF-SPIN				SPIN
Lynch	BFS	DFS	A*	Best-First	DFS
Expanded States	79	50	72	63	47
Generated States	96	52	89	79	50
Witness Length	29	46	29	29	46
Barlett					
Expanded States	82	348	61	26	262
Generated States	99	383	76	33	289
Witness Length	20	246	20	20	251
Mutex					
Expanded States	349	202	150	24	202
Generated States	699	363	300	48	363
Witness Length	15	54	15	15	54
Relay					
Expanded States	707	342	665	151	341
Generated States	2,701	719	2,292	1,069	870
Witness Length	12	190	12	120	190
GARP					
Expanded States	17,798	1,040	18,968	4,727	150
Generated States	53,001	2,818	56,406	13,107	187
Witness Length	29	54	29	39	55

shorter solutions expanding a few states less. On the other side, the ad-hoc algorithm for response properties (A*+DFS) finds the shortest solution in all cases. In the Elevator protocol it expands about 1,000 times more states than the other algorithms, and in the GIOP example it expands about 1,000 times less states. In the elevator case we trace the anomaly back to the heuristic estimate which gave a poor range of values: [0..1]. Heuristic estimates can only improve a search strategy if they have very specific knowledge of the system. A small ranged heurisitc function cannot achieve this. In the GIOP case the range of values was somewhat larger ([0..6]), and obviously this improves the effectiveness of the heuristic search. This observation calls for further refinements of the heuristic functions.

We also performed full validation experiments with a version of the elevator protocol that satisfies the response property and observed that Improved-Nested-DFS executes less transitions (716,715) than classical Nested-DFS (979,336).

7.5 Performance of HSF-SPIN

HSF-SPIN is still a prototype. Therefore, its performance in terms of time and space cannot compete with SPIN. For example, an exhaustive exploration of the state space generated by the GIOP protocol parametrized with 2 clients and 2 servers is performed by SPIN (without partial order reduction) in 226 seconds with a memory consumption of 236 MB, while our tool requires 341 seconds and about 441 MB of space. Further experiments show that SPIN achieves a speedup of about 3 in comparison with HSF-SPIN.

Table 8. Detection of Violation of Liveness Properties in Various Protocols.

	HSF-SPIN			SPIN
Alternating Bit	Nested-DFS	Improved-Nested-DFS	A*+DFS	DFS
Expanded States	33	32	11	24
Generated States	37	36	12	32
Witness Length	64	64	22	46
Elevator				
Expanded States	309	251	217,810	253
Generated States	381	288	1,276,391	401
Witness Length	405	391	377	405
GIOP				
Expanded States	404,799	404,619	113	53,812
Generated States	1,957,563	1,957,390	1,158	107,987
Witness Length	430	158	158	430

8 Conclusion

In this paper we commenced by arguing that there is a need for improving the efficiency of model checking. It is desirable to obtain shorter error witnesses in order to more easily understand errors that the model checker reports. A reduction in the number of visited states during state space search is also desirable since this renders larger models executable. While in previous work the improvements were limited to safety properties, we now present an approach to improving the validation of a large class of non-safety properties. We view this as a step of developing HSF-SPIN into a full-fledged model checker.

The work centers around an algorithm for LTL property checking that is an improvement to nested depth first search. The algorithm exploits the structure of the Never Claim and heuristic estimates in order to find cycles faster. We argued that based on the translation of LTL formulae to Büchi Automata implemented in SPIN we can improve LTL property checking for a large class of specification patterns used in practice. Next we presented heuristics to be used in search algorithms for different classes of properties. We then presented HSF-SPIN, and illustrated its application to a number of protocol examples.

As future work we plan to analyze the proposed improvement of the nested depth-first search algorithm. We plan to perform further experiments to verify the reduction in the number of performed transitions by the new algorithm as well as refinements of the heuristic estimates. It has been shown that nested-depth first search and partial order reductions can coexist [15]. Therefore, we currently investigate how to reconcile partial order reduction and directed search.

Acknowledgements. The authors would like to thank the anonymous referees for helpful references to related work.

References

1. A. Biere. μcke - efficient μ-calculus model checking. In *Computer Aided Verification*, pages 468–471, 1997.
2. D. Brand and P. Zafiropulo. On communicating finite-state machines. *Journal of the ACM*, 30(2):323–342, Apr 1983.
3. E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 2000.
4. T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. The MIT Press, 1990.
5. A. D. E. Muller and P. Schnupp. Alternating automata. the weak monadic theory of the tree and its complexity. In *International Colloquim on Automata, Languages and Programming*.
6. R. Dial. Shortest path forest with topological ordering. *Communications of the ACM*, pages 632–633, 1969.
7. E. W. Dijkstra. A note on two problems in connection with graphs. *Numerische Mathematik*, 1:269–271, 1959.
8. M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Patterns in property specifications for finite-state verification. In *International Conference on Software Engineering*, 1999.
9. S. Edelkamp. *Data Structures and Learning Algorithms in State Space Search*. PhD thesis, University of Freiburg, 1999. Infix.
10. S. Edelkamp, A. L. Lafuente, and S. Leue. Protocol verification with heuristic search. In *AAAI Symposium on Model-based Validation of Intelligence*, 2001.
11. S. Edelkamp and F. Reffel. OBDDs in heuristic search. In *German Conference on Artificial Intelligence (KI)*, pages 81–92, 1998.
12. M. G. Gouda. Protocol verification made simple: a tutorial. *Computer Networks and ISDN Systems*, 25(9):969–980, 1993.
13. P. E. Hart, N. J. Nilsson, and B. Raphael. A formal basis for heuristic determination of minimum path cost. *IEEE Trans. on SSC*, 4:100, 1968.
14. G. Holzmann. An analysis of bitstate hashing. *Formal Methods in System Design*, 13(3):287–305, November 1998. extended and revised version of Proc. PSTV95, pp. 301-314.
15. G. Holzmann, D. Peled, and M. Yannakakis. On nested depth first search. In *The Spin Verification System*, pages 23–32. American Mathematical Society, 1996.
16. G. J. Holzmann. On limits and possibilities of automated protocol analysis. In *Protocol Specification, Testing, and Verification*, 1987.
17. G. J. Holzmann. *Design and Validation of Computer Protocols*. Prentice Hall, 1990.
18. M. Kamel and S. Leue. Formalization and validation of the general inter-orb protocol (GIOP) using Promela and SPIN. In *Software Tools for Technology Transfer*, volume 2, pages 394–409, 2000.
19. M. Kamel and S. Leue. Vip: A visual editor and compiler for v-promela. In *6th International Conference, TACAS 2000*, volume 1785 of *Lecture Notes in Computer Science*, pages 471–486. Springer, 2000.
20. F. J. Lin, P. M. Chu, and M. Liu. Protocol verification using reachability analysis: the state space explosion problem and relief strategies. *ACM*, pages 126–135, 1988.
21. K. McMillan. *Symbolic Model Checking*. Kluwer Academic Press, 1993.
22. D. McVitie and L. Wilson. The stable marriage problem. *Communications of the ACM*, 1971.

23. A. Miller and M. Calder. Analysing a basic call protocol using promela/xspin. In *International SPIN Workshop*, 1998.
24. T. Nakatani. Verification of group address registration protocol using promela and spin. In *International SPIN Workshop*, 1997.
25. K. R. Bloem and F.Somenzi. Symbolic guided search for ctl model checking. In *Conference on Design Automation (DAC-00)*.
26. F. Reffel and S. Edelkamp. Error detection with directed symbolic model checking. In *World Congress on Formal Methods*, pages 195–211. Springer, 1999.
27. F. Somenzi and R. Bloem. Efficient buchi automata from ltl formulae. In *Computer Aided Verification*.
28. P. van Eijk. Verifying relay circuits using state machines. In *International SPIN Workshop*.
29. W. Visser and H. Barringer. Ctl* model checking for spin. *Software Tools for Technology Transfer*, 2000.
30. C. H. Yang and D. L. Dill. Validation with guided search of the state space. In *DAC*, pages 599–604, 1998.

Addressing Dynamic Issues of Program Model Checking

Flavio Lerda and Willem Visser

RIACS/NASA M/S 269-2
Ames Research Center
Moffett Field, CA 94035-1000
USA
{flerda, wvisser}@riacs.edu

Abstract. Model checking real programs has recently become an active research area. Programs however exhibit two characteristics that make model checking difficult: the complexity of their state and the dynamic nature of many programs. Here we address both these issues within the context of the Java PathFinder (JPF) model checker. Firstly, we will show how the state of a Java program can be encoded efficiently and how this encoding can be exploited to improve model checking. Next we show how to use symmetry reductions to alleviate some of the problems introduced by the dynamic nature of Java programs. Lastly, we show how distributed model checking of a dynamic program can be achieved, and furthermore, how dynamic partitions of the state space can improve model checking. We support all our findings with results from applying these techniques within the JPF model checker.

1 Introduction

Software is playing an increasingly important role in our everyday lives, but sadly, so does software failure. At NASA this point was made painfully clear in 1999 when the Mars Polar Lander was lost due to a software related problem (estimated cost was $165 million) [Spa00]. Although most agree that many software failures can, and must, be caught during the design phase, it is however often the case that a design phase is either missing by itself, or the tools and techniques to analyze the designs are missing. Hence, testing an implementation is still the number one way of finding errors in software systems. Testing, however, can be very expensive, but more importantly, it is often incapable of finding subtle errors - e.g. timing errors in a concurrent system.

Model checking has been used extensively to find subtle errors in hardware and protocol designs [BLPV95,CW96,Hol91]. However, until recently, model checking has been deemed inadequate to analyze software code, due to the high level of detail often found in code. Now there are many groups, from both industry and academia, that are analyzing source code by model checking. Many of these source code model checkers are based on a translation from source code

M.B. Dwyer (Ed.): SPIN 2001, LNCS 2057, pp. 80–102, 2001.

to the input notation of a model checker: Bandera [CDH+00], Java PathFinder 1 [HP98], JCAT [DIS99] are Java model checkers, and, AX [Hol00] and SLAM [BR00] are C model checkers. A drawback of the translation approach is that certain language constructs are difficult to translate and hence two of these tools, JCAT (dSPIN [IS99]) and AX have extended their back-end model checker (SPIN in both cases [Hol97a]) to improve efficiency.

We adopted a different approach by creating a custom-made model checker for Java. We call this tool Java PathFinder 2, henceforth referred to as JPF. JPF is an explicit state model checker that takes as input Java bytecode. It is structured as a search algorithm that uses a special Java virtual machine (JVM^{JPF}) to execute the bytecode instructions one at a time. In order to implement a depth first algorithm the JVM^{JPF}needs also to have a backtracking capability. The tool itself is written is Java and it's executed by the Java virtual machine (just JVM from now on). By executing the bytecode we can not only analyze all of Java, but we can also analyze programs without source code (e.g. libraries and code down-loaded over the web), and other languages for which bytecode translations exist [BKR98,Taf96,CD98]. Recently JPF has been integrated with the Bandera system [CDH+00]: in this case Bandera doesn't need to do any translation because our tool is able to handle Java directly, but Bandera's functionality of slicing and abstraction are available to improve the model checking.

If one looks at the history of model checking input notations, then it is clear that there has been an evolution from simple guarded command style notations, to ones where more complex data-structures are used. We believe this trend will continue and soon complex dynamic data-structures as well as other features from typical programming languages will be common place. The purpose of this paper will be to highlight some of the difficulties and possible solutions we have encountered in developing an efficient model checker that can handle dynamically evolving software systems. We hope this will help others when developing similar systems.

Although it is clear that static analysis of a system before model checking can greatly benefit the verification, e.g. slicing a system with respect to a certain property to be checked, or finding independent statements to allow partial-order reductions, here we will focus mostly on purely dynamic optimizations for which no prior information is required. The interested reader is referred to [VHBP00] where we discuss static analysis for partial-order reductions and other techniques, such as abstraction, that JPF employs before doing model checking.

Model checking software is often considered hard due to the complexity of the state of the system (this is the premise of state-less model checking [God97, Sto00]). We address this problem in section 2, by first showing how a "large" state can be collapsed to a smaller one, how this can be exploited to improve explicit-state model checking, how a novel form of symmetry reductions on the state can reduce the size of the state space, and lastly, how garbage collection improves model checking. The state-space explosion problem can be reduced, but it almost never goes away. Hence, the more memory one has the larger the programs that can be checked. In section 3 we extend the distributed model

checking algorithm first used for SPIN [LS99], that exploits the memory of a number of workstations, to work in the dynamic context of Java. Section 4 contains conclusions and directions for future work.

2 Complexity of the State

One of the first issues we had to address was the complexity of the state. A limitation of the current model checking tools is that they cannot handle dynamic structures. In fact dSPIN [IS99], an extension of the model checking SPIN [Hol97a] used as a back-end in the Java model checker tool JCAT [DIS99], introduces direct support for dynamic allocation.

2.1 The Representation of the State

In creating our own model checker, we were free to choose the representation of the state. Our aim was to be able to handle dynamic allocation efficiently and maintain our representation as close as possible to the one suggested by the programming language. The state is composed of three main components:

static area: is an array of entries, one for each class loaded. Each entry contains the values of the static fields of the class and the monitor associated with it. The monitor contains information on the lock for the class: which thread is holding the lock, which threads are waiting for the lock etc. When a new class needs to be loaded a new entry in the static area is created and its fields and monitor are initialized. Once loaded a class will never be unloaded during the execution of an instruction – but it can be unloaded by a backtracking step.

dynamic area: is an array of entries, one for each object. Each entry contains the values of the fields and the monitor[1] associated with it. Objects are created explicitly by specific bytecode instructions. When an object is created an entry is added in the dynamic area and its fields and monitor are initialized. Objects are not destroyed explicitly in Java, but they can be removed if not referenced anymore (see Section 2.5).

thread list: is a list containing the information relative to each thread. It contains the status of the thread together with other information used by the scheduler, and the stack frames created by the method calls. A new entry is created when a new thread is created, and modified each time the execution of a bytecode instruction changes the state of the thread or one of its stack frames.

These three components are dynamic and they can grow and shrink freely during the execution of the program, not imposing any limit on the size of the state. This is a novel feature, since in both SPIN, where process are allocated dynamically, and dSPIN, where also data can be allocated dynamically, there is still a limit imposed on the size of each state.

[1] The fields and monitor structures are the same used in the static area.

2.2 Collapsing the State

The dynamic features of the Java language, namely, class loading, object creation and method invocation, require a complex data-structure to record the state of the system (see for example our state structure in the previous section). Furthermore, in order to do efficient explicit-state model checking one needs to record the states that have been visited (often using a hash-table). From the examples[2] in Table 1 one can clearly see that it is very inefficient to store the states in their original complex form: for both relatively simple Java programs more than 2kB/state are required. Unlike in a tool such as SPIN where state compression is an option, it is clear that for systems that require a more complex state description, compression should be a requirement.

The "collapse" algorithm has been very successful for state compression in SPIN [Hol97b] and hence we decided to extend it for use within JPF. The rationale behind the collapse method is that when a new state is generated large parts of the state are unchanged. This would seem to call for the state to be stored as the difference from the predecessor, but since states need to be compared to determine if a state has been visited before, this would be inefficient. What the collapse does is to associate to a particular part of the state an index. The state can then be collapsed to a list of indexes indicating which components compose the state itself. The decomposition must be unique so that by comparing the indexes it is possible to determine state equality.

In order to generate the indexes we created a set of pools. Each pool is an ordered set without repetitions. Every time a state needs to be stored it is first collapsed: each component is inserted into a pool, the pool returns the index that corresponds to the position of that component in the pool. If the element was not present in the pool it is added at the end, otherwise the index of the copy already present in the pool is returned. The assumption is that the size of the pool is small enough because each single component appears the same in many states.

In SPIN there are pools for the following state components: global variables, processes and asynchronous channels. Asynchronous channels has no counterpart in Java, but one can think of global variables and static fields, and SPIN processes and JAVA threads to be similar. This would seem to imply that a good first try for our state compression should include a pool for the static area, the dynamic area and the threads. This, however, would be inefficient, since each of these three components has further structure that can be exploited. For example, an assignment to a field of an object would make the dynamic area and one thread change, and hence create two large new pool entries. If we rather use a pool for each stack frame in each thread, one for each monitor and one for each fields data entry then the above field assignment would only change one stack frame entry (the one for the method with the assignment, while leaving all other frames in the thread to collapse to their old values) and one fields entry. When deciding on which components to compress one should always pick components that would

[2] The examples used in this and the following tables are available from the JPF webpage http://ase.arc.nasa.gov/jpf

not change too often, in order to get maximum benefit. We therefore choose a pool for each of the following: fields data (from both the static and dynamic area), monitor data (again shared between static and dynamic area), method stack frames and lastly one for other thread information (such as the thread status, that seldom changes).

As can be seen in Table 1, when the compression algorithm is used the number of different elements in the pools is quite small and the reduction of the memory requirements is impressive. Note that the execution time is reduced as well. When a new state is stored it is compared to other states to see if it has already been visited. This operation is highly inefficient when two uncompressed states are compared because of the complexity of the states themselves, but it is quite efficient when the compressed states (namely arrays of integers) are compared.

Table 1. Comparison of JPF using no compression, collapse, and optimized backtrack

	States	**Transitions**	**Pools Entries**
RemoteAgent	66,425	148,825	1,373
	Memory *(MB)*	**Time** *(sec)*	**State Size** *(bytes)*
No Compression	180.79	227.83	2854
Collapse	12.08	138.65	191
Optimized Backtrack	12.08	54.39	191

	States	**Transitions**	**Pools Entries**
BoundedBuffer	105682	275988	583
	Memory *(MB)*	**Time** *(sec)*	**State Size** *(bytes)*
No Compression	504.82	665.90	5009
Collapse	28.17	297.40	445
Optimized Backtrack	28.16	76.82	445

2.3 Optimizing the Backtrack

Although compression made JPF usable, it was clearly still too slow and used too much memory at run-time to handle large examples. Profiling the system revealed that the problem was the way we handled backtracking. Unlike SPIN we decided to store a copy of each state on the depth-first stack for backtracking purposes — SPIN uses a backwards transition to unwind moves when backtracking, and only if the state change is too large a copy of the previous state is used. The reason for this is that we work on the bytecode level and often one

Java statement[3] can correspond to many bytecode instructions, hence unwinding each bytecode instruction seemed too complicated. The graph in Figure 1 shows how the memory usage before optimizing the backtracking varies during the visit as new states are reached. In the same graph (with a different scale) the depth of the stack is shown. It is evident how the memory usage is strictly related to the depth of the stack because the uncompressed state is stored on the stack.

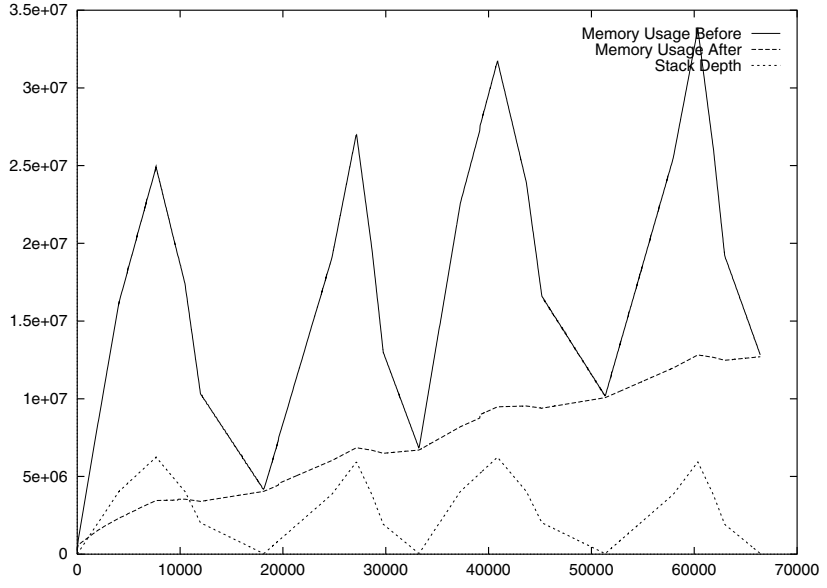


Fig. 1. Memory usage in bytes during execution with and without optimized backtracking. Stack depth is also represented in a different scale

A very simple, and above all novel solution presented itself: store the compressed state on the stack and use the reverse of the collapse operation to recreate the original state when backtracking. In fact, only store a reference to the compressed state on the stack and leave the state itself in the hash-table. Reconstructing the state is quite straight-forward because the collapsed information contains the indexes of the different components of the state that just need to be put back together again via a reverse lookup in the pools (i.e. the original component corresponding to an index must be retrieved). For efficiency only the components that are actually changed are restored. Figure 1 also shows the memory usage after introducing the optimization of the backtrack. A slight

[3] Although JPF executes bytecode instructions, we typically don't use that level of atomicity during model checking, rather we use one JAVA statement or one line of JAVA code as being one transition.

dependency between memory used and stack depth is still present, but now most of the memory is used to store the states. It is interesting to see that the memory usage before and after the optimization intersect where the stack depth comes down to zero – or very close to it.

Table 1 contains the results obtained using this more optimized backtracking technique. It is not evident how the memory usage is optimized (see Figure 1) because the same amount of memory is used when the search is finished. But the execution time is considerably reduced, because it is not necessary to put a copy of the state on the stack anymore.

2.4 Exploiting Symmetries

Symmetries have been used in model checking to reduce the size of the state space [EJ93,ID96,CEJS98,CFJ93,BDH00]. The basic idea is to visit a subset of the state space that is representative of the whole state space based on a symmetry relation that does not influence the properties being checked. Typically symmetry reductions exploit the structure of the system being analyzed, e.g. identical processes, scalar sets etc. [BDH00,ID96]. In keeping with the focus of the paper we exploit symmetries that will be inherent to dynamic systems, and hence we address symmetry issues on the underlying state representation rather than symmetry within the state itself.

```
class A {
  public static int value = 0;
}
```

```
class B {
  public static int value = 1;
}
```

Symmetry	States	Transitions
no	82	152
yes	70	132

```
class Main extends Thread {
  public static void
      main(String[] args) {
    new Main(true).start();
    new Main(false).start();
  }
  public int x;
  private boolean a;
  public Main(boolean a) {
    this.a = a;
  }
  public void run() {
    if(a) {
      x = A.value;
    } else {
      x = B.value;
} } }
```

Fig. 2. Static area symmetry reduction

Specifically we want to exploit the symmetry when classes are loaded into the static area and when objects are created in the dynamic area. Recall from section 2.1 that both the static and dynamic area are implemented as arrays, but the exact position of a class or object in these arrays should not be relevant when comparing states. Nondeterminism, either from concurrency or the environment, can cause classes to be loaded or objects to be created in different orders along different execution paths.

Comparing all possible permutations of the array entries when comparing states is however computationally very expensive, and hence we decided to rather use a canonization function to achieve efficient symmetry reductions. The idea is the following: whenever a state is generated we calculate a canonical representation of the state and use this representation for state comparisons. The use of a canonization function is a well-known technique for achieving symmetry reductions [BDH00,ID96], but calculating this function can be very expensive [CEJS98] in itself. However, since we are only interested in a limited form of symmetry reduction, we can calculate the canonical state representation very efficiently by imposing an order on the entries in the static and dynamic areas. Due to the dynamic nature of Java programs, this ordering must be calculated during model checking. Also, since we use the position of classes and objects as references, an ordering that would require positions to change would be inefficient. The idea is to dynamically map each class (object) to a position in the static (dynamic) array when the class (object) is *first* loaded (created). When backtracking, this mapping is preserved, and reused when executing down a different path.

Figure 2 shows a piece of code where two classes – A and B – are loaded when one of their static fields is accessed. Two threads are executing each one accessing a different class. Depending on the scheduling class A can be loaded before class B or vice-versa. Without symmetry reduction, if classes are allocated in the static area in the order they are loaded, the two interleavings above would lead to two different states, one where A occupies the position before B and the other where B is in front of A. With symmetry reduction we keep a mapping of class names to positions with respect to which class got loaded first, and hence we see a reduction in the number of states (see table in Figure 2). Note that for classes we could have chosen an alphabetical ordering, but this would violate the condition that positions must not change due to the ordering.

Since class names are unique this simple mapping of names to positions is sufficient to achieve a canonical representation for the static area. The dynamic area is not quite as straight-forward, since objects do not have names and hence the different objects created cannot be so easily identified. Our first guess was to use the bytecode instruction that created the object to identify it: this works fine (see results in Figure 3) as long as each instruction is executed at most once during each execution path. However allocation instructions can be executed more than once and create more than one object (e.g. an allocation within a loop). To deal with that we added to the identifier of an object the occurrences of the instruction, i.e. the number of times that instruction has been executed

```
class ThreadOne extends Thread {
  public Object o;
  public void run() {
    o = new Object();
} }
```

```
class ThreadTwo extends Thread {
  public Object o;
  public void run() {
    o = new Object();
} }
```

```
class Main {
  public static void
      main(String[] args) {
    new ThreadOne().start();
    new ThreadTwo().start();
} }
```

Symmetry	States	Transitions
no	54	89
yes	30	51

Fig. 3. Dynamic area symmetry reduction

before. Note that this occurrence number is incremented for an allocation, and also decremented whenever the allocation is backtracked.

```
class ThreadOne extends Thread {
  public Object o;
  public void run() {
    o = Main.newObject();
} }
```

```
class ThreadTwo extends Thread {
  public Object o;
  public void run() {
    o = Main.newObject();
} }
```

```
class Main {
  public static void
      main(String[] args) {
    new ThreadOne().start();
    new ThreadTwo().start();
  }
  public static Object newObject() {
    return new Object();
} }
```

Symmetry	States	Transitions
no	122	237
yes	86	168
thread	68	135

Fig. 4. One instruction can be called by different threads

This is sufficient to identify an object, but it might not lead to the most efficient symmetry reduction results. In Figure 4 the same bytecode instruction is executed by two different threads. The number of occurrences with which each thread executes the instruction depends on the interleaving and therefore so does the position of the object in the dynamic area. One way to address this problem is to add to the identifier of an object the thread that created the objects and count the number of occurrences of the same instruction in each thread separately (see last row in the table in Figure 4).

The same basic approach has been applied to the static and dynamic areas. For the dynamic area – since there is no unique identifier for an object other then the position in the dynamic area itself – it is necessary to use some other information (the bytecode instruction that created the object, the number of occurrences that instruction had before, the thread that created the object) in order to create a dynamic ordering. Although we showed results on small examples in this section, the difference symmetry reductions can make on more realistic examples is quite good: e.g. without symmetry reductions the Bounded-Buffer example from Table 1 has 2502761 states and only 105682 with symmetry reduction (i.e. a 25 fold reduction). Furthermore, the time overhead is almost non-existent: when applying symmetry reduction to an example that have no such reductions we see a 1% increase in the execution time.

The symmetry reduction presented in this paper is orthogonal to other kind of symmetries based on the structure of the system, and therefore they could be used together. Note, that unlike [ID96,BDH00] we don't require an extension of the model checker input language in order to achieve efficient symmetry reductions. Lastly, some of the symmetries that are exploited here are due to different interleavings of some thread execution. Partial order reduction techniques avoid exploring some of those interleavings, and therefore it can mitigate the effect of our symmetry reduction. However our method can reduce symmetries that cannot be avoided by partial order reductions, but more importantly, can be applied without requiring expensive static analysis to determine transition independence.

2.5 Garbage Collection

Java does not offer any primitive to deallocate an object. Once created an object will continue to exist until it is *garbage collected*. An object can be garbage collected when no more references to it are available.

If we want to model check software written in Java we need to take into account garbage collection. Many Java programs rely on its presence, and even very simple examples – see Figure 5 – would have an infinite number of states without garbage collection (each allocation increases the size of the state, hence causing an infinite state-space). Garbage collection during model checking was first introduced in [IS00] and we implemented the same two algorithms, namely reference counting and mark and sweep, in JPF. The results presented in Table 2 show how reference counting and mark and sweep perform in our implementation. Three examples are analyzed:

```
public class Main {
  public static void main(String[] args) {
    Object o;
    while(true) {
      o = new Object();
} } }
```

Fig. 5. Even a simple example can have infinite states

- **TempObj** creates many temporary objects;
- **NoGarbage** adds elements to a list but never removes them, creating no objects to be collected; and
- **DoubleLinked** creates a double linked list and then loses the only pointer to it.

All these examples include concurrency (two or more threads are executing the same operations concurrently). In the table the following have been reported: the number of states, the number of transitions, the memory usage (at the end of the verification), the execution time, and the number of objects collected (with the number of times the algorithm has been activated for mark and sweep).

Table 2. Results obtained with GC

TempObj	**States**	**Transitions**	**Memory** *(MB)*	**Time** *(sec)*	**GC** *objects(runs)*
no GC	609173	1276971	168.26	508.24	-(-)
Mark and Sweep	2923	7110	1.1	5.73	2750(4286)
Reference Count	2923	7110	1.1	6.48	2750(-)

NoGarbage	**States**	**Transitions**	**Memory** *(MB)*	**Time** *(sec)*	**GC** *objects(runs)*
no GC	833766	1812626	216.88	520.26	-(-)
Mark and Sweep	833766	1812626	216.95	646.43	0(1241203)
Reference Count	833766	1812626	216.95	621.90	0(-)

DoubleLinked	**States**	**Transitions**	**Memory** *(MB)*	**Time** *(sec)*	**GC** *objects(runs)*
no GC	124503	310006	71.31	99.18	-(-)
Mark and Sweep	124503	310006	33.45	101.98	35928(231163)
Reference Count	124503	310006	71.37	120.01	0(-)

The first example – TempObj – is heavily affected by garbage collection. The fact that temporary objects are created – which is quite common in Java – adds extra information to the state that makes equivalent states seem different

if garbage collection is not active. The memory requirement are the same for both algorithms, but mark and sweep is slightly faster, since the ratio of objects collected per run is quite high (this is not the case for the other two examples).

The second example – NoGarbage – shows the overhead introduced by the two algorithms, since this example does not produce any object to be collected. The extra memory is about the same for both algorithms, but that's not true for the execution time. The mark and sweet algorithm is activated many times and that is reflected in the higher execution time: almost 70% of the transitions cause the algorithm to be executed. The algorithm is activated only after an instruction that can produce garbage: unfortunately many of the bytecode instructions can.

The last example – DoubleLinked – leads to two considerations. First, the reference count algorithm is not able to detect cycles of garbage. As a consequence no garbage is detected by this algorithm. Secondly, even when garbage is found with the mark and sweep algorithm, the state space is not reduced: the reason is that states with their garbage removed are still different because of other variables in the program. However, there is a reduction in the memory since states from which garbage was collected are now smaller.

Note, unlike the case for dSPIN[IS00], where it was necessary to store the complete state on the stack before garbage collection – because the state was changed in an irreversible way – we do not need to do so because the collapsed version of the state is already stored on the stack.

3 Distributed Memory

Explicit state model checking suffers from the state explosion problem, and when analyzing software programs this problem is more severe due to the higher-level of detail present in such programs. In order to deal with this issue, many different solutions have been tried, distributed model checking being one of them [LS99, SD97]. In this section we present how we improved this technique and adapted it to the dynamic nature of the systems that can be checked with our tool. Our work is based on [LS99], that presents a distributed memory implementation of SPIN. Our goal was to analyze the issues of implementing a distributed model checker when the input model is a dynamic system, in order to guide us in the development of a parallel model checker.

The algorithm presented in [LS99] extends the standard depth-first visit algorithm into a distributed visit algorithm. This new algorithm is no longer depth first but it still visits all the states and paths of the system. The only real issue with this algorithm is that it does not allow LTL model-checking, but this limitation had been overcome in [BBS00].

The basic idea is to divide the state space in partitions. Each node – workstation – will store the states that belong to one of the partitions. Every time a new state is reached a partition function is used to determine which node is the owner of the state and the state is sent there for storage and analysis of the state's successors. If the state has to be visited in the same node then the visit continues depth first from that node. The node the starting state belongs

to will start the visit while the others are waiting for incoming messages. The search is completed when all nodes are waiting and all messages sent have been received. The major issue with this algorithm is picking a partition function that minimizes the memory required by each node, but at the same time limits the number of messages required between the nodes.

3.1 Improvements

The algorithm from [LS99] has been adapted to our system, at first without any modification. After making the tool operational we worked on some extensions, mainly aiming to reduce the communication overhead. In [LS99] a modification of the algorithm, called *sibling storing*, is presented, which reduces the number of messages sent. Each time a new state that needs to be sent to another node – sibling – is reached, a local copy of the state sent is kept in the local hash table. If encountered again, no messages need to be sent, since we know that it has been received by that node before. One issue with this technique is that the number of siblings can grow quickly and consume too much of the memory, taking space that could be allocated to store actual (local) states.

We developed a modified version of this technique, that we called *sibling caching*, that stores the siblings in a cache. When no empty space is left in the cache, the least recently used element is discarded and the new sibling is added. This technique proved to be quite effective – see Table 3 – because a very limited size cache performs, in terms of messages avoided, almost as good as complete sibling storing. Table 3 shows the traditional partition algorithm (that uses a hash function over the complete system state for partitioning) augmented with a number of optimizations techniques and compares these with respect to memory usage, percentage of transitions that generates messages, and lastly the time taken.

Table 3. Results using different optimizations with the RemoteAgent example

RemoteAgent	**Memory** *MB*	**Messages** *%*	**Time** *(sec)*
Normal Distributed	40.69	38	525.16
Sibling Storing	45.70	27	504.65
Sibling Caching (50)	40.10	34	518.52
Sibling Caching (200)	39.60	31	515.26
Sibling Caching (500)	39.19	28	511.77
Children Lookahead	29.67	31	320.75
Children Lookahead + Sibling Storing	33.83	23	296.87
Children Lookahead + Sibling Caching (50)	29.19	28	316.11
Children Lookahead + Sibling Caching (200)	29.14	28	314.61
Children Lookahead + Sibling Caching (500)	28.86	26	319.06

Another extension we developed is called *children lookahead*. This technique tries to avoid sending messages due to short paths that fall into another node's state-space. As can be seen in Figure 6, state $s_{i-1}^{(0)}$ is followed by state $s_i^{(1)}$. This generates a message from node 0 to node 1. When node 1 receives the new state

it generates its successor $s_{i+1}^{(0)}$. This node needs to be stored by node 0 and so a second message is generated. This last message could have been avoided if node 0 had checked the successors of $s_i^{(1)}$ for any state belonging to itself – this operation being not very expensive because state $s_i^{(1)}$ has already been generated by node 0.

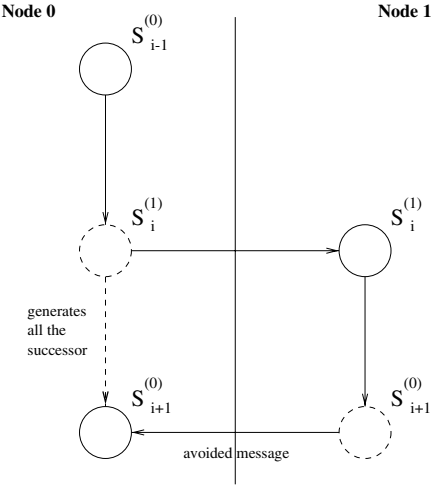


Fig. 6. Example of children lookahead

In order for this technique to work it is necessary to clearly specify who is going to take care of each state. When a message is sent the sender will check for its own states among the successors of the sent state. On the other hand the receiver will skip every state belonging to the sender that falls in the first generation starting from the received state. This algorithm works also if there are more than two parties involved, because the sender will just ignore states belonging to a third party, while the receiver will send the state to the correct node – therefore avoiding duplication. It is important however for the node to check for possible states belonging to itself in the first generation of the state sent to the third node.

This technique avoids messages for runs that last only one state in another node's part of the state space, but the technique can be generalized to an arbitrary number of steps, corresponding to the number of generations that need to be checked. There exists a trade off between the number of generations checked – and therefore the number of possible messages avoided – and the time overhead necessary to generate all the successors – which grows exponentially with respect to the number of generations.

Some results can be seen in Table 3 where the same example has been executed with different combinations of the presented techniques: sibling storing reduces the number of messages more consistently than sibling caching, but in-

creasing the size of the cache – 50, 200, and 500 in the table – the percentage of messages gets closer to the results obtained using storing. Children lookahead appers to be very effective and is orthogonal to the other techniques. The results in terms of message reduction are mirrored by the reduction in execution time. The memory usage should be higher with sibling storing, decrease with the caching and be minimum without any sibling algorithm. Nevertheless the experimental results give an anomalous behavior that we were not able to explain.

State Transfer. A difference between SPIN and JPF is where most of the execution time is spent: in SPIN storing the state uses most of the time, while in JPF execution of the bytecode instructions is the most expensive operation. This is due, in part, to the fact that in JPF transitions are more complex than in SPIN. This difference can affect the design of the distributed version of the two tools. In [LS99] the communication protocol between the nodes had been designed so that a path is sent to identify the state that needs to be visited. This is consistent with the assumption that steps can be executed very efficiently in SPIN. On the other hand, in JPF an execution step is very time consuming so it would sound efficient to send the state, without any need for the path. In JPF however the state is a very complex structure that includes references. At the early stages of the development we tried to send states, but the time necessary to translate the states into something that can be sent on a socket was too high. Therefore our choice was to send the path and use that information to reconstruct the state on the destination node. Although we are sending a path at the moment we believe that when doing the implementation on a parallel architecture sending states would become viable.

Another possibility that we are currently exploring is to send across a compressed version of the state. Since the state is very efficiently compressed by the tool for storing, it would be effective to send the compressed state over the network. This is not at the moment possible because the pools used for the compression are local to each node and therefore it would be impossible to correctly reconstruct the state on the receiving node. One possible approach – that is particularly interesting in a parallel environment, but it's still applicable in a distributed one – would be to centralize the pools used for the compression. This way indexes for components of the state would be global and the compressed state could be easily and quickly transferred between nodes. In order to reduce the communication each node could keep a copy of the entries that it accessed from the centralized pool and only when a new entry has to be added, communication is necessary.

3.2 Partitioning

Partitioning is a crucial point in the distributed algorithm [LS99]. Partitioning aims to achieve two contrasting goals, with an obvious trade off:

- reduce the number of message that need to be transmitted; and
- maintain a fair partitioning of the memory required on each node.

In [LS99] a few heuristics to determine a partition function are suggested. In [Ler00] a more complete approach to the problem is given, and a tool to automatically generate a partition function from static analysis of the input model is presented. However, because of the dynamic nature of the systems we address, these kind of tools are more difficult to implement due to the complexity of the static analysis required. We will therefore focus on partition functions that can be calculated dynamically and compare them to static partitioning functions that do not require any static analysis. In [Ler00] partition functions are classified as:

static: the partitioning is made before the verification is run and no changes are possible once at run-time; or

dynamic: the partition function is adapted at run-time using the information gathered during execution to better suit the system that is being model checked.

These two kinds of partition functions have their advantages and disadvantages: static partitioning does not require further communication to determine which node a state belongs to but it is hard to come up with a good function, i.e. one that achieves both equal partitions and low communication. On the other hand, dynamic partitioning requires a higher level of communication and complexity, but allows more versatility (it is not model dependent) and equal partition size. All the partition functions used in [LS99] are static but the algorithm presented does not rely on that assumption. Static partitioning is especially problematic for a dynamic system, because it is hard to extrapolate what the behavior and structure of the system will be. Therefore a dynamic approach, since it is not dependent on the system structure, seems more appropriate when analyzing dynamic systems.

3.3 Static Partitioning

First we present some examples using static partitioning – Table 4 – that can be used as a reference for the results presented further on. They are also important because – as we will see in Section 3.4 – these partition functions are used as a basis for the dynamic ones.

The results in the table are for the RemoteAgent example using two workstations. The different partition functions are reported in the first column, followed by the percentage of the total memory used by each workstation, the percentage of transitions that cause messages to be sent, and lastly the time taken.

The Global Hash Code partition function uses a hash function to determine the partition: the function is applied to the whole state and the partition is the result modulo the number of partitions. This solution gives a fair division of the state space between the nodes, but at the same time, the number of messages generated is pretty high.

A possible approach is to use the locality principle [LS99]: if the partition function relies only on the information of a particular thread, only when that thread is scheduled is it possible to reach a state that generates a message. As a first step we created a partition function – Local Hash Code – that applies the

Table 4. Different static partition functions on the RemoteAgent example

RemoteAgent	**Memory** *%*	**Messages** *%*	**Time** *(sec)*
Global Hash Code	50/50	38	525.16
Local Hash Code	50/50	46	551.28
Local Hash Code (1)	44/56	11	241.46
Local Hash Code (2)	47/53	13	263.57
Program Counter (1)	54/46	17	342.34
Program Counter (2)	48/52	25	487.43
Program Counters (1)	54/46	17	339.46
Program Counters (2)	43/57	14	326.66

hash function to the thread list only. This implies that the value of the objects are not included in the hashing process – only the stack frames and thread status of all threads. As a further step we limited this process to a specific thread (indicated by a number in the table). The results, in terms of message ratio, are still not very good for the function if applied to the whole thread list because at each step at least one of the program counters will change, but if applied to a single thread, messages are reduced. The choice of the thread is also important: the first thread – zero – is usually a bad choice (hence it was not included in the table), since it is the main thread that often is simply used to create the threads that compose the real system. In general a reduction of the messages is obtained, with a sacrifice in the fairness of the partition.

As said before, in [Ler00] a tool to generate a partition function using static analysis has been presented. Unfortunately, this tool cannot be applied, as is, here because it uses the flow control graph of a thread, that is not as accessible as in SPIN. The idea behind it is to use the current state of a process to determine the partition the state belongs to: on a similar path we tried to use the program counter of threads to do this. Since the static analysis approach cannot be used, we just hashed the program counter. At first sight the program counter seems to be the equivalent of the current state of a Promela process, but because of the stack based approach, each thread has more than one program counter. At first we tried to use the program counter from the topmost stack frame (rows saying Program Counter as partition function), then we tried the same approach using a function of all the program counters from every stack frame (rows marked as Program Counters). The result is a reduction of the percentage of messages, which is higher when the function is applied to a specific thread. Note again, that the main thread (thread zero) is not shown, since as before it is only used to start the rest of the system and hence leads to a very unbalanced partitioning.

An interesting observation is that one can either use too much information to calculate the partitioning, in which case the partition is fair but creates too many messages (see Global Hash Code and Local Hash Code) or one can use too little information (see Program Counter 1 and 2) with similar problems. Using just enough information seemed to give the best results: Local Hash Code for a thread and Program Counters per thread.

3.4 Dynamic Partitioning

The great advantage of dynamic partitioning is that no prior knowledge or static analysis of the system are necessary: run-time information is used to keep the partitioning fair. Dynamic partitioning just means that states can be stored in one node at a certain moment and in another one later on. In general a dynamic partition function will initially assign a subset of the state space to each node and when a certain condition arises – for instance lack of main memory – it will reassign some states to a different node.

In our work we assume that at any given time each node knows where each state is supposed to be stored. This means that a node does not need to interrogate every other node to know if a state has already been visited, but can send it to the legitimate owner. This assumption can be dropped a for limited time after a reassignment with the condition that nodes relay the incoming states that do not belong – anymore – to them to the correct designated node.

One issue that arises at this point is how to represent a partition function that can change with time. It is necessary for some sort of table to specify which state is stored where. It is obvious that the granularity of this table cannot be the single state, otherwise the size of the table would be of the same order of magnitude of the size of the whole state space. States can be grouped together: we called these groups of states *classes*. It is clear that the classification is equivalent to the partitioning. The assumption we made here is that the same techniques used for determining static partition functions can be used to determine a classification function. What is important is that classes do not need to be the exact same size. In fact the number of classes is greater then the number of partitions, and each partition consists of a set of classes: at run-time classes will be grouped together into partitions and when a partition's size is too big, part of it – a class – can be assigned to another node. Still important is to minimize the number of potential messages between two classes, but we do not have a strong trade off like we used to. This solves the problem of having an excellent partition function, since an average classification function can give optimal results.

When a reassignment is issued the states that have already been visited but now belong to a different partition have to be discarded. It is not efficient – at least in a distributed environment – to transfer that information across the network. The node those states are assigned to will rediscover those states – if they will ever be met again – without actually influencing the result of the computation, just extending the search – since some states may be visited more than once.

Table 5 show the results obtained using different dynamic partition functions based on the static ones presented in the previous section. Half of the classes are initially assigned to each node and, if necessary, they will be reassigned. When comparing these results with the static partition results from Table 4 it is clear that in every case the dynamic partition achieves either a similar or better memory distribution and runtime.

An important issue is to decide when a reassignment is necessary: an option would be to start it when the number of states stored in one partition is too

Table 5. Results from dynamic partition with classes split equally

RemoteAgent	**Memory** *%*	**Messages** *%*	**Time** *(sec)*
Global Hash Code	50/50	38	524.02
Local Hash Code	50/50	46	531.17
Local Hash Code (1)	48/53	11	216.56
Local Hash Code (2)	48/52	13	281.25
Program Counter (1)	52/48	17	340.97
Program Counter (2)	48/52	25	487.46
Program Counters (1)	52/48	17	311.22
Program Counters (2)	49/51	13	333.78

big compared to what is currently stored in the others. However this is not a good idea, since having a greater amount of states stored in one partition is not necessarily a problem until memory comes to exhaustion. It is better to wait until the memory become an issue than keep the two partition at the same size during the whole visit – also because a class' size can change, for instance if most of its states will be visited close to the end of the verification.

Another issue is that, even when the memory is abundant, the two nodes have to communicate intensively since the beginning, because the states have been divided as equally as possible before starting. One possible optimization would be to store all the states – at least initially – on the same node and let the reassignment and the dynamic algorithm do the work of obtaining a better partitioning. This last approach leaves all but one node completely useless from the beginning up to the time the first node exhausts its memory resources and starts splitting the state space – and the work – with the others. One disadvantage is that a lot of work might need to be redone, because every time a class is reassigned a part of the state space that has already been visited is lost. Table 6 show the results of doing this form of dynamic partitioning — the fairness of the partitioning is still very good, but now the messages and hence the time is much reduced.

Table 6. Results from having one node start with all the classes

RemoteAgent	**Memory**	**Messages** *%*	**Time** *(sec)*
Global Hash Code	47/53	16	251.74
Local Hash Code	47/54	13	150.09
Local Hash Code (1)	45/55	5	97.57
Local Hash Code (2)	53/47	7	125.09
Program Counter (1)	45/55	11	141.38
Program Counter (2)	44/56	11	134.14
Program Counters (1)	39/61	10	145.38
Program Counters (2)	48/52	14	182.00

One possible way to avoid having idle workstations is to assign to each node a set of states that belong to it, but let them, at the beginning, visit and store also other states. This way all nodes will start visiting the state space at the same time without any need to send messages, because states can be stored in their own hash table. When memory becomes an issue, those states that belong to others can be discarded to make space for local states, but after that messages needs to be sent for those nodes falling in that part of the state space. At first this technique seems very similar to sibling storing but the difference is that states are stored without sending a message. In fact if the successors of a given state are fully visited by one node, the search will be correct even if later on this state will be discarded.

To clarify this technique let's suppose we have only two nodes. Initially both the workstations start the visit until they reach a moment in time when memory becomes scarce. At this point each node will have to discard a part of the state space. Let's assume for simplicity that only two classes were defined: each node will keep one of them and reject the other. With a minimum amount of coordination – to avoid that both reject the same class – both nodes now have only one class stored in their hash table. If we suppose for simplicity again that both nodes got to the exact same point in the visit when they decided to reassign one of their classes, no state would be lost, because each node is keeping what the other rejected – see Figure 7.

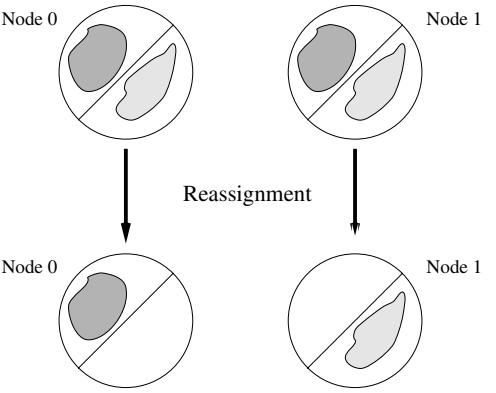


Fig. 7. How states are visited and discarded

This was a simplification: in a more realistic scenario there would be a number of states that are lost, but surely quite limited compared to the same situation where only one node had been running until the first reassignment started. Moreover if more classes than partitions exist the same dynamic techniques applied before can be used. At any time each class can be either stored by a specific node or shared among a set of them. When memory is scarce a shared class can be discarded – making sure ahead of time that not everybody else discard it at

the same time – or a non-shared class can be transfered to another node. Table 7 shows results from this form of dynamic partitioning — note how the messages and time is even further reduced from Table 6.

Table 7. Results from having two nodes start together

RemoteAgent	**Memory** *%*	**Messages** *%*	**Time** *(sec)*
Global Hash Code	50/50	9	251.17
Local Hash Code	48/52	7	120.92
Local Hash Code (1)	52/48	3	74.46
Local Hash Code (2)	48/52	2	80.65
Program Counter (1)	38/62	4	113.21
Program Counter (2)	49/51	4	122.66
Program Counters (1)	48/52	5	112.46
Program Counters (2)	52/48	3	113.84

4 Conclusions

Program model checking is an area of active research since the importance of software and its failures is increasing. Model checking of software presents specific issues that are due to the complexity and the dynamic nature of programs. Translation-based approaches cannot adequately deal with these since they rely on underlying tools that are not designed to exploit programs specific characteristics. We developed our own model checking tool using a programming language, Java, as our input notation in order to be able to overcome this limitation.

We first introduced a representation for the state that respects the paradigm underlying the input notation. In order to be able to explore the state space of a reasonable size we developed a compression algorithm that exploits the structure of the system state. A novel approach to efficient backtracking has been presented, that reconstructs the state from the compressed version present on the stack. A novel approach to symmetry has been introduced, that exploits symmetries inherent to the state representation. Garbage collection is discussed as a further way to reduce the state space.

Even after applying state space reduction techniques programs are often still too large for the memory of a single workstation: a distributed memory algorithm can overcome this. We show how an existing distributed model checking algorithm can be extended to reduce communication overhead and do dynamic memory balancing. We show results supporting our claim that dynamic partitioning of the state space over multiple workstations is well suited to analyze dynamic (Java) programs. Although we did not show that the dynamic partition functions presented here allow the verification to converge, we believe this is the case. We intend to address this issue formally for more complicated functions that we are currently developing.

This paper focussed on techniques that can be applied without any prior knowledge of system structure. We do however believe that many reduction techniques based on a-priori static analysis of the system, such as slicing, partial order reductions, abstractions, etc., can improve the model checking process and should be applied whenever possible.

In the future, we intend to further investigate the combination of static and dynamic reduction techniques to combat the state explosion. Furthermore, we believe that parallel model checking will become more popular in the future due to the use of such machines becoming more widespread. To this end we are currently extending our distributed model checking algorithm to be used on a parallel shared-memory architecture (SGI Origin 2000).

References

[BBS00] J. Barnat, L. Brim, and J. Stribrna. Distributed LTL model-checking in SPIN. Technical Report FIMU-RS-2000-10, Faculty of Informatics, Masaryk University, 2000. Available in this LNCS volume.

[BDH00] D. Bosnacki, D. Dams, and L. Holenderski. Symmetric SPIN. In *Proc. of the 7th International SPIN Workshop*, volume 1885 of *LNCS*. Springer-Verlag, September 2000.

[BKR98] Nick Benton, Andrew Kennedy, and George Russell. Compiling standard ML to Java bytecodes. *SIGPLAN Notices*, 34(1):129–140, September 1998.

[BLPV95] J. Bormann, J. Lohse, M. Payer, and G. Venzl. Model checking in industrial hardware design. In *Proc. of the 32nd Design Automation Conference*, 1995.

[BR00] Thomas Ball and Sriram K. Rajamani. Bebop: A symbolic model checker for boolean programs. In *Proc. of the 7th International SPIN Workshop*, volume 1885 of *LNCS*, pages 113–130. Springer-Verlag, September 2000.

[CD98] L.R. Clausen and O. Danvy. Compiling proper tail recursion and first-class continuations: Scheme on the Java Virtual Machine. *The Journal of C Language Translation*, 6(1):20–32, April 1998.

[CDH+00] J. Corbett, M. Dwyer, J. Hatcliff, S. Laubach, C. Pasareanu, and R. Zheng. Bandera: Extracting finite-state models from Java source code. In *Proc. of the 22nd International Conference on Software Engineering*, June 2000.

[CEJS98] Edmund M. Clarke, E. Allen Emerson, Somesh Jha, and A. Prasad Sistla. Symmetry reductions in model checking. In *Proc. of the 10th International Conference on Computer Aided Verification*, volume 1427 of *LNCS*, pages 147–158. Springer-Verlag, 1998.

[CFJ93] Edmund M. Clarke, T. Filkorn, and S. Jha. Exploiting symmetries in temporal logic model checking. In *Proc. of the 5th International Conference on Computer Aided Verification*, volume 697 of *LNCS*. Springer-Verlag, 1993.

[CW96] Edmund M. Clarke and J. M. Wing. Formal methods: State of the art and future directions. Technical Report CMU-CS-96-178, Carnegie Mellon University, 1996.

[DIS99] Claudio Demartini, Radu Iosif, and Riccardo Sisto. A deadlock detection tool for concurrent Java programs. *Software - Practice and Experience*, 29(7):577–603, 1999.

[EJ93] E. Emerson and C. Jutla. Symmetry and model checking. In *Proc. 5th International Conference on Computer Aided Verification*, volume 697 of *LNCS*. Springer-Verlag, 1993.

[God97] Patrice Godefroid. VeriSoft: A tool for the automatic analysis of concurrent reactive software. In *Proc of the 9th International Conference on Computer Aided Verification*, volume 1254 of *LNCS*, pages 476–479. Springer-Verlag, June 1997.

[Hol91] Gerard J. Holzmann. *Design and Validation of Computer Protocols*. Prentice Hall, 1991.

[Hol97a] Gerard J. Holzmann. The SPIN model checker. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.

[Hol97b] Gerard J. Holzmann. State compression in SPIN: Recursive indexing and compression training runs. In *Proc. of the 3th International SPIN Workshop*, April 1997.

[Hol00] Gerard J. Holzmann. Logic verification of ANSI-C code with SPIN. In *Proc. of the 7th International SPIN Workshop*, volume 1885 of *LNCS*. Springer-Verlag, September 2000.

[HP98] Klaus Havelund and Thomas Pressburger. Model checking Java programs using Java PathFinder. *International Journal on Software Tools for Technology Transfer*, 2(4), April 1998.

[ID96] C. Norris Ip and David L. Dill. Better verification through symmetry. *Formal Methods in System Design*, 9(1/2):47–75, August 1996.

[IS99] Radu Iosif and Riccardo Sisto. dSPIN: A dynamic extension of SPIN. In *Proc. of the 6th International SPIN Workshop*, volume 1680 of *LNCS*, pages 261–276. Springer-Verlag, September 1999.

[IS00] Radu Iosif and Riccardo Sisto. Using garbage collection in model checking. In *Proc. of the 7th International SPIN Workshop*, volume 1885 of *LNCS*, pages 20–33. Springer-Verlag, September 2000.

[Ler00] Flavio Lerda. Model checking: Tecniche di verifica formale in ambiente distributo. Master's thesis, Politecnico di Torino, May 2000.

[LS99] Flavio Lerda and Riccardo Sisto. Distributed-memory model checking with SPIN. In *Proc. of the 5th International SPIN Workshop*, volume 1680 of *LNCS*. Springer-Verlag, 1999.

[SD97] Ulrich Stern and David L. Dill. Parallelizing the Murphi verifier. In *Proc. of the 9th International Conference on Computer Aided Verification*, volume 1254 of *LNCS*, pages 256–278. Springer-Verlag, June 1997.

[Spa00] SpaceViews. Premature engine cutoff likely cause of Mars Polar Lander failure. *http://www.spaceviews.com/2000/03/28b.html*, March 2000.

[Sto00] Scott D. Stoller. Model-checking multi-threaded distributed Java programs. In *Proc. of the 7th International SPIN Workshop*, volume 1885 of *LNCS*, pages 224–244. Springer-Verlag, September 2000.

[Taf96] S. Tucker Taft. Programming the Internet in Ada 95. In *Ada-Europe International Conference on Reliable Software Technologies*, volume 1088 of *LNCS*, pages 1–16. Springer-Verlag, June 1996.

[VHBP00] Willem Visser, Klaus Havelund, Guillaume Brat, and Seung-Joon Park. Model checking programs. In *Proc. of the 15th IEEE International Conference on Automated Software Engineering*, September 2000.

Automatically Validating Temporal Safety Properties of Interfaces

Thomas Ball and Sriram K. Rajamani

Software Productivity Tools
Microsoft Research
http://www.research.microsoft.com/slam/

Abstract. We present a process for validating temporal safety properties of software that uses a well-defined interface. The process requires only that the user state the property of interest. It then automatically creates abstractions of C code using iterative refinement, based on the given property. The process is realized in the SLAM toolkit, which consists of a model checker, predicate abstraction tool and predicate discovery tool. We have applied the SLAM toolkit to a number of Windows NT device drivers to validate critical safety properties such as correct locking behavior. We have found that the process converges on a set of predicates powerful enough to validate properties in just a few iterations.

1 Introduction

Large-scale software has many components built by many programmers. Integration testing of these components is impossible or ineffective at best. Property checking of interface usage provides a way to partially validate such software. In this approach, an interface is augmented with a set of properties that all clients of the interface should respect. An automatic analysis of the client code then validates that it meets the properties, or provides examples of execution paths that violate the properties. The benefit of such an analysis is that errors can be caught early in the coding process.

We are interested in checking that a program respects a set of *temporal safety* properties of the interfaces it uses. Safety properties are the class of properties that state that "something bad does not happen". An example is requiring that a lock is never released without first being acquired (see [24] for a formal definition). Given a program and a safety property, we wish to either validate that the code respects the property, or find an execution path that shows how the code violates the property.

In this paper, we show that safety properties of system software can be validated/invalidated using model checking, without the need for user-supplied annotations (invariants) or user-supplied abstractions. The user only needs to state the safety properties of interest (in our specification language SLIC, described later). As no annotations are required, we use model checking to compute fixpoints automatically over an abstraction of the C code. We construct an

M.B. Dwyer (Ed.): SPIN 2001, LNCS 2057, pp. 103–122, 2001.

appropriate abstraction by (1) obtaining an initial abstraction from the property that needs to be checked, and (2) refining this abstraction using an automatic refinement algorithm.

We model abstractions of C programs using *boolean programs* [3]. Boolean programs are C programs in which all variables have boolean type. Each boolean variable in a boolean program has an interpretation as a predicate over the infinite state space of the C program. Our experience shows that our refinement algorithm finds boolean program abstractions that are precise enough to validate properties. Furthermore, if the property is violated, the process of searching for a suitable boolean program abstraction leads to a manifestation of the violation.

We present the SLAM toolkit for checking safety properties of system software, and report on our experience in using the toolkit to check properties of Windows NT device drivers. Given a safety property to check on a C program, the SLAM process has the following phases: (1) abstraction, (2) model checking, and (3) predicate discovery. We have developed tools to support each of these phases:

- C2BP, a tool that transforms a C program P into a boolean program $\mathcal{BP}(P, E)$ with respect to a set of predicates E over the state space of P [1, 2];
- BEBOP, a tool for model checking boolean programs [3], and
- NEWTON, a tool that discovers additional predicates to refine the boolean program, by analyzing the feasibility of paths in the C program.

The SLAM toolkit provides a fully automatic way of checking temporal safety properties of system software. Violations are reported by the SLAM toolkit as paths over the program P. It never reports spurious error paths. Instead, it detects spurious error paths and uses them to automatically refine the abstraction (to eliminate these paths from consideration). Since property checking is undecidable, the SLAM refinement algorithm may not converge. However, in our experience, it usually converges in a few iterations. Furthermore, whenever it converges, it gives a definite "yes" or "no" answer.

The worst-case run-time complexity of the SLAM tools BEBOP and C2BP is linear in the size of the program's control flow graph, and exponential in the number of predicates used in the abstraction. We have implemented several optimizations to make BEBOP and C2BP scale gracefully in practice, even with a large number of predicates. The NEWTON tool scales linearly with path length and number of predicates.

We applied the SLAM toolkit to check the use of the Windows NT I/O manager interface by device driver clients. There are on the order of a hundred rules that the clients of the I/O manager interface should satisfy. We have automatically checked properties on device drivers taken from the Microsoft Driver Development Kit[1]. While checking for correct use of locks, we found that the SLAM process converges in one or two iterations to a boolean program that is

[1] The code of the device drivers we analyzed is freely available from `http://www.microsoft.com/ddk/W2kDDK.htm`

sufficiently precise to validate/invalidate the property. We also checked a data-dependent property, which requires keeping track of value-flow and aliasing, using four iterations of the SLAM tools.

The remainder of this paper is organized as follows. Section 2 gives an overview of the SLAM approach by applying the tools to verify part of an NT device driver. Sections 3, 4 and 5 give brief descriptions of the three tools that compose the SLAM toolkit and explain how they work in the context of the running example. Section 6 describes our experience applying the tools to various NT device drivers. Section 7 discusses related work and Section 8 concludes the paper.

2 Overview

This section introduces the SLAM refinement algorithm and then applies this algorithm to a small code example, extracted from a PCI device driver. The SLAM toolkit handles a significant subset of the C language, including pointers, structures, and procedures (with recursion and mutual recursion). A limitation of our tools is that they assume a *logical* model of memory when analyzing C programs. Under this model, the expression $p + i$, where p is a pointer and i is an integer, yields a pointer value that points to the same object pointed to by p. That is, we treat pointers as references rather than as memory addresses. Note that this is the same basic assumption underlying most points-to analysis, including the one that our tools use [11].

2.1 Property Specification

We have created a low-level specification language called SLIC (Specification Language for Interface Checking) in which the user states safety properties. A SLIC specification describes a state machine and has two components: (1) a static set of *state variables*, described as a C structure, and (2) a set of *events* and state transitions on the events. The state variables can be of any C type, including integers and pointers.

Figure 1(a) shows a SLIC specification that specifies proper usage of spin locks. There is one state variable `locked` that is initialized to `0`. There are two events on which state transitions happen —returns of calls to the functions `KeAcquireSpinLock` and `KeReleaseSpinLock`. Erroneous sequences of calls to these functions results in the execution of the `abort` statement.

We wish to check if a temporal safety property φ specified using SLIC is satisfied by a program P. We have built a SLIC instrumentation tool that automatically instruments the given program P with property φ to result in a program P' such that P satisfies φ iff the label `SLIC_ERROR` is not reachable in P'. In particular, the tool first creates C code from the SLIC specification, as shown in Figure 1(b). The label `SLIC_ERROR` in the procedure `slic_abort` reflects the finite state machine executing an `abort` statement and moving into a reject state. The tool then inserts calls to the appropriate SLIC C functions

```
state {
   enum { Unlocked=0, Locked=1 }
     state = Unlocked;
}

KeAcquireSpinLock.return {
  if (state == Locked)
    abort;
  else
    state = Locked;
}

KeReleaseSpinLock.return {
  if (state == Unlocked)
    abort;
  else
    state = Unlocked;
}
```

(a)

```
enum { Unlocked=0, Locked=1 }
  state = Unlocked;

void slic_abort() {
  SLIC_ERROR: ;
}

void KeAcquireSpinLock_return() {
  if (state == Locked)
    slic_abort();
  else
    state = Locked;
}

void KeReleaseSpinLock_return {
  if (state == Unlocked)
    slic_abort();
  else
    state = Unlocked;
}
```

(b)

Fig. 1. (a) A SLIC specification for proper usage of spin locks, and (b) its compilation into C code.

in the program P to result in the instrumented program P'. This is known in the model checking community as a "product automaton construction" and is a fairly standard way to encode safety properties. Due to want of space, the formal syntax and semantics of SLIC, and details of the automatic instrumentation tool will be the topic of a future paper.

2.2 Refinement Algorithm

We wish to check if the instrumented program P' can ever reach the label `SLIC_ERROR`. Let i be a metavariable that records the SLAM iteration count. In the first iteration ($i = 0$), we start with a set of predicates E_0 that are present in the conditionals of the SLIC specification. Let E_i be some set of predicates over the state of P'. Then iteration i of SLAM is carried out using the following steps:

1. Apply C2BP to construct the boolean program $\mathcal{BP}(P', E_i)$.
2. Apply BEBOP to check if there is a path p_i in $\mathcal{BP}(P', E_i)$ that reaches the `SLIC_ERROR` label. If BEBOP determines that `SLIC_ERROR` is not reachable, then the property φ is valid in P, and the algorithm terminates.
3. If there is such a path p, then we use NEWTON to check if p is feasible in P. There are two outcomes:

```
void example() {
 do {
  KeAcquireSpinLock();

  nPacketsOld = nPackets;
  req = devExt->WLHV;
  if(req && req->status){
    devExt->WLHV = req->Next;
    KeReleaseSpinLock();

    irp = req->irp;
    if(req->status > 0){
      irp->IoS.Status = SUCCESS;
      irp->IoS.Info = req->Status;
    } else {
      irp->IoS.Status = FAIL;
      irp->IoS.Info = req->Status;
    }
    SmartDevFreeBlock(req);
    IoCompleteRequest(irp);
    nPackets++;
  }
 } while(nPackets!=nPacketsOld);
 KeReleaseSpinLock();

}
```

(a) Program P

```
void example() {
   do {
    KeAcquireSpinLock();
A:  KeAcquireSpinLock_return();
    nPacketsOld = nPackets;
    req = devExt->WLHV;
    if(req && req->status){
      devExt->WLHV = req->Next;
      KeReleaseSpinLock();
B:    KeReleaseSpinLock_return();
      irp = req->irp;
      if(req->status > 0){
        irp->IoS.Status = SUCCESS;
        irp->IoS.Info = req->Status;
      } else {
        irp->IoS.Status = FAIL;
        irp->IoS.Info = req->Status;
      }
      SmartDevFreeBlock(req);
      IoCompleteRequest(irp);
      nPackets++;
    }
   } while(nPackets!=nPacketsOld);
   KeReleaseSpinLock();
C: KeReleaseSpinLock_return();
 }
```

(b) Instrumented Program P'

Fig. 2. (a) A snippet of device driver code P, and (b) instrumented code P' that checks proper use of spin locks.

- "yes": the property φ has been invalidated in P, and the algorithm terminates with an error path p_i (a witness to the violation of φ).
- "no": NEWTON finds a set of predicates F_i that explain the infeasibility of path p_i in P.

4. Let $E_{i+1} := E_i \cup F_i$, and $i := i + 1$, and proceed to the next iteration.

As stated before, this algorithm is potentially non-terminating. However, when it does terminate, it provides a definitive answer.

2.3 Example

Figure 2(a) presents a snippet of (simplified) C code from a PCI device driver that processes interrupt request packets (IRPs). Of interest here are the calls the code makes to acquire and release spin locks (`KeAcquireSpinLock` and `KeReleaseSpinLock`). Figure 2(b) shows the program automatically instrumented by the SLIC tool with respect to the property specification in Figure 1(a). Note that calls to the appropriate SLIC C functions (Figure 1(b)) are introduced (at labels `A`, `B`, and `C`).

The question we wish to answer is: is the label `SLIC_ERROR` reachable in the program P' comprised of the code from Figure 1(b) and Figure 2(b)? The following sections apply the algorithm given above to show that `SLIC_ERROR` is unreachable in this program.

```
decl {state==Locked}, {state==Unlocked};

void slic_abort() begin SLIC_ERROR: skip; end

void KeAcquireSpinLock_return()
begin
  if ({state==Locked})
    slic_abort();
  else
    {state==Locked},{state==Unlocked} := T,F;
end

void KeReleaseSpinLock_return()
begin
  if ({state == Unlocked})
    slic_abort();
  else
    {state==Locked},{state==Unlocked} := F,T;
end
```

Fig. 3. The C code of the SLIC specification from Figure 1(b) compiled by C2BP into a boolean program.

2.4 Initial Boolean Program

The first step of the algorithm is to generate a boolean program from the C program and the set of predicates E_0 that define the states of the finite state machine. We represent our abstractions as *boolean programs*. The syntax and semantics of boolean programs was defined in [3]. Boolean programs are essentially C programs in which the only allowed types are `bool`, with values T (true) and F (false), and `void`. Boolean programs also allow control non-determinism, through the conditional expression "$*$", as shown later on.

For our example, the set E_0 consists of two *global* predicates $(state = Locked)$ and $(state = Unlocked)$ that appear in the conditionals of the SLIC specification. These two predicates and the program P' are input to the C2BP (C to Boolean Program) tool. The translation of the SLIC C code from Figure 1(b) to the boolean program is shown in Figure 3. The translation of the `example` procedure is shown in Figure 4(a). Together, these two pieces of code comprise the boolean program $\mathcal{BP}(P', E_0)$ output by C2BP.

As shown in Figure 3, the translation of the SLIC C code results in the global variables, {`state==Locked`} and {`state==Unlocked`}.[2] For every statement s in the C program and predicate $e \in E_0$, the C2BP tool determines the effect of state-

[2] Boolean programs permit a variable identifier to be an arbitrary string enclosed between "{" and "}". This is helpful for giving boolean variables names to directly represent the predicates in the C program that they represent.

```
    void example()
    begin
     do
      skip;
A:    KeAcquireSpinLock_return();
      skip;
      skip;
      if (*) then
        skip;
        skip;
B:      KeReleaseSpinLock_return();
        skip;
        if (*) then
          skip;
          skip;
        else
          skip;
          skip;
        fi
        skip;
        skip;
        skip;
      fi
     while (*);
     skip;
C:   KeReleaseSpinLock_return();
    end
```

(a) Boolean program $\mathcal{BP}(P', E_0)$

```
    void example()
    begin
     do
      skip;
A:    KeAcquireSpinLock_return();
      b := T;
      skip;
      if (*) then
        skip;
        skip;
B:      KeReleaseSpinLock_return();
        skip;
        if (*) then
          skip;
          skip;
        else
          skip;
          skip;
        fi
        skip;
        skip;
        b := choose(F,b);
      fi
     while (!b);
     skip;
C:   KeReleaseSpinLock_return();
    end
```

(b) Boolean program $\mathcal{BP}(P', E_1)$

Fig. 4. The two boolean programs created while checking the code from Figure 2(b). See text for the definition of the `choose` function.

ment s on predicate e. For example, consider the assignment statement "`state = Locked;` " in Figure 1(b). This statement makes the predicate $(state = Locked)$ true and the predicate $(state = Unlocked)$ false. This is reflected in the boolean program by the parallel assignment statement

```
{state==Locked}, {state==Unlocked} := T,F;
```

in Figure 3. The translation of the boolean expressions in the conditional statements of the C program results in the obvious corresponding boolean expressions in the boolean program. Control non-determinism is used to conservatively model the conditions in the C program that cannot be abstracted precisely using the predicates in E_0, as shown in Figure 4(a).

Many of the assignment statements in the `example` procedure are abstracted to the **skip** statement (no-op) in the boolean program. The C2BP tool uses Das's

points-to analysis [11] to determine whether or not an assignment statement through a pointer dereference can affect a predicate e. In our example, the points-to analysis shows that no variable in the C program can alias the address of the global `state` variable.[3]

We say that the boolean program $\mathcal{BP}(P', E_0)$ *abstracts* the program P', since every feasible execution path p of the program P' also is a feasible execution path of $\mathcal{BP}(P', E_0)$.

2.5 Model Checking the Boolean Program

The second step of our process is to determine whether or not the label `SLIC_ERROR` is reachable in the boolean program $\mathcal{BP}(P', E_0)$. We use the BEBOP model checker to determine the answer to this query. In this case, the answer is "yes". Like most model checkers, the BEBOP tool produces a (shortest) path leading to the error state. In this case, the shortest path to the error state is the path that goes around the loop twice, acquiring the lock twice without an intermediate release, as given by the error path p_0 of labels [`A, A, SLIC_ERROR`].

2.6 Predicate Discovery over Error Path

Because the C program and the boolean program abstractions have identical control-flow graphs, the error path p_0 in $\mathcal{BP}(P', E_0)$ produced by BEBOP is also a path of program P. The question then is: does p_0 represent a feasible execution path of P? That is, is there some execution of program P that follows the path p_0? If so, we have found a real error in P. If not, path p_0 is a spurious error path.

The NEWTON tool takes a C program and a (potential) error path as an input. It then uses verification condition generation (VCGen) to determine if the path is feasible. The answer may be "yes" or "no". [4]

If the answer is "yes", then an error path has been found, and we report it to the user. If the answer is "no" then NEWTON uses a new algorithm to identify a small set of predicates that "explain" why the path is infeasible.

In the running example, NEWTON detects that the path p is infeasible, and returns a single predicate $(nPackets = npacketsOld)$ as the explanation for the infeasibility. This is because the predicate $(nPackets = nPacketsOld)$ is required to be both true and false by path p. The assignment of `nPacketsOld` to `nPackets` makes the predicate true, and the loop test requires it to be false at the end of the **do-while** loop for the loop to iterate, as specified by the path p.

[3] We had to write stubs for the procedures `SmartDevFreeBlock`, and kernel procedures `IoCompleteRequest`, `KeAcquireSpinLock`, and `KeReleaseSpinLock`. The analysis determines that these procedures cannot affect state variables so the calls to them are removed.

[4] Since underlying decision procedures in the theorem prover and our axiomatization of C are incomplete, "don't know" is also a possible answer. In practice, the theorem provers we use [27,13,4] have been able to give a "yes" or "no" answer in every example we have seen so far.

2.7 The Second Boolean Program

In the second iteration of the process, the predicate $(nPackets = nPacketsOld)$ is added to the set of predicates E_0 to result in a new set of predicates E_1. Figure 4(b) shows the boolean program $\mathcal{BP}(P', E_1)$ that C2BP produces. This program has one additional boolean variable (`b`) that represents the predicate $(nPackets = nPacketsOld)$. The assignment statement `nPackets = nPacketsOld;` makes this condition true, so in the boolean program the assignment `b := T;` represents this assignment. Using a theorem prover, C2BP determines that if the predicate is true before the statement `nPackets++`, then it is false afterwards. This is captured by the assignment statement in the boolean program `b := choose(F,b);`. The `choose` function is defined as follows:

```
bool choose(pos, neg)
begin
     if (pos) then return T; elsif (neg) then return F;
  elsif (*)   then return T; else             return F; fi
end
```

The `pos` parameter represents positive information about a predicate while the `neg` parameter represents negative information about a predicate. The `choose` function is never called with both parameters evaluating to true. If both parameters are false then there is not enough information to determine whether the predicate is definitely true or definitely false, so F or T is returned, nondeterministically.

Applying BEBOP to the new boolean program shows that the label `SLIC_ERROR` is not reachable. In performing its fixpoint computation, BEBOP discovers that the following loop invariant holds at the end of the **do-while** loop:

$$(state = Locked \wedge nPackets = nPacketsOld)$$
$$\vee\ (state = Unlocked \wedge nPackets \neq nPacketsOld)$$

That is, either the lock is held and the loop will terminate (and thus the lock needs to be released after the loop), or the lock is free and the loop will iterate. The combination of the predicate abstraction of C2BP and the fixpoint computation of BEBOP has found this loop-invariant over the predicates in E_1. This loop-invariant is strong enough to show that the label `SLIC_ERROR` is not reachable.

3 C2BP: A Predicate Abstractor for C

C2BP takes a C program P and a set $E = \{e_1, e_2, \ldots, e_n\}$ of predicates on the variables of P, and automatically constructs a boolean program $\mathcal{BP}(P, E)$[1, 2]. The set of predicates E are pure C boolean expressions with no function calls. The boolean program $\mathcal{BP}(P, E)$ contains n boolean variables $V = \{b_1, b_2, \ldots, b_n\}$, where each boolean variable b_i represents a predicate e_i. Each

variable in V has a *three-valued* domain: **false**, **true**, and $*$.[5] The program $\mathcal{BP}(P, E)$ is a *sound* abstraction of P because every possible execution trace t of P has a corresponding execution trace t' in B. Furthermore, $\mathcal{BP}(P, E_0)$ is a precise abstraction of P with respect to the set of predicates E_0, in a sense stated and shown elsewhere [2]. Since $\mathcal{BP}(P, E)$ is an abstraction of P, it is guaranteed that an invariant I discovered (by BEBOP) in $\mathcal{BP}(P, E)$, as boolean combinations of the b_i variables, is also an invariant in the C code, where each b_i is replaced by its corresponding predicate e_i. C2BP determines, for every statement s in P and every predicate $e_i \in E$, how the execution of s can affect the truth value of e_i. This is captured in the boolean program by a statement s_B that conservatively updates each b_i to reflect the change. C2BP computes s_B by (1) first computing the weakest precondition of e_i, and its negation with respect to s, and (2) strengthening the weakest precondition in terms of predicates from E, using a theorem prover.

We highlight the technical issues in building a tool like C2BP:

- **Pointers**: We use an alias analysis of the C program to determine whether or not an update through a pointer dereference can potentially affect an expression. This greatly increases the precision of the C2BP tool. Without alias analysis, we would have to make very conservative assumptions about aliasing, which would lead to invalidating many predicates.
- **Procedure calls**: Since boolean programs support procedure calls, we are able to abstract procedures modularly. The abstraction process for procedure calls is challenging, particularly in the presence of pointers. After a call, the caller must conservatively update local state that may have been modified by the callee. We provide a sound and precise approach to abstracting procedure calls that takes such side-effects into account.
- **Precision-efficiency tradeoff**. C2BP uses a theorem prover to strengthen weakest pre-conditions in terms of the given predicate set E. Doing this strengthening precisely requires $O(2^{|E|})$ calls to the theorem prover. We have explored a number of optimization techniques to reduce the number of calls made to the theorem prover. Some of these techniques result in an equivalent boolean program, while others trade off precision for computation speed. We currently use two automatic theorem provers Simplify [27,13] and Vampyre [4]. We are also investigating using other decision procedures, such as those embodied in the Omega test [30] and PVS [28].

Complexity. The runtime of C2BP is dominated by calls to the theorem prover. In the worst-case, the number of calls made to the theorem prover for computing $\mathcal{BP}(P, E)$ is linear in the size of P and exponential in the size of E. We can compute sound but imprecise abstractions by considering only k-tuples of predicates in the strengthening step. In all examples we have seen so far we find that we lose no precision for $k = 3$. Thus, in practice the complexity is cubic in the size of E.

[5] The use of the third value $*$, is encoded using control-nondeterminism as shown in the `choose` function of Section 2. That is, "$*$" is equivalent to "`choose(F,F)`".

4 BEBOP: A Model Checker for Boolean Programs

The BEBOP tool [3] computes the set of reachable states for each statement of a boolean program using an interprocedural dataflow analysis algorithm in the spirit of Sharir/Pnueli and Reps/Horwitz/Sagiv [34,31]. A state of a boolean program at a statement s is simply a valuation to the boolean variables that are in scope at statement s (in other words, a bit vector, with one bit for each variable in scope). The set of reachable states (or invariant) of a boolean program at s is thus a set of bit vectors (equivalently, a boolean function over the set of variables in scope at s).

BEBOP differs from typical implementations of dataflow algorithms in two crucial ways. First, it computes over sets of bit vectors at each statement rather than single bit vectors. This is necessary to capture correlations between variables. Second, it uses binary decision diagrams [5] (BDDs) to implicitly represent the set of reachable states of a program, as well as the transfer functions for each statement in a boolean program. BEBOP also differs from previous model checking algorithms for finite state machines, in that it does not inline procedure calls, and exploits locality of variable scopes for better scaling. Unlike most model checkers for finite state machines, BEBOP handles recursive and mutually recursive procedures. BEBOP uses an explicit control-flow graph representation, as in a compiler, rather than encoding the control-flow with BDDs, as done in most symbolic model checkers. It computes a fixpoint by iterating over the set of facts associated with each statement, which are represented with BDDs.

Complexity. The worst-case complexity of BEBOP is linear in the size of the program control-flow graph, and exponential in the maximum number of boolean variables in scope at any program point. We have implemented a number of optimizations to reduce the number of variables needed in support of BDDs. For example, we use live variable analysis to find program points where a variable becomes dead and then eliminate the variable from the BDD representation. We also use a global MOD/REF analysis of the boolean program in order to perform similar variable eliminations.

5 NEWTON: A Predicate Discoverer

NEWTON takes a C program P and an error path p as inputs. For the purposes of describing NEWTON, we can identify the path p as a sequence of assignments and assume statements (every conditional is translated into an assume statement). The `assume` statement is the dual of `assert`: `assume`(e) never fails. Executions on which e does not hold at the point of the `assume` are simply ignored [15].

The internal state of NEWTON has three components: (1) *store*, which is a mapping from locations to symbolic expressions, (2) *conditions*, which is a set of predicates, and (3) a *history* which is a set of past associations between locations and symbolic expressions. The high-level description of NEWTON is given in Figure 5. The functions LEval and REval evaluate the l-value and r-value of a given expression respectively. NEWTON maintains the dependencies

```
Input: A sequence of statements p = s_1, s_2, ..., s_m.
store := null map;
history := null set;
conditions := null set;
/* start of Phase 1 */
for i = 1 to m do {
  switch( s_i ) {
    "e_1 := e_2" :
      let lval = LEval (store, e_1) and
      let rval = REval(store, e_2) in {
          if(store[lval] is defined)
              history := history ∪ {(lval, store[lval])}
          store[lval] := rval
      }
    "assume(e)" :
      let rval = REval(store, e) in {
          conditions := conditions ∪ {rval}
          if(conditions is inconsistent){
              /*Phase 2 */
              Minimize size of conditions while maintaining inconsistency
              /*Phase 3 */
              predicates := all dependencies of conditions using store and
              history
              Say "Path p is infeasible"
              return(predicates)
          }
      }
  }
}
Say "Path p is feasible"
return
```

Fig. 5. The high-level algorithm used by NEWTON.

of each symbolic expression on the elements in *store*, to be used in Phase 3. It also uses symbolic constants for unknown values. We illustrate these using an example. Consider a path with the following four statements:

```
s1:   nPacketsOld = nPackets;
s2:   req = devExt->WLHV;
s3:   assume(!req);
s4:   assume(nPackets != nPacketsOld);
```

This path is a projection of the error path from BEBOP in Section 2.

Figure 6 shows four states of NEWTON, one after processing each statement in the path. The assignment `nPacketsOld = nPackets` is processed by first introducing a symbolic constant α for the variable `nPackets`, and then assigning it to `nPacketsOld`. The assignment `req = devExt->WLHV` is processed by first introducing a symbolic constant β for the value of `devExt`, then introducing a

```
s1:   nPacketsOld = nPackets;
s2:   req = devExt->WLHV;
s3:   assume(!req);
s4:   assume(nPackets != nPacketsOld);
```

loc.	value	deps.	conds.	deps.
1. *nPackets*:	α	()		
2. *nPacketsOld*:	α	(1)		

after `s1`

loc.	value	deps.	conds.	deps.
1. *nPackets*:	α	()		
2. *nPacketsOld*:	α	(1)		
3. *devExt*:	β	()		
4. $\beta \rightarrow$ *WLHV*:	γ	(3)		
5. *req*:	γ	(3,4)		

after `s2`

loc.	value	deps.	conds.	deps.
1. *nPackets*:	α	()	!(γ)	(5)
2. *nPacketsOld*:	α	(1)		
3. *devExt*:	β	()		
4. $\beta \rightarrow$ *WLHV*:	γ	(3)		
5. *req*:	γ	(3,4)		

after `s3`

loc.	value	deps.	conds	deps.
1. *nPackets*:	α	()	!(γ)	(5)
2. *nPacketsOld*:	α	(1)	(α!= α)	(1,2)
3. *devExt*:	β	()		
4. $\beta \rightarrow$ *WLHV*:	γ	(3)		
5. *req*:	γ	(3,4)		

after `s4`

Fig. 6. A path of four statements and four tables showing the state of NEWTON after simulating each of the four statements.

second symbolic constant γ for the value of β`->WLHV`, and finally assigning γ to `req`. The conditional `assume(!req)` is processed by adding the predicate `!(`γ`)` to the condition-set. The dependency list for this predicate is `(5)` since its evaluation depended on entry `5` in the store. Finally, the conditional `assume(nPackets != nPacketsOld)` is processed by adding the (inconsistent) predicate `(`α `!=` α`)` to the condition-set, with a dependency list `(1,2)`. At this point, the theorem prover determines that the condition-set is inconsistent, and NEWTON proceeds to Phase 2.

Phase 2 removes the predicate `!(`γ`)` from the condition store, since the remaining predicate `(`α`!=` α`)` is inconsistent by itself. Phase 3 traverses store entries `1` and `2` from the dependency list. A post processing step then determines that the symbolic constant α can be unified with the variable `nPackets`, and NEWTON produces two predicates: ($nPacketsOld = nPackets$) and ($nPacketsOld \neq nPackets$). Since one is a negation of the other, only one of the two predicates needs to be added in order for the path to be ruled out in the boolean program. Though no symbolic constants are present in the final set of predicates in our example, there are other examples where the final list of predicates have symbolic constants. C2BP is able to handle predicates with symbolic constants. We do not discuss these details here due to want of space. The *history* is used when a location is overwritten with a new value. Since no location was written more than once in our example, we did not see the use of *history*. NEWTON also handles error paths where each element of the path is also provided with values to the boolean variables from BEBOP, and checks for their consistency with the concrete states along the path.

```
VOID
SerialDebugLogEntry(IN ULONG Mask, IN ULONG Sig,
  IN ULONG_PTR Info1, IN ULONG_PTR Info2, IN ULONG_PTR Info3)
{
    KIRQL irql;
    irql = KeGetCurrentIrql();
    if (irql < DISPATCH_LEVEL) {
        KeAcquireSpinLock(&LogSpinLock, &irql);
    } else {
        KeAcquireSpinLockAtDpcLevel(&LogSpinLock);
    }
    // other code (deleted)
    if (irql < DISPATCH_LEVEL) {
        KeReleaseSpinLock(&LogSpinLock, irql);
    } else {
        KeReleaseSpinLockFromDpcLevel(&LogSpinLock);
    }
    return;
}
```

Fig. 7. Code snippet from serial-port driver.

Complexity. The number of theorem-prover calls made by NEWTON on a path p is $O(\mid p \mid)$, where $\mid p \mid$ is the length of the path.

6 NT Device Drivers: Case Study

This section describes our experience applying the SLAM toolkit to check properties of Windows NT device drivers. We checked two kinds of properties: (1) Locking-unlocking sequences for locks should conform to allowable sequences (2) Dispatch functions should either complete a request, or make a request pending for later processing. In either case, a particular sequence of Windows NT specific actions should be taken.

The two properties have different characteristics from a property-checking perspective.

- The first property depends mainly on the program's control flow. We checked this property for a particular lock (called the "Cancel" spin lock) on three kernel mode drivers in the Windows NT device driver tool kit. We found two situations where spurious error paths led our process to iterate. With its inter-procedural analysis and detection of variable correlations, the SLAM tools were able to eliminate all the spurious error paths with at most one iteration of the process. In all the drivers, we started with 2 predicates from the property specification and added at most one predicate to rule out spurious error paths.

– The second property is data-dependent, requiring the tracking of value flow and alias relationships. We checked this property on a serial port device driver. It took 4 iterations through the SLAM tools and a total of 33 predicates to validate the property.

The drivers we analyzed were on the order of a thousand lines of C code each. In each of the drivers we checked for the first property, the SLAM tools ran in under a minute on an 800MHz Pentium PC with 512MB RAM. For the second property on the serial driver, the total run time for all the SLAM tools was about three minutes to complete all the four iterations.

We should note that we did not expect to find errors in these device drivers, as they are supposed to be examplars for others to use. Thus, the fact that the SLAM tools did not find errors in these program is not too surprising. We will report on the defect detection capabilities of the tools in a future paper.

6.1 Property 1

We checked for correct lock acquisition/release sequences on three kernel mode drivers: MCA-bus, serial-port and parallel-port. The SLAM tools validated the property on MCA-bus and parallel-port drivers without iteration. However, interprocedural analysis was required for checking the property, as calls to the acquire and release routines were spread across multiple procedures in the drivers. Furthermore, in the serial-port driver, the SLAM tools found one false error path in the first iteration, which resulted in the addition of a single predicate. Then the property was validated in the second iteration. The code-snippet that required the addition of the predicate is shown in Figure 7. The snippet shows that the code has a dependence on the interrupt request level variable (`irql`) that must be tracked in order to eliminate the false error paths. At most three predicates were required to check this property.

6.2 Property 2

A dispatch routine to a Windows NT device driver is a routine that the I/O manager calls when it wants the driver to perform a specific operation (e.g, read, write etc.) The dispatch routine is "registered" by the driver when it is initialized. A dispatch routine has the following prototype:

```
NTSTATUS DispatchX(IN PDEVICE_OBJECT DeviceObject, IN PIRP irp)
```

The first parameter is a pointer to a so called "device object" that represents the device, and the second parameter is a pointer to a so called "I/O request packet", or "IRP" that contains information about the current request.

All dispatch routines must either process the IRP immediately (call this *option A*), or queue the IRP for processing later (call this *option B*). Every IRP must be processed under one of these two options. If the driver chooses option A, then it has to do the following actions in sequence:

1. Set the `irp->IoS.status` to some return code other than `STATUS_PENDING` (such as `STATUS_SUCCESS`, `STATUS_CANCELLED` etc.)

```
state {
  enum {Init, Complete, Pending}
   s = Init;
  PIRP gIrp = 0;
}

Dispatch.entry {
  s, gIrp = Init, $2;
}

IoCompleteRequest.call{
  if(gIrp == $1) {
    if( s != Init) abort;
    else s = Complete;
  }
}
```

```
IoMarkIrpPending.call{
  if(gIrp == $1) {
    if( s != Init) abort;
    else s = Pending;
  }
}

Dispatch.exit{
  if (s == Complete) {
    if( $return == STATUS_PENDING)
      abort;
  } else if (s == Pending) {
    if( $return != STATUS_PENDING)
      abort;
  }
}
```

Fig. 8. SLIC specification for completing an IRP or marking it as pending.

2. Call `IoCompleteRequest(irp)`
3. Return the same status code as in step 1.

If the driver chooses option B, then it has do the following actions in sequence:

1. Set `irp->IoS.status` to `STATUS_PENDING`
2. Call the kernel function `IoMarkIrpPending(irp)`
3. Queue the IRP into the driver's internal queue using the kernel function `IoStartPacket(irp)`
4. Return `STATUS_PENDING`

Note that this is a partial specification for a dispatch routine —just one of several properties that the dispatch routine must obey. Figure 8 shows a SLIC specification for this property. The variable `$1` is used by SLIC to denote the first parameter of the function, and the variable `$return` is used to denote the return value. Note that we first store the IRP at the entry of the dispatch routine in a state variable `gIrp` and then later check if the calls to `IoCompleteRequest` and `IoMarkIrpPending` refer to the same IRP.

Checking the instrumented driver. We started the first iteration of SLAM with 7 predicates from the SLIC specification. It took 4 iterations of the SLAM tools and a total of 33 predicates to discover the right abstraction to validate this property. The discovered predicates kept track of the value of the flow of the `irp` pointer and the status value through several levels of function calls. We found one bug in the fourth iteration, which turned out to be due to an error in the SLIC specification. After fixing it, the property passed.

7 Related Work

SLAM seeks a sweet spot between tools based on verification condition generation(VCGen) [14,25,26,6] that operate directly on the concrete semantics, and model checking or data flow-analysis based tools [8,21,18,16] that operate on abstractions of the program. We use VCGen-based approach on finite (potentially interprocedural) paths of the program, and use the knowledge gained to construct abstract models of the program. NEWTON uses VCGen on the concrete program, but as it operates on a single finite interprocedural path at a time, it does not require loop-invariants, or pre-conditions and post-conditions for procedures. C2BP also reasons about the statements of the C program using decision procedures, but does so only locally, one statement at a time. Global analysis is done only on the boolean program abstractions, using the model checker BEBOP. Thus, our hope is to scale without losing precision, as long as the property of interest allows us to do so, by inherently requiring a small abstraction for its validation or invalidation.

SLAM generalizes Engler et al.'s approach [18] in three ways: (1) it is sound (modulo the assumptions about memory safety); (2) it permits interprocedural analysis; (3) it avoids spurious examples through iterative refinement (in some of the code Engler et al. report on, their techniques generated three times as many spurious error paths as true error paths, a miss rate of 75%.[6]) In fact, with a suitable definition of abstraction, and choice of initial predicates, the first iteration of the SLAM process is equivalent to performing Engler et al.'s approach interprocedurally.

Constructing abstract models of programs has been studied in several contexts. Abstractions constructed by [18] and [22] are based on specifying transitions in the abstract system using a pattern language, or as a table of rules. Automatic abstraction support has been built into the Bandera tool set [17]. They require the user to provides finite domain abstractions of data types. Predicate abstraction, as implemented in C2BP is more general, and can capture relationships between variables. The predicate abstraction in SLAM was inspired by the work of Graf and Saidi [20] in the model checking community. Efforts have been made to integrate predicate abstraction with theorem proving and model checking [32]. Though our use of predicate abstraction is related to these efforts, our goal is to analyze software written in common programming languages.

The SLAM tools C2BP and BEBOP can be used in combination to find loop-invariants expressible as boolean functions over a given set of predicates. The loop-invariant is computed by the model checker BEBOP using a fixpoint computation on the abstraction computed by C2BP. Prior work for generating loop invariants has used symbolic execution on the concrete semantics, augmented with widening heuristics [35,36]. The Houdini tool guesses a candidate set of annotations (invariants,preconditions, postconditions) and uses the ESC/Java checker to refute inconsistent annotations until convergence [19].

[6] Jon Pincus, who led the development of an industrial-strength error detection tool for C called PREfix [6], observes that users of PREfix will tolerate a false alarm rate in the range 25-50% depending on the application [29].

Boolean programs can be viewed as abstract interpretations of the underlying program [9]. The connections between model checking, dataflow analysis and abstract interpretation have been explored before [33] [10]. The model checker BEBOP is based on earlier work on interprocedural dataflow analysis [34,31]. Automatic iterative refinement based on error paths first appeared in [23], and more recently in [7]. Both efforts deal with finite state systems.

An alternative approach to static validation of safety properties, is to provide a rich type system that allows users to encode both safety properties and program annotations as types, and reduce property validation to type checking [12].

8 Conclusions

We have presented a fully automated methodology to validate/invalidate temporal safety properties of software interfaces. Our process does not require user supplied annotations, or user supplied abstractions. When our process converges, it always give a definitive "yes" or "no" answer.

The ideas behind the SLAM tools are novel. The use of boolean programs to represent program abstractions is new. To the best of our knowledge, C2BP is the first automatic predicate abstraction tool to handle a full-scale programming language, and perform a sound and precise abstraction. BEBOP is the first model checker to handle procedure calls using an interprocedural dataflow analysis algorithm, augmented with representation tricks from the symbolic model checking community. NEWTON uses a path simulation algorithm in a novel way, to generate predicates for refinement.

We have demonstrated that the SLAM tools converge in a few iterations on device drivers from the Microsoft DDK.

The SLAM toolkit has a number of limitations that we plan to address. The logical model of memory is a limitation, since it is not the actual model used by C programs. We plan to investigate using a physical model of memory. We also are exploring what theoretical guarantees we can give about the termination of our iterative refinement. Finally, we plan to evolve the SLAM tools by applying them to more code bases, both inside and outside Microsoft.

Acknowledgements. We thank Rupak Majumdar and Todd Millstein for their hard work in making the C2BP tool come to life. Thanks to Andreas Podelski for helping us describe the C2BP tool in terms of abstract interpretation. Thanks also to the members of the Software Productivity Tools research group at Microsoft Research for many enlightening discussions on program analysis, programming languages and device drivers, as well as their numerous contributions to the SLAM toolkit.

References

1. T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani. Automatic predicate abstraction of C programs. In *PLDI 01: Programming Language Design and Implementation(to appear)*. ACM, 2001.

2. T. Ball, A. Podelski, and S. K. Rajamani. Boolean and cartesian abstractions for model checking C programs. In *TACAS 01: Tools and Algorithms for Construction and Analysis of Systems(to appear)*. Springer-Verlag, 2001.
3. T. Ball and S. K. Rajamani. Bebop: A symbolic model checker for Boolean programs. In *SPIN 00: SPIN Workshop*, LNCS 1885, pages 113–130. Springer-Verlag, 2000.
4. D. Blei and et al. Vampyre: A proof generating theorem prover — http://www.eecs.berkeley.edu/ rupak/vampyre.
5. R. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, 1986.
6. W. R. Bush, J. D. Pincus, and D. J. Sielaff. A static analyzer for finding dynamic programming errors. *Software-Practice and Experience*, 30(7):775–802, June 2000.
7. E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *CAV 00: Computer Aided Verification*, LNCS 1855, pages 154–169. Springer-Verlag, 2000.
8. J. Corbett, M. Dwyer, J. Hatcliff, C. Pasareanu, Robby, S. Laubach, and H. Zheng. Bandera: Extracting finite-state models from Java source code. In *ICSE 2000: International Conference on Software Engineering*, pages 439–448. ACM, 2000.
9. P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for the static analysis of programs by construction or approximation of fixpoints. In *POPL 77: Principles of Programming Languages*, pages 238–252. ACM, 1977.
10. P. Cousot and R. Cousot. Temporal abstract interpretation. In *POPL 00: Principles of Programming Languages*, pages 12–25. ACM, 2000.
11. M. Das. Unification-based pointer analysis with directional assignments. In *PLDI 00: Programming Language Design and Implementation*, pages 35–46. ACM, 2000.
12. R. DeLine and M. Fähndrich. Enforcing high-level protocols in low-level software. In *PLDI 01: Programming Language Design and Implementation(to appear)*. ACM, 2001.
13. D. Detlefs, G. Nelson, and J. Saxe. Simplify theorem prover – http://research.compaq.com/src/esc/simplify.html.
14. D. L. Detlefs, K. R. M. Leino, G. Nelson, and J. B. Saxe. Extended static checking. Technical Report Research Report 159, Compaq Systems Research Center, December 1998.
15. E. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.
16. M. Dwyer and L. Clarke. Data flow analysis for verifying properties of concurrent programs. In *FSE 94: Foundations of Software Engineering*, pages 62–75. ACM, 1994.
17. M. Dwyer, J. Hatcliff, R. Joehanes, S. Laubach, C. Pasareanu, Robby, W. Visser, and H. Zheng. Tool-supported program abstraction for finite-state verification. In *ICSE 01: Software Engineering (to appear)*, 2001.
18. D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *OSDI 00: Operating System Design and Implementation*. Usenix Association, 2000.
19. C. Flanagan, R. Joshi, and K. R. M. Leino. Annotation inference for modular checkers. *Information Processing Letters (to appear)*, 2001.
20. S. Graf and H. Saidi. Construction of abstract state graphs with PVS. In *CAV 97: Computer Aided Verification*, LNCS 1254, pages 72–83. Springer-Verlag, 1997.
21. G. Holzmann. The Spin model checker. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.
22. G. Holzmann. Logic verification of ANSI-C code with Spin. In *SPIN 00: SPIN Workshop*, LNCS 1885, pages 131–147. Springer-Verlag, 2000.

23. R. Kurshan. *Computer-aided Verification of Coordinating Processes*. Princeton University Press, 1994.
24. L. Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, SE-3(2):125–143, 1977.
25. K. R. M. Leino and G. Nelson. An extended static checker for Modula-3. In *CC 98: Compiler Construction*, LNCS 1383, pages 302–305. Springer-Verlag, 1998.
26. G. Necula. Proof carrying code. In *POPL 97: Principles of Programming Languages*, pages 106–119. ACM, 1997.
27. G. Nelson. Techniques for program verification. Technical Report CSL81-10, Xerox Palo Alto Research Center, 1981.
28. S. Owre, S. Rajan, J. Rushby, N. Shankar, and M. Srivas. PVS: Combining specification, proof checking, and model checking. In *CAV 96: Computer-Aided Verification*, LNCS 1102, pages 411–414. Springer-Verlag, 1996.
29. J. Pincus. personal communication, October 2000.
30. W. Pugh. A practical algorithm for exact array dependence analysis. *Communications of the ACM*, 35(8):102–114, August 1992.
31. T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *POPL 95: Principles of Programming Languages*, pages 49–61. ACM, 1995.
32. H. Saidi and N. Shankar. Abstract and model check while you prove. In *CAV 99: Computer-aided Verification*, LNCS 1633, pages 443–454. Springer-Verlag, 1999.
33. D. Schmidt. Data flow analysis is model checking of abstract interpretation. In *POPL 98: Principles of Programming Languages*, pages 38–48. ACM, 1998.
34. M. Sharir and A. Pnueli. Two approaches to interprocedural data dalow analysis. In *Program Flow Analysis: Theory and Applications*, pages 189–233. Prentice-Hall, 1981.
35. N. Suzuki and K. Ishihata. Implementation of an array bound checker. In *POPL 77: Principles of Programming Languages*, pages 132–143. ACM, 1977.
36. Z. Xu, B. P. Miller, and T. Reps. Safety checking of machine code. In *PLDI 00: Programming Language Design and Implementation*, pages 70–82. ACM, 2000.

Verification Experiments on the MASCARA Protocol*

Guoping Jia and Susanne Graf

VERIMAG** – Centre Equation – 2, avenue de Vignate – F-38610 Gières – France
Guoping.Jia@imag.fr, Susanne.Graf@imag.fr
http://www-verimag.imag.fr/PEOPLE/graf

Abstract. In this paper, we describe a case study on the verification of a real industrial protocol for wireless ATM, called MASCARA. Several tools have been used: SDL has been chosen as the specification language and the commercial tool *Object*GEODE has been used for creating and maintaining SDL descriptions. The IF tool-set has been used for generation, minimization and comparison of system models and verification of expected properties. All specification and verification tools are connected via the IF language, which has been defined as an intermediate representation for timed asynchronous systems as well as an open validation environment. Due to the complexity of the protocol, static analysis techniques, such as live variable analysis and program slicing, were the key to the success of this case study. The results obtained give some hints concerning a methodology for the formal verification of real systems.

1 Introduction

Model checking [CE81,QS82] is by now a well established method for verifying properties of reactive systems. The main reason for its success is the fact that it works fully automatically and it is able to reveal subtle defects in the design of complex concurrent systems. Different academic tools have been developed for supporting these techniques. Not only hardware but also telecommunication industries are beginning to incorporate them as a component of their development process. For example, the commercial SDL design tools *Object*GEODE [Ver96] and TAU [TA99] provide some verification facilities going beyond interactive or random simulation. A major challenge in model checking is dealing with the well-known *state explosion* problem. This limits its large scale use in practice. In order to limit this problem, different techniques have been developed, such as on-the-fly model-checking [JJ89a,Hol90], symbolic model-checking [BCM+90,McM93], partial order reduction [God96,GKPP94], abstraction [CGL94,LGS+95], compositional minimization [GS90,KM00] and more recently static analysis reduction

* This work was supported by the VIRES project (Verifying Industrial REactive Systems, Esprit Long Term Research Project No.23498)

** VERIMAG is a research laboratory associated with CNRS, Université Joseph Fourier and Institut Nationale Polytechnique de Grenoble

M.B. Dwyer (Ed.): SPIN 2001, LNCS 2057, pp. 123–142, 2001.

[BFG00a]. We try to show here that the right combination of these techniques allows to tackle the verification of large software systems.

We present a detailed report on the verification of an industrial protocol, called MASCARA (Mobile Access Scheme based on Contention And Reservation for ATM) [DPea98]. The protocol is a specific medium access control (MAC) protocol, which has been designed for wireless ATM communication and has been developed within the WAND (Wireless ATM Network Demonstrator) consortium [WAN96]. SDL [IT94] has been chosen as the specification language by the designers and we have used the commercial tool *Object*GEODE for maintaining the SDL description of the protocol. The IF tool-set [BFG+99b,BFG+99a] has been used for analysis of the protocol. All specification and verification tools of this tool-set have been connected via the IF language, which is an intermediate representation for timed asynchronous systems in an open validation environment. In order to deal with the complexity of the protocol, all the available reduction techniques of the tool have been used in combination to reduce the state graphs of the protocol. The results obtained give some hints on a methodology for the formal verification of large systems.

The paper is organized as follows. Section 2 gives an overview on the MASCARA protocol and a brief description of the IF language and its validation environment. In Section 3, we describe in detail the verification of the protocol. The results are compared and discussed.

2 The Context

2.1 The IF Language and Validation Environment

The IF language. [BFG+99a,BFG+99b] has been defined as an intermediate representation for timed asynchronous systems. In IF, a system is described by a set of parallel processes communicating either asynchronously through a set of buffers, or by rendez-vous via a set of gates. Buffers can have various queuing policies (fifo, bag, etc.), can be bounded or unbounded, reliable or lossy, and delayable or with zero delay. Processes are timed automata with urgencies [BST97], extended with discrete variables. Process transitions are triggered by inputs and/or guards and perform variable assignments, and clock settings and signal outputs. An urgency out of *eager, delayable, lazy* is associated with each transition, defining its priority with respect to time progress. This makes IF very flexible and appropriate on one hand as underlying semantics for high level specification formalisms such as SDL[BKM+01] or ESTELLE used in commercial design tools and on the other hand as an intermediate representation for tool interconnections as it is powerful enough to express the concepts of the languages used in the main verification tools of the domain, such as LOTOS [BB88] and PROMELA [Hol91].

The IF validation environment. provides a complete validation tool chain allowing to transform high level SDL descriptions through the intermediate representation into the input formats of several verification tools (see Figure 1) [BFG+00b]:

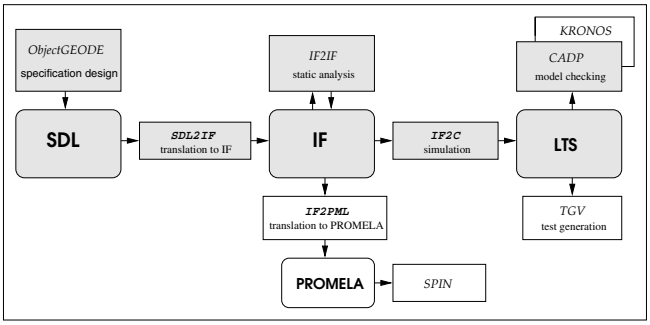


Fig. 1. The IF validation environment

– **The specification level tools.** IF does not itself provide facilities to edit SDL specifications. For this it relies on the commercial tool-set *Object*GEODE developed by TELELOGIC and supporting SDL, MSC and TTCN. It includes graphical editors and compilers for each of these formalisms and provides step-by-step and random-walk simulation, but also a model-checking facility using observers to help the user to debug an SDL specification. In the verification experiment, *Object*GEODE has mainly been used for maintaining the SDL description of the MASCARA protocol, for visualizing MSCs generated from diagnostic sequences generated by verification tools and for replaying these MSC on the SDL specification to ease error identification.

– **The intermediate level tools.** Based on an API provided by *Object*GEODE, a translator, SDL2IF, generates operationally equivalent IF specification from SDL [BFG+99b]. At the IF level, a tool called IF2IF implements various static analysis techniques such as *dead variable analysis*(LIVE), *slicing*(SLICE), *constant propagation* and *clock reduction*. IF2IF transforms with a small cost a given IF description into a semantically equivalent one (with respect to a set of properties) with a reduced model, where a typical factor of reduction observed in many examples is between 1 and 3 orders of magnitude. Any back-end tool connected to IF can profit from these reductions. For example, the SPIN tool [Hol91] which has been connected via a translator IF2PML [BDHS00].

– **The verification tools.** CADP [FGK+96] has been designed as a tool-set for the verification of LOTOS specifications. Its model generation and analysis tools, such as ALDEBARAN and EVALUATOR, have been adapted for IF-specifications and can be used for generation, minimization, comparison of state graphs and verification of properties specified as alternation-free μ-calculus formulas either on-the-fly or on an already generated model. Diagnostic sequences are computed and can be translated into MSCs to be displayed in a user friendly manner. Other tools, such as KRONOS [Yov97] for the verification of real-time properties described in TCTL and TGV [FJJV97] for automatic test generation, can work directly on IF specifications.

IF is presently used in several projects of different nature concerned with verification, test case generation, performance analysis and UML modeling; several significant case studies in the domain of verification have already been carried out using IF [BFG+98,BML00]

2.2 The MASCARA Protocol

The MASCARA (Mobile Access Scheme based on Contention And Reservation for ATM) protocol [DPea98] is a special medium access control (MAC) protocol designed for wireless ATM (Asynchronous Transfer Mode) communication and developed by the WAND (Wireless ATM Network Demonstrator) consortium [WAN96]. A wireless ATM network extends transparently services to *mobile terminals* (MT) via a number of geographically distributed *access points* (AP). The task of the MASCARA protocol is to mediate between APs and MTs via a wireless link. The protocol has a layered structure, where we consider only the highest layer, the MASCARA control layer.

The Purpose of the MASCARA Control Layer (MCL). is to ensure that mobile terminals can initially establish and then maintain an association with an access point with good quality and minimal interruptions as long as the ATM connection is valid. It carries out a periodical monitoring of the current radio link quality (gathering the information about radio link qualities of its neighboring APs to hand-over to in the case of deterioration of the current association link quality) and switching from one AP to another in the hand-over procedure. Since several MTs communicate through a single AP with the ATM core network, MCL is different on the AP and the MT side.

MCL consists of two main parts: *dynamic control* and *static control*. We describe in detail the dynamic control part, which we have concentrated on in this case study.

Dynamic Control (DC). The main task of the dynamic control is to set up and release *associations* and virtual *connections*. It consists of the following entities: *dynamic generic agent*, *association agent* and MAC *virtual channel agent*.

The dynamic generic agent is the top-level entity of the dynamic control and its task is *association management*. It dispatches upper layer requests concerning existing associations and connections to the right association agent, manages MAC-layer addresses for the associations, and informs the upper layer about lost associations.

The association agent of an MT and its peer entity in the (to be) associated AP are responsible for managing and supervising a single *association*. Each association can carry a variable number of connections via virtual channels. The task of the association agent peers is to create the virtual channel agents, map the addresses between the ATM-layer and the MAC-layer connection identifiers and forward requests. Since each MT is at any time associated with at most one AP, there exists one association agent per MT. While, whereas each AP has one association agent for every associated MT.

An MVC agent of an MT and its peer entity in the AP manage a single *connection* on a virtual channel. Beside address mapping from the ATM-layer to the MAC-layer connection identifiers, the MVC agents are in charge of *resource allocation* for the connection they manage over the wireless channels.

3 Verification of the MASCARA Protocol

The overall description of the MASCARA protocol which we got is *300* pages of SDL textual code. This makes it impossible to envisage to push the "**verify**" button on the protocol as a whole. That fact that the descriptions of the lower layers of the protocol were very incomplete made that the protocol was even not simulatable as a whole. We concentrate on the verification of the MASCARA control layer, for which the SDL description could be made reasonably complete. Here we report on the verification of the dynamic control. Another verification experiment has been carried out on static control [BDHS00]. In this section, we first present the experiment system and the assumptions and simplifications we made. Then, we list some of the correctness properties to be verified and describe in detail the approaches to perform the verification. Finally, we present the verification results and discuss some problems encountered.

3.1 The Experimental System

Architecture. Dynamic control has an interface to the MASCARA management layer (called upper layer here), and exchanges control signals with lower layer entities of the MASCARA protocol. For verification, we abstract all lower MASCARA layers to a pair of buffered communication channels. We have considered these channels as non-lossy, but with a transmission delay that is possibly longer than expected by the timers, so that these channels represent all relevant behaviours of the lower levels of the protocol. The architecture of the SDL model used for verification can be seen in Figure 2 and consists of the following parts:

- AP Dynamic Control has itself a hierarchical structure: all signals from outside are received by the *Generic Dynamic Control* process, and either treated directly or forwarded to the relevant *Association Agent*, which on turn, either treats the signal itself or forwards it to the relevant *Virtual Channel (*MVC*) Agent*.
- MT Dynamic Control has the same architecture as AP Dynamic Control, but the implemented protocols are of an asymmetric nature: association is always initiated by MT, whereas channel opening can be initiated on both sides, in such a way that MT will always "win" in case of concurrent opening of the same channel on both sides.
- An "*Environment*" process which consists of abstract versions of the upper layer and of the other MASCARA Control entities, in particular steady state control.

We assume that only one MT can be associated and only one connection can be opened, i.e., only one pair of association agents (AAA/MAA) and one pair of

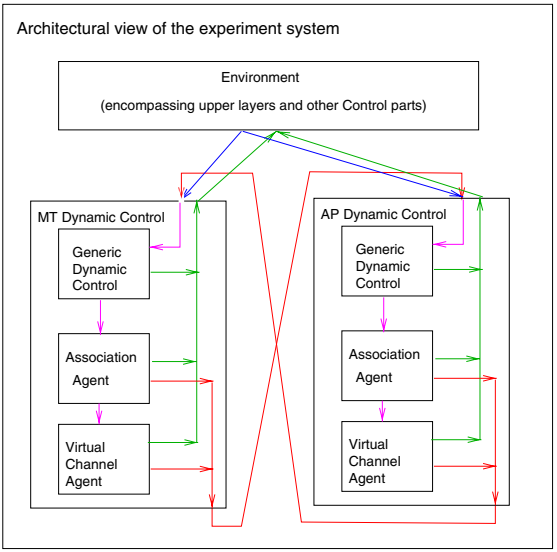


Fig. 2. Architectural view of the experimental system

MVC agents (AMA/MMA) are considered, which is sufficient for the correctness properties we have verified.

Environment. Verification should be carried out under a general environment with realistic restrictions. As we have not obtained information on the MASCARA upper layer, we considered initially an open system with an unconstrained upper layer, which would allow us to obtain the "*most general*" verification results. But communication is via unbounded channels as in SDL, leads to infinitely growing channel contents and thus an infinite state model in case that the environment sends requests too often, which is typically the case in "reactive" systems always ready to to treat requests from the environment.

Object GEODE offers a facility for avoiding this problem which however leads to a drastic restriction of the possible behaviours, allowing a new request from the environment only when no system transition is enabled any more, that is, the previous request has been completely treated. This is not an adequate approach for the MASCARA Protocol which is precisely designed for dealing with several, even conflicting, requests in parallel.

The approach we have chosen to provide a more restricted, but still realistic environment consists in restricting the number of requests it can make per time unit. We assume that within one time unit, no more than "N" requests can be sent by the environment. Considering system transitions as *eager* – that means that time can only progress when no system transition is enabled – this provides an environment suitable for our purpose. The system has never to deal with more than "N" requests "simultaneously" which leads, in the MASCARA protocol, to

bounded channel contents. The success of the method depends on the use of a realistic bound. We use $N = 4$.

Simultaneously means here in the "same time slice", meaning that the environment can send a new request even if there are previous ones still to be treated. Using Rendez-vous or bounded and blocking channels like in LOTOS or PROMELA leads to similar restrictions as the environment is blocked as soon as all buffers are full. Our solution makes it however easier to exactly qualify the applied restriction.

The role of time. This protocol makes use timeouts, essentially to avoid indefinite waiting for a response that will never come. This type of systems can usually be verified as an untimed system, where the occurrence of a timeout is treated as a non-deterministic event. As we use time as a convenient means to slow down the environment, we cannot adopt exactly this solution. We consider

- the transmission delay through the channels between AP and MT as 1 (the value is arbitrary) and all other transmission delays as zero and
- the maximal waiting time for response as 2
- all system transitions as "eager" so that the system is forced to progress until standstill before time can pass.

This has as consequence that responses and corresponding timeouts occur in the same "time slice" and thus can occur in any order, and still time can be used to slow down the environment as it can send in each time slice only a limited number of requests.

3.2 Properties

As it is often the case, the designers of the system did not provide us with requirements that the system must satisfy. Therefore, we considered generic properties such as deadlocks. And for each request from the environment (such as association, deassociation, connection opening, connection release,...) a set of "*response properties*", where in a first time we verified very weak properties, such as "potential response", and the more the system became debugged, the more we strengthened them. As an example, we show the strongest response property considered for "association request".

Association Establishment. This property refers to association establishment between an MT and an AP which is obtained by a four-way handshake protocol initiated by MT. The signals between the system and the upper layer are:

- the request for association establishment ("*a_req*") from the upper layer to MT.
- the response to association request ("*a_cnf_(rc)*")[1] by MT to the upper layer.

[1] A return code is attached with the response signal which indicates positive (e.g. *success*) or negative (e.g. *failed*, *already associated* or *no response*) result of the corresponding association request.

– the indication of association establishment ("*a_ind*") AP to the upper layer.

the response property we considered for checking the correctness of the association establishment, is the following one:

> "Any *association request* received by the MT is *eventually* followed by an *association confirmation* with either a negative or a positive result. In the second case, an *association indication* has already been or will *eventually* be emitted by the AP."

Expression of Properties. We have used several formalisms for the expression of properties:

1. We expressed some properties by sets of temporal logic formulas. Temporal logics such as Computational Tree Logic (CTL)[CGP99] and Linear Temporal Logic (LTL)[MP92] are widely used. The model-checker EVALUATOR is based on the use of the alternation-free μ-calculus[Koz83] which is more expressive, but more difficult to use. However, there exist macros going beyond the modalities of CTL or LTL, such that most properties can be expressed conveniently at least by an expert. For verification (see Section 3.3), we decomposed the above correctness criterion into simpler properties (Table 2). The following formula expresses the requirement *Req1* of this table, where "*all*", "*inev*" and "*TRUE*" are macros which should be intuitive enough.

$$all[a\_req](inev \ < a\_cnf\_* > \ TRUE)$$

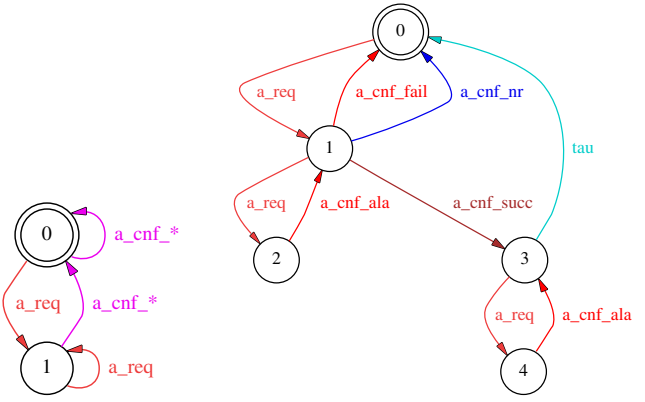


Fig. 3. *Req1* and a refinement of *Req1* expressed as LTS

2. CTL and LTL are not powerful enough to express all useful properties. Moreover, they are difficult to use by a non expert for the expression of complex state and event sequencing properties often required for protocols, especially

if they are not covered by the set of macros. Finite automata are another formalism for the expression of properties which is powerful yet relatively easy to use. In our tools we use labeled transition systems (LTS), where the transition labels are a set of observable events, which are in our case a subset of the signal input and output events, possibly with their parameters. Figure 3 shows on the left side an LTS expressing *Req1* of Table 2 and on the right side a refinement of it. This refined version, which corresponds more to the really expected behaviour, is easier to express by an LTS than by a temporal logic formula.

3. We have tried to apply so called "*visual verification*" which gives often very satisfactory results. It consists in "*computing*" the *exact property* of the system with respect to a set of observable events: the (completely constructed) state-graph is minimized with respect to an appropriate equivalence relation, such as observational[Mil80] or safety[BFG$^+$91] equivalence. In this particular protocol, the obtained minimal state-graphs were most of the time still too complicated to be inspected visually. The reason for this seems to be the counting effect introduced by limiting the number of simultaneous requests (to 4).
4. We have used Message Sequence Charts (MSC). Nevertheless, we have not written them ourselves in order to express negations of safety properties, but our tools generated them from diagnostic sequences, showing the violation of some property formulated in another formalism. In Figure 5 an example of such a "diagnostic MSC" can be found.

3.3 Verification Methodology

Given a property φ and a system $\mathcal{S}$, described as a parallel composition of a number of subsystems $\mathcal{S}_i$, the goal of model checking is to verify whether $\mathcal{S}_1 \parallel \cdots \parallel \mathcal{S}_n \models \varphi$ holds.

It is well-known that the size of the state graph grows exponentially with the number of components $\mathcal{S}_i$ (and also the number of variables). For the MASCARA protocol, even if only a single association and a single connection is considered, the state space is too large to be analyzed without application of any reduction technique.

We combined the use of all reduction techniques available in our tools and applied them in the order depicted in Figure 4. We explain the results observed using the different techniques in the following paragraphs.

Static Analysis Reduction Techniques: *static analysis techniques* is applied at the program level to reduce the number of states and transitions of the model associated with the specification and thus make model checking feasible:

1. Dead variable analysis transforms an IF specification into an equivalent one by adding systematic reset statements of "*dead*" variables. A variable is "*dead*" at some control point if its value is not used before it is assigned again. This transformation preserves all event-oriented properties of the original specification while the global state space and consequently the exploration time are reduced.

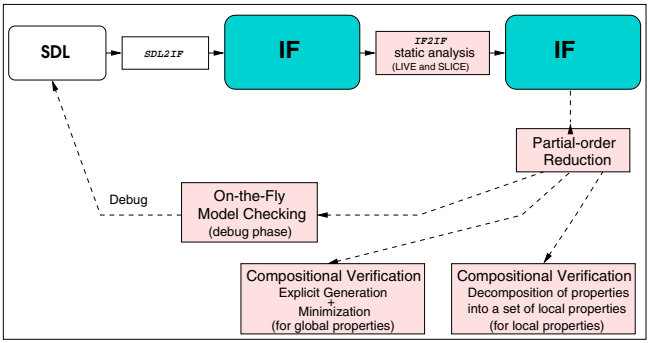


Fig. 4. Overview of the verification methodology

2. Program slicing automatically extracts portions of a program relevant for a specific property, called *slicing criterion*: for example, a set of variables or signals at program points of interest for the given correctness property. The reduction is obtained by eliminating program parts which do not influence the slicing criterion. There is a different model for each property (or a set of properties). This transformation preserves safety properties depending only on the slicing criterion, while it results smaller IF-program.

These reductions are performed on the structural program level description of the system, before model generation and their complexity is completely unproblematic. They can, and should, be always applied, independently of the verification method or tool being used later on.

Partial Order Reduction. [God96,GKPP94] is a method which consists in verifying properties of a concurrent system without exploring all interleavings of concurrent executions of independent transitions. It preserves properties on orderings of events as long as they are considered dependent. A simple version of this method has been implemented in the IF tool-set. In order to use partial order reduction jointly with compositional techniques, we need to consider all signal input from the "*environment*" as dependent, no matter the property to be verified. It is well-known that partial order reduction allows a significant reduction of the size of the state-graph, and should therefore always be applied during generation or traversal of the model.

Atomicity Reduction. A well-known reduction method consists in considering *sequences of internal steps* as "model steps". This is correct as long as each sequence contains at most one read or write action of a global variable and at most one modification of observable variables.

SDL greatly facilitates the use of this method: all transitions start by reading a signal from the process's message buffer, and then execute a set of actions consisting in local assignments and signal outputs to buffers of other processes.

All properties we consider are expressed in terms of signal inputs and outputs and in MASCARA there are never two observable outputs sent within a single SDL transition; thus we can consider complete SDL transitions as atomic steps. The reduction obtained by atomicity reduction is in general tremendous. As it is applied by default in the step-function of the IF tool-set, we can however not show its effect.

On-the-Fly Model Checking: In "*on-the-fly*" model checking [JJ89b,FM91, Hol91] verification is performed during a possibly exhaustive traversal of the model. This method is very helpful, in particular at the first stage of verification for the debugging of the initial specification, as it exposes fast many errors and omissions, even of systems which cannot be verified completely. It should be noted, however, that only for very simple safety properties, the underlying model of the on-the-fly analysis has the same size as the system model alone. For more complex properties, on-the-fly verification explores a model which can be significantly bigger than the system model alone, and some of the advantage of not storing transitions vanishes. In the particular case of the MASCARA protocol, there was no verification of a non-trivial property that we could do on-the-fly, but for which we could not generate the state graph.

Compositional Verification: We have applied two different types of *compositional verification*. The first one is based on property decomposition [Kur94], and the second one is based on compositional generation of a state graph minimized with respect to a behavioral equivalence [GS90]. For the application of both methods, we split the system into smaller functional parts, namely, AP dynamic control and MT dynamic control.

1. a) decompose a global property of a system into a set of *local* properties of the considered subsystems;
   b) verify each local property on the corresponding subsystem — using a particular environment representing an abstraction of the remaining subsystems.

   All safety properties which hold on a small configuration hold also on the complete system. This method is very convenient to use, under the condition that the global properties can be decomposed and that it is sufficient to include a very abstract view of the neglected subsystems. For example, *Req1 Req2*, *Req3a* and *Req3b* of Table 2 below, are such local properties of MT which can be verified on a state graph, generated by abstracting the AP part almost as the Chaos process (making sure however that it is not too active). see Table 1.

2. a) generate the state graph of each subsystem (AP and MT) separately, considering the same weak abstraction of the other subsystem as in the first method, and reduce it with respect to weak bisimulation using the ALDEBARAN tool;
   b) apply parallel composition on the reduced models (as communication between AP and MT is via a pair of buffers, these buffers are the global

variables of the system and need to be considered as such for parallel composition [KM00])

c) verify the global correctness properties on the generated global model.

This method preserves all safety properties on observable events. *Req3* of Table 2 below, for example, can be evaluated on the state graph DC1 mentioned in Table 1.

The first method allows to work with smaller models than the second one as no abstraction of the global state graph need to be constructed. Unfortunately, it can sometimes be quite difficult to find an appropriate decomposition of properties and to provide a useful abstraction of the "*non-considered parts*" of the system. For example, the decomposition of *Req3* into *Req3a* and *Req3b* is only correct if the communication channels between AP and MT can be considered as reliable. Notice that the second method does not necessarily rely on a correct environment abstraction [GS90], but this variant is not implemented in our tool for systems communicating through asynchronous buffers.

3.4 Complexity

Table 1 gives an overview of a subset of the state graphs we have generated using different reduction techniques and allows to compare their sizes.

Execution Time. With respect to execution time, the following observation can be made: execution times are roughly proportional to the size of the generated graphs, which means that the different reduction methods do not introduce any significant overhead. For static analysis reduction this result is not surprising. For partial order reduction it is the case because we use a simple static dependency relation. Table 1 shows only minimization results for relatively small graphs (AP4a and MT4a) so that minimization time is small anyway. Nevertheless, it can be seen that minimization for observational equivalence is more expensive than for safety equivalence, as the computation of the transitive closure transition relation "$\tau * a\tau*$" is required (where τ represents a non-observable and a an observable transition).

State Graph Size. We can see that application of dead variable analysis (LIVE) and partial order reduction (PO) alone reduces the original state graph by 1 to 2 orders of magnitude. The combination of LIVE and PO gives more than the combined reduction of each of these techniques applied in isolation. Notice that for AP, the efficiency of PO and LIVE are about the same, whereas for MT, LIVE performs better; in other case studies we also had the situation where PO performed better, so that one can say that with a negligible cost, applying them together, most of time, one obtains good reduction (here 3 orders of magnitude).

Obviously the reduction obtained by the application of slicing depends heavily on the considered properties, and it is impossible to make general statements. For the considered system, we get similar reductions when slicing according to the 4 main sub-protocols (1 to 2 additional orders of magnitude), where connection opening is slightly more complicated than the others (it

Table 1. State graphs of the dynamic control

Entities	Generation Approaches		N. of Tran.	N. of Stat.	Time (hh:mm:ss)
AP Dynamic Control (with an abstract version of MT part)	AP1: Direct generation		30 689 244	7 308 400	3:27:35
	AP2: Partial-order reduction alone		1 807 095	895 249	37:26
	AP3*a*: LIVE optimization alone		1 536 699	351 202	12:22
	AP4*a*: LIVE + Partial-order		52 983	28 069	1:53
	AP4*b*: Minimization of AP4*a* (strong bisimulation)		38 952	20 312	18
	AP4*c*: Minimization of AP4*a* (hide/rename + observ. equivalence)		18 521	11 265	1:04
	AP4*d*: Minimization of AP4*a* (hide/rename + safety equivalence)		12 210	3 502	22
	AP5: SLICE + LIVE (w.r.t. properties)	AP5*a*: Association Establishment	12 664	4 556	9
		AP5*b*: Deassociation	10 739	3 940	5
		AP5*c*: Conn. Open	36 603	11 616	28
		AP5*d*: Conn. Release	15 958	5 636	9
	AP6: SLICE + LIVE + Partial-order (w.r.t. properties)	AP6*a*: Association Establishment	2 885	1 630	3
		AP6*b*: Deassociation	2 972	1 703	3
		AP6*c*: Conn. Open	8 099	4 583	9
		AP6*d*: Conn. Release	4 262	2 476	5
MT Dynamic Control (with an abstract version of AP part)	MT1: Direct generation		12 811 961	4 388 765	2:51:58
	MT2: Partial-order reduction alone		7 433 859	3 099 928	1:30:57
	MT3*a*: LIVE optimization alone		325 312	63 628	1:03
	MT4*a*: LIVE + Partial-order		20 913	6 580	7
	MT4*b*: Minimization of MT4*a* (strong bisimulation)		12 241	3 927	8
	MT4*c*: Minimization of MT4*a* (hide/rename + observ. equivalence)		1 804	1 148	13
	MT4*d*: Minimization of MT4*a* (hide/rename + safety equivalence)		1 380	499	3
	MT5: SLICE + LIVE (w.r.t. properties)	MT5*a*: Association Establishment	16 854	4 018	5
		MT5*b*: Deassociation	16 522	3 967	5
		MT5*c*: Conn. Open	15 823	3 754	4
		MT5*d*: Conn. Release	16 135	3 820	4
	MT6: SLICE + LIVE + Partial-order (w.r.t. properties)	MT6*a*: Association Establishment	2 845	977	3
		MT6*b*: Deassociation	2 411	985	2
		MT6*c*: Conn. Open	2 801	969	2
		MT6*d*: Conn. Release	2 162	855	2
Dynamic Control (AP + MT)	DC1: Composition of models (AP4*c* × MT4*c*)		1 142 215	218 130	–

involves more signal exchanges than the others), and thus we get a bit less reduction.

It was impossible to generate the state graph of the global system as a whole, thus we started to consider AP and MT in isolation (see first two parts of the Table 1). Finally, we were able to compositionally generate a reduced model of the global system using compositional generation, under the condition to use both LIVE and partial order reduction for the generation of the subsystems.

3.5 Verification Results

We did a large number of verification experiments with increasing complexity. Initially, many deadlocks were found which were mainly due to the interaction the different "*request/response*" sub-protocols. It should also be mentioned that the feature of implicit (that is silent) discarding unexpected signals in SDL made the analysis of the generated diagnostic sequences of deadlock traces more difficult. Using a different translation from SDL to IF, this problem has disappeared.

As we had obtained almost no information on the environment of the MASCARA layer, we considered initially the case where the request from the environment can be sent in any order. This lead to a number of error traces which we considered to be "probably because of too loose assumptions on the environment" and we added corresponding restrictions for subsequent verifications. The state graphs mentioned in Table 1 have been obtained using the most restrictive environment.

Table 2. Properties and Verification results for Association Establishment

Property: **Association Establishment**		
Req1.	After reception of an association request by MT, an association confirmation with either *positive* or *negative* return value will be eventually sent by MT to the upper layer.	**TRUE**
Req2.	After reception of an association request by MT, there exists an execution path where an association confirmation with *positive* return value is sent by MT to the upper layer.	**TRUE**
Req3.	Whenever the association confirmation with *positive* return value is sent by MT, an association indication will be or has already been sent by AP to the upper layer.	**FALSE**
Req3a.	Whenever AP receives the third handshake message (MPDU_MT_AP_addr_received), it will eventually send the fourth handshake message (MPDU_AP_MT_Assoc_ack) to MT, and an indication of the association (a_ind) to the upper layer.	**TRUE**
Req3b.	Whenever MT receives the fourth handshake message (MPDU_AP_MT_Assoc_ack) from AP, it will eventually send a successful confirmation (a_cnf_succ) to the upper layer.	**FALSE**

Table 2 lists the verification results for the properties concerning association establishment. We performed the verification in an incremental manner, starting with weak properties, weaker than those mentioned in the table, for each subsystem (AP and MT), and finally ending up with the strong properties of the table which we verified either on the relevant subsystem or on the reduced version of the global system.

The local property *Req3b* (as well as the global property *Req3*) does not hold. A diagnostic was produced by the tool EVALUATOR. Figure 5 gives an MSC scenario of such a trace. Its analysis shows that this violation occurs when a deassociation request is sent before the association has been confirmed. In case of a deassociation request, a negative association confirmation is sent to the environment independently of the success of the handshake protocol with AP; and this is a correct behaviour. Thus, *Req3a* should be replaced by the following weaker requirement (which holds):

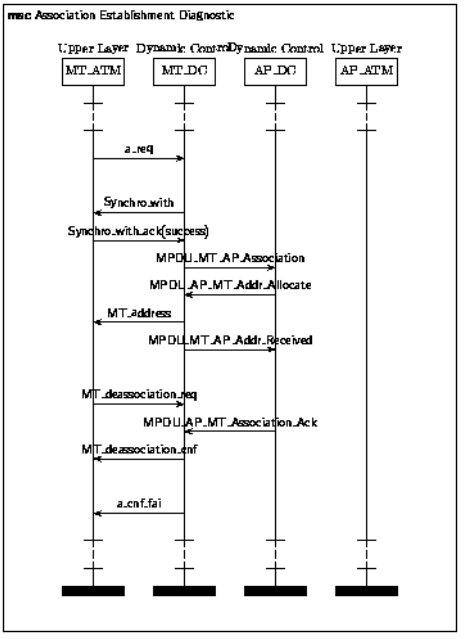


Fig. 5. MSC showing the non satisfaction of *Req3b*

When MT receives an MPDU_AP_MT_Assoc_ack from AP, it will eventually a successful confirmation (a_cnf_succ) to the upper layer, except if it has already received or meanwhile receives a deassociation request from the upper layer.

4 Conclusion and Perspectives

In this paper, we have presented an experiment report on the verification of an industrial ATM protocol. The aim of this verification was much more the experimentation and improvement of the verification facilities of the IF toolset and the analyze of the difficulties occurring with such a big protocol and their solution rather than the actual verification of the protocol. We believe that we have at least partially succeeded. The main difficulties that we encountered together with some responses or some to-do list are the following ones, in the order in which they appeared:

1. **How to extract a subsystem from a large SDL description?** (such as a single layer from a whole protocol stack) The fact that we did not enlarge verification from the initially considered sub-system to a larger subsystem is partially due to the fact that it is such a time consuming hand-work to cut large SDL descriptions in pieces, or to recombine separately modified subsystems at a later stage. Hopefully, the integration of on one hand UML oriented

features in SDL design tools, which allow to trace interface changes, and on the other hand static analysis methods allowing to "cut out" subsystems in a clean manner, will eliminate this problem.

2. **How to get reasonable abstraction of the neglected parts of the system?** In a protocol stack,
   - the lower layers can often easily be abstracted by communication channels with particular properties: they may be reliable or lossy, ordered or unordered, delayable or immediate.
   - The upper layers can often be considered as unconstraint or only weak order constraints are necessary, a part from the fact that the flooding of the system with infinitely fast sequences of requests must be avoided in order to make state space exploration tractable. Fortunately, for the verification of safety properties it is always reasonable to limit the number of requests of the upper layer per time unit.

   For other subsystems which are not related in such a simple way with the subsystem under study, slicing is one way to get a simplified description, but in our example this was not sufficient. General abstraction techniques as those implemented in the InVest tool [BLO98] will be connected with IF.
3. **How to get requirements?** A part from "deadlock-freedom", there exist only few "generic" properties which must hold for any system. Communication protocols, can often be viewed as reactive systems which must react "in an appropriate way" to a number of "requests" from the environment; moreover this is true for every sub-layer of the protocol. In an ideal case, the designer should be able to help in expressing the appropriate "response properties". In absence of help we used the following strategy, which turned out to work quite well: we started with very weak response properties and strengthened them as long as they hold. When we found a violated property, we analyzed a number of diagnostic sequences (or graphs) produced by the model-checkers in order to find out if it was likely to be
   - a real problem of the system,
   - a too loose environment
   - or a too strong property

   and we made corresponding changes.
4. **How to express properties?** For simple properties, temporal logic is very convenient, but for more complicated ones, for example taking into account a number of exceptions under which the desired response need not to occur, temporal logic is cumbersome. Labeled transition systems allow to express more complicated properties on message exchanges. MSCs express the existence of sequences with certain characteristics, and are therefore more appropriate for the representation of "bad scenarios" or "never claims". We believe that a generalization of MSCs, such as *Live Sequence Charts* [DH99], could be very useful.
5. **How to analyze negative verification results?** In the case where the violation of a property is due to (a set of) execution sequences, we translated these sequences into message sequence charts which we than replayed on the SDL specification using a facility of *Object*GEODE. This was convenient as

long the sequences were not too long. Using an abstraction criterion makes the sequences shorter, but introduces non-determinism, which is another source of complexity, and hides sometimes away the problematic point. We found that only once we had a good understanding of the protocol (which should be the case for the designers), we could detect subtle errors with a reasonable effort.

6. **How far can we go with system verification?** We hope that we have demonstrated that using an appropriate strategy, we can verify automatically reasonably small subsystems or components of large systems. For the verification of global properties of large systems, automatic verification using state space enumeration, combined with whatever reduction strategies, seems out of reach. To go a step further we applied two approaches — out of the large number of compositional approaches proposed in the literature — which could be applied using the facilities of our tool-set.
   - compositional construction of a state-graph reduced with respect to some equivalence relation. Our results show that this method will probably not scale, unless the interfaces between the subsystems are very concise, or we can provide automatic means for getting precise enough abstractions of large parts of a system.
   - compositional verification based on property decomposition. We believe that this method can scale, even if there are at least two problems which can make its application difficult:
     - the decomposition of a global system with a large number of subsystems can be very hard (we applied it to a system with only two subsystem and a very simple communication structure)
     - as for the first method an abstraction of the environment of the considered subsystem is needed, even if one can hope that less concise abstractions are enough.

Acknowledgments. We would like to thank Marius Bozga, Lucian Ghirvu and Laurent Mounier for fruitful discussion and kind help during the verification experiment.

References

[BB88] T. Bolognesi and E. Brinksma. Introduction to the ISO Specification Language LOTOS. *Computer Networks and ISDN Systems*, 14(1):25–29, January 1988.

[BCM+90] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *Proceedings of the International Conference on Logic in Computer Science'90*, 1990.

[BDHS00] D. Bosnacki, D. Dams, L. Holenderski, and N. Sidorova. Model Checking SDL with SPIN. In *Proceedings of TACAS'2000, Berlin, Germany*, LNCS, 2000.

[BFG+91] A. Bouajjani, J.Cl. Fernandez, S. Graf, C. Rodriguez, and J. Sifakis. Safety for Branching Time Semantics. In *18th ICALP*, number 510 in LNCS, July 1991.

[BFG+98] M. Bozga, J.-C. Fernandez, L. Ghirvu, C. Jard, T. Jéron, A. Kerbrat, P. Morel, and L. Mounier. Verification and Test Generation for the SSCOP Protocol. *SCP*, 1998.

[BFG+99a] M. Bozga, J.-C. Fernandez, L. Ghirvu, S. Graf, J.P. Krimm, and L. Mounier. IF: An Intermediate Representation and Validation Environment for Timed Asynchronous Systems. In *Proceedings of FM'99, Toulouse, France*, LNCS, 1999.

[BFG+99b] M. Bozga, J.-C. Fernandez, L. Ghirvu, S. Graf, J.P. Krimm, L. Mounier, and J. Sifakis. IF: An Intermediate Representation for SDL and its Applications. In *Proceedings of SDL-FORUM'99, Montreal, Canada*, June 1999.

[BFG00a] M. Bozga, J-C. Fernandez, and L. Ghirvu. Using Static Analysis to Improve Automatic Test Generation. In *Proceedings of TACAS'2000, Berlin, Germany*, LNCS, 2000.

[BFG+00b] M. Bozga, J.Cl. Fernandez, L. Ghirvu, S. Graf, J.P. Krimm, and L. Mounier. IF: A Validation Environment for Timed Asynchronous Systems. In E.A. Emerson and A.P. Sistla, editors, *Proceedings of CAV'00 (Chicago, USA)*, LNCS. Springer, July 2000.

[BKM+01] Marius Bozga, Susanne Graf Alain Kerbrat, Laurent Mounier, Iulian Ober, and Daniel Vincent. Timed extensions for sdl. In *Proceedings, SDL Forum 2001*, June 2001.

[BLO98] S. Bensalem, Y. Lakhnech, and S. Owre. Computing Abstractions of Infinite State Systems Compositionally and Automatically. In A. Hu and M. Vardi, editors, *Proceedings of CAV'98 (Vancouver, Canada)*, volume 1427 of *LNCS*, pages 319–331, June 1998.

[BML00] M. Bozga, L. Mounier, and D. Lesens. Model Checking Ariane-5 Flight Program. Technical report, Verimag, Grenoble, France, December 2000.

[BST97] S. Bornot, J. Sifakis, and S. Tripakis. Modeling Urgency in Timed Systems. In *International Symposium: Compositionality - The Significant Difference (Holstein, Germany)*, volume 1536 of *LNCS*. Springer, September 1997.

[CE81] E.M. Clarke and E.A. Emerson. Design and Synthesis of Synchronisation Skeletons using Branching Time Temporal Logic. In *Proceedings of IBM Workshop on Logics of Programs*, volume 131 of *LNCS*, 1981.

[CGL94] E.M. Clarke, O. Grumberg, and D.E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, September 1994.

[CGP99] E.M. Clarke, O. Grumberg, and D.A. Peled. *Model Checking*. The MIT Press, 1999.

[DH99] W. Damm and D. Harel. Lscs: Breathing live into message sequence charts. In *FMOODS'99 IFIP TC6/WG6.1 Third International Conference on Formal Methods for Open Object-Based Distributed Systems*, 1999.

[DPea98] I. Dravapoulos, N. Pronios, and S. Denazis et al. *The Magic WAND, Deliverable 3D5, Wireless ATM MAC, Final Report*, August 1998.

[ES94] E. A. Emerson and A. P. Sistla. Utilizing symmetrie when model checking under fairness assumptions. submitted to POPL95, 1994.

[FGK+96] J.-C. Fernandez, H. Garavel, A. Kerbrat, R. Mateescu, L. Mounier, and M. Sighireanu. CADP: A Protocol Validation and Verification Toolbox. In *Proceedings of CAV'96, New Brunswick, USA*, volume 1102 of *LNCS*, July 1996.

[FJJV97] J.C. Fernandez, C. Jard, T. Jéron, and C. Viho. An Experiment in Automatic Generation of Test Suites for Protocols with Verification Technology. *Science of Computer Programming*, 29, 1997.

[FM91] J.-C. Fernandez and L. Mounier. "On the fly" Verification of Behavioural Equivalences and Preorders. In *Workshop on Computer-Aided Verification, Aalborg University, Denmark*, LNCS, July 1991.

[GKPP94] R. Gerth, R. Kuiper, D. Peled, and W. Penczek. A partial order approach to branching time logic model checking, April 1994.

[God96] P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State Explosion Problem*, volume 1032 of *LNCS*. Springer, January 1996.

[GS90] S. Graf and B. Steffen. Compositional Minimisation of Finite State Processes. In *Proceedings of CAV'90, Rutgers*, number 531 in LNCS, 1990.

[Hol90] G.J. Holzmann. Algorithms for automated protocol validation. *AT&T technical Journal*, 60(1):32–44, January 1990.

[Hol91] G.J. Holzmann. *Design and Validation of Computer Protocols*. Prentice Hall Software Series, 1991.

[IT94] ITU-T. *Recommendation Z.100. Specification and Description Language (SDL)*. 1994.

[JJ89a] C. Jard and T. Jeron. On-Line Model Checking for Finite Linear Temporal Logic Specifications. In *Workshop on Automatic Verification Methods for Finite State Systems, Grenoble*. LNCS 407, Springer Verlag, 1989.

[JJ89b] C. Jard and T. Jeron. On-Line Model-Checking for Finite Linear Temporal Logic Specifications. In J. Sifakis, editor, *Proceedings of the 1st Workshop on Automatic Verification Methods for Finite State Systems (Grenoble, France)*, volume 407 of *LNCS*, pages 189–196. Springer Verlag, Juin 1989.

[KM00] J.P. Krimm and L. Mounier. Compositional State Space Generation with Partial Order Reductions for Asynchronous Communicating Systems. In *Proceedings of TACAS'2000, Berlin, Germany*, LNCS, 2000.

[Koz83] D. Kozen. Results on the Propositional μ-Calculus. In *Theoretical Computer Science*. North-Holland, 1983.

[Kur94] R.P. Kurshan. *Computer-Aided Verification of Coordinating Processes: The Automata-Theoretic Approach*. Princeton University Press, Princeton, New Jersey, 1994.

[LGS+95] C. Loiseaux, S. Graf, J. Sifakis, A. Bouajjani, and S. Bensalem. Property preserving abstractions for the verification of concurrent systems. *Formal Methods in System Design*, 6, Issue 1:11–44, jan 1995. first published in CAV'92, LNCS 663.

[McM93] K.L. McMillan. *Symbolic Model Checking: an Approach to the State Explosion Problem*. Kluwer Academic Publisher, 1993.

[Mil80] R. Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer-Verlag, 1980.

[MP92] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer-Verlag, 1992.

[QS82] J.P. Queille and J. Sifakis. Specification and Verification of Concurrent Programs in CESAR. In *International Symposium on Programming*, volume 137 of *LNCS*, 1982.

[TA99] Sweden Telelogic A.B., Malmö. *Telelogic TAU SDL suite Reference Manuals*. `http://www.telelogic.se`, 1999.

[Ver96] Verilog. Object*GEODE SDL Simulator - Reference Manual*. http://www.verilogusa.com/products/geode.htm, 1996.

[WAN96] WAND. *Magic WAND - Wireless ATM Network Demonstrator*. http://www.tik.ee.ethz.ch/wand, 1996.

[Yov97] S. Yovine. KRONOS: A Verification Tool for Real-Time Systems. *Software Tools for Technology Transfer*, 1(1+2):123–133, December 1997.

Using SPIN for Feature Interaction Analysis – A Case Study

Muffy Calder and Alice Miller

Department of Computing Science
University of Glasgow
Glasgow, Scotland.

Abstract. We show how SPIN is applied to analyse the behaviour of a real software artifact – feature interaction in telecommunications services. We demonstrate how minimal abstraction techniques can greatly reduce the cost of model-checking, and how analysis can be performed automatically using scripts.

Keywords: telecommunications services; Promela/SPIN; communicating processes; distributed systems; formal modelling; analysis and reasoning techniques; feature interaction

1 Introduction

In software development a *feature* is a component of additional functionality – additional to the main body of code. Typically, features are added incrementally, often by different developers. A consequence of adding features in this way is *feature interaction*, when one feature affects, or modifies, the behaviour of another feature. Although in many cases feature interaction is quite acceptable, even desirable, in other cases interactions lead to unpredictable and undesirable results. The problem is well known within the telecommunications (telecomms) services domain (for example, see [2]), though it exhibits in many other domains such as email and electronic point of sales.

Techniques to deal with feature interactions can be characterised as design time or run time, interaction detection and/or resolution. Here, we concentrate on detection at design time, resolution will be achieved through re-design.

When there is a proliferation of features, as in telecomms services, then automated detection techniques are essential. In this paper, we investigate the feasibility of using Promela and SPIN [16].

Our approach involves considering a given service (and features) at two different levels of abstraction: communicating finite state automata and temporal logic formulae, represented by Promela specifications, labelled transition systems and Büchi automata. We make contributions at several levels, including

- a low level call service model in Promela that permits truly independent call control processes with asynchronous communication, asymmetric call control and a facility for adding features in a structured way,

M.B. Dwyer (Ed.): SPIN 2001, LNCS 2057, pp. 143–162, 2001.

- state-space reduction techniques for Promela which result in tractable state-spaces, thus overcoming classic state-explosion problems,
- interaction analysis of a basic call service with six features, involving four users with full functionality. There are two types of analysis, static and dynamic, the latter is completely automated, making extensive use of Perl scripts to generate the model-checking runs.

Related work is discussed below. The overall approach to interaction detection, and the role of SPIN, is given in section 2; section 3 contains an introduction to feature interaction analysis. Sections 4, 5 and 6 give an overview of the finite state automata, temporal properties, the Promela implementation of the basic call service, and state-space reduction techniques. Sections 7, 8and 9 contain an overview of the features and their implementations. The interaction analysis is described in sections 10 and 11 and in section 12 we discuss how the Promela models and SPIN model-checking runs required for the analysis are automated. We conclude in section 13.

1.1 Related Work

Model-checking for feature interaction analysis has been investigated using SMV [22], Caesar [23], COSPAN [10] and SPIN [18]. In the last, the Promela model is extracted mechanically from call processing software code; no details of the model are given and so it is difficult to compare results. In the SMV work, the authors are restricted to two subscribers of the service with full functionality (plus two users with half functionality), due to state-explosion problems. For similar reasons, call control is not independent. Nevertheless, we regard this as a benchmark paper and aim at least to demonstrate a similar set of properties within our context. In the COSPAN work, features and the basic service are described only at an abstract level by temporal descriptions. State-explosion is avoided, but interactions arising from implementation detail, such as race conditions, cannot be detected. Our layered approach permits detection of interactions at this level, building on earlier work by the first author (of this paper), [23], using process algebra. This too suffered from limitations of state-explosion and the lack of (explicit) asynchronous communication; these limitations motivated the current investigation using Promela and SPIN. Initial attempts to model the basic call service using Promela and SPIN are described in [4].

2 Approach

Our approach has two phases; in the first phase we consider only the basic call service, as depicted in figure 1(a). The aim of the first phase is to develop the right level of abstraction of the basic service and to ensure that we have effective reasoning techniques, before proceeding to add features.

Our starting point is the top and left hand side of figure 1(a): the automata and properties. Neither need be *complete* specifications; this is a virtue of the

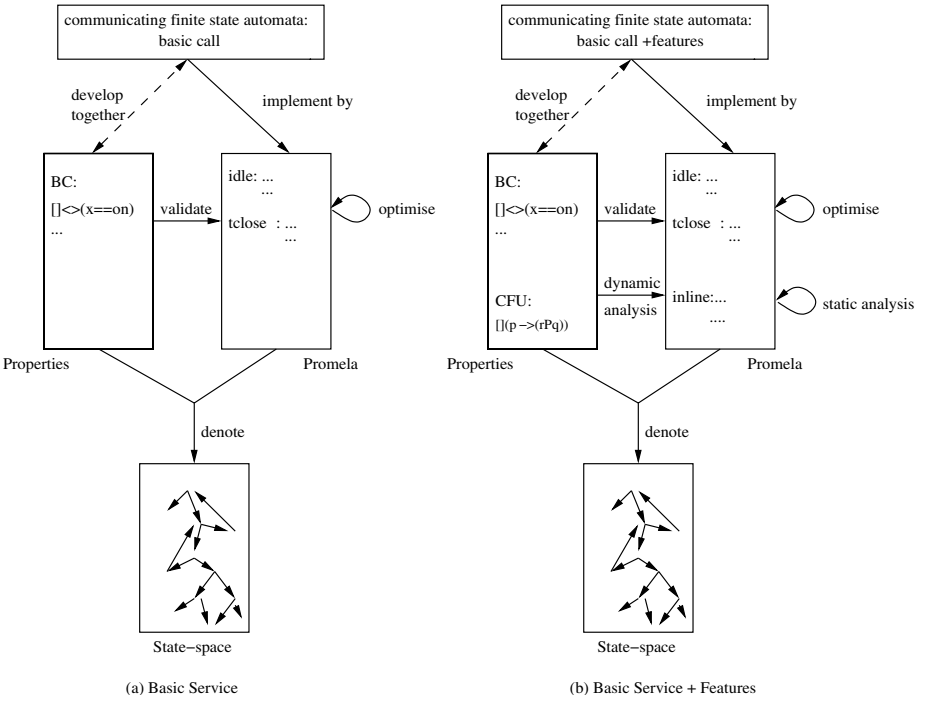


Fig. 1. Overall Approach

approach and, for example, allows us to avoid the frame problem. The Promela description on the rhs of figure 1(a) is regarded as the implementation; a crucial step therefore is validation of the implementation, i.e. checking satisfaction of the properties, using SPIN. Initial attempts fail, due to state-explosion, however, an examination of the underlying state-space (bottom of figure 1(a)) leads us to discover simple, but very effective state-reduction strategies (via code "optimisation").

The second phase, when we add features, is depicted in figure 1(b). Again, the starting point is finite state automata and properties. The Promela implementation is augmented with the new feature behaviour, primarily through the use of an *inline* function (see Section 9.1), and then validated. Interaction detection analysis takes two forms: *static* analysis, (syntactic) inspection of the Promela code, and *dynamic* analysis, reasoning over combinations of sets of logical formulae and configurations of the feature subscribers, using SPIN. The results of (either) analysis is interaction *detection*. The distinction between the two analyses is discussed in more detail in Section 11.

3 Background – Features and Interactions

Control of the progress of calls is provided by a service at an exchange (a *stored program control* exchange). This software must respond to events such as handset on or off hook, as well as sending control signals to devices and lines such as ringing tone or line engaged. A *service* is a collection of functionality that is usually self-sustaining. A *feature* is additional functionality, for example, a *call forwarding capability*, or *ring back when free*; a user is said to *subscribe* to a feature. When features are added to a basic service, there may be *interactions* (i.e. behavioural modifications) between both the features offered within that service, as well as with features offered in another service.

For example, if a user who subscribes to *call waiting* (CW) and *call forward when busy* (CFB) is engaged in a call, then what happens when there is a further incoming call? (Full details of all features mentioned here are given in section 7.) If the call is forwarded, then the CW feature is clearly compromised, and vice versa. In either case, the subscriber will not have his/her expectations met. This is an example of a single user, single component (SUSC) [5] interaction – the conflicting features are subscribed to by a single user. More subtle interactions can occur when more than one user/subscriber are involved, these are referred to as multiple user, multiple component (MUMC) interactions. Consider when user A subscribes to *originating call screening* (OCS), with user C on the screening list, and user B subscribes to CFB to user C. If A calls B, and the call is forwarded to C, as prescribed by B's CFB, then A's OCS is compromised. If the call is not forwarded, then we have the converse. These kind of interactions can be particularly difficult to detect (and resolve), since different features are activated at different stages of a the call.

Ideally, interactions are detected and resolved at service creation time, though this may not always be possible when third-party or legacy services are involved (for example, see [3]).

4 Basic Call Service

Figure 2 gives a diagrammatic representation of the automaton for the basic call service (following the IN (*Intelligent Networks*) model, distributed functional plane [19]).

States to the left of the idle state represent *terminating* behaviour, states to the right represent *originating* behaviour. Events observable by service subscribers label transitions: *user-initiated* events at the terminal device, such as (handset) on and (handset) off, are given in plain font, *network-initiated* events such as *unobt* and *engaged* are given in italics. Note that there are two "ring" events, *oring* and *tring*, for originating and terminating ring tone, respectively; call behaviour is asymmetric. Not all transitions are labelled.

The automata must communicate, in order to coordinate call set up and clear down. To implement communication, we associate a channel with each call process. Each channel has capacity for at most one message: a pair consisting of

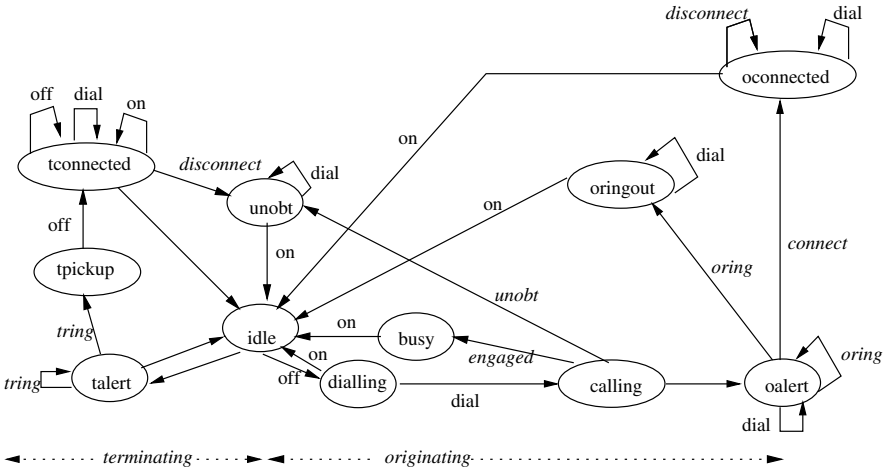


Fig. 2. Basic Call - States and Events

a channel name (the other party in the call) and a status bit (the status of the connection). Figure 3 describes how messages are interpreted.

5 Basic Call Service Properties in LTL

Below, we give a set of temporal properties describing the behaviour of the basic call service. Before doing so, we make a few comments about our use of LTL in SPIN.

When using SPIN's LTL converter (or otherwise – we use the conversion tool of Etessami, [9]) it is possible to check whether a given property holds for *All Executions* or for *No Executions*. A universal quantifier is implicit at the beginning of all LTL formulas and so, to check an LTL property, it is natural to choose the *All Executions* option. However, we sometimes wish to check that

Contents of Channel A	*Interpretation*
empty	A is free
(A,0)	A is engaged, but not connected
(B,0)	A is engaged, but not connected B is terminating party B is attempting connection
(B,1)	If channel B contains (A,1) then A and B are connected

Fig. 3. States of a Communication Channel in the Protocol

a given property (p say) holds for *some state* along *some execution path* (or "p is possible"), we can do so by showing that "$\langle\rangle p$ holds for *No Executions*" is **not** true (via a never-claim violation), which is equivalent. In Property 1 below, we use the notation E to mean *for some path* in place of the usual implicit *for all paths*. Additionally, we can use SPIN to prove properties of the form "p is true in the next state relative to process i". (That is p is true after the next time that process i is active.) This is done via judicious use of SPIN's *_last* operator, details are omitted here. We use the shorthand $\circ_{proci}$ to mean the next global state in which process *proci* has made a local transition. In addition the operators $\mathcal{W}$ (*weak* until) and $\mathcal{P}$ (precedes) are defined by $f\mathcal{W}g = \Box f \vee (fUg)$ and $f\mathcal{P}g = \neg(\neg fUg)$.

The LTL is given here alongside each property. This involves referring to variables (eg. *dialled* and *connect*) contained within the Promela code (an extract of which is given in section 6.1). We use symbols to denote predicates, for example "$\Box p$ where p is $dialled[i] == i$". This provides a neater representation, and the LTL converter requires properties to be given in this way.

Property 1. *A connection between two users is possible.*
That is: $E\Diamond p$, where p is $connect[i].to[j] == 1$, for $i \neq j$.

Property 2. *If you dial yourself, then you receive the engaged tone before returning to the idle state.*
That is: $\Box(p \rightarrow ((\neg r)\mathcal{W}q))$ where p is $dialled[i] == i$, q is $network\_event[i] == engaged$ and r is $user[proci]@idle$.

Property 3. *Busy tone or ringing tone will directly (that is, the next time that the process is active) follow calling.*
That is: $\Box(p \rightarrow \circ_{proci} q)$ where p is $event[i] == call$ and q is $((network\_event[i] == engaged) \vee (network\_event[i] == oring))$.

Property 4. *The dialled number is the same as the number of the connection attempt.*
That is: $\Box(p \rightarrow q)$ where p is $dialled[i] == j$ and q is $partner[i] == chan\_name[j]$.

Property 5. *If you dial a busy number then either the busy line clears before a call is attempted, or you will hear the engaged tone before returning to the idle state.*
That is: $\Box((p \wedge v \wedge t) \rightarrow (((\neg s)\mathcal{W}(w)) \vee ((\neg r)\mathcal{W}q)))$ where p is $dialled[i] == j$, v is $event[i] == dial$, t is $full(chan\_name[j])$, s is $event[i] == call$, w is $len(chan\_name[i]) == 0$, r is $user[proci]@idle$ and q is $network\_event[i] == engaged$, for $i \neq j$.

Note that the operator *len* is used to define w in preference to the function *empty* (or $nfull$). This is because SPIN disallows the use of the negation of these functions (and $\neg w$ arises within the never-claim).

Property 6. *You can not make a call without having just (that is, the last time that the process was active,) dialled a number.*
That is: $\Box(p \rightarrow q)$ where p is $user[proci]@calling$ and q is $event[i] == dial$.

6 Basic Call Service in Promela

6.1 Unoptimised Code

Each call process (see figure 2) is described in Promela as an instantiation of the (parameterised) proctype User declared thus:

```
proctype User (byte selfid;chan self)
```

Promela is a state-based formalism, rather than event-based. Therefore, we represent events by (their effect on) variables, and states (e.g. calling, dialling, etc.) by labels. Since each transition is implemented by several compound statements, we group these together as an *atomic* statement, concluding with a *goto*.

An example of the original (unoptimised) Promela code (as described in [4]) associated with the *idle*, *dialling*, *calling* and *oconnected* states and their outgoing transitions is given below. (For the full optimised code, contact the authors.) The global/local variables and parameters should be self-explanatory. We note in passing that any variable about which we intend to reason should not be updated more than once within any atomic statement; also d_steps, while more efficient than atomic steps, are not suitable here because they do not allow a process to jump to a label out of scope. There are numerous assertions within the code, particularly at points when entering a new (call) state, and when reading and writing to communication channels.

```
idle:
  atomic{
             assert(dev == on);
             assert(partner[selfid]==null);
     /* either attempt a call, or receive one */
         if
             :: empty(self)->event[selfid]=off;
                dev[selfid]=off;
                self!self,0;goto dialling
     /* no connection is being attempted, go offhook */
     /* and become originating party */
         :: full(self)-> self?<partner[selfid],messbit>;
     /* an incoming call */
                if
                ::full(partner[selfid])->
                  partner[selfid]?<messchan,messbit>;
                  if
               :: messchan == self /* call attempt still there */
                     ->messchan=null;messbit=0;goto talert
                  :: else -> self?messchan,messbit;
     /* call attempt cancelled */
                     partner[selfid]=null;partnerid=6;
                 messchan=null;messbit=0;goto idle
              fi
                ::empty(partner[selfid])->
                  self?messchan,messbit;
     /* call attempt cancelled */
                  partner[selfid]=null;partnerid=6;
              messchan=null; messbit=0; goto idle
                fi
        fi};


dialling:
  atomic{
```

```
            assert(dev == off);assert(full(self));
            assert(partner[selfid]==null);
     /* dial or go onhook */
        if
            :: event[selfid]=dial;
     /* dial and then nondeterministic choice of called party */
           if
               :: partner[selfid] = zero;dialled[selfid] = 0;
                  partnerid=0
               :: partner[selfid] = one;dialled[selfid] = 1;
                  partnerid=1
               :: partner[selfid] = two;dialled[selfid] = 2;
                  partnerid=2
               :: partner[selfid] = three;dialled[selfid] = 3;
                  partnerid=3
               :: partnerid= 7;
               fi
            :: event[selfid]=on; dev[selfid]=on;
           self?messchan,messbit;assert(messchan==self);
               messchan=null;messbit=0;goto idle
    /*go onhook, without dialling */
        fi};

calling:/* check number called and process */
  atomic{
        event[selfid]=call;
            assert(dev == off);assert(full(self));
            if
        :: partnerid==7->goto unobtainable
            :: partner[selfid] == self -> goto busy
   /* invalid partner */
        :: ((partner[selfid]!=self)&&(partnerid!=7)) ->
           if
               :: empty(partner[selfid])->partner[selfid]!self,0;
                        self?messchan,messbit;
                     self!partner[selfid],0;goto oalert
   /* valid partner, write token to partner's channel*/
          :: full(partner[selfid]) -> goto busy
   /* valid partner but engaged */
           fi
        fi};

oconnected:
  atomic{
                assert(full(self));assert(full(partner[selfid]));
  /* connection established */
        connect[selfid].to[partnerid] = 1;
        goto oclose};
```

Any number of call processes can be run concurrently. For example, assuming the global communication channels `zero`, `one`, etc. a network of four call processes is given by:

```
atomic{run User(0,zero);run User(1,one); run User(2,two);run User(3,three)}
```

6.2 Options and State-Space Reduction Techniques

Initial attempts to validate the properties against a network of four call processes fail because of state-explosion. In this section we examine the causes, the applicability of standard solutions and how the Promela code can be transformed (optimised) to reduce the size of the state-space.

SPIN Options. The default Partial order reduction (POR) option was applied throughout, but did not reduce the size of the state-space sufficiently. This is due to the scarcity of statically defined "safe" operations (see [17]) in our model. Any assignments to local variables are embedded in large atomic statements that are not safe. Furthermore the use of non-destructive read operations (to test the contents of a channel) prevents the assignment of exclusive read/send status to channels. Such a test is crucial: often behaviour depends on the exact contents of a channel.

States can be compressed using *minimised automaton encoding* (MA) or *compression* (COM). When using the former, it is necessary to define the maximum size of the state-vector, which of course implies that one has searched the entire space. However one can often find a reasonable value by choosing the (uncompressed) value reported from a preliminary verification with a deliberate assertion violation. While MA and COM together give a significant memory reduction, the trade-off in terms of time was simply unacceptable.

Other State-space Reduction Strategies. A simple but stunningly effective way to reduce the state-space is to ensure that each visit to a *call* state is indeed a visit to the same underlying Promela state. This means that as many variables as possible should be initialised and then reset to their initial value (reinitialised) within Promela loops. For example, in virtually every call state it is possible to return to *idle*. An admirable reduction is made if variables such as `messchan` and `messbit` are initialised before the first visit to this label (*call* state), and then reinitialised before subsequent visits. This is so that global states that were previously *distinguished* (due to different values of these variables at different visits to the *idle* call state) are now *identified*.

The largest reduction is to be found when such variables are routinely reset before progressing to the next *call* state. Unfortunately, this is not always possible, as it would result in variables *about which we wish to reason* being updated more than once within an atomic statement (as discussed in section 6.1). However, there is a solution: add a further state where variables are reinitialised. For example, we have added a new state *preidle*, where the variables *network_event* and *event* are reinitialised, before progression to *idle*. Therefore every occurrence of *goto idle* becomes *goto preidle*.

We note that although the (default) data-flow optimisation option available with SPIN attempts to reinitialise variables automatically, we have found that this option actually *increases* the size of the state-space of our model. This is due to the initial values of our variables often being non-zero (when they are of type mtype for example). SPIN's data-flow optimisation always resets variables to zero. Therefore we *must* switch this option off, and reinitialise our variables manually.

The size of the state-space can be greatly reduced if any reference to (update of) a global variable which is not needed for verification, is commented out. Furthermore, by including all references to *all* of the *event* variables (say) when any such variable is needed for verification (see for example Property 3), the size of the state-space can be increased by an unnecessarily large amount. For

example, to prove that Property 3 holds for $user[i]$, we are only interested in the value of $event[i]$, not of $event[j]$ where $i \neq j$. The latter do not need to be updated. Thus an inline function, $event\_action(eventq)$ has been introduced to enable the *update of specific variables*. That is, it allows us to update the value of $event[i]$ to the value $eventq$, and leave the other event variables set to their default value. So, for example, if $i = 0$, the event_action inline becomes:

```
inline event_action (eventq)
{
 if
 ::selfid==0->event[selfid]=eventq
 ::selfid!=0->skip
 fi
}
```

Any reference to this inline definition is merely commented out when no *event* variables are needed for verification. (Another inline function is included to handle the *network_event* variables in the same way.)

We note that this reduction is not implemented in SPIN. SPIN does, however, issue a warning "variable never used" in situations where such a reduction would be beneficial.

These transformations (which we refer to as *code optimisation*) not only lead to a *dramatic* reduction of the underlying state-space, the search depth required was reduced to 10 percent of the initial value, but they do not involve abstraction away from the original model. On the contrary, if anything, they could be said to reduce the level of abstraction.

Unlike other abstraction methods (see for example [6], [11] and [13]) our techniques are simple, and merely involve making simple checks that unneccessary states have not been unintentionally introduced. We believe that these kinds of state-space explosions are not uncommon. All SPIN users should be aware that they may be introducing spurious states when coding their problem in Promela.

6.3 Basic Call Service Validation

It was possible to verify all six properties listed in section 5 well within our 1.5 Gbyte memory limit. State compression was used throughout. The verification of property 3 took the longest (21 mins) and a greater search-depth was reached in this case. This is partially due to the fact that both the *event* and *network_event* variables for the process under consideration had to be included for this property. In addition, the use of the *_last* operator precludes the use of partial order reduction, which could have helped to reduce the complexity in this case.

7 Features

Now that the state-space is tractable, we can commence the second phase: adding a number of features to the basic service.

7.1 Features

The set of features that we have added include:

- **CFU – call forward unconditional.** All calls to the subscriber's phone are diverted to another phone.
- **CFB – call forward when busy.** All calls to the subscriber's phone are diverted to another phone, if and when the subscriber is busy.
- **OCS – originating call screening.** All calls by the subscriber to numbers on a predefined list are inhibited. Assume that the list for user x does not contain x.
- **ODS – originating dial screening.** The dialling of numbers on a predefined list by the subscriber is inhibited. Assume that the list for user x does not contain x.
- **TCS – terminating call screening.** Calls to the subscriber from any number on a predefined list are inhibited. Assume that the list for user x does not contain x.
- **RBWF – ring back when free.** The subscriber has the option to call the last recorded caller to his/her phone.

Two further features that are straightforward to implement are originating call behaviour (e.g. a pay phone) and terminating call behaviour (e.g. a teen line). However we give no details of such features here.

We do not give automata for all the features, but in figure 4 we give the additional behaviour prescribed by the RBWF feature. While the automaton appears to be non-deterministic, it is not because the dial event is associated with data. The data determines the choice (e.g. 7 will result in the transition to ringback). Notice that this feature introduces a new call state (namely *ringback*); it is the only feature to do so.

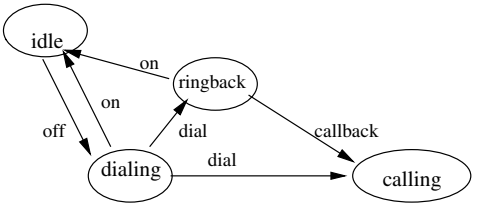


Fig. 4. Part of the Finite State Automaton for RBWF

8 Temporal Properties for Features

The properties for features are more difficult to express than those for the basic service. In order to accurately reflect the behaviour of each feature great attention must be paid to the *scope* of each property (within the corresponding LTL

formula). For example, in property 8 (see below), it is essential that (for the CFB feature to be invoked) the forwarding party has a full communication channel *whilst the dialling party is in the dialling state*. This can only be expressed by stating that the forwarding party must have a full channel continuously between two states, the first of which must occur *before* the dialling party enters the dialling state, and the second *after* the dialling party emerges from the dialling state.

The values of the variables i, j and k depend on the *particular pair of features* and the corresponding *property* that is being analysed. These variables are therefore updated prior to each verification either manually (by editing the Promela code directly), or automatically during the running of a model-generating script (see section 12).

Property 7 – CFU. Assume that user j forwards to k.
If user i rings user j then a connection between i and k will be attempted before user i hangs up.
That is: $[](p \rightarrow (r \mathcal{P} q))$, where p is $dialled[i] == j$, r is $partner[i] == chan\_name[k]$, and q is $dev[i] == on$.

Property 8 – CFB. Assume that user j forwards to k.
If user i rings user j when j is busy then a connection between i and k will be attempted before user i hangs up.
That is: $[](((u \wedge t) \wedge ((u \wedge t) U ((\neg u) \wedge t \wedge p))) \rightarrow (r \mathcal{P} q))$, where p is $dialled[i] == j$, t is $full(chan\_name[j])$, r is $partner[i] == chan\_name[k]$, u is $user[proci]@dialling$ and q is $dev[i] == on$.

Property 9 – OCS. Assume that user i has user j on its screening list.
No connection from user i to user j is possible.
That is: $[](\neg p)$, where p is $connect[i].to[j] == 1$.

Property 10 – ODS. Assume that user i has user j on its screening list.
User i may not dial user j .
That is: $[](\neg p)$, where p is $dialled[i] == j$.

Property 11 – TCS. Assume that user i has user j on its screening list.
No connection from user j to user i is possible.
That is: $[](\neg p)$, where p is $connect[j].to[i] == 1$.

Property 12 – RBWF. Assume that user j has automatic call back.
It is possible for an attempted call from i to j to eventually result in a successful call from j to i (without j ever dialling i) .

That is: $E(\diamond((p \wedge t \wedge \diamond q) \wedge (r \mathcal{P} q))$, where p is $dialled[i] = j$, q is $dialled[j] = i$, r is $connect[j].to[i] == 1$ and t is $event[i] == call$.

9 The Features in Promela

Relevant changes that need to be made to the Promela model are given below. Before this, we make a few observations:

- To implement the features we have included a "feature_lookup" function (see below) that implements the features and computes the transitive closure of the forwarding relations (when such features apply to the same call state).
- We distinguish between call and dial screening; the former means a call between user A and B is prohibited, regardless of whether or not A actually dialled B, the latter means that if A dials B, then the call cannot proceed, but they might become connected by some other means. The latter case might be desirable if screening is motivated by billing. For example, if user A dials C (a local leg) and C forwards calls to B (a trunk leg) then A would only pay for the local leg.
- Currently we restrict the size of the lists of screened callers (relating to the OCS, ODS and TCS features) to one. That is, we assume that it is impossible for a single user to subscribe to two of the *same* screening feature. This is sufficient to demonstrate some feature interactions, and limits the size of the state-space.
- The addition of RBWF, while straightforward, increases the complexity of the underlying state-space greatly. This is due both to the addition of the new *ringback* state and to the fact that it involves recording (in a structure indexed by call processes) the last connection attempt. The issue is not just that there is a new global variable, but that *call* states that were previously identified are now distinguished by the contents of that record (see discussion about variable reinitialisation in section 6.2).
- To ensure that all variables are initialised, we use 6 as a default value. This is particularly useful when a user does not subscribe to a particular feature. The value 7 is used to denote both an unobtainable number (e.g. an incorrect number) and to denote the "button press" in RBWF. We do not use an additional value for the latter, so as not to increase the state space.

9.1 Implementation of Features: The *Feature_lookup* Inline

In order to enable us to add features easily, all of the code relating to *feature behaviour* is now included within an *inline* definition. The *feature_lookup* inline is defined as follows:

```
inline feature_lookup(part_chan,part_id,st)
{
 do
 ::((st==st_dial)&&(ODS[selfid]==part_id))->st=st_unobt
 ::((st==st_dial)&&(RBWF[selfid]==1)&&(part_id==7))->st=st_rback
 ::((part_id!=7)&&(st==st_dial)&&(CFU[part_id]!=6))
          ->part_id=CFU[part_id];part_chan=chan_name[part_id]
 ::((part_id!=7)&&(st==st_dial)&&(CFB[part_id]!=6)&&(len(part_chan)>0))
          ->part_id=CFB[part_id];part_chan=chan_name[part_id]
 ::((st==st_call)&&(OCS[selfid]==part_id))->st=st_unobt
 ::((st==st_call)&&(TCS[part_id]==selfid))->st=st_unobt
 ::else->break
 od
 }
```

The parameters *part_chan*, *part_id*, and *st* take the values of the current partner, partnerid and state of a user when a call to the *feature_lookup* inline is made.

Statements within *feature_lookup* pertaining to features that are not currently active are automatically commented out (see section 12).

We note that in some sense *feature_lookup* encapsulates centralised intelligence in the switch, as it has "knowledge" of the status of processes and data concerning feature configuration. While on the one hand one might argue that this is against the spirit of an *IN* switch, on the other hand we maintain that MUMC feature interactions simply cannot be detected in a completely distributed architecture.

9.2 Feature Validation

Each feature was validated (via SPIN verification) against the appropriate set of properties (1–12). Each verification took place within 30 minutes and took place well within our our 1.5 Gbyte memory limit. For brevity, however, we do not give details here.

10 Static Analysis

Static analysis is an analysis of the *structure* of the feature descriptions, i.e. an examination of the *syntax*. Specifically, we look for *overlapping* guards (two or more guards which evaluate to true, under an assignment to variables) with diverging consequences. A more operational explanation is the detection of shared *triggers* of features. Because we have collected additional feature behaviour together within the inline *feature_lookup*, we need only consider overlapping guards within this function. If there is an overlap, and the consequences diverge, then we have non-determinism and hence a potential interaction.

For example, consider the following overlap between CFU and CFB:

```
::((part_id!=7)&&(st==st_dial)&&(CFU[part_id]!=6))
       ->part_id=CFU[part_id]; part_chan=chan_name[part_id]
::((part_id!=7)&&(st==st_dial)&&(CFB[part_id]!=6)&&(len(part_chan)>0))
       ->part_id=CFB[part_id];part_chan=chan_name[part_id]
```

The overlap occurs under the assignment $st = st\_dial$, $CFU[part\_id] = x$, $len(part\_chan) > 0$, and $CFB[part\_id] = y$ where $x, y \neq 6$. When $x \neq y$, the first consequent assigns x to *part_id*, the second assigns y to *part_id*. These are clearly divergent, and so we have found an interaction.

SUSC and MUMC interactions are distinguished by considering the roles of *part_id* and $selfid$ as indices. If the same index is used for the feature subscription, e.g. $CFU[part\_id]$ and $CFB[part\_id]$, then the interaction is SUSC, if different indices are used, it is MUMC. In this example, the interaction is clearly SUSC.

An overlap is not always possible. For example, consider the first two choices:

```
::((st==st_dial)&&(ODS[selfid]==part_id))
        ->st=st_unobt
::((st==st_dial)&&(RBWF[selfid]==1)&&(part_id==7))
        ->st=st_rback
```

As 7 is not a valid number to be in a screening list there is no overlap and hence no interaction.

In all, there are 7 pairs to consider (4 clauses for *st_dial*, leading to 6 pairs, and two clauses for *st_call*, leading to one pair). Results of the static analysis are given in the tables of figure 5. A $\surd$ indicates an interaction whereas a $\times$ indicates none. The tables are symmetric.

	CFU	CFB	OCS	ODS	TCS	RBWF
CFU	–	$\surd$	$\times$	$\times$	$\times$	$\times$
CFB	$\surd$	–	$\times$	$\times$	$\times$	$\times$
OCS	$\times$	$\times$	–	$\times$	$\times$	$\times$
ODS	$\times$	$\times$	$\times$	–	$\times$	$\times$
TCS	$\times$	$\times$	$\times$	$\times$	–	$\times$
RBWF	$\times$	$\times$	$\times$	$\times$	–	–

(a) SUSC

	CFU	CFB	OCS	ODS	TCS	RBWF
CFU	–	$\times$	$\times$	$\surd$	$\times$	$\times$
CFB	$\times$	–	$\times$	$\surd$	$\times$	$\times$
OCS	$\times$	$\times$	–	$\times$	$\times$	$\times$
ODS	$\surd$	$\surd$	$\times$	–	$\times$	$\times$
TCS	$\times$	$\times$	$\times$	$\times$	–	$\times$
RBWF	$\times$	$\times$	$\times$	$\times$	$\times$	–

(b) MUMC

Fig. 5. Feature Interaction Results - Static Analysis

Static analysis is a very simple yet very effective mechanism for finding some interactions – those which arise from new non-determinism. It is based on equational reasoning techniques and the process of finding overlapping guards (known as superposition) can be automated. The process of considering whether the consequent statements are divergent is more difficult and a complete solution would require a thorough axiomatic description of the Promela language. However, it would be possible to automate a relatively effective approach based on simple assignment. For the purposes of this paper, we rely on manual inspection of the function *feature_lookup*. In any case, we note that the ease and contribution of static analysis depends very much on the structure of the specification.

We now turn our attention to a *dynamic* form of analysis.

11 Dynamic Analysis

Dynamic analysis depends upon logical properties that are satisfied (or not) by pairs of users subscribing to combinations of features.

Consider two users, $u1$ and $u2$. Then $u1_{f_i} \cup u2_{f_j}$ is the *configuration*, or *scenario*, in which $u1$ subscribes to feature f_i and $u2$ subscribe to feature f_j. Two features f_i and f_j *interact* if a property that holds for f_i alone, no longer holds in the presence of another feature f_j. More formally stated: for a property ϕ, we have $u1_{f_i} \models \phi$ but $u1_{f_i} \cup u2_{f_j} \not\models \phi$. When $u1 == u2$, then the interaction is SUSC, otherwise it is MUMC. Note that there are no constraints on i and j, ie. $i = j$ or $\neq j$.

Note that the analysis is *pairwise*, known as 2-way interactions. While at first sight this may be limiting, empirical evidence suggests there is little motivation

to generalise, 3-way interactions that are not detectable as a 2-way interaction are exceedingly rare [21].

An initial approach is to consider *any* property above as a candidate for ϕ. However, it is easy to see that in this case all features interact. A more selective approach is required: we consider only the properties associated with the features under examination, i.e. for features f_i and f_j, consider only properties ϕ_i and ϕ_j. An SUSC (MUMC) interaction between f_i and f_j, resulting from a violation of property ϕ_i is written $(f_i, f_j)_S$ ($(f_i, f_j)_M$).

11.1 Dynamic Analysis – Feature Interaction Results

The tables in figure 6 gives the interactions found for pairs of features in both the SUSC case and the MUMC case. A $\surd$ in the row labelled by feature f_i means that the property ϕ_i is violated whereas a $\times$ indicates that no such violation has occurred. Two features f_i and f_j interact if and only if there is a $\surd$ in position (f_i, f_j) and/or a $\surd$ in position (f_j, f_i). BC is excluded as every feature interacts with it in some way.

	CFU	CFB	OCS	ODS	TCS	RBWF
CFU	–	$\surd$	×	×	×	×
CFB	$\surd$	–	×	×	×	×
OCS	×	×	–	×	×	×
ODS	×	×	×	–	×	×
TCS	×	×	×	×	–	×
RBWF	$\surd$	×	$\surd$	$\surd$	$\surd$	–

(a) SUSC

	CFU	CFB	OCS	ODS	TCS	RBWF
CFU	$\surd$	$\surd$	×	×	×	×
CFB	$\surd$	$\surd$	×	×	×	×
OCS	×	×	×	×	×	×
ODS	$\surd$	$\surd$	×	×	×	×
TCS	×	×	×	×	×	×
RBWF	×	×	$\surd$	$\surd$	$\surd$	×

(b) MUMC

Fig. 6. Feature Interaction Results - Dynamic Analysis

New SUSC interactions are detected by the dynamic analysis, namely those associated with the RBWF feature. For example, there is an $(RBWF, CFU)_S$ interaction because the CFU feature prevents the *record* variable pertaining to the subscriber being set to a non-default value. Therefore the subscriber is unable to perform a ring-back.

The tables are not symmetric. For example, there is an $(ODS, CFU)_M$ interaction, but not a $(CFU, ODS)_M$ interaction. To understand why, observe that static analysis detects an MUMC interaction under the assignment $ODS[0] = 1$, and $CFU[1] = 2$. Dynamic analysis also detects an interaction violation – indeed our analysis script (see section 12) generates exactly this scenario: an $(ODS, CFU)_M$ interaction with $i = 0$ and $j = 1$ (i.e. user 0 rings user 1). Consider those computations where $feature\_lookup$ takes the ODS branch. One could understand this as ODS having precedence. There is no interaction in this case: both property 7 and property 10 are satisfied. However, there is a computation where the CFU branch is taken; in this case CFU has precedence and

property 10 is violated because user 0 has *dialled* user 1 – before the call is forwarded to user 2 (although clearly property 7 is satisfied). Often, understanding why and how a property is violated will give the designer strong hints as to how to resolve an interaction.

The interactions uncovered by dynamic analysis depend very much on the *properties* and how the features are *modelled*. When the properties are *adequate*, we would expect every statically detected interaction to be detected dynamically, but not vice versa. This is borne out by our case-study. We may regard the static analysis step as a cheap method of uncovering some interactions, as well as providing an indication of whether or not we have a good set of behavioural properties. But, note that the properties are not complete descriptions, in particular they do not state what should *not* happen (i.e. the frame problem). For example, one might expect a $(CFU, TCS)_M$ interaction but this is not the case because although TCS will block the forwarded call, the *partner* variable will be set appropriately, thus satisfying property 7. Perhaps one should strengthen the property for CFU, to insist that the connection is made (rather than just setting *partner* appropriately). But it is not that simple, the forwarded party may be engaged, or have a forwarded feature (or any other kind of feature); the possibilities are endless. Therefore, we consider the CFU property to be quite adequate.

12 Automatic Model Generation and Feature Interaction

Originally, before features were added to the basic call model, global variables were manually "turned off" (ie. commented out) or replaced by local variables when they are not needed for verification. The addition of features has led to even more variables requiring to be selectively turned on and off, and set to different values. For example if an *originating call screening* feature is selected the *orig_call_sreen* array has to be included and its elements set to the appropriate values. In addition the *feature_lookup* inline must be amended to include those lines pertaining to the originating call screening feature. If no *ring back when free* feature is chosen, the entire *ringback* call state must be commented out.

Making all of the necessary changes before every SPIN run was extremely time-consuming and error prone. Therefore, we now use a Perl script to enable us to perform these changes automatically. Specifically this enables us to generate, for any combination of features and properties, a model from a template file. Each generated model also includes a header containing information about which features and properties have been chosen in that particular case, which makes it easier to monitor model-checking runs.

Dynamic feature interaction analysis is combinatorially explosive: we must consider all pairs of features *and* combinations of suitable instantiations of the free variables i,j and k occurring in the properties. For example, for the SUSC case alone this gives 36 different scenarios (though not all are valid). To ease this burden and to speed up the process, a further Perl script is used to enable

- systematic selection of pairs of features and parameters i,j and k, and generation of corresponding model,
- automatic SPIN verification of model and recording of feature interaction results.

Note that scenarios leading to feature interactions *are* recorded. Depending on the property concerned, a report of 1 error (properties 7–11) or 0 errors (property 12) from the SPIN verification indicates an interaction. Once (if) an SUSC interaction is found the search for MUMC interactions commences. If an MUMC interaction is found the next pair of features is considered. The following example of output demonstrates the complete results for CFU and CFB with property 7.

```
/*The features are 1 and 2 */

/*New combination of features:CFU[0]=1 and CFB[0]=0 */
feature 2 is meaningless

/*New combination of features:CFU[0]=1 and CFB[0]=1 */
with property 7
with parameters 0,0 and 1 errors: 0

with parameters 1,0 and 1 errors: 0

with parameters 2,0 and 1 errors: 0

with parameters 3,0 and 1 errors: 0


/*New combination of features:CFU[0]=1 and CFB[0]=2 */
with property 7
with parameters 0,0 and 1 errors: 1 FEATURE INTERACTION: SUSC

/*New combination of features:CFU[0]=1 and CFB[1]=0 */
potential loop, test seperately

/*New combination of features:CFU[0]=1 and CFB[1]=1 */
feature 2 is meaningless

/*New combination of features:CFU[0]=1 and CFB[1]=2 */
with property 7
with parameters 0,0 and 1 errors: 1 FEATURE INTERACTION: MUMC
```

13 Conclusions and Future Directions

We have used Promela and SPIN to analyse the behaviour of a software artifact – feature interaction in a telecomms service. Our approach involves two different levels of abstraction: communicating finite state automata and temporal logic formulae, represented by Promela specifications, labelled transition systems and Büchi automata.

We have demonstrated the approach with an analysis of a basic call service with six features, involving four users with full functionality. There are two types of analysis, static and dynamic; the latter is completely automated, making extensive use of Perl scripts to generate the SPIN runs.

The distinction between static and dynamic analysis is important; the latter is more comprehensive, but the former provides a simple yet effective initial step, and a check for the temporal properties upon which the latter depends.

State-explosion is a major concern in feature interaction analysis: understanding how a Promela model can be optimised, in order to generate tractable state-spaces, is important. We have outlined a simple but effective state-space reduction technique for Promela that does not abstract away from the system being modelled, on the contrary, it may be understood as reducing the gap between the Promela representation and the system under investigation. The technique involves reinitialising variables and results in a reduction of 90 per cent of the state-space. Thus, we overcome classic state-explosion problems and our interaction analysis results are considerably more extensive than those in [22]. We believe that both our reduction technique and the use of Perl scripts could be useful to the SPIN community in general.

Finally, we note that understanding why an interaction occurs can help the redesign process. For example, static analysis indicates shared triggers and dynamic analysis indicates in-built precedences between features, when the results of the analysis are not symmetric. Both can indicate how to alter precedences between features, in order to resolve interactions. How to do so in a structured way is a topic for further work.

Acknowledgements. The authors thank Gerard Holzmann for his help and advice, and the Revelation project at Glasgow for computing resources. The second author was supported by a Daphne Jackson Fellowship from the EPSRC.

References

1. L. G. Bouma and H. Velthuijsen, editors. *Feature Interactions in Telecommunications Systems*. IOS Press (Amsterdam), May 1994.
2. M. Calder and E. Magill, editors. *Feature Interactions in Telecommunications and Software Systems*, volume VI. IOS Press, Amsterdam, 2000.
3. M. Calder, E. Magill, and D. Marples. A hybrid approach to software interworking problems: Managing interactions between legacy and evolving telecommunications software. *IEE Proceedings - Software*, 146(3):167–180, June 1999.
4. Muffy Calder and Alice Miller. Analysing a basic call protocol using Promela/XSpin. In *[15]*, pages 169–181, 1998.
5. E. J. Cameron, N. Griffeth, Y.-J. Lin, M. E. Nilson, and W. K. Schnure. A feature interaction benchmark for IN and beyond. In *[1]*, pages 1–23, May 1994.
6. E.M. Clarke, O. Gumberg, and D Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, September 1994.
7. Costas Courcoubetis, editor. *Proceedings of the Fifth International Conference on Computer Aided Verification (CAV '93)*, volume 697 of *Lecture Notes in Computer Science*, Elounda,Greece, June/July 1993. Springer-Verlag.
8. P. Dini, R. Boutaba, and L. Logrippo, editors. *Feature Interactions in Telecommunication Networks IV*. IOS Press (Amsterdam), June 1997.

9. K. Etessami. Stutter-invariant languages, ω-automata, and temporal logic. In *[12]*, pages 236–248, 1999.
10. A. Felty and K. Namjoshi. Feature specification and automatic conflict detection. In *[2]*, pages 179–192, May 2000.
11. Susanne Graf and Claire Loiseaux. A tool for symbolic program verificaion and abstration. In *[7], pages = 71–84, year = 1993,*.
12. Nicolas Halbwachs and Doron Peled, editors. *Proceedings of the eleventh International Conference on Computer-aided Verification (CAV '99)*, volume 1633 of *Lecture Notes in Computer Science*, Trento, Italy, July 1999. Springer-Verlag.
13. Constance L. Heitmeyer, James Jr. Kirby, Bruce Labaw, Myla Archer, and Ramesh Bharadwaj. Using abstraction and model checking to detect safety violations in requirements specifications. *IEEE Transactions on Software Engineering*, 24(11), November 1998.
14. D. Hogrefe and S. Leue, editors. *Proceedings of the Seventh International Conference on Formal Description Techniques (FORTE '94)*, volume 6 of *International Federation For Information Processing*, Berne, Switzerland, October 1994. Kluwer Academic Publishers.
15. Gerard Holzmann, Elie Najm, and Ahmed Serhrouchni, editors. *Proceedings of the 4th Workshop on Automata Theoretic Verification with the Spin Model Checker (SPIN '98)*, Paris, France, November 1998.
16. Gerard J. Holzmann. The model checker Spin. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.
17. Gerard J. Holzmann and Doron Peled. An improvement in formal verification. In *[14]*, pages 197–211, 1994.
18. G.J. Holzmann and Margaret H. Smith. A practical method for the verification of event-driven software. In *Proceedings of the 1999 international conference on Software engineering (ICSE99)*, pages 597–607, Los Angeles, CA, USA, May 1999. ACM Press.
19. *IN Distributed Functional Plane Architecture*, recommmendation q.1204, ITU-T edition, March 1992.
20. K. Kimbler and L.G. Bouma, editors. *Feature Interactions in Telecommunications and Software Systems V*. IOS Press (Amsterdam), September 1998.
21. M. Kolberg, E. H. Magill, D. Marples, and S. Reiff. Results of the second feature interaction contest. In *[2]*, pages 311–325, May 2000.
22. M. Plath and M. Ryan. Plug-and-play features. In *[20]*, pages 150–164, 1998.
23. M. Thomas. Modelling and analysing user views of telecommunications services. In *[8]*, pages 168–182, 1997.

Behavioural Analysis of the Enterprise JavaBeans^{TM} Component Architecture

Shin Nakajima[1] and Tetsuo Tamai[2]

[1] NEC Corporation, Kawasaki, Japan
[2] Graduate School of the University of Tokyo, Tokyo, Japan

Abstract. Rigorous description of protocols (a sequence of events) between components is mandatory for specifications of distributed component frameworks. This paper reports an experience in formalizing and verifying behavioural aspects of the Enterprise JavaBeans^{TM} specification with the SPIN model checker. As a result, some potential flaws are identified in the EJB 1.1 specification document. The case study also demonstrates that the SPIN model checker is an effective tool for behavioural analysis of distributed software architecture.

1 Introduction

Software component technology is gaining importance in constructing large-scale distributed applications such as E-Commerce systems, and has also been widely accepted in the industry as a new technology for object-oriented software reuse [21]. Notable examples are COM/DCOM, JavaBeans/Enterprise JavaBeans, and a new component model of CORBA proposed by OMG. Systems can be constructed by implementing a component that encapsulates application logic and making it run with existing components in a pre-defined execution environment.

A *component* is a constituent of a system, and is a reusable unit that has a definite interface for exchanging information with other constituents. The main feature of the technology is an integration framework that is based on a specific computational model for components and supports the basic information exchange protocols among them. User-defined components can only be run successfully if they conform to the specification that the framework assumes. This implies that the integration framework specification should be described in an unambiguous manner [10].

Current component frameworks, however, have not been successful in providing precise specifications in their documents. Specification documents use a natural language (English) and informal diagrams for illustrative purposes. It is not uncommon to find ambiguities, inconsistencies or even *bugs* in the informal documents. Thus, the application of formal techniques is necessary for creating rigorous specification documents for component frameworks. Actually, Sullivan et al. [19] have identified some ambiguities in the description of the COM aggregation and interface negotiation specification by using a precise description written in the Z notation. Sousa et al. [18] have shown that modeling

M.B. Dwyer (Ed.): SPIN 2001, LNCS 2057, pp. 163–182, 2001.

with Wright [1], an Architecture Description Language (ADL), is effective for behavioural analysis of the EJB 1.0 specification.

This paper reports an experience using the SPIN model checker [5] for behavioural analysis of the Enterprise JavaBeans component architecture. The main contributions of the present work can be summarized as follows. (1) Results show that the SPIN model checker is an adequate tool for behavioural analysis of distributed software architecture as compared with other tools such as Wright [1] or Darwin [12]. (2) The case identifies there are some potential flaws in the EJB 1.1 document [20] in terms of behavioural specification.

2 Enterprise JavaBeans

Enterprise JavaBeans^{TM} is a component architecture for distributed business applications. This section briefly describes the Enterprise JavaBeans (EJB) component architecture and presents some of its behavioural specifications. The material in this section is based on the EJB 1.1 specification document [20].

2.1 The Component Architecture

Figure 1 illustrates an overview of the EJB component architecture, and introduces several principal participants. `Client` is a client Java application. First it uses a `JNDI` Naming Server to look up an object reference of a `Home` interface. `Home` is responsible for creating a new `Bean` instance, and returns an object reference of a `Remote`. The `Remote` acts as a proxy to the `Bean`. The `Remote` accepts requests from the `Client` and delegates them to the `Bean`. `Bean`, then, executes these requests under the control of the `Container`.

The underlying EJB Server provides `Bean` with several runtime-services such as passivation, persistency, and garbage collection. All client requests to the `Bean` are made through the `Home` or `Remote` object. The `Container` may intercept the original requests and then control invocations on the `Bean`, while making it possible for the runtime-services to interleave between client request invocations.

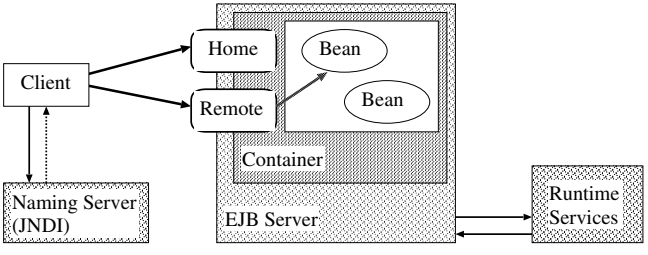


Fig. 1. The EJB Architecture

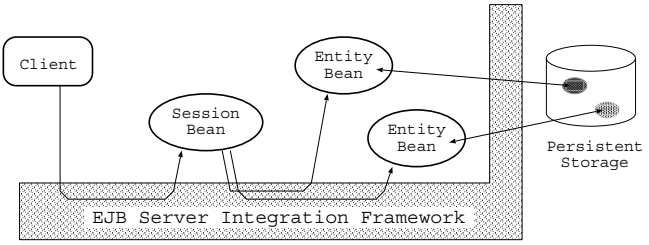


Fig. 2. Session and Entity Beans

There are two types of Enterprise JavaBeans : Session Beans and Entity Beans (Figure 2). An Entity Bean represents data stored in persistent storage and is a shared resource. The internal values of an Entity Bean are synchronized with the contents of persistent storage through the runtime service. A Session Bean, on the other hand, executes on behalf of a single client, and may implement long-lived transaction style application logic to access several Entity Beans.

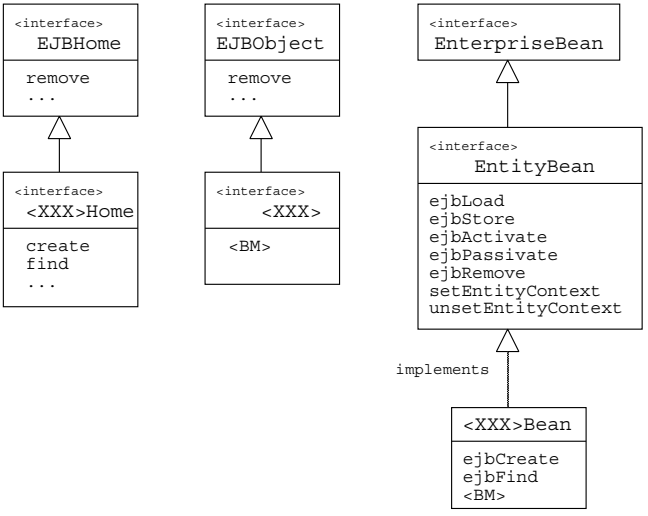


Fig. 3. Class Diagram

Figure 3 presents a fragment of the class diagram for Entity Beans. In the figure, `EJBHome`, `EJBObject`, `EnterpriseBean`, and `EntityBean` are the four Java interfaces that the EJB component framework provides. The `EJBHome` interface describes the client's view of the `Home` shown in Figure 1, while `EJBObject`

corresponds to the `Remote`. The `EntityBean` interface specifies the basic APIs that should be implemented by every Entity Bean. The `EnterpriseBean` is a common super-interface for both `EntityBean` and `SessionBean`[1].

In order to develop a new Entity Bean, three Java constructs must be defined: (1) `<XXX>Home` as a sub-interface of `EJBHome`, (2) `<XXX>` as a sub-interface of `EJBObject`, and (3) `<XXX>Bean` as an application class for implementing the `EntityBean`. The interface `<XXX>Home` adds application-specific `create` and `find` methods. The interface `<XXX>` should have application-specific business methods, which are denoted here by `<BM>`.

The class `<XXX>Bean` is a concrete bean definition that implements all of the methods specified by the `EntityBean` interface, as well as provides application-specific methods: `ejbCreate` and `ejbFind` are the counterparts of `create` and `find` of `<XXX>Home`, respectively. `<BM>` directly corresponds to `<BM>` of `<XXX>`. When a client invokes, for example, create method of `<XXX>Home`, the `Container` intercepts the request and then delegates it to `<XXX>Bean` instance in the form of an `ejbCreate` method invocation. Most of the methods defined in `EntityBean` are APIs for runtime-services. The passivation service uses `ejbPassivate` and `ejbActivate`, while the persistency service invokes `ejbLoad` and `ejbStore`.

The EJB 1.1 specification document describes the roles of the participants and all of the APIs in the form of Java interface descriptions. It is important to note that most of the methods are accompanied by exceptions in addition to normal functionalities. All explanations are written in a natural language.

2.2 Behavioural Specifications

Behavioural specifications show an external view of component architecture in terms of event sequences. With the EJB architecture, an event corresponds to a method invocation. Thus, the behavioural specifications consist of traces of method invocations. What follows shows some example descriptions of behavioural aspects adapted from the EJB 1.1 specification document.

Figure 4 is a lifecycle model of an Entity Bean (adapted from Figure 23 on page 102 [20]). The lifecycle consists of three states, and each method contributes to a transition between the states. For example, a business method invoked by a client can only be executed when the Entity Bean is in its `ready` state. In view of the lifecycle model, the passivation service is an event initiator that makes the Entity Bean move between the `ready` and `pooled` states by using `ejbPassivate` and `ejbActivate`.

In addition, the document employs Object Interaction Diagrams (OIDs) to illustrate typical event sequences. Figure 5 is an example of an OID for the Entity Beans (simplified and adapted from Figure 29 on page 139 [20])[2]. The diagram depicts various portions of event sequences that relate to a passivation service and business method invocation by a client.

[1] It is not shown in the figure.

[2] Division of labour between the two participants in a rectangle, +EJBObject+ and +Container+, is dependent on a particular implementation, and thus the rectangle may sometimes be treated as one entity (on page 62 [20]).

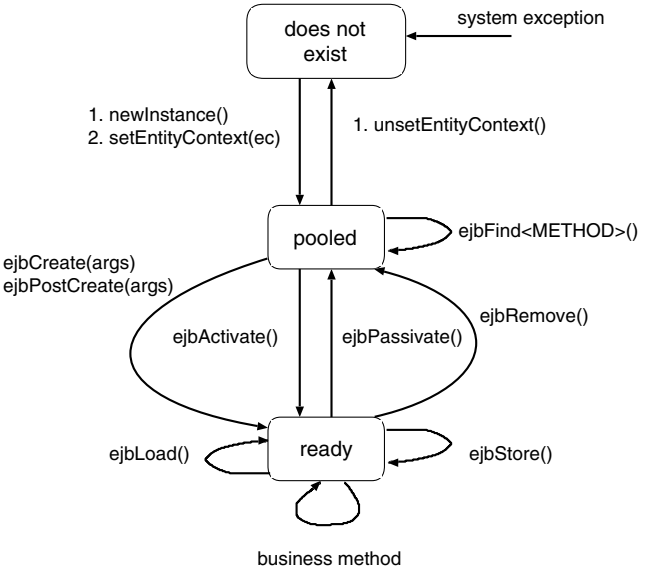


Fig. 4. Lifecycle of Entity Beans

A client request is directly delegated to the Entity Bean when the bean is in the `ready` state (see top of the diagram). The passivation service may spontaneously decide that the bean should be swapped out. This is accomplished through a sequence of method invocations; `ejbStore` followed by `ejbPassivate`. In Figure 4, it can be seen that the bean moves to the `pooled` state. At this point, a client may invoke another business method (see middle of the diagram). Since the bean is not in the `ready` state, the business method cannot be executed, and is thus suspended. The EJB server is then responsible for returning the bean to the `ready` state by issueing `ejbActivate` and `ejbLoad`. Finally, the suspended request is executed by the bean.

In addition to lifecycle models, the EJB 1.1 specification document describes behavioural aspects such as those described through OIDs. An OID, however, is just an example trace and is not a complete specification. One must infer from OIDs the intention of the original designer, as well as obtain precise specification descriptions from them. Furthermore, the document uses a natural language to describe temporal constraints related to particular runtime-service methods. For example, it says on p. 113 that the `Container`[3] invokes at least one `ejbLoad` between `ejbActivate` and the first business method in the instance. These constraints are used as sources for behavioural properties to be verified.

[3] It is identified here with the entity represented by the rectangle in Figure 5.

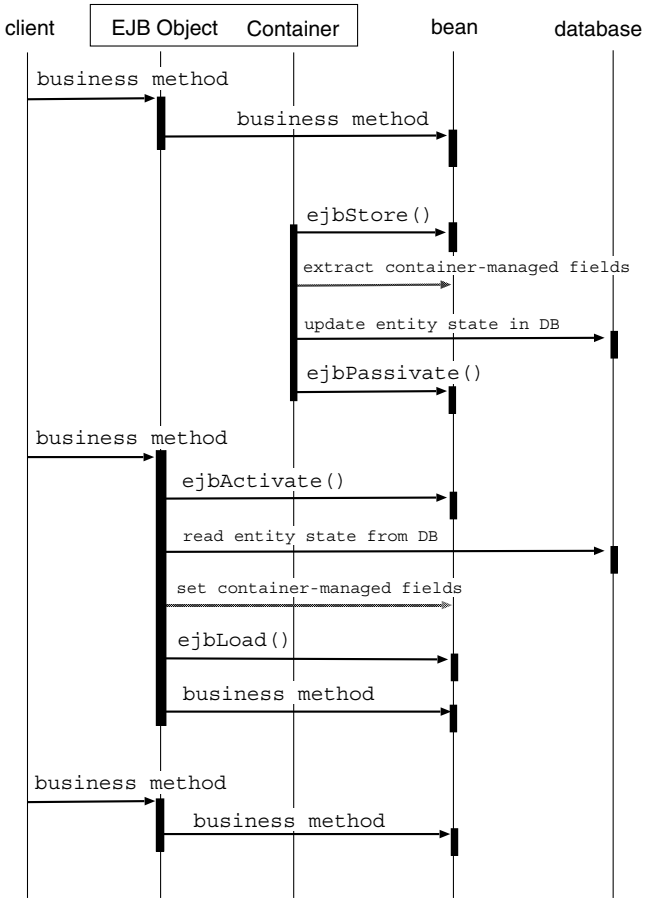


Fig. 5. Object Interaction Diagram

3 Formalization

This section focuses on the issues and approach related to formalizing the component architecture in Promela as compared with other specificands such as communication protocol and distributed algorithm [5] or program verification [6] for which Promela has been used successfully.

3.1 Issues and Approach

Since the specificand, the EJB framework, has different characteristics from those that have been formalized and analyzed in Promela/SPIN, there are several issues that must be considered before formalization. First, the original specification is centered around APIs, and an API defines exceptional cases as well as

a normal functionality[4]. Each API is a Java method accompanied by possible application-specific and system exceptions.

Second, one cannot predetermine the behaviour of clients that access bean instances. The EJB framework must show valid behaviour regardless of the client. In particular, some clients may terminate, while others may not.

Third, since behavioural specification is based on event sequences, Linear-time Temporal Logic (LTL) formulae to represent properties should involve atomic propositions that describe the occurrence of particular events, method invocations. In contrast, validity of atomic propositions in LTL is determined by states. This makes it necessary to devise a way to encode an event occurrence.

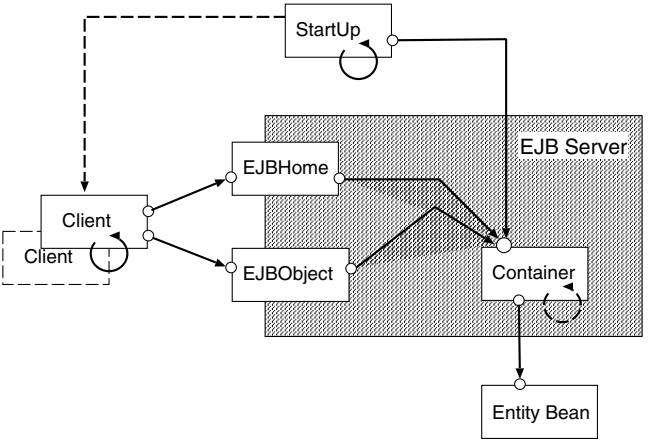


Fig. 6. Promela Processes

Figure 6 shows the overall configuration of the Promela processes. Three processes with a cyclic arrow are event initiators. The `Client` process issues a request such as create or business method, and is thus considered to be an event initiator. The role of `Container` is dual. It accepts events from other processes as well as generates events that implement runtime services such as passivation or persistency. The `StartUp` process needs further explanation. The `StartUp` process is responsible for setting up the initial environment necessary for a `Client` and a `Bean` to be executed properly. For example, Entity Beans are not required to be created by a client, but are created in advance. The `StartUp` process for such a case is responsible for creating and initializing the Entity Beans. Thus, introducing the `Client` and `StartUp` processes as event initiators is an approach that can be used to deal with the second issue above. This approach also contributes to creating a well-structured formal model because the

[4] The Wright group has observed similar characteristics in the cases [2][18].

functionality of the EJB server and processes for setting up verification environment are clearly separated. With respect to the first and third issues, one can make use of the Promela language constructs, which will be discussed in detail using actual Promela code in Section 3.2.

3.2 Promela Model

In order to describe method invocation on an object and return from the method, one first introduces two communication channels between the caller and calleé processes. And an extra channel is defined for transporting possible exceptions from the object.

```
#define  NBUF  1
chan mthd = [NBUF] of { short };
chan retv = [NBUF]  of { short };
chan excp = [NBUF] of { short };
```

The `chan mthd` is used for invoking method and messages flow from the `Container` to the `EntityBean`. The other two, `retv` and `excp`, are for messages going in the opposite direction. The channel definitions assume that the method name and the values are `short` and properly `#define`d.

The `EntityBean` in Figure 6 is represented by a simple Promela process that waits for method invocation from the `mthd` channel. The fact that any method can be invoked at any time is expressed using a `do...od` statement with an `endLoop` label. Each entry in `do...od` corresponds to a method, and its body is just an `if...fi` statement, which in turn has more than one branch. Each branch corresponds to either an exception or a normal termination. Since the `if...fi` has channel send statements in its guard positions, any of the branches can be executed at any time, which simulates a non-deterministic choice between exceptions and normal terminations. Therefore, one can encode exceptions in the Promela model, which resolves the first issue in Section 3.1 in a compact manner.

```
#define  ejbActivate   106
#define  ejbPassivate  110
#define  BM            120
...

proctype EntityBean ()
{
endLoop:
 do
  :: mthd?ejbActivate -> if :: retv!Void  :: excp!SysError fi
  :: mthd?ejbPassivate -> if :: retv!Void  :: excp!SysError fi
  :: mthd?BM -> if :: retv!Value :: excp!AppError :: excp!SysError fi
  ...
 od
}
```

The `EJBObject` in Figure 6 is an example process that terminates. It is a proxy object for accepting requests from the client, and delegating them to the `Container`, and it ends its lifecycle after handling of `remove` method by the client. Since an `EJBObject` is capable of accepting requests from more than one client[5], it must distinguish between each of the clients' requests. In the Promela description below, the `remote` channel carries two channels as well as a method name as its formal parameters. By sending different channel parameter values each time, it is possible to simulate a situation in which each message sent corresponds to a different method invocation event.

```
chan remote = [NBUF] of { short, chan, chan };
chan retval[NC] = [NBUF] of { short };
chan except[NC] = [NBUF] of { short };

proctype EJBObject()
{
   chan returnValue;   chan exceptionValue;   short value;

progressLoop:
endLoop:
 do
  :: remote?remove,returnValue,exceptionValue
    -> { request!reqRemove; retvalFC?value
       -> returnValue!value; goto endTerminate }
     unless { exceptFC?value -> exceptionValue!Error; goto endTerminate }

  :: remote?BusinessMethod,returnValue,exceptionValue
    -> { request!reqBM; retvalFC?value -> returnValue!value }
     unless { exceptFC?value -> exceptionValue!Error }
  od;

endTerminate:
   skip
}
```

The `Client` in Figure 6 is a Promela process that accepts two channel parameters when it runs. Since it invokes a request either on a `Home` or `Remote`, the `Client` is an event initiator that always sends messages via, for example, the `remote` channel, and waits for completion of the invoked method execution.

```
proctype Client(chan ret; chan exc)
{
  ...
MainLoop:
    do
     :: remote!BusinessMethod,ret,exc ->
```

[5] More precisely, the +EJBObject+ for Session Bean should be ready for multiple requests while the one for Entity Bean need not.

```
        if :: ret?value -> skip  :: exc?value -> goto endClient fi
    :: remote!remove,ret,exc ->
        if :: ret?value -> goto wrapUp :: exc?value -> goto endClient fi
    :: home!remove,ret,exc;
        if :: ret?value -> goto wrapUp :: exc?value -> goto endClient fi
   od;
  ...
}
```

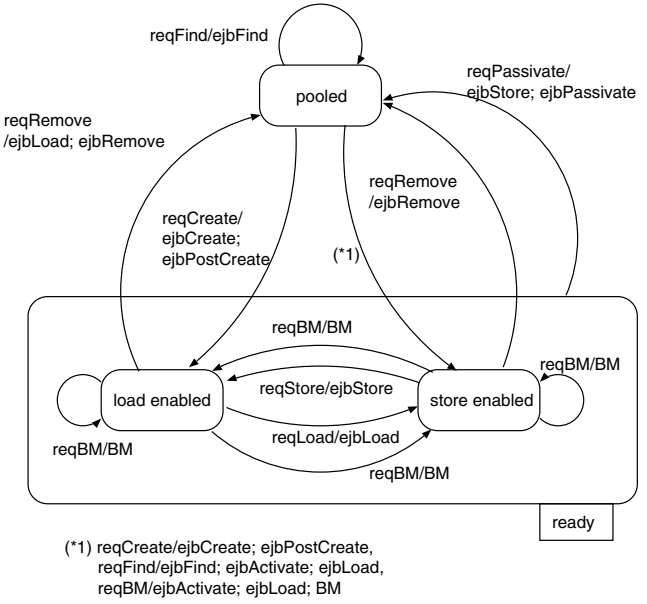


Fig. 7. Container for Entity Beans

As for the `Container` process, Figure 7 shows only the main behaviour in terms of the state transition diagram a lá Statechart [14] because the Promela code is too lengthy to be shown here[6]. The diagram is mainly derived from the lifecycle model in Figure 4, and appropriately augmented by studying the descriptions in other parts of the original document. The `ready` and `pooled` states in the diagram correspond to the states with the same labels in Figure 4; however, the `ready` state is modeled as a super-state with two sub-states. Such elaboration is needed for a proper interpretation of potential interference related to the persistency service, which was found necessary during the formalization process.

[6] It is some 250 lines of Promela codes.

The diagram shows all of the necessary transitions for implementing the main behaviour of the `Container`. For example, a transition from `pooled` to `load enabled` is notated as

```
reqCreate / ejbCreate;ejbPostCreate,
```

which indicates the following behaviour: when the `Container` receives a `reqCreate` from the `EJBHome`, the `Container` invokes two consecutive methods, `ejbCreate` and `ejbPostCreate`, on the `EntityBean`[7].

Finally, one goes back to the third issue in Section 3.1, which is related to describing the occurrence of a particular event as an atomic proposition in LTL formulae. In the above description of `EntityBean`, one can observe that the process is, at a certain time, in a state in which it is ready to receive a particular event. When, for example, an `ejbActivate` message is at the head of the `mthd` channel, the `EntityBean` process can be considered to be in a state where `mthd?ejbActivate` is executable. The condition can be checked by an expression `mthd?[ejbActivate]` because it becomes true if a message `ejbActivate` is at the head of the chanel `mthd`. Therefore, one may use the square brackets notation as a required atomic proposition representing an occurence of a particular event. One should also be very careful in stating formulae to be checked in order to ensure that an executable event cocincide with the next event occurred. For the formulae defined for events occurring on the same `EntityBean`, this is automatically satisfied because the corresponding messages are queued in the buffered channel `mthd` in an FIFO manner.

The SPIN feature for automatic translation of LTL formulae can also be used. For example, as described in the last paragraph of Section 2.2, the EJB specification requires that the `Container` invokes at least one `ejbLoad` between `ejbActivate` and the first business method in the instance. To verify the property, the SPIN can be used to generate a *never automaton*, where the process description needs appropriate `#define`s.

```
spin -f "! [](q8 && <>q2) -> (! q2 U q12))"

#define  q2  mthd?[BusinessMethod]
#define  q8  mthd?[ejbActivate]
#define  q12 mthd?[ejbLoad]
```

Note that the LTL formula is *negated* because the SPIN model checker handles negative claims in the form of never automaton for verification.

4 Behavioural Analysis

This section presents some concrete examples of behavioural properties and the results of analysis.

[7] As can also be seen in Figure 4.

4.1 Entity Beans

As discussed in Section 3.1, an appropriate client process is necessary for analyzing the behaviour of the EJB server. Three varieties of clients are formulated. Their behaviour can be compactly expressed in terms of event sequences generated by the client. (1) client-1 is a standard client that generates $\{create, find\}$; BM^*; $remove$, (2) client-2 starts with a find method ($find$; BM^*; $remove$), and (3) client-3 does not end with remove ($\{create, find\}$; BM^*). The individual client uses a somewhat different `StartUp` process because each run needs a different event sequence for setting up the environment to successfully start the client. A standard checking command for deadlock freedom (`run -q`) is executed on each client model, and all succeed.

Next, various properties, formulated in terms of LTL formulae, are verified against the standard client-1. The first property of interest is the behaviour of invoking a business method (BM) on a passivated Entity Bean (Figure 4), and can be validated by the following two LTL formulae. When an Entity Bean instance is in the `pooled` state and a client requests a BM on the instance, `ejbActivate` is eventually executed (E1). The formula (E2) states that an `ejbLoad` is invoked between the `ejbActivate` and the BM on the instance.

$$\text{(E1)}\quad \Box(pooled \wedge BM\_Client \rightarrow \Diamond ejbActivate)$$

$$\text{(E2)}\quad \Box(ejbActivate \wedge \Diamond BM \rightarrow (\neg BM\ \mathbf{U}\ ejbLoad))$$

In a similar fashion, the EJB server invokes `ejbStore` between the last business method (`BM`) and `ejbPassivate`. (E3) is also satisfied.

$$\text{(E3)}\quad \Box(BM \wedge \Diamond ejbPassivate \rightarrow (\neg ejbPassivate\ \mathbf{U}\ ejbStore))$$

The EJB server must obey a set of "trivial progress" properties. The properties can be compactly expressed as (E4), showing that a client request M *leads to* an actual method invoked on an instance.

$$\text{(E4)}\quad \Box(M\_Client \rightarrow \Diamond M\_Bean)$$

At first, this seems to be of no interest at all because such properties are trivially satisfied. However, they actually reveal several interesting characteristics of the EJB server.

In the case of the client's create request, in which M_Client is `create` and M_Bean is `ejbCreate`, the expected formula is satisfied. The `ejbCreate` method, however, should be accompanied by an `ejbPostCreate` (Figure 4), and thus the property to be checked should be (E5).

$$\text{(E5)}\quad \Box(create \rightarrow \Diamond(ejbCreate \wedge \Diamond ejbPostCreate))$$

The formula becomes false because there are situations in which `ejbCreate` raises a system exception and thus `ejbPostCreate` is not executed. This shows that there is some complexity in taking into account the possible occurrence of exceptions, which is mandatory for behavioural analysis of the EJB server.

The next property has to do with "spontaneous allocation" of a fresh bean instance. The EJB server has the liberty of keeping more than one behaviourally equivalent Entity Beans and allocating them for client requests. (E6) is a property to represent one such behaviour, which is found to be true.

$$\text{(E6)}\quad \Box(ejbRemove \rightarrow \Diamond(BM\_Client \rightarrow \Diamond BM))$$

Behaviour relating to a remove request is of particular interest because the trivial progress property (E7) does not hold.

$$\text{(E7)}\quad \Box(remove \rightarrow \Diamond ejbRemove)$$

Analyzing the output trace reveals that a possible *livelock* caused by an infinite iteration of `ejbStore` and `ejbLoad` prevents the EJB server from the expected progress. In addition, (E7) can be checked by setting the weak-fairness flag (`run -f -a`). This filters out situations that has the previous livelock. The formula, however, still fails because of potential interference between `ejbRemove` and `ejbPassivate`. To study the situation, please refer to Figures 4 and 7. The state-diagrams show that two transitions, `ejbRemove` and `ejbPassivate`, are possible from the `ready` to the `pooled` state. Also `ejbRemove` is initiated by a client `remove`, while `ejbPassivate` is an event from the runtime passivation service that is generated in an asynchronous way independent of any client requests. The identified situation is that the bean instance moves to the `pooled` state by `ejbPassivate` while a client explicitly requests a `remove`. And thus it does not lead to an occurence of `ejbRemove`. The interference is a potential flaw in the EJB 1.1 specification document [20] that can be seen only by a careful examination of (E7).

It is also possible to try a trivial progress for the case of business method (BM).

$$\text{(E8)}\quad \Box(BM\_Client \rightarrow \Diamond BM\_Bean)$$

The formula proves to be false because of a possible livelock of the `ejbStore` and `ejbLoad` as discussed in relation to (E7). Analysis under the weak-fairness condition also leads to a failure. This is because an internal exception occurs in the EJB server. For example, when a client issues a BM request on an instance in the `pooled` state, it involves more than one method execution (see E1 and E2). It is possible that one of the methods, for example `ejbActivate`, raises an exception. If this is the case, the EJB server cannot continue the service on the instance, and thus the property does not hold. If the property (E9), which takes into account the exception, is used, it is found to be successful under the weak-fairness condition.

$$\text{(E9)}\quad \Box(BM\_Client \rightarrow \Diamond(BM\_Bean \vee Exception))$$

As mentioned above, several trivial progress properties are not satisfied due to either possible exceptions or some other anomalous situation. However, it is easy to confirm that there exists at least one desirable sequence for satisfying each property. One may try a negation of a property that corresponds to a

desirable behaviour expressed in a deterministic manner. (E10) is an example case for a business method (BM).

$$\text{(E10)}\quad \neg(\Diamond(BM\_Client \wedge \Diamond BM\_Bean))$$

The SPIN protocol analyzer (`pan`) fails to show the correctness of the property and generates a sequence that leads to a failure. The sequence is exactly what one would expect. What is shown below is an edited output of the trace.

```
q\p   Container   EJBObject   Bean   Client
-------------------------------------------------
 13   .           .           .      remote!BM,6,8
 13   .           remote?BM,6,8
  1   .           req!BM
  1   req?BM
  3   mthd!BM
  3   .           .           mthd?BM
```

Therefore, it is true that at least one event sequence fulfills the trivial progress property for BM.

4.2 Session Beans

Session Beans have two options that can be specified at the time of deployment: STATEFUL or STATELESS. The two options are not very different to the bean developer; however, they exhibit quite different runtime behaviour. Thus, formalization and analysis of Session Beans are actually conducted to model two independent behaviours, although some common behaviours are effective for both models. As in the case of Entity Beans, one introduces a variety of clients with an adequate `StartUp` process for each client case.

Common Behaviour. One important property in terms of behavioural specification is in relation to concurrency control. According to the document, if a client request arrives at an instance while the instance is executing another request, the container must throw an exception for the second request. Session Beans must satisfy such a "non-reentrant" property. The property can be formulated in an LTL formula such as (S1).

$$\begin{aligned}\text{(S1)}\quad \Box(&Invoked\_1 \wedge \Diamond BM\_2 \\ &\wedge (\neg(Return\_1 \vee Exception\_1)\ \mathbf{U}\ Exception\_2) \\ &\rightarrow \Diamond(Return\_1 \vee Exception\_1))\end{aligned}$$

Some remarks on the formula are in order: (a) the server is in a state in which it has already accepted the first BM ($Invoked\_1$), (b) the second BM is requested afterward ($\Diamond BM\_2$), (c) the first BM is not completed before the second BM request ends with an exception ($\neg(Return\_1 \vee Exception\_1)\ \mathbf{U}\ Exception\_2$), and then (d) the first BM request results either in a normal termination or in an exception ($\Diamond(Return\_1 \vee Exception\_1)$).

As for the concurrency control, two varieties of clients are used to examine the behaviour: (1) client-s1 is a simple process for invoking only one BM, (2) client-s2 is an infinite process that generates BMs iteratively (BM^+). In addition, the standard check for deadlock freedom is executed on the following three runs: (a) two client-s1's are involved, (b) two client-s2's are involved, and (c) three clients-s2's are involved. All are found to be deadlock-free; however, the size of the searched state space is different for each run (Table 1)[8].

Table 1. Analysis of Non-reentrant Property

Run	♯States	♯Transitions	Depth
(a)	3,742	8,217	66
(b)	6,161	12,225	295
(c)	75,769	167,884	986

Although the state space for the present EJB server model is not large in comparison to other published cases of practical interest, reducing the size by an appropriate abstraction is still important for an efficient checking since the sizes drastically increase as in Table 1. To prove the property (S1), the case (a) was used because of the simplicity.

STATEFUL Session Beans. In addition to the three client models explained above, two other models are used: (1) *create*; BM^*; *remove*, and (2) *create*; BM^*. The first client model is a standard one used in most analyses. The second one, however, is mandatory for properties involving *timeout*. The EJB server must generate a *timeout* event for garbage-collecting of possibly unused STATEFUL Session Bean instances either in the `method_ready` or in the `passive` state. In the case of the `method_ready` state, the timeout event must be followed by an `ejbRemove` on the instance. From the `passive` state, on the contrary, instances just disappear without any further events.

The timeout makes the situation somewhat complicated for a STATEFUL Session Bean. For example, the trivial progress property for business method (BM) is not satisfied. This is because an instance is forced to leave the `ready` state due to a possible timeout event. The only alternative is to verify the property (S2) that takes into account the timeout situation.

$$\text{(S2)}\quad \Box(BM\_Client \rightarrow \Diamond(BM\_Bean \vee timeout))$$

STATELESS Session Beans. For a client model of STATELESS Session Beans, the simplest one capable of generating BM^* is sufficient. This is because

[8] The metrics are the case for a STATELESS Session Bean.

the EJB server is responsible for the creation and deletion of STATELESS Session Beans. A create request by a client becomes *no-op* for STATELESS Session Beans.

STATELESS Session Beans do not have any conversational state, and all bean instances are equivalent when they do not serve a client-invoked method. STATELESS Session Beans are similar to Entity Beans in that the EJB server will allocate a suitable bean instance even if one has already been removed. This is because a STATELESS Session Bean has a "spontaneous allocation" property that an Entity Bean has. This property can be confirmed by checking the LTL formula (E6) mentioned above.

A property specific to STATELESS Session Beans is "spontaneous removal or automatic garbage collection." STATELESS Session Beans do not allow an explicit remove operation by clients. The EJB server is responsible for deciding whether an instance is no longer necessary and invoking an `ejbRemove` method on it. In the present Promela model, the `Container` invokes `ejbRemove` in a non-deterministic manner. The LTL formula for the property is (S3), which says that `ejbRemove` is eventually invoked when the instance is in the `ready` state.

(S3) $\Box(ready \rightarrow \Diamond ejbRemove)$

The result is false due to a *livelock* of continuous BMs. The counter example sequence (livelock of BMs) is exactly the case in which the client accesses the bean heavily and thus the EJB server does not issue any `ejbRemove` on the bean. This confirms that timeout is generated only when the client is in a *think* time and does not invoke any BM at all. This is in accordance with the EJB 1.1 specification.

4.3 Discussion

The EJB 1.1 document [20] describes specifications from various viewpoints, thereby making it necessary to have a consistent model incorporating all of the viewpoints scattered throughout the document. As most of the descriptions are written in a natural language, the document uses many ambiguous "words." One specific example is related to the persistence service; invocations of the `ejbLoad` and `ejbStore` can be *arbitrarily* mixed with invocations of business methods. The intention of the author of the document is understandable, but, the word "arbitrarily" needs a concrete interpretation. The model in Section 3.2 adapts an interpretation that can be seen in the state-model in Figure 7. The model still leads to a livelock situation, which is revealed in the analysis.

Although the lifecycle model such as the one in Figure 4, can be used as a basis for formalizing the behavioural aspect, the model in the document is too simple. Reaching the final model in Figure 7 requires a thorough reading of the document and feedback from the analysis. This could be made easier if the EJB document had chapter(s) that concentrated on precise descriptions of behavioural aspects.

After obtaining a formal model, it is possible to conduct behavioural analysis in which properties are expressed in terms of LTL formulae. Thanks to numerous

literature on the use of LTL formulae [3][8][13], one finds it not hard (though not easy either) to formulate properties using LTL formulae. It, however, still requires a trial-and-error in formulating and checking the LTL formulae. This is partly because a naive *leads-to* property such as (E4) does not hold in nearly all cases. It was a surprise at first, but was found immediately from the fault traces that many interesting situations, either possible exceptions or other run-time anomalies, were not considered in the formulae. The trial-and-error process was a great help in understanding the behaviour of the EJB server. The SPIN model checker could be described as a *light-weight* design caclulator [16] used in iterative processes for refining the formal models.

Some LTL formulae were deemed to be false because of a possible livelock in the EJB server, some of which could be avoided by setting a weak-fairness flag. One must be very careful about properties proved under the fairness condition because a real system may incorporate a scheduler that behaves differently from what was assumed at the time of the analysis [4]. In the present study, however, the source of livelock is where the actual implementation should be taken care of. For example, a livelock with `ejbLoad` and `ejbStore` can be attributed to the present design artifact written in Promela, which is believed to be a faithful model in accordance with the presentation of the original document. This is understandable because the document does not mention anything about the implementation. As in proving (E7) and (E9), analysis under the fairness condition revealed other interesting situations, which are important to the present study. As shown in Section 4.1, proving (E7) results in a failure, which indicates that the bean instance moves to the `pooled` state by `ejbPassivate` even when a client explicitly requests a `remove`. The analyses have revealed potential flaws in the EJB 1.1 document.

In summary, most of the problems are identified in the formalization process in which a consistent model to integrate various aspects is obtained. Early stages of behavioural analysis can contribute to *debugging* the model as well as being a great help in understanding the specificand. Finally, note that the EJB 1.1 specification document [20] is not a design document. It is more or less abstract and lacks information detailed enough for use in software development. However, because the document is used as a *reference* for those involved in the EJB technology, more detailed description is mandatory. As the present study revealed with the formalization and analysis, improvements must be made in the EJB 1.1 specification document.

5 Comparisons

Behavioural analysis has been most successful in software architecture. Wright [1] and Darwin [12] are the two practical tools that have been applied to distributed software infrastructure of non-trivial complexities [2][11]. Comparing Wright and Darwin with the approach using the SPIN model checker in some degrees of detail is mandatory for discussing whether the SPIN model checker is an adequate tool for behavioural analysis of distributed software architecture.

Wright [1] follows the *Component-Connector* model of software architecture [17], which provides a general framework for describing various architecture styles from a unified viewpoint. To explain briefly, a *component* is a computational entity while a *connector* is an abstract infrastructure that *connects* the participating components. *Connector* is responsible for how the components exchange information, which basically describes the behavioural specification of the specificand architecture. In particular, Wright adopts Communicating Sequential Process (CSP) as a rigorous notation for describing the behavioural specifications of *connectors*. The concrete syntax of Wright can be considered as a syntax-sugaring of a large CSP process description in a structured manner.

CSP is the essence of Wright in terms of behavioural specification, and its formal analysis can be conducted by means of FDR (Failures/Divergences Refinement) [15], a model-checker for CSP. Since FDR is based on failure-divergence semantics, various properties such as deadlock freedom are checked through a refinement test ($\sqsubseteq$). Wright formulates several verification problems using its surface syntax. Using FDR to analyze behavioural aspects of software architecture requires some familiarity with failure-divergence semantics and how to express various properties in terms of the refinement relationship, thus making it less accessible for a wide audience. It is also not certain from the published works whether Wright can do behavioural analysis of systems with more than one connector. All examples in the papers use only one connector to model and analyze systems [1][2][18].

Sousa et al. [18] apply Wright to formalizing and analyzing the EJB component integration framework as defined in the EJB 1.0 specification. The entire specification of the EJB server, including the Container and two client-accessible proxies, is modeled as a single large connector. Thus, traceability between the original specification and the resultant formal model is weak. The present formal model written in Promela (Section 3.2) shows a more intuitive mapping between the two; an object in the original document is modeled as a Promela process.

Additionally, Sousa et al. identify a potential flaw relating to an interference between delegation of business method and `ejbPassivate`, as well as provide a remedy. They also discuss that the flaw may be due to their modeling but not to the EJB 1.0 specification. The same flaw, however, did not manifest itself in the present case study with Promela. On the contrary, the study in Section 4.1 identifies other flaws such as one relating to a potential interference between an execution of `remove` request and `ejbPassivate`.

Darwin [12] uses diagram notation for the structural aspects and FSP (Finite State Processes), a variant of CSP, to describe behavioural specifications, which can be analyzed by the LTSA (Labeled Transition System Analyzer) model checker. The basic model of Darwin is the *Component-Port* model, and does not have explicit notion of connector. The model is basically equivalent to the Promela model; a component and a port can be mapped to a Promela process and a channel respectively. Darwin, however, allows hierarchical models and components, which Promela does not support.

LTSA is a well-designed model-checker that can be used even by a novice. LTSA is easy to use, but restricts itself in terms of the power of verification. For safety analysis, a deterministic FSP process (property automaton) is used to

show *correct* behavior. LTSA model checks a product of the target and the property automaton. For liveness analysis, FSP provides a declarative way to specify a set of progress labels, which is equivalent to checking $\Box\Diamond q$ and $\Box(p \rightarrow \Diamond q)$ if expressed as LTL formulae, and is less expressive than the full LTL used in the SPIN model checker.

Sullivan et al. [19] formalize structural aspects of the COM model in the Z notation, and point out that there is a conflict between aggregation and interface negotiation in the original specification. This is a successful non-trivial result of applying formal methods to component architectures. Jackson and Sullivan [7] employ Alloy to formalize the model, which was originally formulated in the Z notation, and uses Alcoa, an automatic analysis tool, to show that the same flaws manifest themselves in the specification. Thus, they demonstrate the effectiveness of the automatic analysis tool. The present case study deals with the behavioural aspects of the EJB framework, and it does not deal with the structural ones. The use of both approaches may be necessary for analyzing advanced component architectures.

Kobryn [9] uses a UML Collaboration diagram to model component architectures. First, a pattern for component frameworks is introduced, and then the pattern is instantiated to the two important frameworks, EJB and COM+. Since the approach makes use of UML Collaboration diagram, the main concern is to illustrate the participant roles and structural relationships between the participants. Behavioural analysis is not conducted. The present case study concentrates on behavioural analysis using the SPIN model checker, but was limited to the EJB framework. However, it can be easily applied to other component frameworks such as COM+ because COM+ and EJB can be instantiated from a general pattern, as illustrated by Kobryn in his paper. Finally, the configuration of the Promela processes in Figure 6 is almost identical to that of Kobryn's pattern[9]. This ensures that the model in the present case study is *natural*, and thus shows a sufficient traceability with the original document.

6 Conclusion

This paper describes how the SPIN model checker was used for behavioural analysis of the Enterprise JavaBeans component architecture. This is a case study on applying a model-checking technique to a non-trivial real-world software artifact. Using concrete examples, the present work was able to demonstrate that the SPIN model checker can be used as an effective tool for behavioural analysis of distributed software architecture. Further, the case was also able to successfully identify several potential flaws in the EJB 1.1 specification document.

Finally, further work is needed on integration of the UML-based approach, for example, in [9] and the SPIN-based model that is amenable to automatic behavioural analysis. This is inevitable for the model-checking technology to gain a wide acceptance from software engineers.

[9] The authors did not know about the work in [9] before writing this paper.

References

1. Allen, R. and Garlan, D.: Formalizing Architectural Connection, Proc. ACM/IEEE ICSE'94 (1994).
2. Allen, R., Garlan, D., and Ivers, J.: Formal Modeling and Analysis of the HLA Component Integration Standard, Proc. ACM SIGSOFT FSE'98, pp.70-79 (1998).
3. Dwyer, M.B., Avrunin, G.S., and Corbett, J.C.: Patterns in Property Specifications for Finite-State Verification, Proc. ACM/IEEE ICSE'99 (1999).
4. Godefroid, P. and Holzmann, G.J.: On the Verification of Temporal Properties, in Proc. PSTV'93, pp.109-124 (1993).
5. Holzmann, G.J.: The Model Checker SPIN, IEEE trans. SE, vol.23, no.5, pp.279-295 (1997).
6. Holzmann, G.J. and Smith, M.H.: Software Model Checking: Extracting Verification Models from Source Code, Proc. FORTE/PSTV'99 (1999).
7. Jackson, D. and Sullivan, K.: COM Revisited: Tool-Assisted Modelling and Analysis of Complex Software Structures, Proc. ACM SIGSOFT FSE'00 (2000).
8. Janssen, W., Mateescu,R., Mauw, S., Frennema, P., and van der Stappen, P.: Model Checking for Managers, Proc. 6th SPIN Workshop (1999).
9. Kobryn, C.: Modeling Components and Frameworks with UML, Comm. ACM, vol.43 no.10, pp.31-38 (2000).
10. Leavens, G. and Sitaraman, M. (ed.): *Foundations of Component-based Systems*, Cambridge University Press 2000.
11. Magee, J., Kramer, J., and Giannakopoulou, D.: Analysing the Behaviour of Distributed Software Architectures: a Case Study, Proc. IEEE FTDCS'97 (1997).
12. Magee, J., Kramer, J., and Giannakopoulou, D.: Software Architecture Directed Behavior Analysis, Proc. IEEE IWSSD'98, pp.144-146 (1998).
13. Manna, Z. and Pnueli, A.: *The Temporal Logic of Reactive and Concurrent Systems: Specification*, Springer-Verlag 1991.
14. OMG: OMG Unified Modeling Language Specification v1.3 (2000).
15. Roscoe, A.W.: *The Theory and Practice of Concurrency*, Prentice Hall 1998.
16. Rushby, J.: Mechanized Formal Methods: Where Next?, Proc. FM'99, pp.48-51 (1999).
17. Shaw, M. and Garlan, D.: *Software Architecture*, Prentice Hall 1996.
18. Sousa, J. and Garlan, D.: Formal Modeling of the Enterprise JavaBeans^{TM} Component Integration Framework, Proc. FM'99, pp.1281-1300 (1999).
19. Sullivan, K., Marchukov, M., and Socha, J.: Analysis of a Conflict Between Aggregation and Interface Negotiation in Microsoft's Component Object Model, IEEE trans. SE, vol.25, no.4, pp.584-599 (1999).
20. Sun Microsystems, Inc.: Enterprise JavaBeans^{TM} Specification, v1.1 (1999).
21. Szyperski, C.: Components and the Way Ahead, in [10], pp.1-20 (2000).

p2b: A Translation Utility for Linking Promela and Symbolic Model Checking (Tool Paper)

Michael Baldamus[1] and Jochen Schröder–Babo

University of Karlsruhe
Institute for Computer Design and Fault Tolerance
Formal Methods Group
http://goethe.ira.uka.de/fmg

Abstract. p2b is a research tool that translates Promela programs to boolean representations of the automata associated with them. These representations conform to the input syntax of the widely–used symbolic model checker SMV; it is then possible to verify the automata with SMV, as opposed to enumerative model checking with SPIN, the classical Promela verifier. SMV and SPIN are focussed on verifying branching or linear time temporal properties, respectively, and often exhibit different performance on problems that are expressible within both frameworks. Hence we envisage that p2b will provide the missing link in establishing a verification scenario that is based on Promela as modeling language, and where one chooses different logics and verification methods as needed. The present paper provides an introduction to p2b, a description of how it works and two benchmark examples.

1 Introduction

An important ingredient of model checking is an expressive language that can be used for model description. Such a language must have a precise semantics, yet it must also be suitable for its application domain and easy to use. Promela [7], the input language of the SPIN model checker [8], is an asynchronous concurrent modeling language. It naturally does have a precise semantics and it arguably fulfills the other criteria too. SPIN then performs enumerative model checking of linear time temporal properties (LTL) over Promela programs. Two major optimizations realized by SPIN are on–thy–fly state space traversal and partial order reduction; they shorten runtime often dramatically. Murφ [5] is another well–known enumerative model checker that contains several optimizations of the basic procedure. Successful applications in various practical fields have shown how powerful enumerative model checking can be.

That success, however, does not come in all cases. A property may hold, for instance, meaning that it does *a priori* not help that on–the–fly traversals often find counterexamples without visiting every state that is relevant. Moreover,

[1] Michael Baldamus's work is supported by the Deutsche Forschungsgemeinschaft within the Project Design and Design Methodology of Embedded Systems.

M.B. Dwyer (Ed.): SPIN 2001, LNCS 2057, pp. 183–191, 2001.

partial order reduction may greatly reduce the number of relevant states, but the method requires specific preconditions with regard to the way processes communicate with each other: the efficiency gain suffers the more communication relationships violate those conditions.

Another method besides the enumerative one is symbolic model checking [4, 10]. Its basic idea consists of working with reduced ordered binary decision diagrams (BDDs, [3]) to represent finite automata and sets of states. The "secret" is partly that many systems with large state spaces can be represented with comparatively small BDDs; besides that, most algorithms on BDDs have moderate complexity. Hence it has been possible to verify practical examples whose state sets are astronomically large. The main applications of symbolic model checking have to date been in verifying synchronous digital hardware. There are, however, encouraging results on verifying also asynchronous and interleaved processes [6, 1,9,2], as they are typical for software–like systems. This situation was the reason for us to develop p2b. The objective was to perform symbolic model checking on such systems and, at the same time, to profit from Promela's versatility as a modeling language.

Efficient symbolic model checkers are readily available. The easiest way to achieve the objective of p2b is therefore to translate Promela programs to boolean representations of the automata associated with them, as symbolic model checkers usually understand this kind of input. More specifically, p2b generates code that conforms to the input syntax of the well–known and widely–used symbolic model checker SMV [10].

With SPIN, p2b and SMV, we have carried out various experiments. They indicate that enumerative and symbolic model checking may indeed exhibit rather different efficiency when applied to the automaton of one and the same Promela program. Sometimes SPIN is significantly faster, sometimes SMV. (cf. Section 3). Another possible benefit from p2b consists of the fact that symbolic model checking can be used to verify both branching and linear time properties, as opposed to the LTL world to which SPIN belongs. Hence we envisage that p2b will provide the missing link in establishing a verification scenario that is based on Promela as modeling language, and where one chooses different logics and verification methods as needed.

We have to mention that SMV starts the actual model checking procedure strictly after it has built the BDD that represents the automaton of the model under consideration. For this reason, it may be somewhat difficult to represent a dynamically evolving system of concurrent processes. Such systems, on the other hand, can easily be modeled with Promela with the help of the keyword `run`, which spawns a new process instance. p2b supports this feature only in so far as all process instances of a model — *proctype* instances in Promela terminology — can easily be determined before verification or simulation takes place. More specifically, every instantiation must refer to a proctype that is defined earlier in the program text and there must be no `run` within a loop or in the vicinity of a `goto`. To our experience, these restrictions still allow one to model many practical

systems, notably within the realms of embedded systems and communication protocols.

The remainder of the present paper is structured as follows: Section 2 describes the basics of how p2b works; Section 3 presents two benchmark examples, the dining philosophers problem and a mutual exclusion protocol over asynchronous channels; Section 4 briefly concludes the paper.

The p2b homepage is located at

`http://goethe.ira.uka.de/~baldamus/p2b`.

The package can be downloaded from a subpage there.

2 How p2b Works

p2b is a command line utility. It works in batch mode in the sense that a run consists of parsing a Promela program and generating ASCII output that conforms to the input syntax of the SMV model checker. The basic idea is to identify every proctype instance of the program. The automaton of each individual instance is described in isolation; by putting together these descriptions, the automaton of the program as a whole is described.

2.1 SMV Code Generated by p2b

Then the raw structure of the output will in general be as follows:

```
MODULE main

VAR
  ≪ declarations of current state variables ≫

INIT
  ≪ initialization of current state variables ≫

DEFINE
  ≪ boolean equations ≫

TRANS
  ≪ top expression ≫

SPEC
  ≪ temporal formula ≫
```

Only the `SPEC` part may be missing (see Section 2.1). In the sequel of this subsection, we briefly discuss each individual part.

Variables and Variable Initialization. If P is the program, then the output represents the automaton of P employing *current state variables* and *next state variables* to encode automaton states. First of all there are boolean variables that mostly correspond to control flow locations of the proctype instances of P but may also have auxiliary roles. Besides that, there may be *data variables*, which

correspond to data variables and channel entries in P. The ordinary, explicitly declared SMV variables of both kinds are the current state variables. For every such variable, say x, there is a unique next state variable, which appears in the SMV code as `next(`x`)`. p2b does not have to allocate any next state variable since SMV does that automatically. The current state variables are declared in the `VAR` part; their initial values are assigned in the `INIT` part. This assignment encodes the initial state of the automaton of P.

Boolean Equations. The variable declarations and initializations are followed by a `DEFINE` part. This part contains a collection of equations of the form *identifier* := *boolean expression*. Every right hand side may contain state variables or identifiers defined by other expressions. The collection of equations is essentially a bottom-up description of the automata of the proctype instances of P. To give an impression of that, let A be an active proctype in P that has k instances, $k \geq 1$, and let $l_1, \ldots, l_n$ be the control flow locations in A, $n \geq 1$. Then there are equations of the form $A\_i\texttt{-}l_j\texttt{\_enabled-} :=$ *boolean expression* for all $i \in \{1, \ldots, k\}$ and all $j \in \{1, \ldots, n\}$; they become true iff the corresponding control flow location is enabled. There are also equations of the form $A\_i\texttt{-at-}l_j :=$ *boolean expression* for all $i \in \{1, \ldots, k\}$ and all $j \in \{1, \ldots, n\}$; these ones describe the local and global effects of a transition of A that starts at the respective l_j, referring to $A\_i\texttt{-}l_j\texttt{\_enabled-}$ on the right hand side. Furthermore, there are equations of the form

$$A\_i\texttt{-} := \bigvee_{j=1}^{n} A\_i\texttt{-at-}l_j$$

for all $i \in \{1, \ldots, k\}$, describing the entire automaton of the respective instance of A. Besides that, there are equations of the form $A\_i\texttt{-idle-} :=$ *boolean expression* for all $i \in \{1, \ldots, k\}$; their role is to describe the idling of the corresponding instance of A if another proctype instance is active.

Top Expression. Apart from the equation system, there is a top expression, which puts all proctype instance automata together. This expression makes up the content of the `TRANS` part. To see what it basically looks like, let $pre_1, \ldots, pre_m$ be the identifier prefixes that correspond to proctype instances of P, $m \geq 1$. — An example of such a prefix is A_i, $i \in \{1, \ldots, k\}$. — Then the top expression has the form

$$\left(\bigvee_{i=1}^{m} \left(pre_i\texttt{-} \bigwedge_{j \in \{1,\ldots,m\} \setminus \{i\}} pre_j\texttt{-idle-} \right) \right) \vee \text{Term},$$

where Term is an expression that describes the idling of the entire system once every proctype instance has terminated. The idea is that a transition of the automaton of P, as long as it has not terminated, involves a transition of the

automaton of some proctype instance of P and, at the same time, leaves all other instances idle. This scheme works under the assumption that all channels have a capacity of at least one, meaning that there is no synchronization via channels. At the moment, p2b does indeed not support channels of capacity zero. To our experience, this restriction is not too severe in terms of what Promela models of practical systems can still be translated with p2b.

Optional Temporal Specification. SMV can read temporal formulas that are to be verified, so p2b also allows the user to include such a temporal specification in the Promela code as a specific pragma ignored by SPIN. The pragma must appear at the end of the input file, and it must be of the form

`/*p2b: SPEC` $\ll$ temporal formula $\gg$ `*/`.

The formula contained in it will appear at the end of the output file behind the keyword `SPEC`. Non–trivial temporal properties will usually be formulated with the help of the variables from the `VAR` part.

Complexity of the Translation. The complexity of the translation is quadratic in the number of proctype instances; the reason of that is the syntactic structure of the top expression in the `TRANS` part. The complexity of generating the output up to and including the `DEFINE` part is linear in the number of proctype instances.

2.2 Supported Constructs

p2b rejects every input rejected by SPIN. The current version of p2b *does not accept* every input accepted by SPIN either, since it does not support all Promela constructs. This situation is mostly due to the limited manpower that could be allocated to the p2b project; it is only to a small extent due to any principal difficulty in translating Promela in the way adopted for p2b, that is, by generating a boolean representation of the program automaton over current and next state variables. The constructs supported by the current version are as follows:

- **Data Types.** The supported data types are `bit`, `bool`, `byte`, `short` and `int`.
- **Channels and Variables.** Channels must have a capacity of at least 1 and must be global. Variables may be either global or local. The _–variable is supported.
- **Expressions.** p2b supports numeric constants, the usual boolean and arithmetic operators, bracketing, variable access and the `empty`, `nempty`, `full`, `nfull` and `?[..]`–operators for channel polling including the `eval` constraint in the case of `?[..]`.
- **Elementary Statements.** p2b supports `skip`, assignments, expressions that appear as statements, standard send and receive operations on channels

including `eval`, `goto`, the `xs`, `xw` and `xu` declarations, `printf` and `assert`. Among these statements, `xs`, `xw` and `xu` declarations and `printf` do not affect symbolic model checking, so they are treated like `skip`. `assert` is also treated like `skip`, as this statement runs somewhat contrary to the paradigm of breadth–first search used in symbolic model checking. If the functionality of `assert` is desired, then it should mostly be possible to use a temporal specification instead (see Section 2.1).

- **Statement Constructors.** p2b supports sequential composition, `if..fi`, `if..::else..fi`, `do..od`, `do..::else..od`, `atomic`, `unless` and `{..}`.
- **Labeling.** p2b supports labels wherever SPIN permits labels in Promela code.
- **Proctypes, `init` and `run`.** p2b supports parameterized proctypes, parameterized active proctypes with and without numeric instantiators, `init` and `run`. Active proctypes must not have channel parameters.
  Apart from the restrictions mentioned in the introduction, actual channel parameters occurring within `run` statements must be global channels; in other words, p2b does not support passing on formal channel parameters of an enclosing proctype.
- **`never`–Claims, `trace` and `notrace`.** p2b ignores any `never`–claim as well as `trace` and `notrace`. Temporal properties to be verified by SMV should be specified using the pragma described in Section 2.1.
- **`#define`.** p2b supports C–style macros.

2.3 Specifying Variable Ranges

Symbolic model checking is sometimes affected by the fact that the representation of data operations may lead to large BDD sizes. A prime example is multiplication, since in its case every BDD representation must be exponential in the width of the data path. For this reason, symbolic model checking may be greatly helped if it is known to what extent the data variables of the program are utilized. p2b allows the user to supply such information by means of pragmas that are ignored by SPIN. Such a construct has the form

`/*p2b:` ≪ smallest possible value ≫ `..` ≪ highest possible value ≫ `*/`

and may occur behind the types `byte`, `short` and `int`. Its effect is that p2b generates syntax that instructs SMV to allocate just enough BDD variables to accommodate the specified range.

3 Benchmark Examples

We have studied several examples with p2b. In each case we have verified a Promela program or a class of such programs with SPIN and — after translation — with SMV. This section reports on our results with regard to two scalable examples, the dining philosophers problem and a mutual exclusion protocol over asynchronous channels. We used SPIN V. 3.4.3 and Cadence Berkeley SMV V. 08-08-00p3 on an 800 MHz Pentium III processor under Linux with 700 MB of available RAM.

3.1 The Dining Philosophers Problem

The model of the dining philosophers problem represents philosophers as proctypes and chop sticks as channels of capacity one. As chop sticks can be considered passive items, we contend that this kind of model is natural. Due to the ring topology of the problem, every channel is then shared by two proctypes, which both read from and sent to the channel. Figure 1 shows measurements obtained from a solution without deadlock; the deadlock–preventing component is a dictionary, which is also represented as a channel of capacity one. All proctypes initially try to read from that channel or from a channel representing a chop stick. Only one proctype can succeed in reading from the channel representing the dictionary and it will write back to this channel before trying to read from any other one. Furthermore, the initial read from a channel representing a chop stick is guarded by the condition that the channel representing the dictionary be empty.

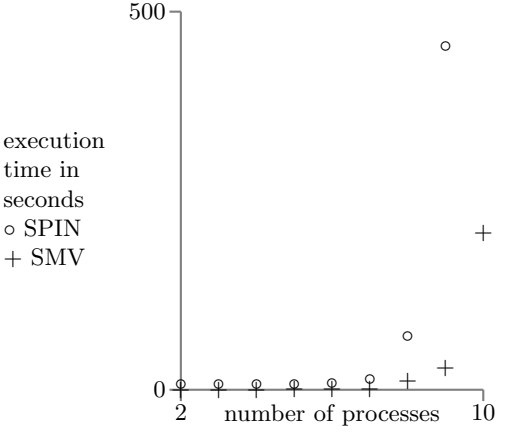


Fig. 1. Execution time measurements from verifying a deadlock–free solution to the dining philosophers problem. SMV was used via its graphical interface and the following options were set: Use heuristic variable ordering, Use modified search order, Restrict model checking to reachable states, Turn off transition relation clustering, Turn off conjunctively partitioned relations, Turn off partitioned relations. The other options of the graphical interface were not set.

The result of this experiment was that SMV was significantly faster than SPIN from seven philosophers onwards. Moreover, SPIN ran out of memory from ten philosophers onwards even when compression was turned on. This situation was probably due to the combination of two facts: first, the on–the–fly strategy could not bear fruit since the property to be verified always held; second, the exclusive send and the exclusive read condition with regard to channels are violated in all cases, so partial order reduction was less effective than usual.

3.2 A Mutual Exclusion Protocol over Asynchronous Channels

We have observed converse tendencies in the case of a mutual exclusion protocol over asynchronous channels (Figure 2). That protocol consists of n processes that communicate with an arbiter via channels a_i, b_i and c, $1 \leq i \leq n$, where each channel is of capacity one. A process P_i sends its request for entering a critical section to the arbiter via a_i; the arbiters elects one such process, say P_j, and sends it a grant for entering the critical section via b_j. That process sends a notification via c to the arbiter once it has left the critical section.

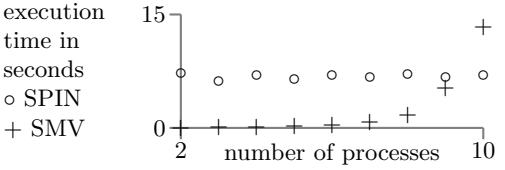


Fig. 2. Execution time measurements from verifying a mutual exclusion protocol over asynchronous channels.

As shown by the figure, SPIN's execution time remained virtually constant over the scaling range considered in the experiment. SMV's execution time, by contrast, was significantly higher for ten processes and showed a clear tendency to staying so for larger numbers of processes. This situation was probably due to the fact that nearly all communication relationships in the model satisfy the exclusive send or the exclusive read condition, meaning that partial order reduction could be effective.

4 Conclusion

The preceding sections have given an introduction to the objective of p2b, which consists of linking Promela and symbolic model checking. It was also described how p2b works and two scalable benchmark examples were presented. From the first example, we conclude that it indeed makes sense to supplement enumerative model checking of Promela programs with symbolic model checking; from the second example, however, we conclude that p2b will not entail that enumerative model checking is replaced by symbolic model checking.

As for future work, it would of course be desirable to extend the range of Promela constructs supported by p2b. Another topic might consist of incorporating results on the combination of partial order reduction and symbolic model checking [1,9].

References

1. R. Alur, R. Brayton, T. Henzinger, S. Qadeer, and S. Rajmani. Partial–Order Reduction in Symbolic State Space Exploration. In *Computer–Aided Verification*, pages 340–351. Springer–Verlag, 1997. Proceedings CAV '97.
2. M. Baldamus and K Schneider. The BDD Space Complexity of Different Forms of Concurrency, 2001. Accepted for ICACSD '01.
3. R. Bryant. Graph–Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, C–35(8):677–691, 1986.
4. J. Burch, E. Clarke, K. McMillan, D. Dill, and L. Hwang. Symbolic Model Checking: 10^{20} States and Beyond. In *Logic in Computer Science*, pages 1–33. IEEE Computer Society Press, 1990. Proceedings LICS '90 symposium.
5. D. Dill, A. Drexler, A. Hu, and C. Han Yang. Protocol Verification as a Hardware Design Aid. In *Computer Design: VLSI in Computers and Processors*, pages 522–525, 1992. IEEE Conference Proceedings.
6. R. Enders, T. Filkorn, and D. Taubner. Generating BDDs for Symbolic Model Checking in CCS. In *Computer–Aided Verification*, LNCS 575, pages 203–213. Springer–Verlag, 1991. Proceedings CAV '91 conference.
7. G. Holzmann. *Design and Validation of Computer Protocols*. Prentice Hall, 1991.
8. G. Holzmann. The Model Checker SPIN. *IEEE Transactions on Computer Engineering*, 23:279–295, 1997.
9. Kurshan, R. and Levin, V. and Peled, D. and Yenigün, H. Static Partial Order Reduction. In *Tools and Algorithms for the Construction and Analysis of Systems*, LNCS 1384, pages 345–357. Springer–Verlag, 1998. Proceedings TACAS '98 conference.
10. K. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

Transformations for Model Checking Distributed Java Programs*

Scott D. Stoller and Yanhong A. Liu

Computer Science Dept., SUNY at Stony Brook, Stony Brook, NY 11794-4400

Abstract. This paper describes three program transformations that extend the scope of model checkers for Java programs to include distributed programs, *i.e.*, multi-process programs. The transformations combine multiple processes into a single process, replace remote method invocations (RMIs) with local method invocations that simulate RMIs, and replace cryptographic operations with symbolic counterparts.

1 Introduction

There is growing interest in model checking of programs written in standard programming languages. Java's support for remote method invocation (RMI), an object-oriented version of remote procedure call (RPC) [BN84], makes writing distributed programs relatively painless. The current generation of model checkers for Java programs, *e.g.*, [BHPV00,PSSD00,Sto00,CDH+00], work for multi-threaded single-process programs but not distributed programs. This paper describes three program transformations that extend the scope of these model checkers to include distributed programs, *i.e.*, programs that involve communication among multiple processes.

Centralization: merge all processes into a single process. This yields a non-distributed program.

RMI removal: replace RMIs with ordinary method invocations that simulate RMIs.

Pseudo-crypto: replace cryptographic operations with symbolic counterparts, which we call *pseudo-cryptographic* operations.

All three transformations improve performance of model checking. Centralization avoids the overhead of initializing multiple processes and Java Virtual Machines (JVMs). RMI removal replaces genuine RMIs with faster simulated RMIs. Pseudo-crypto replaces computationally expensive cryptographic operations, such as generation of public-private key pairs and verification of digital signatures, with symbolic counterparts that are at least an order of magnitude faster.

* This work is supported in part by NSF under Grant CCR-9876058 and by ONR under grants N00014-99-1-0132, N00014-99-1-0358, and N00014-01-1-0109.
{stoller,liu}@cs.sunysb.edu http://www.cs.sunysb.edu/~{stoller,liu}

M.B. Dwyer (Ed.): SPIN 2001, LNCS 2057, pp. 192–199, 2001.

RMI removal and pseudo-crypto eliminate calls to native methods. Standard implementations of RMI invoke native methods for serialization (*i.e.*, conversion of data into a format suitable for network transmission) and network communication. Cryptographic operations are typically implemented using native methods for efficiency. The simulated RMIs and pseudo-cryptographic operations introduced by RMI removal and pseudo-crypto, respectively, do not invoke native methods. This is significant, because supporting native methods in a model checker is a non-trivial task. Furthermore, native methods for network communication maintain state outside the JVM. Supporting such native methods in a model checker is extremely problematic, because there is no general way for the model checker to save and restore their state.

Centralization enables current model checkers for Java to be used to systematically test and verify distributed programs. It also suggests that there may be little incentive to extend those model checkers to directly handle distributed programs. Centralization can be used without RMI removal, because a single process may be both the client and the server in a RMI.

Centralization by itself does not attempt to control non-determinism from scheduling or other sources. Centralization is particularly useful in conjunction with tools that do provide some control. This includes debuggers as well as model checkers. For example, consider using the Java debugger jdb to debug a distributed program. jdb supports breakpoints, but a breakpoint halts only a single process. If jdb is used to debug the centralized program, then the breakpoint halts the entire system, which is often what the user wants.

Many distributed programs are designed to work over an insecure network, such as the Internet, and therefore use cryptography. During model checking, such programs are usually executed together with a program that simulates an adversary that controls communication over the insecure network. Cryptography causes a special problem for model checking, in addition to the issues of performance and native methods mentioned above. Specifically, if the program sends actual ciphertexts and does not help the adversary program determine their contents, then the adversary program would be extremely inefficient. For example, to determine whether it can decrypt an intercepted ciphertext, it would have to attempt the decryption with every key it knows. This problem arises in testing, as well as model checking. Pseudo-cryptographic operations enable the adversary program to efficiently determine the contents of intercepted ciphertexts.

All three transformations rely on the assumption that the original program is not real-time and does not use reflection in a way that would detect the transformation's effects.

2 Centralization

We refer to programs produced by the centralization transformation as *centralized programs*. The transformed program is equivalent to the original program in the sense that it has essentially the same possible executions as the original program. More precisely, (1) there exists a refinement mapping f such that,

for each execution of the original program, there exists a stuttering-equivalent [Lam94] execution of the transformed program relative to f, and (2) *vice versa*.

The basic idea underlying centralization is simple. Suppose the original system consists of a process P_1 with three threads and a process P_2 with two threads. Then the centralized program creates five threads, three of which correspond to threads of P_1, and two of which correspond to threads of P_2. We assume that the original program does not count the total number of instances of `Thread` or `ThreadGroup` in the process, because this would detect the effect of the transformation.

Centralization involves four steps. The first step generates a driver class whose main method starts up the system. The driver is generated from a startup file supplied by the user, which contains (roughly) a list of command lines of the form "`java` $options_i$ $class_i$ $args_i$" used to start the processes of the original system. We refer to the process created by the i'th line of this file as process i. We assume that these processes can run on a single host, *i.e.*, different hostnames are not hardwired in the code. We currently do not support dynamic creation of processes, though it would not be difficult. The main method of the driver class creates, for each i, a thread that executes the main method of $class_i$ with arguments $args_i$.

The second step deals with static fields (*i.e.*, fields that are associated with a class rather than an instance of a class). In the original system, each process has its own copy of each class (the copy is created when the class is loaded by the JVM) and therefore its own copy of each static field. In the transformed system, there is only one copy of each class. The transformation introduces arrays to simulate the effect of having multiple copies of the class. For example, suppose the program uses a class C that contains a static field x of type T. The transformed version of class C declares a static field of type T`[]`, for an array whose elements have type T. Threads that correspond to threads of the i'th process access only C`.`x`[`i`]`. To allocate and initialize the arrays, we transform class initialization code—`<clinit>` and methods invoked directly or indirectly from `<clinit>`—appropriately.

Instructions that access static fields are transformed to access the appropriate element of the array. The index into the array is the number of the "process" to which the thread belongs. This value cannot easily be maintained in a global variable, because the JVM does not provide hooks that would invoke user-supplied code at every context switch. To determine this value efficiently, the transformation replaces all uses of `Thread` in non-library classes (*i.e.*, classes not in the Java 2 API) with `CentralizedThread`.[1] Class `CentralizedThread` extends (*i.e.*, inherits from) `Thread`, declares an instance field `int procNum`, and overrides the constructors of `Thread` with constructors that initialize `procNum` appropriately. When transforming an access to a static field, the index into the array is the value of the `procNum` field of the current thread; specifically, it is `((CentralizedThread)Thread.currentThread()).procNum`.

[1] Recall that, in Java, each thread is associated with an instance of class `Thread` or a subclass of `Thread`.

The third step deals with static synchronized methods, *i.e.*, methods that are associated with a class rather than an instance of a class and whose bodies implicitly start with an acquire operation on the lock associated with the class and end with a release operation on that lock.[2] In the original system, each process has its own copy of each class and the associated lock. To simulate this, for each class C that declares static synchronized methods, the transformation introduces a static variable `C.locks` that points to an array of new objects, and, for each static synchronized method $C.m$, it inserts an acquire on `C.locks[((CentralizedThread)Thread.currentThread()).procNum]` at the beginning of $C.m$, inserts a matching release at the end of $C.m$, and marks $C.m$ as not synchronized.

The fourth step deals with the method `System.exit`, which terminates the process. In the transformed program, `System.exit` should terminate only threads with the same value of `procNum` as the invoker. Java does not directly provide a mechanism for one thread to terminate another thread. Transforming the program to incorporate such a mechanism is non-trivial. Currently, we transform calls to `System.exit` so that they throw `java.lang.ThreadDeath`, which should terminate the calling thread. This is correct if all other threads with the same value of `procNum` have terminated; this condition could easily be checked dynamically.

The current implementation does not transform static fields of library classes.

Centralization is independent of the communication mechanisms used in the program. It could be used in conjunction with a socket removal transformation as well as RMI removal.

3 RMI Removal

Java RMI works roughly as follows. A process, called a server, makes an object available to other processes, called clients, by registering the object in the RMI registry, which is a simple database that maps strings (names of services) to objects. A client locates a remote object by looking up a service name in the RMI registry. A successful lookup creates a new object o_{stub} in the client. o_{stub} is called a *stub* and contains the address of a server S and a reference to an object o in S. The stub is an instance of an automatically generated class, called a *stub class*. The stub class for class C is named `C_Stub`. A *remote reference* is a reference to a stub. For each remotely invokable method m of C, the automatically generated method `C_Stub.`m on the client serializes its arguments *args* and sends them to server S; S unserializes the arguments, executes $o.m(args)$ in a new thread, serializes the return value (or exception), and sends it to o_{stub} on the client; the client unserializes the return value (or exception) and uses it as the result of the RMI. As an optimization, the JVM may maintain a pool of re-usable threads, rather than creating a new thread for each RMI. We do not describe here how interfaces are used to indicate which methods are remotely invokable; our transformation handles this aspect easily.

[2] Recall that, in Java, each class and each object implicitly contains a unique lock.

RMI removal replaces a RMI with an ordinary method invocation that simulates the RMI. The semantics of method invocation in Java is call-by-value for primitive data, and call-by-reference for remote references and ordinary references. The semantics of RMI is different, because serialization followed by unserialization effectively performs copying. Specifically, the semantics of RMI is call-by-value for primitive data, call-by-reference for remote references (although the stub object is copied, the copy refers to the same remote object, not to a copy of the remote object), and call-by-deep-copy for local references. "Deep copy" means that the entire subgraph of the heap reachable from the arguments is copied; the copy is isomorphic to the original subgraph. In all cases, the semantics for passing return values is the same as for passing arguments.

The transformed program uses a simulated RMI registry. Currently, the simulated RMI registry expects to find stub classes in the `CLASSPATH`.

Which thread should execute a remote invocation? To ensure a faithful simulation of RMI, the transformed code could create a new thread to execute each RMI. This is easy to implement but inefficient and typically unnecessary, in the sense that most applications are insensitive to the identity of the thread that handles the RMI. Maintaining a pool of re-usable threads is not as easy to implement. In our current implementation, the calling thread executes the "remote" invocation; we assume that the application does not detect this difference. While the calling thread is executing a remote invocation of a method of an object o, the thread's `procNum` should be set to the number of the server that created o, because that is the number of the process on which the method would be executed in the original system. This requires an efficient mechanism for determining which process created each instance of each class with remotely-invokable methods. Accordingly, we insert a field `procNum` in each such class C and modify each constructor for C to initialize that field with the current thread's `procNum`.

Implementing copying using reflection is tempting, but reflection uses native methods, and we want to eliminate uses of native methods, so copying is implemented as follows. The transformation identifies classes whose instances might appear in arguments or return values of RMIs. For each such class C, it generates a method named C`.copyRMI`. Method C`.copyRMI` has a parameter `h` that indicates which objects have already been copied; it is used to ensure that the original subgraph and the copy are isomorphic. C`.copyRMI(h)` returns `this` if `this` is a remote reference,[3] and returns a deep copy of `this` if `this` is a local reference. In the latter case, C`.copyRMI(h)` starts by checking whether `this` is in the hash map `h`. If so, `this` has already been copied, so C`.copyRMI(h)` finds the copy using `h` and returns it. Otherwise, C`.copyRMI(h)` creates a new instance o of C, adds the mapping `this` $\mapsto o$ to `h`, copies from `this` to o the value of each field of C with primitive type, recursively invokes `copyRMI` on the value of each field of C with non-primitive type and stores the result in the corresponding field of o, and returns o. Creation and initialization of o require special treat-

[3] A more faithful alternative would be to clone (*i.e.*, create a shallow copy of) the stub, but the identity of a stub should not be significant to a normal application, so this cloning would be unnecessary overhead.

ment when C has a non-serializable superclass other than `java.lang.Object`; we omit details of how the transformation handles this.

The transformation also generates stub classes. The standard stub classes produced by the compiler are not used by the transformed program, so we reuse their names for our stub classes. Thus, the stub class generated for class C is named C`_Stub` and declares an instance field `target` with type C, which refers to the object registered by the server. We still say that a reference to an instance of a stub class is a "remote reference". The generated method C`_Stub.`$m(args)$ creates new hashmaps `hArgs` and `hRet`, invokes `copyRMI(hArgs)` on arguments that are local references, sets the current thread's `procNum` to `this.target.procNum`, and invokes `this.target.`m on a combination of original arguments (that are not local references) and copies of arguments (that are local references); when the invocation of m returns, C`_Stub.`$m(args)$ restores the previous value of the current thread's `procNum`, invokes `copyRMI(hRet)` on the return value v of `this.target.`m if v is a local reference, and returns v (if v is not a local reference) or the copy of v (if v is a local reference) to the caller.

To efficiently check whether a reference is remote, the transformation introduces an interface `StubInterface`. All stub classes implement `StubInterface`. Thus, (`r instanceof StubInterface`) is true iff r is a remote reference.

Java RMI allows the user to specify a security policy that controls which remote methods may be invoked by which clients. Currently, we do not simulate checking of security policies; in effect, we assume that all RMIs performed by the original program are permitted by the security policy.

4 Pseudo-Cryptography

`java.security` provides a standard API for cryptography libraries. We assume that the original program uses this API. The original program is transformed by replacing all occurrences of `java.security` with `PseudoCrypto`; for example, `java.security.Signature.sign` is replaced with `PseudoCrypto.Signature.sign`.

The central issue in designing `PseudoCrypto` is the representation of ciphertexts. To solve the problem discussed in Section 1, the obvious approach is to use a symbolic representation. For example, the result of encrypting a plaintext t with a key k would be an object containing t and k. Given such a symbolic "ciphertext", the adversary program can trivially determine the key used to create it. This approach is standard in model checking of security protocols with traditional model checkers such as FDR and Murϕ [Low96,MMS97].

However, in the `java.security` API, ciphertexts are not an abstract data type. Ciphertexts are byte arrays, and this representation is visible to the application. Transforming the application to accommodate a different representation of ciphertexts would be difficult. Our approach is to maintain byte arrays and a symbolic representation. The symbolic representation is used only within `PseudoCrypto` and the adversary program; it is not visible to the application. A hash map is used to map the byte array representation of a ciphertext to its

symbolic representation. A mapping is inserted in the hash map every time a ciphertext is created. The adversary program looks up intercepted ciphertexts in the hash map.

It is the responsibility of the author of the adversary program to simulate an adversary that controls an insecure network. For example, the author must determine which cryptographic keys stored in the hash table are known to the adversary.

If the original cryptographic operations are computed in the transformed program, then this transformation does not affect the contents of the byte arrays seen by the application and hence does not affect the behavior of the application.

Computing cryptographic operations during state-space exploration is often impractical, because of performance and native methods, as discussed in Section 1. Therefore, `PseudoCrypto` computes "pseudo-cryptographic" operations that maintain the byte array and symbolic representations, as above, except the byte arrays are filled with pseudo-random data, not genuine ciphertexts. This change in the contents of the byte arrays could affect the behavior of the application, *e.g.*, if the application does not trust its cryptography library and therefore checks whether ciphertexts have the expected format. Typical applications treat the byte arrays as "atomic values", merely passing them around and then using them as arguments to other cryptographic operations. Such applications are not affected by this transformation. Currently, manual inspection of the original program is used to check whether this could happen. A conservative automatic check based on static analysis could be developed.

Currently, we assume that all operations that produce ciphertexts involve pseudo-random initialization, so invoking an operation twice on the same arguments produces different ciphertexts. Operations that are "functional" (*i.e.*, whose return value depends only on the arguments) could easily be accommodated using memoization [CLR90].

A separate issue is that the adversary program needs to efficiently determine which parts of intercepted "messages" might be ciphertexts. Currently, the author of the adversary program must deal with this. We are working on a static program analysis, based on points-to escape analysis [WR99], that aims to automate this and some other aspects of producing the adversary program.

Package `PseudoCrypto` currently supports commonly used methods for generation of public-private key pairs and generation and verification of digital signatures. Support for symmetric-key cryptography can easily be added.

5 Implementation and Case Study

The implementation of all three transformations is mostly complete. The implementation transforms bytecode using the Byte Code Engineering Library (BCEL), formerly called JavaClass [Dah99]. We hope to incorporate these transformations into Bandera [CDH+00], which provides an excellent framework for model checking and the associated program analyses. The first public release of Bandera is expected soon.

The first major case study will be a secure and scalable distributed voting system, whose design is described in [MR98]. Students implemented that design in Java as a course project. We plan to model check it using the above transformations and Java PathFinder [BHPV00]. Java PathFinder is not yet available to us, but its first public release is imminent. We anticipate completion of this case study before SPIN 2001 and expect to present the results there.

Acknowledgments. Ziyi Zhou and Dezhuang Zhang implemented centralization. Srikant Sharma and Kartik Gopalan implemented RMI removal.

References

[BHPV00] Guillaume Brat, Klaus Havelund, Seung-Joon Park, and Willem Visser. Model checking programs. In *IEEE International Conference on Automated Software Engineering (ASE)*, September 2000.

[BN84] A. D. Birrell and B. J. Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2:39–59, February 1984.

[CDH+00] James C. Corbett, Matthew Dwyer, John Hatcliff, Corina Pasareanu, Robby, Shawn Laubach, and Hongjun Zheng. Bandera: Extracting finite-state models from Java source code. In *Proc. 22nd International Conference on Software Engineering (ICSE)*, June 2000.

[CLR90] Thomas Cormen, Charles Leiserson, and Ronald Rivest. *Introduction to Algorithms*. MIT Press and McGraw-Hill, 1990.

[Dah99] Markus Dahm. Byte code engineering with the JavaClass API. Technical Report B-17-98, Institut für Informatik, Freie Universität Berlin, 1999. Available via http://www.inf.fu-berlin.de/~dahm/JavaClass/.

[Lam94] Leslie Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.

[Low96] Gavin Lowe. Breaking and fixing the Needham-Schroeder public-key protocol using FDR. In *Proc. Workshop on Tools and Algorithms for The Construction and Analysis of Systems (TACAS)*, volume 1055 of *Lecture Notes in Computer Science*, pages 147–166. Springer-Verlag, 1996.

[MMS97] John C. Mitchell, Mark Mitchell, and Ulrich Stern. Automated analysis of cryptographic protocols using Murϕ. In *Proc. 18th IEEE Symposium on Research in Security and Privacy*, pages 141–153. IEEE Computer Society Press, 1997.

[MR98] Dahlia Malkhi and Michael Reiter. Secure and scalable replication in phalanx. In *17th IEEE Symposium on Reliable Distributed Systems (SRDS)*, pages 51–60, October 1998.

[PSSD00] David Y.W. Park, Ulrich Stern, Jens U. Skakkebaek, and David L. Dill. Java model checking. In *Proc. First International Workshop on Automated Program Analysis, Testing, and Verification*, 2000.

[Sto00] Scott D. Stoller. Model-checking multi-threaded distributed Java programs. In *Proc. 7th Int'l. SPIN Workshop on Model Checking of Software*, volume 1885 of *Lecture Notes in Computer Science*, pages 224–243. Springer-Verlag, August 2000.

[WR99] John Whaley and Martin Rinard. Compositional pointer and escape analysis for Java programs. In *Proc. ACM Conference on Object-Oriented Systems, Languages and Applications (OOPSLA)*, pages 187–206. ACM Press, October 1999. Appeared in *ACM SIGPLAN Notices* 34(10).

Distributed LTL Model-Checking in SPIN*

Jiri Barnat[1], Lubos Brim[1], and Jitka Stříbrná[2]

[1] Faculty of Informatics, Masaryk University Brno,
Botanická 68a, Brno, Czech Republic
{xbarnat,brim}@fi.muni.cz
[2] Department of Computer and Information Science, University of Pennsylvania
200 South 33rd Street, Philadelphia, PA 19104, USA
jitkas@saul.cis.upenn.edu

Abstract. In this paper we propose a distributed algorithm for model-checking LTL. In particular, we explore the possibility of performing nested depth-first search algorithm in distributed SPIN. A distributed version of the algorithm is presented, and its complexity is discussed.

1 Introduction

Verification of complex concurrent systems requires techniques to avoid the state-explosion problem. Several methods to overcome this barrier have been proposed and successfully implemented in automatic verification tools. As a matter of fact, in application of such tools to practical verification problems the computational power available (memory and time) is the main limiting factor. Recently, some attempts to use multiprocessors and networks of workstations have been undertaken.

In [UD97] the authors described a parallel version of the verifier Murφ. In their approach, the table of all reached states is partitioned over the nodes of the parallel machine, which allows the table to be larger than on a single node. The explicit state enumeration is then performed in parallel. For the model-checker SPIN [Hol97], a similar approach towards distributed reachability analysis was proposed in [LS99]. This distributed version of SPIN uses different ways to partition the state space than Parallel Murφ. Yet another distributed reachability algorithm was proposed in [AAC87], but has not been implemented. Other recent attempts to use distributed environment of workstations for parallel model-checking are [HGGS00,BDHGS00].

Unlike the original SPIN, the distributed algorithm proposed in [LS99] performs a non-nested, depth-first search of the state space. It carries out reachability analysis, however, it does not implement model-checking of LTL formulas. In this paper, we propose a distributed algorithm based on nested depth-first search, that model-checks LTL formulas. The basic idea is as follows. The algorithm starts to explore the state space in (not necessarily nested) distributed

* This work has been supported in part by the Grant Agency of Czech Republic grants No. 201/00/1023, and 201/99/D026.

M.B. Dwyer (Ed.): SPIN 2001, LNCS 2057, pp. 200–216, 2001.

depth-first manner, similarly to the distributed SPIN of [LS99]. When an accepting state of the Büchi automaton (called a *seed*) is visited, it is stored in a special data structure (*dependency structure*). Nested DFS procedures for seeds are initialised separately in an appropriate order determined by the dependency structure. Only one nested DFS procedure can be started at a time. The algorithm thus performs a limited nested depth-first search which requires some synchronisation during its execution. Our aim was to experimentally explore how much will synchronisation influence the overall behaviour of the tool. To our surprise, even using this very simple method we were able to deal with verification problems larger that those that could be analysed with a single workstation running the standard (non-distributed) version of SPIN.

An experimental version of the algorithm has been implemented and a series of preliminary experiments has been performed on a cluster of nine PC based Linux workstations interconnected with a 100Mbps Ethernet, using the MPI library.

The rest of the paper is organised as follows. We start with a section which briefly describes the already existing distributed version of SPIN. The following section explores specific difficulties in direct extention of the distributed SPIN for the purposes of checking LTL formulas, and also proposes a possible solution. Then we describe the additional data structures required by our algorithm and present the pseudo-code. Finally, the complexity and effectiveness of the algorithm are discussed.

2 Distributed SPIN

In this section we briefly summarise the main idea of the distributed version of SPIN as presented in [LS99]. The algorithm partitions the state space into subsets according to the number of network nodes (computers). Each node is responsible for its own part of the state space. When a node computes a new state s, it runs the procedure `Partition(s)` to find out whether or not the state belongs to its own state subset. If the state is local, the node continues locally, otherwise a message containing the state is sent to its owner node. Local computations proceed in the depth-first search manner using the procedure DFV (a slight modification of SPIN's DFS procedure). However, due to distribution of work, the *global* searching does not follow the depth-first order, which is one of the reasons for the algorithm's inadequacy for LTL model-checking. An auxiliary data structure `U[i]` is constructed to hold information about pending requests on a node `i`. The distributed algorithm terminates when all `U[i]` queues are empty and all nodes are idle. A *manager process* is used to detect termination. Each node may send two kinds of messages to the manager. One message will be sent whenever a node becomes idle, a different message will dispatched when it becomes busy. Correct termination requires each node to reconfirm, and the overall number of messages sent and received be equal.

The pseudo-code bellow illustrates the original algorithm used in the distributed version of SPIN.

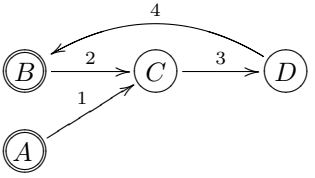
```
procedure START(i, start_state);
begin
  V[i] := {}; { already visited states }
  U[i] := {}; { pending queue }
  j := Partition(start_state);
  if i = j then
  begin
    U[i] := U[i] + start_state;
  end;
  VISIT(i);
end;

procedure VISIT(i);
begin
  while true do
  begin
    while U[i] = {} do begin end;
    S := extract(U[i]);
    DFV(i,S);
  end;
end;

procedure DFV(i, state);
begin
  if not state in V then
  begin
    V[i] := V[i] + state;
    for each sequential process P do
    begin
      nxt = all transitions of P enabled in state;
      for each st in nxt do
      begin
        j = Partition(st);
        if j = i then
        begin
          DFV(i, st);
        end
        else
        begin
          U[j] := U[j] + st;
        end;
      end;
    end;
  end;
end;
```

3 Problems with Extending the Distributed SPIN

When we want to adopt directly the technique of nested DFS approach to distributed computing we encounter two main problems. By simply allowing more nested DFS procedures for different seeds we may obtain an incorrect result, which can be demonstrated by the following example:

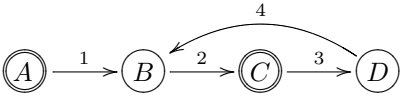


If two nested DFS procedures were run simultaneously on both seeds A and B then the cycle through the state B might not be detected. The outcome depends on the relative speeds of both nested DFS procedures. If the nested DFS procedure initialised from A visits the state C first, the nested DFS procedure starting in B will not be allowed to continue through C and, as a result, the cycle B, C, D, B will not get detected.

In general, whenever the subgraphs generated by two different seeds do not have an empty intersection, there is a possibility that some cycle may not be detected. A simple criterion to determine whether it is possible to run two or more nested DFS procedures in parallel is to find out whether or not the corresponding intersections are empty. However, verifying this condition could result in searching the entire state space.

An obvious solution to this problem is to store for each state the information in which nested DFS procedure it was visited. This would mean to store a nontrivial amount of information because each state might be a seed, in which case the space complexity of the additional storage may turn out to be quadratic with respect to the number of states. This is the reason why we do not allow to run several nested DFS procedures simultaneously.

Another problem is the following. The original distributed version of SPIN does not preserve the depth-first-search order of visited nodes. This is not a problem in case of reachability analysis because the only relevant information is whether or not a given state was visited. However, this may pose a threat to the correctness of the full model-checking procedure, which is shown on the following example:



A correct run through this graph (when DFS procedure goes back we put a dash over the name of the edge, runs of nested DFS are put into brackets), in which the cycle through C is detected, is:

$$1, 2, 3, 4, \overline{4}, \overline{3}, [3, 4, 2, \circ]^C$$

An incorrect run, in which the correct order of seeds is not preserved, is for instance

$$1, 2, 3, 4, \overline{4}, \overline{3}, \overline{2}, \overline{1}, [1, 2, 3, 4, \overline{4}, \overline{3}, \overline{2}, \overline{1}]^A, [3, \overline{3}]^C$$

A possible solution to this problem was proposed in [LS99]. It is based on the original distributed SPIN procedure for reachability analysis and consists of adding synchronisation to the distributed algorithm. The synchronisation is done by sending suitable acknowledgements that will exclude the possibility of a seed being processed before another seed that is "below", thus avoiding incorrect runs where cycles may not be detected. This solution cuts off any parallelism and so in fact does not perform better than the plain sequential algorithm.

What we propose here is an attempt to provide a more subtle solution that makes it possible to effectively tackle larger tasks than standard SPIN within reasonable space limits. The idea is to reduce the necessary synchronisation by using additional data structures with limited space requirements.

Yet another important issue that must be taken into consideration is that any distributed extention of SPIN should incorporate the main memory and complexity reduction techniques available in SPIN, such as state compression, partial order reduction, and bit state hashing. The approach that we present here is compatible with such mechanisms as much as possible.

4 Distributed Model-Checking Algorithm

From the discussion in the previous section it is clear that it is crucial to take care about the order in which individual nested DFS procedures are performed. We need to make sure that the following invariant will always hold for each distributed computation:

> *A nested DFS procedure is allowed to begin from a seed S if and only if all seeds below S have already been tested for cycle detection.*

Different states, hence also seeds, of the state space are processed on different nodes in accordance with the partition function. Hence, it may occur that for a seed S, a computation of the subgraph generated by S is interrupted and continues on a different node. We need to keep track of such situations.

In order to represent dependencies between states (corresponding to computations transferred to other nodes) we shall build a dynamic structure that will keep this essential information. We need to remember the states which caused the computation to be transferred to other nodes. We call these states *transfer states*. Two border states are involved in the transfer of computation and we need to remember only one of them. A *border state of node m* is a state belonging to the node m whose incoming or outgoing edge crosses to another node. We shall also include all the seeds that appear during the computation in order to ensure the correct order of performed nested DFS procedures. Each node maintains its own structure and we call it *DepS* (Dependency Structure).

4.1 Dependency Structure

Dependency structure (DepS) for node n is a graph whose vertices are either *seeds* of the local state space of node n or *transfer states for node n*. Vertices can be indexed by a set of node names. A transfer state for node n is either a border state of node n whose predecessor is a border state of another node, or a border state of another node whose predecessor is a border state of node n (see Figure 1 for graphical explanation). The starting state of the entire state space is also a vertex in DepS for the node running manager process. Note that a seed (or the starting state) can be a transfer state. In this case it will occur only once in the structure (as a seed *and* and a transfer state at the same time). Indexes in vertices corresponding to transfer states represent the nodes from which the computation was transferred to the transfer state.

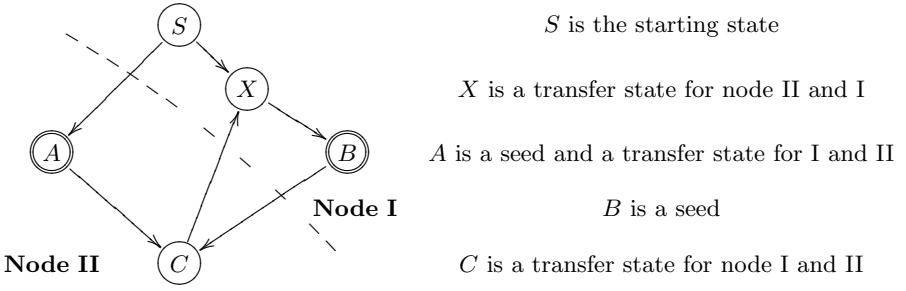


Fig. 1. Transfer States and Seeds

The edges in DepS for node n represent the reachability relation between states (and provide crucial information to perform nested DFS procedures in a "correct order"). The dependency structure is a forest-like structure and is built dynamically during the computation of the DFS procedure. All vertices in the structure are the states actually *visited* by the algorithm during depth-first search. For each vertex its immediate successors contain the states (seeds or transfer states) which the vertex is "waiting for", in the sense that the states must be processed (checked for nested DFS in case of seeds) before the state corresponding to the vertex can be processed.

The structure DepS is changing dynamically as the computation continues. Whenever the DFS procedure reaches an unvisited seed or a transfer state, a new vertex corresponding to the state is added to the structure as an immediate successor of the last visited state which is present in the structure. Moreover, in the case of a transfer state a request to continue with DFS procedure is sent to the owner node of the transfer state (if not already sent before).

During the computation each node receives requests from other nodes to continue a walk through the state space. These requests are remembered in

a local queue of pending requests and the queue is processed in the standard FIFO manner. For each request it is first checked whether the state is in the set of visited states. If it is, then it is checked whether there is any vertex in the (local) DepS structure containing the same state. If yes, then the "name" of the node who sent the request is added to its index. If the request is not in the DepS structure and it is in the set of visited states, then an acknowledgement of the request is sent back. If the request is not in the set of visited states (hence not in DepS) a new *root* vertex is added to DepS with "name" of the sender node as its index.

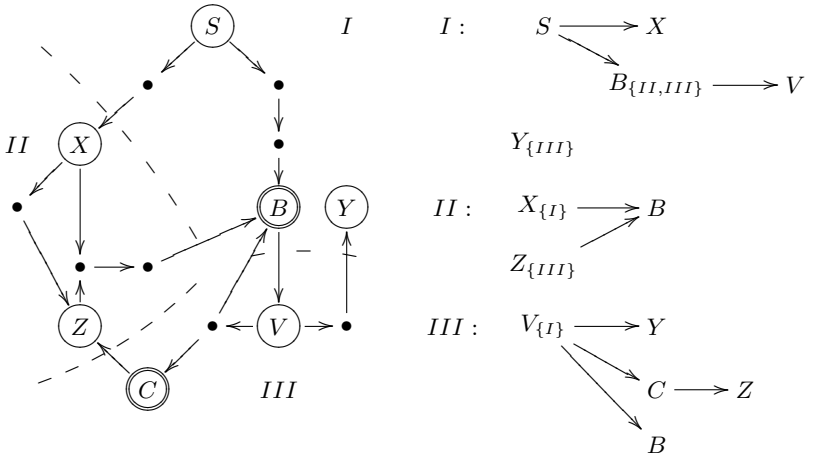


Fig. 2. Example of dependency structures

Removing a vertex from the dependency structure is possible only after the DFS procedure has gone up through the respective state. The removal procedure first checks whether the state associated with the vertex is a seed, in which case the state is added to the global queue of seeds waiting for the nested DFS procedure. For any state all acknowledgements are generated according to the (vertexes) index. Then the vertex is taken out of the structure. Removing any vertex may make its parents' successor list empty and thus cause elimination of the parent and parent's parent, and so on.

Whenever the DFS procedure goes up through a state with a vertex in DepS, it is checked whether the vertex is a leaf of the tree. If it is the case the removal procedure is initiated. Removal procedure can also be initiated by incoming acknowledgements.

In Figure 2, the dependency structures for nodes I, II and III are shown before any vertex has been removed.

4.2 Manager Process

Distributed SPIN uses a manager process that starts the verification program on a predetermined set of network nodes. After it has detected termination, the manager process stops the program, and collects the results. In our version of distributed SPIN we will in addition require that the manager process is in charge of initiating nested DFS procedures, in accordance with the method described in previous sections.

All nodes involved in the computation communicate with the manager process by sending information about their status. The status of each node is determined by: *DFS status, nDFS status, numbers of sent and received requests (DFS and nDFS packets)*. DFS status can be *busy*, *idle-empty-DepS* and *idle-nonempty-DepS*, whereas nDFS status can be only *busy* or *idle*. In this way, the manager process has a local representation of the current state of all the nodes.

Nested DFS Procedures

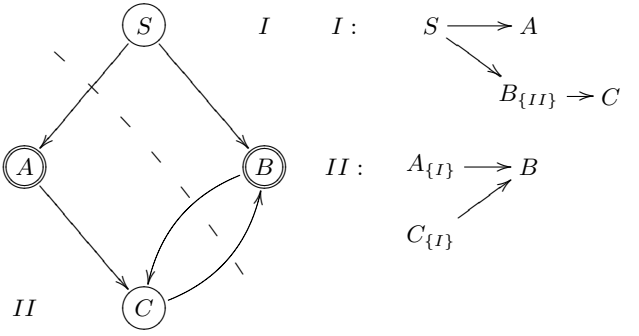
As we have mentioned before, our approach relies on the requirement that only one nested DFS procedure is allowed at a time. This is the reason for sending the seed to the manager process instead of starting a nested DFS procedure immediately during the DFS procedure. The manager process maintains a global queue of seeds waiting for their nested DFS procedures. It starts a nested DFS procedure for the first seed in the queue only if no other nested DFS procedure is running. This will be determined by checking that the nDFS status of all nodes is equal to *idle* and the overall number of all sent nDFS packets equals the overall number of all received nDFS packets, as it was already implemented in [LS99]. After the manager process has started a nested DFS procedure for a seed, it will be removed from the queue.

Termination Detection

The distributed algorithm must terminate when there is no waiting seed in the global queue, no nested DFS procedure is running, the overall numbers of sent and received DFS packets are equal, and all nodes have an *idle-empty-DepS* DFS status.

Termination detection can be handled in a similar way as it is done by distributed SPIN. The only exception is the situation when the overall numbers of sent and received DFS packets are equal, no computer has *busy* DFS status, *but* some node has *idle-nonempty-DepS* DFS status. In this case the manager process asks all the nodes with *idle-nonempty-DepS* DFS status for information about their *DepS* structures. The nodes reply by sending the following elements of their *DepS* structures: $X_Z \stackrel{u}{\rightarrow} Y$, where X is a node from the *DepS* structure which has a nonempty index Z, and Y is the node representing the transfer state for which X is waiting. The edge has a label u that represents the presence of a seed on the path from X to Y, including X and excluding Y, in the original *DepS* structure. So u can be either $1 =$ *with a seed* or $0 =$ *without a seed*.

The manager process constructs a temporary graph out of the received elements, and using standard Tarjan's algorithm [Tar72], it finds a maximal strongly connected component with no outgoing edges in the graph. Note that such a strongly connected component must exist. The manager process checks for the presence of an edge with label 1 in the strongly connected component. If there is no such edge then an acknowledgement is generated for an arbitrary node from the found component, and distributed computation continues in the standard way. In the other case, i.e. when there is a cycle labelled by 1 in the temporary graph, it is clear that a cycle through an accepting state must exist in the original state space and therefore the verified property does not hold. In this case the algorithm terminates immediately. The whole situation is shown in Figure 3.



Elements sent to the manager process are: $B_{\{II\}} \xrightarrow{1} C, A_{\{I\}} \xrightarrow{1} B, C_{\{I\}} \xrightarrow{0} B$

The constructed graph is: $A_{\{I\}} \xrightarrow{1} B_{\{II\}} \underset{0}{\overset{1}{\rightleftarrows}} C_{\{I\}}$

A strongly connected component without outgoing edges is: $B_{\{II\}} \underset{0}{\overset{1}{\rightleftarrows}} C_{\{I\}}$

A cycle through an accepting state has been found.

Fig. 3. Example

4.3 The Algorithm

Our algorithm extends the distributed algorithm from [LS99] in the sense that it employs the same distributed method for reachability analysis. This ensures that we do not exclude use of the main memory and complexity reduction techniques available in SPIN.

The underlying idea behind our distributed model-checking algorithm is the following. The whole reachable graph is partitioned into as many regions as the

number of network nodes. Each node performs the computation on the states that belong to its own region. When a successor belonging to another region is generated, a message containing the new state is sent to its owner. Received messages are stored in a (remote) queue and processed sequentially. For both DFS and nested DFS procedures, only the yet unvisited states are explored. In contrast with the original algorithm, not all visited states are permanently stored in the set of visited states, only transfer states and seeds. Each node keeps the set of permanently stored states in array `PV[i]`. To prevent cycling through not permanently stored states, the algorithm keeps track of all visited states within the processing of each received request. This temporary set is kept in array `V[i]`, which is initialised to $\emptyset$ before processing another request (state) from the queue of pending requests `U[i]`.

To ensure the right order of nested DFS procedure calls, an additional data structure `DepS` is maintained (see subsection 4.1). This structure is built by procedures `CREATE_IN_DepS(s)` and `ADD_TO_DepS(s1,s2)`, using a temporary pointer `Last_visited`. A root vertex in the dependency structure, which corresponds to state `s`, is created by procedure `CREATE_IN_DepS(s)`. Procedure `ADD_TO_DepS(s1,s2)` creates a vertex corresponding to state `s2`, if it does not exist yet, and adds an edge between the vertices corresponding to states `s1` and `s2`. A vertex representing state `s`, written as ⟨`s in DepS`⟩ in the pseudo-code, is composed of several components (fields): *parent*, *successors*, *state*, *DFS_gone* and *index*. The field *parent* points to the vertex for which this vertex has been created as a successor. The field *index* is a set of node names. The field *DFS_gone* is a flag indicating whether the DFS procedure has already walked through the state up. The meaning of the fields *successors* and *state* is obvious.

Some additional local variables are used in the algorithm. The meaning of `Seed`, `state`, `came_from` and `tmp` is obvious. Variable `toplevel` is used to distinguish whether procedure `DFV` was called recursively from `DFV` or whether it was called from procedure `VISIT`. Variable `Seed_queue` is a global variable which is maintained by `MANAGER_PROCESS`. It represents the queue of seeds waiting for their nested DFS procedures to be started.

All nodes execute the same code. For simplicity we assume that the master node runs the manager process only. The DFS procedure is started by calling procedure `START(i,starting_state)`. The value of `i` is the name of the node (integer). The procedure puts `starting_state` into the queue of pending requests `U[i]` at the proper node `i`. Procedure `VISIT(i)` is called for all nodes except the master node at the end. Procedure `MANAGER_PROCESS` is called at the master node. The task of procedure `MANAGER_PROCESS` was explained in the subsection 4.2.

Procedure `PROCESS_INCOMING_PACKETS` is actually a boolean function which returns `false` if the computation should be stopped for some reason, and returns `true` otherwise. This function plays the role of the client side of the manager process. It updates the numbers of locally received and sent packets and sends this numbers and information about the node status to the manager process. It also processes all incoming packets. The requests are stored in `U[i]` queue, the

acknowledgements are collected and procedure `REMOVE` is called for them. Also all control packets are handled by it.

Procedure `VISIT` waits for queue `U[i]` to be nonempty. It collects a request from the queue, and resets variables `toplevel` and `V[i]`. In case that the request is a new one (it is not in the set `PV[i]`), procedure `DFV` is called. It is necessary to distinguish between the first DFS procedure and the nested DFS procedure. In the case of nested DFS procedure variable `Seed` must be set, on the other hand in the case of DFS procedure appropriate actions on the `DepS` structure are performed. The states already processed, which are requested again, are checked for presence in the `DepS` structure. If the corresponding vertex exists, only its index is updated, otherwise the acknowledgement is generated.

The `DFV` procedure checks whether the state belongs to the same node. If not, the message containing the state is sent to the node owning the state and the `DepS` structure is updated. (Note: In the case of nested DFS procedure the seed, for which the nested search is running, is included in the message as well.) When the `DFV` procedure is called from the `VISIT` procedure, a new root vertex must be added to the `DepS` structure, and the `state` must be stored in the set of permanently visited states `PV[i]`. In case that `state` is a seed it is necessary to update the `DepS` structure. Note that the `DepS` structure is maintained only for the first DFS procedure and not for the nested DFS procedure. Conversely, the check whether the reached `state` is `Seed` is done only in the nested DFS procedure. The `CYCLE_FOUND` procedure informs the manager process about the fact that a cycle has been found. Before all the successors of `state` are generated, `state` is added to the set of actually visited states (`V[i]`). The unvisited successors are handled by recursive calling of procedure `DFV`. If a successor is a seed or a transfer state which is already contained in the `DepS` structure (hence it must appear in `PV[i]`), the appropriate edge is added to the `DepS` structure. After all successors of `state` have been processed and if `state` is a seed, the check whether there are any more successors of the corresponding vertex in the `DepS` structure is performed. In case there are no successors, the vertex is removed from `DepS` by procedure `REMOVE`.

Procedure `REMOVE(vertex)` is crucial with respect to maintaining the `DepS` structure. It is responsible not only for (recursive) deleting of the vertices from the memory but also for sending the seeds to the global queue (`Seed_queue`), and for sending all appropriate acknowledgements, which is done by procedure `ACK(vertex)` (see the pseudo–code for details).

Note that the same state can be visited twice, in the DFS procedure and in the nested DFS procedure. To store the information about the fact that the state has or has not been visited in one or both DFS procedures only one additional bit is required (see [Hol91]). We assume that this one additional bit is included in the bit vector representing the `state`. That is why the bit vectors representing the same state differ in the case of DFS procedure and nested DFS procedure.

The `Partition` function is used for partitioning of the state space. The choice of a "good" partition function is crucial in the distributed algorithm since a "bad" partition of states among the nodes may cause communication overhead.

This issue was addressed in [LS99], where several partition functions were proposed and tested.

The pseudo-code of our proposed algorithm follows:

```
procedure START(i,start_state)
begin
  DepS := {};
  U[i] := {};
  PV[i] := {};
  Last_visited := nil;
  Seed := nil;
  if (i = Partition(start_state)) then
  begin
    U[i] := U[i] + {start_state};
  end;
  if (i = 0) then
  begin
    MANAGER_PROCESS();
  end
  else
  begin
    VISIT(i);
  end;
end.

procedure VISIT(i)
begin
  while (PROCESS_INCOMING_PACKETS()) do
  begin
    if (U[i] <> {}) then
    begin
      get (state, came_from) from U[i];
      toplevel := true;
      V[i] := {};
      if (state not in PV[i])) then
      begin
        if (Nested(state)) then
        begin
          Seed := state.seed;
          DFV(i,state);
        end
        else
        begin
          DFV(i,state);
          if (<state in DepS>.successors = {}) then
          begin
            REMOVE(<state in DepS>);
          end;
        end;
      end
```

```
      else
      begin
        if (state in DepS) then
        begin
          <state in DepS>.index := <state in DepS>.index
                                   + {came_from};
        end
        else
        begin
          ACK(<state in DepS>);
        end;
      end;
    end;
  end;
end.

procedure REMOVE(vertex)
begin
  if Accepting(vertex.state) then
  begin
    Seed_queue := Seed_queue + {vertex.state};
  end;
  ACK(vertex);
  tmp := vertex.predecessors;
  for (i in tmp) do
  begin
    i.successors := i.successors - {vertex};
  end;
  free(vertex);
  for (i in tmp) do
  begin
    if ((i.successors = {}) and (i <> nil)
                            and (i.DFS_gone)) then
    begin
      REMOVE(i);
    end;
  end;
end.

procedure DFV(i,state)
begin
  if (PARTITION(state) <> i) then
  begin
    U[PARTITION(state)] := U[PARTITION(state)] + {state};
    if (not Nested(state)) then
    begin
      ADD_TO_DepS (Last_visited, state);
    end;
    return;
  end;
```

```
  if (toplevel) then
  begin
    PV[i] := PV[i] + state;
    if (not Nested(state)) then
    begin
      CREATE_IN_DepS(state);
      Last_visited := state;
    end;
  end;
  if (Accepting(state)) then
  begin
    if (not(toplevel) and (not Nested(state))) then
    begin
      ADD_TO_DepS(Last_visited,state);
      Last_visited := state;
      PV[i] := PV[i] + state;
    end;
  end;
  toplevel := false;
  V[i] := V[i] + {state};
  for (newstate in successors of state) do
  begin
    if (Nested(state) and (Seed=newstate)) then
    begin
      CYCLE_FOUND();
    end;
    if (newstate not in (V + DepS + PV[i])) then
    begin
      DFV(i,newstate);
    end
    else
    begin
      if ((newstate in DepS) and (not in 1stDFS stack)) then
      begin
        ADD_TO_DepS (Last_visited,newstate);
      end;
    end;
    if (Accepting(state) and (not Nested(state))) then
    begin
      Last_visited := <Last_visited in DepS>.parent;
      <state in DepS>.DFS_gone := true;
      if (<state in DepS>.successors = {}) then
      begin
        REMOVE(<state in DepS>);
      end;
    end;
  end;
end.
```

5 Complexity and Effectiveness

We shall try to estimate the overall size of dependency structures constructed during distributed computations. For any node, there are two kinds of states stored in the structure – all *seeds* that are visited by this node, and all states that arise in the computation but are transferred to another node in accordance with the partition function (*transfer states*). The number of the latter is crucial because this can be in the worst case quadratic wrt the overall number of states.

The partition function we employ was originally proposed in [LS99] where it was also shown that with this function, the fraction of transfer states in the global state space is at most $\frac{2}{P}$, where P is the number of processes. The number of transfer states T is bounded by the expression $S \times R$, where S is the number of states and R is the maximum of out-going degrees over all states. R is at most $P \times ND$, where ND is the maximal number of nondeterministic choices of a process. Thus we get that $T \leq (S \times P \times ND)$ and the average number of transfer states is $\frac{2}{P} \times S \times P \times ND = 2S \times ND$. Hence, the number of states stored in the dynamic structure is on average $S + 2(2S \times ND)$ which works out to be $O(S \times ND)$. In most real systems the amount of non-determinism (represented by ND) is limited and small. We may conclude that the memory complexity of the distributed algorithm is on average linear in the size of the state space and the factor given by non-determinism.

We will compare our approach with the simple method of synchronisation proposed in [LS99]. We will look in detail at how the first and nested depth-first-search procedures work. The first DFS searches through the state space and marks accepting states (seeds) with a particular kind of flag. The computation of first DFS is completely asynchronous and never stops before the whole state space is searched trough. The nested DFS searches in a similar, i.e. distributed, way the subgraph rooted in the currently processed seed. If there are seeds ready to be processed, the nested DFS is running in parallel with the first DFS. Therefore our method allows parallel computations which are almost disabled in the simple synchronisation technique.

We conclude the complexity analysis by a brief remark on the time complexity of the algorithm. As we do not permanently store all the visited states, we have to recompute some of them if required. This could in the worst case result in the theoretically not very significant increase of time complexity from quadratic to cubic in the size of the state space. Moreover, for SPIN with its very fast time performance the space (memory) is the real critical resource limitation in practice.

6 Conclusions and Future Research

We have proposed an extension to the existing distributed algorithm used in the verification checker SPIN, which allows to model-check LTL formulas. This problem was suggested in [LS99] and left unsolved. The method used is very simple and requires some synchronisation between nodes.

The experimental version of the algorithm has been implemented and a series of preliminary experiments has been performed on a cluster of nine 366 MHz Pentium PC Linux workstations with 128 Mbytes of RAM each interconnected with a fast 100Mbps Ethernet and using MPI library.

We have compared the performance of our algorithm with the standard sequential version of SPIN running on a single computer. Although the implementation of the *Dependency structure* was far from being optimal, we were able to receive some promising results. Our results show that it is possible to increase the capability of SPIN to check LTL formulas in a distributed manner. For testing we have used some of the scalable problems from the SPIN distribution. The experimental version was mainly used to get a fast feedback on applicability of the method. We intend to perform an extensive testing on a variety of different examples using the new and more sophisticated implementation currently under development.

There are several problems we intend to consider in the future. Firstly, we have used MPI library to get a fast prototype implementation. However, the overhead caused by this communication infrastructure is quite high. We assume that using another communication infrastructure (like TCP/IP) will certainly lead to better performance. Secondly, our algorithm utilises a partition function which is based on the function used by distributed SPIN. We realize that partition function plays a crucial role in the effectiveness of the algorithm and so we want to investigate other techniques that might be better suited to model-checking LTL formulas. Finally, for simplicity reasons we allow only one nested DFS procedure to be performed at a time. Under certain circumstances, which can be effectively recognised from the dependency structure, it is possible to start more than one nested DFS procedure at a time and we want to extend our distributed algorithm by partial parallelisation of nested DFS procedures.

References

[AAC87] S. Aggarwal, R. Alonso, and C. Courcoubetis. Distributed reachability analysis for protocol verification environments. In P. Varaiya and H. Kurzhanski, editors, *Discrete Event Systems: Models and Application*, volume 103 of *LNCIS*, pages 40–56, Berlin, Germany, August 1987. Springer-Verlag.

[BBS00] J. Barnat, L. Brim, and J. Stříbrná. Distributed LTL Model-Checking in SPIN. Technical Report FI-MU-10/00, Masaryk Univeristy Brno, 2000.

[BDHGS00] S. Ben-David, T. Heyman, O. Grumberg, and A. Schuster. Scalable distributed on-the-fly symbolic model checking. In *third International Conference on Formal methods in Computer-Aided Design (FMCAD'00), Austin, Texas*, November 2000.

[Dil96] David L. Dill. The murφ verification system. In *Conference on Computer-Aided Verification (CAV '96)*, Lecture Notes in Computer Science, pages 390–393. Springer-Verlag, July 1996.

[HGGS00] Tamir Heyman, Danny Geist, Orna Grumberg, and Assaf Schuster. Achieving scalability in parallel reachability analysis of very large circuits. In Orna Grumberg, editor, *Computer Aided Verification, 12th International Conference*, volume 1855 of *Lecture Notes in Computer Science*, pages 20–35. Springer-Verlag, June 2000.

[Hol91] G.J. Holzmann. *Design and Validation of Computer Protocols*. Prentice-Hall, Englewood Cliffs, New Jersey, 1991.

[Hol97] Gerard J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997. Special Issue: Formal Methods in Software Practice.

[LS99] F. Lerda and R. Sisto. Distributed-memory model checking with SPIN. In *SPIN workshop*, number 1680 in LNCS, Berlin, 1999. Springer.

[Tar72] Robert Tarjan. Depth first search and linear graph algorithms. *SIAM journal on computing*, pages 146–160, Januar 1972.

[UD97] U.Stern and D. L. Dill. Parallelizing the murφ verifier. In O. Grumberg, editor, *Proceedings of Computer Aided Verification (CAV '97)*, volume 1254 of *LNCS*, pages 256–267, Berlin, Germany, 1997. Springer.

[WL93] P. Wopler and D. Leroy. Reliable hashing without collision detection. In *Conference on Computer-Aided Verification (CAV '93)*, Lecture Notes in Computer Science, pages 59–70. Springer-Verlag, 1993.

Parallel State Space Construction for Model-Checking

Hubert Garavel, Radu Mateescu, and Irina Smarandache

Inria Rhône-Alpes / Vasy, 655, avenue de l'Europe
F-38330 Montbonnot Saint Martin, France
Hubert.Garavel@inria.fr, Radu.Mateescu@inria.fr, Irina.Sturm@st.com

Abstract. The verification of concurrent finite-state systems by model-checking often requires to generate (a large part of) the state space of the system under analysis. Because of the state explosion problem, this may be a resource-consuming operation, both in terms of memory and Cpu time. In this paper, we aim at improving the performances of state space construction by using parallelization techniques. We present parallel algorithms for constructing state spaces (or Labeled Transition Systems) on a network or a cluster of workstations. Each node in the network builds a part of the state space, all parts being merged to form the whole state space upon termination of the parallel computation. These algorithms have been implemented within the Cadp verification tool set and experimented on various concurrent applications specified in Lotos. The results obtained show close to ideal speedups and a good load balancing between network nodes.

Keywords: distributed algorithms, labeled transition system, Lotos, model-checking, state space construction, verification

1 Introduction

As formal verification becomes increasingly used in the industry as a part of the design process, there is a constant need for efficient tool support to deal with real-size applications. Model-checking [20,10] is a successful verification method based on reachability analysis (state space exploration) and allows an automatic detection of early design errors in finite-state systems. Model-checking works by constructing a model (state space) of the system under design, on which the desired correctness properties are verified.

There are essentially two approaches to model-checking: *symbolic* verification [9,10] represents the state space in comprehension, by using various encoding techniques (e.g., Bdds), and *enumerative* verification [32,11,12,19] represents the state space in extension, by enumerating all reachable states. Enumerative model-checking techniques can be further divided in *global* techniques, which require to entirely construct the state space before performing the verification, and *local* (or *on-the-fly*) techniques, which allow to construct the state space simultaneously with the verification.

M.B. Dwyer (Ed.): SPIN 2001, LNCS 2057, pp. 217–234, 2001.

In this paper, we focus on enumerative model-checking, which is well-adapted to asynchronous, non-deterministic systems containing complex data types (records, sets, lists, trees, etc.). More precisely, we consider the problem of constructing a *Labeled Transition System* (LTS), which is the natural model for high-level, action-based specification languages, especially process algebras such as CCS [30], CSP [18], ACP [4], or LOTOS [21]. An LTS is constructed by exploring the transition relation starting from the initial state (*forward reachability*). During this operation, all explored states must be kept in memory in order to avoid multiple exploration of a same state. Once the LTS is constructed, it can be used as input for various verification procedures, such as bisimulation/preorder checking and temporal logic model-checking. Moreover, when the verification requires to explore the entire LTS (e.g., when verifying invariant temporal properties or checking bisimulation), since the state contents is abstracted away in a constructed LTS, the memory consumed is generally much smaller than for on-the-fly verification on the initial specification.

State space construction may be very consuming both in terms of memory and execution time: this is the so-called *state explosion* problem. During the last decade, different techniques for handling state explosion have been proposed, among which partial orders and symmetries; however, for industrial-scale systems, these optimizations are not always sufficient. Moreover, most of the currently available verification tools work on sequential machines, which limits the amount of memory (between 0.5 and 2 GBytes on usual configurations), and therefore the use of clusters or networks of workstations is desirable.

In this paper, we investigate an approach to parallelize state space construction on several machines, in order to benefit from all the local memories and CPU resources of each machine. This allows to reduce both the amount of memory needed on each machine and the overall execution time. We propose algorithms for parallel construction of LTSs, developed using the generic environments BCG and OPEN/CÆSAR [13] for LTS manipulation provided by the CADP verification tool set [12]. Since these environments are language independent, our algorithms can be directly used not only for LOTOS, but also for every language connected to the OPEN/CÆSAR application programming interface, such as UML [22].

The implementation is based on standard sockets, available everywhere, and was experimented on two different configurations: a typical network of workstations (Sparc workstations running Solaris and PCs running Linux, connected using 100 Mb/s ETHERNET), and a cluster of PCs (with 450 MHz processor and 512 MBytes main memory) connected using SCI (*Scalable Coherent Interface*). Each machine in the network is responsible for constructing a part of the LTS, this part being determined using a static partition function. Upon termination of the parallel computation, which is detected by means of a virtual ring-based distributed algorithm, all parts are merged to form the complete LTS.

We experimented with our algorithms on three non-trivial protocols specified in LOTOS: the home audio-video (HAVi) protocol of Philips [33], the TOKENRING leader election protocol [14], and the SCSI-2 bus arbitration protocol [3].

Related work. Distributed state space construction has been studied in various contexts, mostly for the analysis of low-level formalisms. such as Petri nets, stochastic Petri nets, discrete-time and continuous-time Markov chains [5,6,2,1, 8,31,27,16,23].

All these approaches share a common idea: each machine in the network explores a subset of the state space. However, they differ on a number of design principles and implementation choices such as: the choice between a shared memory architecture and a message-passing one, the use of hash tables or B-trees to store states on each machine, the way of partitioning the state space using either static hash functions or dynamic ones that allow dynamic load balancing, etc.

As regards high-level languages for asynchronous concurrency, a distributed state space exploration algorithm [26] derived from the SPIN model-checker [19] has been implemented for the PROMELA language. The algorithm performs well on homogeneous networks of machines, but it does not outperform the standard, sequential implementation of SPIN, except for problems that do not fit into the main memory of a single machine. Several SPIN-specific partition functions are experimented, the most advantageous one being a function that takes into account only a fraction of the state vector.

Another distributed state enumeration algorithm has been implemented in the MURφ verifier [34]. The speedups obtained are close to linear and the hash function used for state space partition provides a good load balancing. However, experimental data reported concerns relatively small state spaces (approximatively 1.5 M states) on a 32-node UltraSPARC Myrinet network of workstations.

There also exist approaches, such as [24], in which parallelization is applied to "partial" verification, i.e., state enumeration in which some states can be omitted with a low probability. In the present paper, we only address exact, exhaustive verification issues.

For completeness, we can also mention an alternative approach [17] in which symbolic reachability analysis is distributed over a network of workstations: this approach does not handle states individually, but sets of states encoded using BDDs.

Paper outline. Section 2 gives some preliminary definitions and specifies the context of our work. Section 3 describes the proposed algorithms for parallel construction of LTSs. Section 4 discusses implementation issues and presents various experimental results. Finally, Section 5 gives some concluding remarks and directions for future work.

2 Definitions

A (monolithic) Labeled Transition System (LTS) is a tuple $M = (S, A, T, s_0)$, where S is the set of states, A is the set of actions, $T \subseteq S \times A \times S$ is the transition relation, and $s_0 \in S$ is the initial state. A transition $(s, a, s') \in T$ indicates that the system can move from state s to state s' by performing action a. All states in S are assumed to be reachable from s_0 via (sequences of) transitions in T.

In the model-checking approach by state enumeration, there are essentially two ways to represent an LTS:

explicitly, by enumerating all its states and transitions. In this case, the contents of states becomes irrelevant, since the essential information is given by actions (transition labels). Therefore, when storing an LTS as a computer file, it is sufficient to encode states as natural numbers. An explicit representation of LTSs is provided by the BCG (*Binary Coded Graph*) file format of the CADP verification tool set [12]. The BCG format is based upon specialized compression algorithms, allowing compact encodings of LTSs.

implicitly, by giving its initial state s_0 and its successor function $succ : S \rightarrow 2^T$ defined by $succ(s) = \{(s, a, s') \mid (s, a, s') \in T\}$. An implicit representation of LTSs is provided by the generic, language independent environment OPEN/CÆSAR [13] of CADP. OPEN/CÆSAR offers primitives for accessing the initial state of an LTS and for enumerating the successors of a given state, as well as various data structures (state tables, stacks, etc.), allowing straightforward implementations of on-the-fly verification algorithms.

Our objective is to translate LTSs from an implicit to an explicit representation by using parallelization techniques.

In order to represent a monolithic LTS $M = (S, A, T, s_0)$ on N machines (numbered from 0 to $N-1$), we introduce the notion of *partitioned* LTS $D = (M_0, \ldots, M_{N-1}, s_0)$, where: $S = \cup_{i=0}^{N-1} S_i$ and $S_i \cap S_j = \emptyset$ for all $0 \leq i, j < N$ (the state set is partitioned into N classes, one class per machine), $A = \cup_{i=0}^{N-1} A_i$, $T = \cup_{i=0}^{N-1} T_i$ and $(s, a, s') \in T_i \Rightarrow s' \in S_i$ for all $0 \leq i < N$ (transitions between two states belonging to different classes are part of the transition relation of the component LTS containing the target state). Note that initial states of the component LTSs M_i are irrelevant, since the corresponding subgraphs may be not connected (i.e., not reachable from a single state). A partitioned LTS can be represented as a collection of BCG files encoding the components $M_0, \ldots, M_{N-1}$.

3 Parallel Generation of LTSs

In this section we present two complementary algorithms allowing to convert an implicit LTS (defined using the OPEN/CÆSAR interface) to an explicit one (represented as a BCG file) using N machines connected by a network. These algorithms operate in two steps:

- Construction of a partitioned LTS represented as a collection of BCG files. This is done by using an algorithm called DISTRIBUTOR, which is executed on every machine in order to generate a BCG file encoding a component of the partitioned LTS.
- Conversion to a monolithic LTS represented as a single BCG file. This is done using an algorithm called BCGMERGE, which is executed on a sequential machine in order to generate a single BCG file containing all the states and transitions of the partitioned LTS.

Once the BCG file encoding the initial LTS has been constructed, it can be used as input for the EVALUATOR 3.0 model-checker [28] of CADP, which allows linear-time verification of temporal formulas expressed in regular alternation-free μ-calculus.

3.1 Construction of Partitioned LTSs

We consider a network of N machines numbered from 0 to $N-1$ and an LTS $M = (S, A, T, s_0)$ given implicitly by its initial state s_0 and its successor function *succ*. Machine i can send a message m to machine j by invoking a primitive named SEND (j, m), and can receive a message by invoking another primitive RECEIVE (m). There are four kinds of messages: *Arc*, *Rec*, *Snd*, and *Trm*, the first one being used for sending LTS transitions and the others being related to termination detection. SEND and RECEIVE are assumed to be non-blocking. RECEIVE returns a boolean answer indicating whether a message has been received or not.

The parallel generation algorithm DISTRIBUTOR that we propose is shown on Figure 1. Each machine executes an instance of DISTRIBUTOR and explores a part of the state space S. The states explored by each machine are determined using a static partition function $h : S \to [0, N-1]$. Machine i explores all states s such that $h(s) = i$ and produces a BCG file $B_i = (S_i, A_i, T_i)$. The computation is started by the machine called *initiator*, having the index $h(s_0)$, which explores the initial state of the LTS.

The states visited and explored by machine i during the forward traversal of the LTS are stored in two sets V_i ("visited") and E_i ("explored"), respectively. V_i and E_i are implemented using the state table handling primitives provided by the OPEN/CÆSAR environment. The transitions to be written to the local BCG file B_i are temporarily kept in a work list L_i. It is worth noticing that the DISTRIBUTOR algorithm only keeps in memory the state set S_i of the corresponding component of the partitioned LTS, whilst the transition relation T_i is stored in the BCG file B_i. The DISTRIBUTOR algorithm consists of a main loop, which performs three actions:

(a) A state $s \in V_i$ is explored by enumerating all its successor transitions $(s, a, s') \in succ(s)$. If a target state s' belongs to machine i (i.e., $h(s') = i$), the corresponding transition is kept in the list L_i and will be processed later. Otherwise, the transition is sent to machine $h(s')$ as a message $Arc(n_i(s), a, s')$, where $n_i(s)$ is the number associated by machine i to s. Machine $h(s')$ is responsible for writing the transition to its local BCG file and for exploring state s'. Note that there is no need to send the contents of state s itself, but only its number $n_i(s)$.

(b) A transition is taken from L_i and is written to the BCG file B_i by computing appropriate numbers for the source and target states. In order to obtain a bijective numbering of LTS states across the N BCG files, each state s explored by machine i is assigned a number $n_i(s)$ such that $n_i(s)\ mod\ N = i$. This is done using a counter c_i, which is initialized to i and incremented by N every time a new state is visited.

```
procedure DISTRIBUTOR (i, s_0, succ, h, N) is
   initiator_i := (h(s_0) = i); L_i := ∅; E_i := ∅; A_i := ∅; T_i := ∅; c_i := i;
   if initiator_i then
      terminit := false; n_i(s_0) := c_i; V_i := {s_0}; S_i := {n_i(s_0)}
   else
      V_i := ∅; S_i := ∅
   endif;
   terminated_i := false; nbsent_i := 0; nbrecd_i := 0;
   while ¬terminated_i do
(a)   if V_i ≠ ∅ then
         let s ∈ V_i; V_i := V_i \ {s}; E_i := E_i ∪ {s};
         forall (s, a, s') ∈ succ(s) do
            if h(s') = i then
               L_i := L_i ∪ {(n_i(s), a, s')}
            else
               SEND (h(s'), Arc(n_i(s), a, s')); nbsent_i := nbsent_i + 1
            endif
         endfor
(b)   elsif L_i ≠ ∅ then
         let (n, a, s) ∈ L_i; L_i := L_i \ {(n, a, s)};
         if s ∉ E_i ∪ V_i then
            c_i := c_i + N; n_i(s) := c_i; V_i := V_i ∪ {s}; S_i := S_i ∪ {n_i(s)};
         endif;
         A_i := A_i ∪ {a}; T_i := T_i ∪ {(n, a, n_i(s))}
      endif;
(c)   if RECEIVE (m) then
         case m is
            Arc(n, a, s) → L_i := L_i ∪ {(n, a, s)}; nbrecd_i := nbrecd_i + 1
            Rec(k) → if ¬initiator_i then
                        SEND ((i + 1) mod N, Rec(k + nbrecd_i))
                     elsif terminit then
                        totalrecd := k; SEND ((i + 1) mod N, Snd(nbsent_i))
                     endif
            Snd(k) → if ¬initiator_i then
                        SEND ((i + 1) mod N, Snd(k + nbsent_i))
                     elsif terminit ∧ totalrecd = k then
                        SEND ((i + 1) mod N, Trm)
                     else
                        terminit := false
                     endif
            Trm →    if ¬initiator_i then
                        SEND ((i + 1) mod N, Trm)
                     endif;
                     terminated_i := true
         endcase
      endif;
      if L_i = ∅ ∧ V_i = ∅ ∧ initiator_i ∧ ¬terminit then
         terminit := true; SEND ((i + 1) mod N, Rec(nbrecd_i))
      endif
   endwhile
end
```

Fig. 1. Parallel generation of an LTS as a collection of BCG files

(c) An attempt is made to receive a message m from another machine. If m has the form $Arc(n, a, s)$, it denotes a transition (s', a, s), where n is the source state number $n_j(s')$ assigned by the sender machine of index $j = n \ mod \ N$. In this case, the contents of m is stored in the list L_i; otherwise, m is related to termination detection (see below). Thus, the BCG file B_i will contain all LTS transitions whose target states are explored by machine i.

In order to detect the termination of the parallel LTS generation, we use a virtual ring-based algorithm inspired by [29]. According to the general definition, (global) termination is reached when all local computations are finished (i.e., each machine i has neither remaining states to explore, nor transitions to write in its BCG file B_i) and all communication channels are empty (i.e., all sent transitions have been received).

The principle of the termination detection algorithm used in DISTRIBUTOR is the following. All machines are supposed to be on an unidirectional virtual ring that connects every machine i to its successor machine $(i + 1) \ mod \ N$. Every time the initiator machine finishes its local computations, it checks whether global termination has been reached by generating two successive waves of *Rec* and *Snd* messages on the virtual ring to collect the number of messages received and sent by all machines, respectively. A message $Rec(k)$ (resp. $Snd(k)$) received by machine i indicates that k messages have been received (resp. sent) by the machines from the initiator up to $(i - 1) \ mod \ N$. Each machine i counts the messages it has received and sent using two integer variables $nbrecd_i$ and $nbsent_i$, and adds their values to the numbers carried by *Rec* and *Snd* messages. Upon receipt of the $Snd(k)$ message ending the second wave, the initiator machine checks whether the total number k of messages sent is equal to the total number *totalrecd* of messages received (the result of the *Rec* wave). If this is the case, it will inform the other machines that termination has been reached, by sending a *Trm* message on the ring. Otherwise, the initiator concludes that termination has not been reached yet and will generate a new termination detection wave later.

In practice, to reduce the number of termination detection messages, each machine propagates the current wave only when its local computations are finished (for simplicity, we did not specify this in Figure 1). Experimental results have shown that in this case there is almost no termination detection overhead, two waves being always sufficient. This distributed termination detection scheme seems to use less messages than the centralized termination detection schemes used in the parallel versions of SPIN [26] and MURφ [34], which in all cases require several broadcast message exchanges between a coordinator machine and all other machines.

3.2 Merging of Partitioned LTSs into Monolithic LTSs

After constructing a collection of N BCG files representing a partitioned LTS by using the DISTRIBUTOR algorithm, the next step is to convert them into a unique BCG file in order to make it usable by the verification tools of CADP.

Since the states contained in different BCG files have been given unique numbers by the DISTRIBUTOR algorithm (i.e., every state belonging to the BCG file B_i has an index k such that $k \ mod \ N = i$ and two states belonging to the same BCG file have different numbers), this could simply be done by concatenating all transitions of the N BCG files.

However, since the partition function h is not perfect, a simple concatenation may result in a BCG file with an initial state number different from 0 (when $h(s_0) \neq 0$) and with "holes" in the numbering of states (when $|S_i| \neq |S_j|$ for two BCG files B_i and B_j). For example, for an LTS with 7 states and $N = 2$, DISTRIBUTOR could produce $S_0 = \{0, 2, 4, 6, 8\}$, $S_1 = \{1, 3\}$, and $h(s_0) = 1$, which would lead by concatenation to a BCG file with $S = \{0, 1, 2, 3, 4, 6, 8\}$ instead of $S = \{0, 1, 2, 3, 4, 5, 6\}$. A contiguous renumbering of the states would be more suitable for achieving a better compaction of the final BCG file.

The conversion algorithm BCGMERGE that we propose (see Figure 2) takes as inputs a partitioned LTS represented as a collection of BCG files $B_0, \ldots, B_{N-1}$ generated using DISTRIBUTOR from an LTS $M = (S, A, T, s_0)$, and the index i_0 $(= h(s_0))$ of the BCG file containing s_0. BCGMERGE constructs a BCG file that encodes M by numbering the states contiguously from 0 to $|S| - 1$.

```
procedure BCGMERGE (B_0, ..., B_{N-1}, i_0) is
    c := 0;
    forall k = 0 to N - 1 do
        i := (i_0 + k) mod N;
        c_i := c;
        c := c + |{q ∈ S_i | q mod N = i}|
        c := c + |{q ∈ S_i | q mod N = i}|
    end;
    Q := ∅; A := ∅; R := ∅; q_0 := 0;
    forall k = 0 to N - 1 do
        i := (i_0 + k) mod N;
        forall (q, a, q') ∈ T_i do
            Q := Q ∪ {c_{q mod N} + (q div N), c_i + (q' div N)};
            A := A ∪ {a};
            R := R ∪ {(c_{q mod N} + (q div N), a, c_i + (q' div N))}
        end
    end
end
```

Fig. 2. Merging of a collection of BCG files into a single one

Let $N_i = |\{q \in S_i \mid q \ mod \ N = i\}|$ be the number of states belonging to file B_i, i.e., the states $s \in S$ of the original LTS having $h(s) = i$. The idea is to assign to each BCG file B_i (for i going from i_0 to $(i_0 + N - 1) \ mod \ N$) a new range $[c_i, c_i + N_i - 1]$ of contiguous state numbers such that $c_{i_0} = 0$ and

$c_{(i+1) \ mod \ N} = c_i + N_i$. The final BCG file $B = (Q, A, R, q_0)$ is then obtained by concatenating the transitions of all BCG files B_i, each state number $q \in S_i$ corresponding to a state $s \in S$ with $h(s) = i$ being replaced by $c_i + (q \ div \ N)$ (where *div* denotes integer division). Thus, the initial state s_0 will get number $q_0 = c_{i_0} + (i_0 \ div \ N) = 0$ and all states will be numbered contiguously.

It is worth noticing that the BCGMERGE algorithm processes only one BCG file B_i at a time and does not require to load in memory the transition relation of B_i. State renumbering is performed on-the-fly, resulting in a low memory consumption, independent from the size of the input BCG files.

4 Experimental Results

We implemented the DISTRIBUTOR and BCGMERGE algorithms within the CADP verification tool set [12] by using the OPEN/CÆSAR [13] and BCG environments. To ensure maximal portability, the communication primitives of DISTRIBUTOR are built on top of TCP/IP using standard UNIX sockets. An alternative implementation using the MPI (*Message Passing Interface*) standard [15] would have been possible; we chose sockets because they are built-in in most operating systems and because the DISTRIBUTOR algorithm was simple enough not to require the higher-level functionalities provided by MPI.

We experimented DISTRIBUTOR and BCGMERGE on three industrial-sized protocols specified in LOTOS:

(a) The HAVi protocol [33], standardized by several companies, among which Philips, in order to solve interoperability problems for home audio-video networks. HAVi provides a distributed platform for developing applications on top of home networks containing heterogeneous electronic devices and allowing dynamic plug-and-play changes in the network configuration. We considered a configuration of the HAVi protocol with 2 device control managers (1,039,017 states and 3,371,039 transitions, state size of 80 bytes).

(b) The correct TOKENRING leader election protocol [14] for unidirectional ring networks, which is an enhanced version of the protocols proposed by Le Lann [25] and by Chang & Roberts [7]. This TOKENRING protocol corrects an error in Le Lann's and Chang & Roberts' protocols, by allowing to designate a unique leader station in presence of various faults of the system, such as message losses and station crashes. We considered a configuration of the TOKENRING protocol with 3 stations (12,362,489 states and 45,291,166 transitions, state size of 6 bytes).

(c) The arbitration protocol for the SCSI-2 bus [3], which is designed to provide an efficient peer-to-peer I/O bus for interconnecting computers and peripheral devices (magnetic and optical disks, tapes, printers, etc.). We considered SCSI-2 configurations consisting of a controller device and several disks that accept data transfer requests from the controller. Two versions of the specification have been used: v1, with 5 disks (961,546 states and 5,997,701 transitions, state size of 13 bytes) and v2, with 6 disks (1,202,208 states and 13,817,802 transitions, state size of 15 bytes).

The experiments have been performed on a cluster of 450 MHz, 512 MBytes PCs connected via SCI (the DISTRIBUTOR and BCGMERGE have been developed and debugged on an ETHERNET network of three SUN workstations; however, using a dedicated SCI network with more machines was more appropriate for performance measurement). Our performance measurements concern three aspects: speedup, partition function, and use of communication buffers.

4.1 Speedup

Figure 3 shows the speedups obtained by generating the LTSs of the aforementioned LOTOS specifications in parallel on a cluster with up to 10 PCs. For the TOKENRING and HAVi protocols, the speedups observed on N machines are given approximately by the formulas $S_N = t_1/t_N = 0.4N$ and $S_N = 0.3N$ (t_k being the execution time on k machines). For the v1 and v2 versions of the SCSI-2 protocol, the speedups obtained are close to ideal.

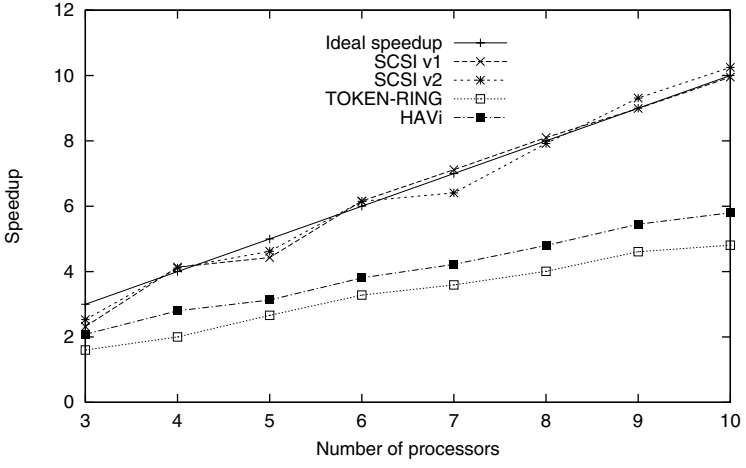


Fig. 3. Speedup measurements for the HAVi, TOKENRING, and SCSI-2 protocols

These results can be explained by examining the implementation of the DISTRIBUTOR algorithm. The state sets explored by each machine in the network are stored locally using generic hash tables provided by the OPEN/CÆSAR library. Since these hash tables use open hashing (with a fixed number of hash entries), the number of states contained in each table is not bounded (except by the amount of available memory on each machine) and the search time in the hash table grows linearly with the number of states already inserted in the table. Therefore, splitting the state set among N machines is likely to reduce by N the overall search time. Also, parallelization becomes efficient when the time spent in generating state successors is important, which happens for LOTOS

specifications having many parallel processes and complex synchronization patterns. This explains why the speedup obtained for the SCSI-2 is better than for the TOKENRING: the SCSI-2 example involves complex data computations (handling of disk buffers and of device status kept by the controller) and synchronizations (multiple rendezvous between 6 or 7 devices to gain bus access), whereas the TOKENRING example has very simple computations and only binary synchronizations between stations and communication links.

The speedups obtained show a good overlapping between computations and communications during the execution of DISTRIBUTOR. This is partly due to a buffered communication scheme with well-chosen dimensions of transmission buffers (see Section 4.3).

4.2 Choosing a Good Partition Function

In order to increase the performance of the parallel generation algorithm, it is essential to achieve a good load balancing between the N machines, meaning that the N parts of the distributed LTS should contain (nearly) the same number of states. As indicated in Section 3.1, we adopted a static partition scheme, which avoids the potential communication overhead occurring in dynamic load balancing schemes. Then, the problem is to choose an appropriate partition function $h : S \rightarrow [0, N-1]$ associating to each state a machine index.

Because we target at language independent state space construction, we cannot assume that state contents exhibit structural properties (e.g., constant fields, repeated patterns, etc.) particular to a given language. All that we can assume is that state contents are uniformly distributed bit strings.

The OPEN/CÆSAR environment [13] of CADP offers several hashing functions $f(s, P)$ that compute, for a state s and a prime number P, a hash-code between 0 and $P-1$. The function we chose[1] performs very well, i.e., it uniformly distributes the states of S into P chunks, each one containing $|S|$ *div* P states. To distribute these P chunks on N machines, the simplest way is to take the remainder of the hash-code modulo N, yielding a partition function of the form $h(s) = f(s, P)$ *mod* N. In order to guarantee that h also distributes states uniformly among the N machines, we must choose an appropriate value for P.

Still assuming that state contents are uniformly distributed, the partition function h will allocate $(P \ div \ N) + 1$ chunks on each machine $j \in \{0, \ldots, (P \ mod \ N) - 1\}$ and $P \ div \ N$ chunks on each other. If N is prime, the obvious choice for P is $P = N$, leading to a distribution of a single chunk on each machine. If N is not prime, a choice of P such that $P \ mod \ N = 1$ ensures that only machine 0 has one chunk more than the others. In this case, P should be sufficiently big, in order to reduce the size $|S| \ div \ P$ of a chunk. For the experiments presented in this paper, we chose P around 1,600,000 (which gives a limit of 10 states per chunk).

[1] This hashing function, called `CAESAR_STATE_3_HASH()` in the OPEN/CÆSAR library, calculates the remainder modulo a prime number of the state vector (seen as an arbitrarily long integer number).

Figures 4 and 5 show the distribution of the states on 10 machines for the main protocols described above. In order to evaluate the quality of the distribution, we calculated the standard deviation $\sigma = \sqrt{(\sum_{i=0}^{N-1}(|S_i| - |S|/N)^2)/N}$ between the sizes $|S_i|$ of the state sets explored by each machine i in the network. For all examples considered, the values obtained for σ are very small (less than 1% of the mean value $|S|/N$), which indicates a good performance of the partition function h.

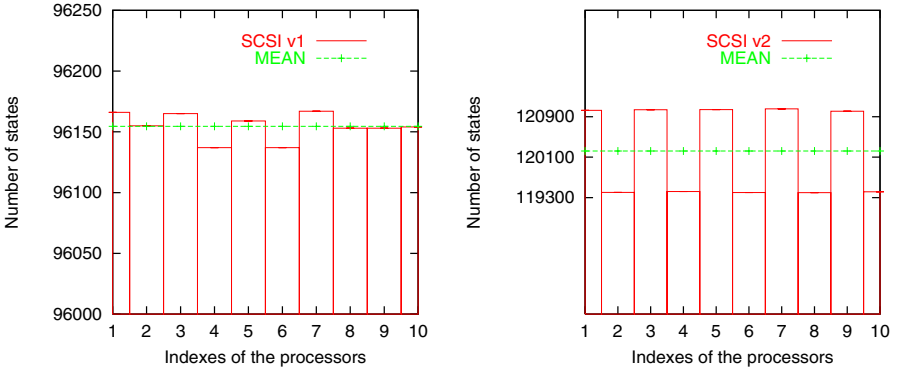


Fig. 4. State distributions for the SCSI-2 protocol on 10 machines

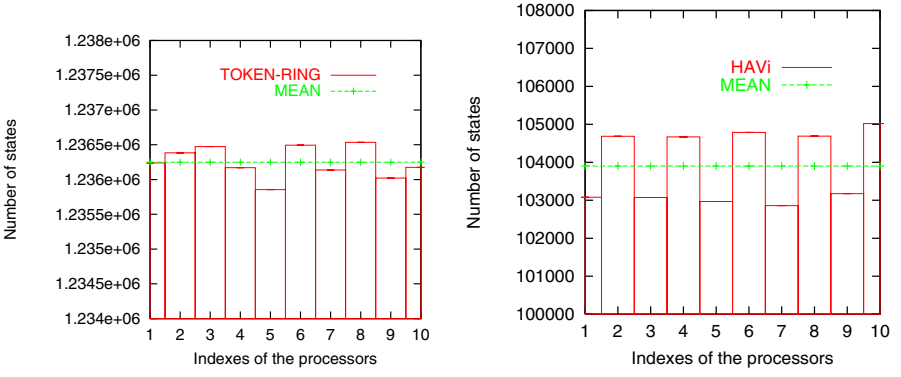


Fig. 5. State distribution for the TOKENRING and HAVI protocols on 10 machines

The quality of a partition function could also be estimated according to the number of "cross-border" transitions of the partitioned LTS (i.e., transitions hav-

ing the source state in a component and the target state in another component). This number should be as small as possible, since it is equal to the number of *Arc* messages sent over the network during the execution of DISTRIBUTOR. However, in practice, reducing the number of cross-border transitions would require additional information about the structure of the program, and therefore must be language dependent. Since DISTRIBUTOR is built using the language independent OPEN/CÆSAR environment, we did not focused on developing language dependent (e.g., LOTOS-specific) partition functions. This might be done in the future, by extending the OPEN/CÆSAR application programming interface to provide more information about the internal structure of program states.

4.3 Using Communication Buffers

To reduce the overhead of message transmission and to increase the overlapping between communications and computations, we chose an asynchronous, non-blocking implementation of the SEND and RECEIVE primitives used in the DISTRIBUTOR algorithm. Also, to reduce communication latency, these primitives actually perform a buffering of messages (LTS transitions) instead of sending them one by one as indicated in Figure 1.

The implementation is based on TCP/IP and standard UNIX communication primitives (sockets). In practice, for each machine $0 \leq i \leq N-1$, there is a virtual channel (i, j) to every other machine $j \neq i$ with a corresponding logical buffer of size L used for storing messages transmitted on the channel. The $N-1$ virtual channels associated with each machine share the same physical channel (socket), which has an associated buffer of size L_p. For a given size d of messages (which depends on the application), we observed that the optimal length of the logical transmission buffer is given by the formula $L_{opt} = L_p/d(N-1)$. Experiments show that for this value, all transitions accumulated in the logical transmission buffers can be sent at the physical level by the next call to SEND. Figure 6 illustrates the effect of buffering on DISTRIBUTOR's speedup for the SCSI-2 and the TOKENRING protocols. A uniform increase of speedup is observed between the variants $L = 1$ (no buffering) and $L = L_{opt}$. The difference in speedup is greater for the TOKENRING protocol because the percentage of communication time w.r.t. computation time is more important than for the SCSI-2 protocol. Therefore, the value L_{opt} seems a good choice for ensuring a maximal overlapping of communications and computations.

5 Conclusion and Future Work

We presented a solution for constructing an LTS in parallel using N machines connected by a network. Each machine constructs a part of the LTS using the DISTRIBUTOR algorithm, all resulting parts being combined using the BCGMERGE algorithm to form the complete LTS. These algorithms have been implemented within the CADP tool set [12] using the generic environments OPEN/CÆSAR [13] and BCG for implicit and explicit manipulation of LTSs.

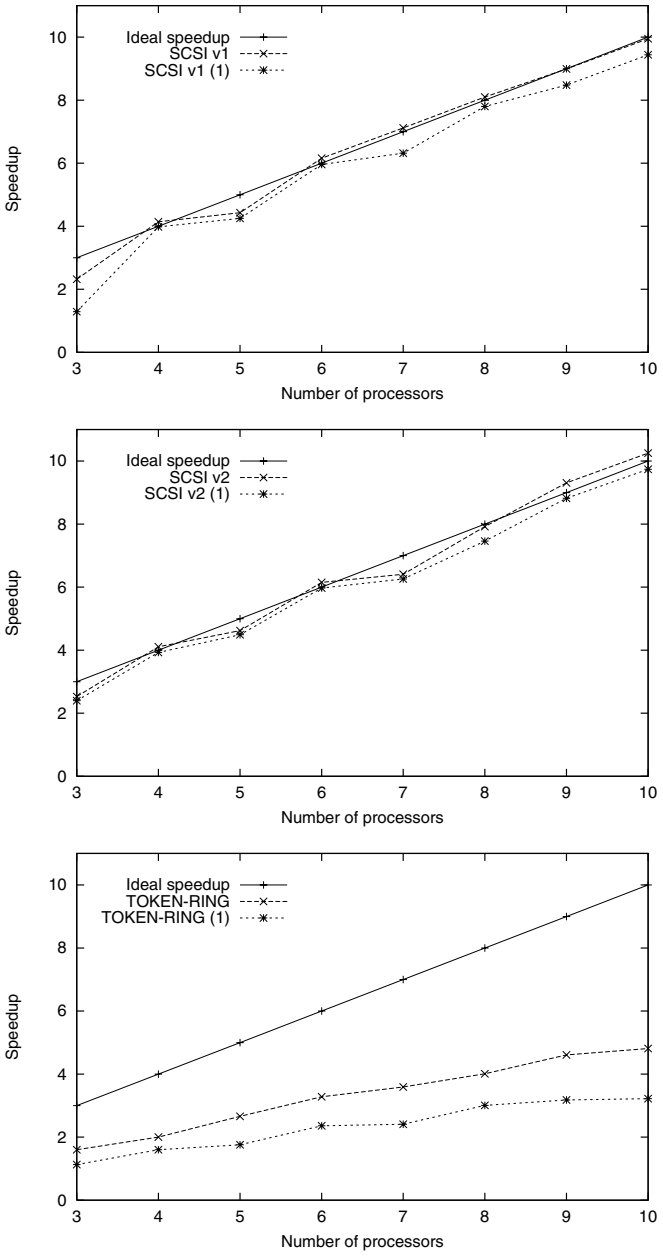


Fig. 6. Speedup measurements for the SCSI-2 and TOKENRING protocols for transmission buffers of size 1 and L_{opt}

Being independent from any specification language is a difference between our approach and other related work. To our knowledge, all published algorithms but [8] are dedicated to a specific low-level formalism (Petri nets, Markov chains, etc.) or high-level language (Murφ, Promela, etc.). On the contrary, as the Open/Cæsar and Bcg environments are language independent, the Distributor and BcgMerge tools can be used not only for Lotos, but also for every language having a connection to the Open/Cæsar interface, such as the Umlaut compiler for Uml [22].

Another distinctive feature of our approach relies in the scheme used by Distributor and BcgMerge to assign unique numbers to states. Although the Distributor algorithm is similar to the *ExploreDistributed* algorithm of [8], we manage to number states with mere integers, whereas [8] uses pairs of the form $\langle$*processor number*, *local state number*$\rangle$.

We experimented our approach on several real-size Lotos specifications, for which we generated large Ltss (up to 12 million states and 45 million transitions). Compared to the data reported for other high-level languages such as Murφ [34] and Promela [26], respectively, we were able to generate larger (11 times and 4.2 times, respectively) state spaces.

We believe that the memory overhead required by distribution (i.e., hash table auxiliary data structures, communication buffers, etc.) is negligible. Moreover, our experimental results show that parallel construction of Ltss provides linear speedups. This is due both to the good quality of the partition function used to distribute the state space among different machines, and to well-dimensioned communication buffers. The speedups obtained are more important for the specifications involving complex data computations and synchronizations, because in this case the traversal of Lts transitions becomes time expensive and can be distributed profitably across different machines.

In this paper, we focused on the problem of constructing Ltss in parallel, with a special emphasis on resource management issues such as state storage in distributed memories and transition storage in distributed filesystems. For a proper separation of concerns, we deliberately avoided to mix parallel state space constructions with other issues such as on-the-fly verification. Obviously, it would be straightforward to enhance the parallel algorithms with on-the-fly verification capabilities such as deadlock detection, invariant checking, or more complex properties. However, this was not suitable to obtain meaningful experimental results (especially, the sizes of the largest state spaces that can be constructed using the parallel approach), because on-the-fly verification may either terminate early without exploring the entire state space, or explore a larger state space when relying on automata product techniques.

This work can be continued in several directions. Firstly, we plan to pursue our experiments on new examples and assess the scalability of the approach using a more powerful parallel machine, a cluster of 200 Pcs that is currently under construction at Inria Rhône-Alpes.

Secondly, we plan to extend the Distributor tool in order to handle specifications containing dynamic data structures, such as linked lists, trees, etc.

This will require the transmission of variable length, typed data values over a network, contrary to the current implementation of DISTRIBUTOR, which uses messages of fixed length.

Finally, we will seek to determine at which point the sequential verification algorithms available in CADP (for model-checking of temporal logic formulas on LTSs, comparison and minimization of LTSs according to equivalence/preorder relations) will give up. As the sizes of LTSs constructed by DISTRIBUTOR will increase, it will be necessary to parallelize the verification algorithms themselves. Two approaches can be foreseen: parallel algorithms operating on-the-fly during the exploration of the LTS, or sequential algorithms working on (already constructed) partitioned LTSs.

Acknowledgements. We are grateful to Xavier Rousset de Pina and to Emmanuel Cecchet for interesting discussions and for providing valuable assistance in using the PC cluster of the SIRAC project of INRIA Rhône-Alpes. We also thank Adrian Curic and Frédéric Lang for their careful reading and comments on this paper.

References

1. S. Allmaier, S. Dalibor, and D. Kreische. Parallel Graph Generation Algorithms for Shared and Distributed Memory Machines. In *Proceedings of the Parallel Computing Conference PARCO'97 (Bonn, Germany)*. Springer-Verlag, 1997.
2. S. Allmaier, M. Kowarschik, and G. Horton. State Space Construction and Steady-State Solution of GSPNs on a Shared-Memory Multiprocessor. In *Proceedings of the 7th IEEE International Workshop on Petri Nets and Performance Models PNPM'97 (Saint Malo, France)*, pages 112–121. IEEE CS-Press, 1997.
3. ANSI. Small Computer System Interface-2. Standard X3.131-1994, American National Standards Institute, January 1994.
4. J. A. Bergstra and J. W. Klop. Process Algebra for Synchronous Communication. *Information and Computation*, 60:109–137, 1984.
5. S. Caselli, G. Conte, F. Bonardi, and M. Fontanesi. Experiences on SIMD Massively Parallel GSPN Analysis. In G. Haring and G. Kotsis, editors, *Computer Performance Evaluation: Modelling Techniques and Tools*, volume 794. Lecture Notes in Computer Science, Springer-Verlag, 1994.
6. S. Caselli, G. Conte, and P. Marenzoni. Parallel State Space Exploration for GSPN Models. In G. De Michelis and M. Diaz, editors, *Applications and Theory of Petri Nets 1995*, volume 935, pages 181–200. Lecture Notes in Computer Science, Springer-Verlag, 1995.
7. Ernest Chang and Rosemary Roberts. An Improved Algorithm for Decentralized Extrema-Finding in Circular Configurations of Processes. *Communications of the ACM*, 22(5):281–283, may 1979.
8. G. Ciardo, J. Gluckman, and D. Nicol. Distributed State Space Generation of Discrete-State Stochastic Models. *INFORMS Journal of Computing*, 1997.
9. A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri. NUSMV: a New Symbolic Model Checker. *Springer International Journal on Software Tools for Technology Transfer (STTT)*, 2(4):410–425, April 2000.
10. E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 2000.

11. D. Dill. The Murφ Verification System. In R. Alur and T. Henzinger, editors, *Proceedings of the 8th International Conference on Computer-Aided Verification CAV'96*, volume 1102 of *Lecture Notes in Computer Science*, pages 390–393. Springer Verlag, July 1996.
12. Jean-Claude Fernandez, Hubert Garavel, Alain Kerbrat, Radu Mateescu, Laurent Mounier, and Mihaela Sighireanu. CADP (CÆSAR/ALDEBARAN Development Package): A Protocol Validation and Verification Toolbox. In Rajeev Alur and Thomas A. Henzinger, editors, *Proceedings of the 8th Conference on Computer-Aided Verification (New Brunswick, New Jersey, USA)*, volume 1102 of *Lecture Notes in Computer Science*, pages 437–440. Springer Verlag, August 1996.
13. Hubert Garavel. OPEN/CÆSAR: An Open Software Architecture for Verification, Simulation, and Testing. In Bernhard Steffen, editor, *Proceedings of the First International Conference on Tools and Algorithms for the Construction and Analysis of Systems TACAS'98 (Lisbon, Portugal)*, volume 1384 of *Lecture Notes in Computer Science*, pages 68–84, Berlin, March 1998. Springer Verlag. Full version available as INRIA Research Report RR-3352.
14. Hubert Garavel and Laurent Mounier. Specification and Verification of Various Distributed Leader Election Algorithms for Unidirectional Ring Networks. *Science of Computer Programming*, 29(1–2):171–197, July 1997. Special issue on Industrially Relevant Applications of Formal Analysis Techniques. Full version available as INRIA Research Report RR-2986.
15. W. Gropp, S. Huss-Lederman, A. Lumsdaine, E. Lusk, B. Nitzberg, W. Saphir, and M. Snir. *MPI: The Complete Reference, Vol. 2 — The MPI-2 Extensions.* MIT Press, 1998.
16. B. Haverkort, H. Bohnenkamp, and A. Bell. On the Efficient Sequential and Distributed Evaluation of Very Large Stochastic Petri Nets. In *Proceedings PNPM'99 (Petri Nets and Performance Models)*. IEEE CS-Press, 1999.
17. T. Heyman, D. Geist, O. Grumberg, and A. Schuster. Achieving Scalability in Parallel Reachability Analysis of Very Large Circuits. In E. A. Emerson and A. P. Sistla, editors, *Proceedings of the 12th International Conference on Computer-Aided Verification CAV'2000 (Chicago, IL, USA)*, volume 1855 of *Lecture Notes in Computer Science*, pages 20–35. Springer Verlag, July 2000.
18. C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, 1985.
19. G. Holzmann. The Model Checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.
20. Gerard J. Holzmann. *Design and Validation of Computer Protocols.* Software Series. Prentice Hall, 1991.
21. ISO/IEC. LOTOS — A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour. International Standard 8807, International Organization for Standardization — Information Processing Systems — Open Systems Interconnection, Genève, September 1988.
22. J-M. Jézéquel, W.M. Ho, A. Le Guennec, and F. Pennaneac'h. UMLAUT: an Extendible UML Transformation Framework. In R.J. Hall and E. Tyugu, editors, *Proceedings of the 14th IEEE International Conference on Automated Software Engineering ASE'99.* IEEE, 1999. Also available as INRIA Technical Report RR-3775.
23. W. J. Knottenbelt and P. G. Harrison. Distributed Disk-Based Solution Techniques for Large Markov Models. In *Proceedings of the 3rd International Meeting on the Numerical Solution of Markov Chains NSMC'99*, Zaragoza, Spain, September 1999.

24. W. J. Knottenbelt, M. A. Mestern, P. G. Harrison, and P. Kritzinger. Probability, Parallelism and the State Space Exploration Problem. In *Proceedings of the 10th International Conference on Modelling, Techniques and Tools (TOOLS '98)*, pages 165–179. LNCS 1469, September 1998.
25. Gérard Le Lann. Distributed Systems — Towards a Formal Approach. In B. Gilchrist, editor, *Information Processing 77*, pages 155–160. IFIP, North-Holland, 1977.
26. F. Lerda and R. Sista. Distributed-Memory Model Checking with SPIN. In D. Dams, R. Gerth, S. Leue, and M. Massink, editors, *Proceedings of the 5th and 6th International SPIN Workshops on Theoretical and Practical Aspects of SPIN Model Checking SPIN'99*, volume 1680 of *Lecture Notes in Computer Science*, pages 22–39. Springer Verlag, July 1999.
27. P. Marenzoni, S. Caselli, and G. Conte. Analysis of Large GSPN Models: a Distributed Solution Tool. In *Proceedings of the 7th International Workshop on Petri Nets and Performance Models*, pages 122–131. IEEE Computer Society Press, 1997.
28. Radu Mateescu and Mihaela Sighireanu. Efficient On-the-Fly Model-Checking for Regular Alternation-Free Mu-Calculus. In Stefania Gnesi, Ina Schieferdecker, and Axel Rennoch, editors, *Proceedings of the 5th International Workshop on Formal Methods for Industrial Critical Systems FMICS'2000 (Berlin, Germany)*, GMD Report 91, pages 65–86, Berlin, April 2000. Also available as INRIA Research Report RR-3899.
29. F. Mattern. Algorithms for Distributed Termination Detection. *Distributed Computing*, 2:161–175, 1987.
30. Robin Milner. *Communication and Concurrency*. Prentice-Hall, 1989.
31. D. Nicol and G. Ciardo. Automated Parallelization of Discrete State-Space Generation. *Journal of Parallel and Distributed Computing*, 47:153–167, 1997.
32. Y.S. Ramakrishna and S.A. Smolka. Partial-Order Reduction in the Weak Modal Mu-Calculus. In A. Mazurkiewicz and J. Winkowski, editors, *Proceedings of the 8th International Conference on Concurrency Theory CONCUR'97*, volume 1243 of *Lecture Notes in Computer Science*, pages 5–24. Springer Verlag, 1997.
33. Judi Romijn. Model Checking the HAVi Leader Election Protocol. Technical Report SEN-R9915, CWI, Amsterdam, The Netherlands, June 1999. submitted to Formal Methods in System Design.
34. U. Stern and D. Dill. Parallelizing the Murφ Verifier. In *Computer Aided Verification*, volume 1254, pages 256–267. Lecture Notes in Computer Science, Springer-Verlag, 1997.

Model Checking Systems of Replicated Processes with Spin

Fabrice Derepas[1] and Paul Gastin[2]

[1] Nortel Networks, 1, Place des frères Montgolfier,
78928 Yvelines Cedex 09, France.
fderepas@nortelnetworks.com
[2] LIAFA, UMR 7089 Université Paris 7, 2 place Jussieu,
F-75251 Paris Cedex 05, France.
Paul.Gastin@liafa.jussieu.fr

Abstract. This paper describes a reduction technique which is very useful against the state explosion problem which occurs when model checking distributed systems with several instances of the same process. Our technique uses symmetry which appears in the system. Exchanging those instances is not as simple as it seems, because there can be a lot of references to process locations in the system. We implemented a solution using the Spin model checker, and added two keywords to the Promela language to handle these new concepts.

1 Introduction

When a new protocol for a distributed system is designed, the behavior of each actor is known, but there is a need to validate the overall system behavior. Distributed systems are difficult to verify using model checkers due to the state explosion problem. For n independent components, with S states and T transitions the number of states and transitions of the system may be as large as S^n and nTS^{n-1}.

Very often in a distributed system several actors have similar roles, such as several clients regarding a server for instance. This is implemented as several processes running the same code. We say these actors are replicated processes. In this case, the system presents some symmetry that should be exploited during its verification. This has already been studied and implemented, e.g. in the Murφ model checker, where a special data structure with a restricted usage is introduced and used to describe the symmetric part of the system [1,2,3]. An implementation of scalarsets has been done under the Spin model Checker [4], in the Symmetric Spin package [5].

But in a distributed system with communicating processes, a process often keeps addresses of other processes in some variables, e.g. in order to send (or receive from) them messages. For instance we have several copies of a variable representing the process identifier in our system. This kind of variables does not fulfill the above requirements, hence, in this case, one cannot use the aforementioned techniques and tools.

M.B. Dwyer (Ed.): SPIN 2001, LNCS 2057, pp. 235–251, 2001.

The aim of this paper is to present an abstraction through symmetry which works also when using variables holding references to other processes. We give the theoretical background of our method and we describe an implementation for the Spin model Checker. We have introduced two new constructs to the Promela input language of Spin in order to allow the automatic reduction through symmetry. Our implementation translates the extended Promela description into a classical one and stores some extra information that is used later by the model checker. We have changed the next-state function of Spin so that it calls a reduction function after each newly generated state. The reduction function basically maps each state to a state which is equivalent up to the symmetry of the system. This reduction uses the extra information which was stored above.

We give some experimental results which show that our abstraction may induce a huge reduction of the number of states visited during the model checking.

A more theoretic approach to reduction using symmetry has been given in [6]. There, it is shown that one can reduce a system $\mathcal{M}$ to its quotient by a subgroup of $\mathrm{auto}(\mathcal{M}) \cap \mathrm{auto}(\varphi)$ where $\mathrm{auto}(\mathcal{M})$ is the group of automorphisms of $\mathcal{M}$ and $\mathrm{auto}(\varphi)$ is a similar group for the specification formula φ. For an automatic implementation, the difficulty is to compute a suitable subgroup. Our paper may be seen as a particular heuristic for this problem. By introducing the new constructs to the Promela Language and reducing the kind of formula one may write, we are able to compute a suitable subgroup that is contained in $\mathrm{auto}(\mathcal{M}) \cap \mathrm{auto}(\varphi)$. Actually, we do not use the formalism introduced in [6] and we give a direct proof of our result.

Part 2 presents the generic notions we are adding to Promela [4] to implement our abstractions. Soundness of the abstraction requires some constraints, verified by the model checker. These constraints are described.

Part 3 presents a formal approach to justify our abstraction. This is similar to proofs in [6], but it also allows to understand the former constraints.

Then part 4 gives some experimental results, and some heuristics for other systems.

2 Permutation of Processes

This section details the need we have to permute processes. Notions required for this operation: references and public variables, are presented. Modifications to the Promela language follow.

2.1 Our Goal

We are interested in systems where there are several processes running the same source code. This corresponds to several instances of the same procedure. Two instances of the same procedure are considered to be relatively equivalent, that is, for two such processes A_1 and A_2, we consider that the system state where A_1 is in state s_1 and A_2 in state s_2, is equivalent to the system state where A_1 is in state s_2 and A_2 in state s_1.

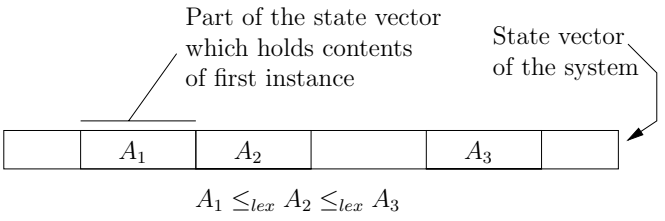


Fig. 1. Ordered instances in state vector

A simple idea, regarding all instances $A_1, \ldots, A_n$ of the same procedure, is to sort them in - for instance - lexicographic order in the state vector. This is shown in Figure 1. We have to apply this sorting procedure after each transition of the system in order to have only states where all instances of a same procedure are ordered. There can be two problems with this method:

- some global variables of the system are sometimes used as public variables attached to the processes. For instance, if there are N instances of process User one can imagine that each process has a boolean flag which may be used by other processes and hence must be declared as a global array. When exchanging two instances i and j of type User one should swap the global variables $flag[i]$ and $flag[j]$. Instead of defining such a flag as a global variable, a much better solution would be to declare it inside each process as a public variable.
- some variables in the system explicitly refer to the position of the process in the state vector. This is the case for the *_pid* variable. We should pay attention to the fact that if two processes are exchanged, then those variables should be updated since the positions in the state vector have changed. This is the notion of reference. If a byte variable n is a reference to the position of some process, then the value of n must be updated when two processes are exchanged.

The discussion above explains the reduction that can be obtained by sorting the state vector. Actually, we are considering an equivalence relation on the system states and we are taking a representative for each equivalence class. Sorting is just one possible algorithm to get such a representative but the technique extends indeed to other algorithms giving a representative.

2.2 New Keywords for Promela

In order to use the two notions previously described we add two keywords to the Promela language, `ref` to declare reference variables, and `public` to declare local variables with a public scope. A reference variable of proctype P is used to access the public variables of a process of proctype P. A complete example is given in Figure 9.

For instance let us imagine a server with its own channel of messages. A regular declaration in Promela would be:

```
chan q[2] = [4] of {mtype};

active [2] proctype server () {
  q[_pid]?req;
  ...
}

active [3] proctype client () {
  if
  :: q[0]!req;
  :: q[1]!req;
  fi
}
```

The channel array `q` has to be a global variable since any process should be able to send messages to these channels. When two processes are exchanged, mailbox contents should also be exchanged. We propose to declare such a mailbox variable as a public element of the process:

```
active [2] proctype server () {
  public chan q = [4] of {mtype};
  q?req;
  ...
}

active [3] proctype client () {
  server[anyRef(server)].q!req
}
```

Actually the second declaration does not change the way the array is implemented in the model checker. The keyword public implicitly declares a global array. The size of the array is the number of instances of the proctype. We use a short cut `anyRef` which allows to choose a random reference in the servers.

Now if we want the client to send its identifier in the request. A local variable named `_lid` - for local identifier - is also defined. Its value is the number of the instance in the type, starting at 0. This enables to easily index array using the `_lid` variable. We get the following code:

```
active [2] proctype server () {
  public chan q = [4] of {mtype, ref client};
  ref client cli;
  q?req,cli;
  ...
}

active [3] proctype client () {
  server[anyRef(server)].q!req,_lid
}
```

2.3 Requirements

In order to use this abstraction some precise requirements must be fulfilled.

- One can only assign to a reference variable r of proctype P the values `_undef_ref`, or `anyRef`(P), or `_lid` if we are inside the proctype P, or s if s is also a reference variable of proctype P. In particular, it is not allowed to assign a constant such as 1 to a reference variable.
- One can only test whether a reference variable r of proctype P is equal to `_undef_ref`, or to `_lid` if we are inside the proctype P, or to s if s is also a reference variable of proctype P. It is neither allowed to use another comparison such as $<$ or $\leq$, nor to test whether r equals some constant such as 1. Some examples are presented below.

```
ref client cli;
int n;
chan q = [2] of {ref client};
ref server ser;
```

Allowed in client	Not allowed in client
`cli==_lid`	`cli==n`
`cli=_lid`	`cli=1`
`cli=_undef_ref`	`cli<_lid`
`q!_lid`	`q!3`
`q!anyRef(client)`	`q!ser`
`ser=anyRef(server)`	`ser==cli`

The `anyRef` function used in previous listings returns a reference of the specified type. From the Promela writer point of view this can be considered as a function which returns a random reference. From the model checker writer this is expanded as n transitions where n is the number of instances of the considered proctype. For instance the following code from a previous example:

```
q[anyRef(server)]!req
```

is expanded as:

```
if
:: q[0]!req
:: q[1]!req
fi
```

3 Abstraction Based on Process Permutations

This section gives a formal framework for the abstraction we are performing.

3.1 Syntactic Definition

Our aim is to study a system consisting of an asynchronous product of n automata $\mathcal{A}_i = (Q, V \cup W, P \cup R \cup S, I_i, T_i)$ where Q is a finite set of states. The sets V and W contain the public and the global (non-reference) variables of $\mathcal{A}_i$. The private (non-reference) variables of $\mathcal{A}_i$ can be encoded into its states, hence we do not need to keep them explicitly. Our reduction will permute the automata $\mathcal{A}_i$ and since the values of the reference variables must be updated, we need to keep them explicitly even if they are private variables. The sets P, R, S contains respectively the private, the public and the global reference variables of the system. Without loss of generality, we have assumed the same sets of states and of variables for all automata.

The set of initial states is $I_i \subseteq Q$. The set T_i consists of transitions which are tuples of the form (q, g, a, q') where $q, q' \in Q$ are the source and target states of the transition, g is the guard which must evaluate to true in order to enable the transition, and a is the action which modifies the variables of the system.

We define the set of visible references without indirection by $X_0 = P \cup R \cup S$. We define the set of visible references with k indirection by $X_k = R \times X_{k-1}$.

For instance, in the following code the level of indirection is $k = 2$.

```
ref server servid;
clientRef[serverRef[servid]]=lid;
```

We define the set X of visible references by

$$X = \bigcup_{0 \le k \le k_{max}} X_k$$

where k_{max} is the maximum level of indirection which can be found in the code. We also define $Y = W \cup V \cup (V \times X)$ to be the set of visible (non-reference) variables.

A guard g of a transition is a boolean function with domain

$$\mathbb{N}^Y \times \{0,1\}^{X \times X} \times \{0,1\}^{X \times \{\texttt{undef}\}} \times \{0,1\}^{X \times \{\texttt{lid}\}}.$$

Intuitively, the truth of the guard g depends only on the values of the visible non-reference variables ($\mathbb{N}^Y$) and on whether a visible reference variable from X equals another visible reference variable ($\{0,1\}^{X \times X}$), or is undefined ($\{0,1\}^{X \times \{\texttt{undef}\}}$), or is the local identifier ($\{0,1\}^{X \times \{\texttt{lid}\}}$).

An action a is a (possibly empty) sequence of assignments to visible variables from $X \cup Y$. An assignment is either a pair (v, f) where $v \in Y$ and f is a mapping from $\mathbb{N}^Y$ to $\mathbb{N}$ (the new value of a visible non-reference variable depends only on the old values of the visible variables); or a pair (r, h) where $r \in X$ and $h \in X \cup \{\texttt{undef}, \texttt{lid}, \texttt{anyRef}(A)\}$ for some $A \subseteq \{1, \ldots, n\}$. The intuition is that one can only assign to a reference variable the value of another reference variable or one of the values `undef`, `lid`, $\texttt{anyRef}(A)$.

3.2 Semantic Definition

In the asynchronous product $\mathcal{M} = \prod \mathcal{A}_i = (\mathcal{S}, \mathcal{I}, \mathcal{T})$, each automaton $\mathcal{A}_i$ must use a separate copy of the local variables V, P and R. Hence, the actual non-reference variables of the system are $\mathcal{V} = W \cup (V \times \{1, \ldots, n\})$ and the actual reference variables of the system are $\mathcal{R} = S \cup (P \times \{1, \ldots, n\}) \cup (R \times \{1, \ldots, n\})$.

The set of concrete states of $\mathcal{M}$ is $\mathcal{S} = Q^n \times \mathbb{N}^{\mathcal{V}} \times \{0, \ldots, n\}^{\mathcal{R}}$ and the set of concrete initial states of $\mathcal{M}$ is $\mathcal{I} = (\prod_{1 \le i \le n} I_i) \times \{0\}^{\mathcal{V}} \times \{0\}^{\mathcal{R}}$.

Since we are considering an asynchronous product, the set of transitions is defined by $\mathcal{T} = \mathcal{T}_1 \cup \cdots \cup \mathcal{T}_n \subseteq \mathcal{S} \times \mathcal{S}$ where $(q, \nu, \rho, q', \nu', \rho') \in \mathcal{T}_i$ if there exists some transition $(p, g, a, p') \in T_i$ such that $q_i = p$ and the guard g evaluates to true in state (i, ν, ρ) and the new state q' is defined by $q'_i = p'$ and $q'_j = q_j$ for all $j \neq i$ and (ν', ρ') is obtained by executing the sequence of assignments defined by the action a.

In order to give a formal definition, we first define the actual variables associated with the visible variables from $X \cup Y$ in state (i, ρ).

- For $r \in X$, the actual variable $r_{i,\rho}$ associated with r in state (i, ρ) is defined by $r_{i,\rho} = r$ if $r \in S$, $r_{i,\rho} = (r, i)$ if $r \in P \cup R$, and $r_{i,\rho} = (s, \rho(t_{i,\rho}))$ if $r = (s, t) \in R \times X_k$ for some $k \ge 0$.
- For $v \in Y$, the actual variable $v_{i,\rho}$ associated with v in state (i, ρ) is defined by $v_{i,\rho} = v$ if $v \in W$, $v_{i,\rho} = (v, i)$ if $v \in V$, and $v_{i,\rho} = (w, \rho(r_{i,\rho}))$ if $w = (w, r) \in V \times X$.

Now the evaluation of the guard g in state (i, ν, ρ) is

$$g(\nu(v_{i,\rho})_{v \in Y}, (\rho(r_{i,\rho}) = \rho(s_{i,\rho}))_{r,s \in X}, (\rho(r_{i,\rho}) = 0)_{r \in X}, (\rho(r_{i,\rho}) = i)_{r \in X})$$

which must be true for the transition to be enabled.

An assignment (v, f) to a visible non-reference variable $v \in Y$ changes the current value of the variable $v_{i,\rho}$ to $f((\nu(u_{i,\rho}))_{u \in Y})$.

An assignment (r, h) to a visible reference variable $r \in X$ changes the current value of the variable $r_{i,\rho}$ to $\rho(h_{i,\rho})$ if $h \in X$, or to 0 if $h = \texttt{undef}$, or to i if $h = \texttt{lid}$, or to any value from A if $h = \texttt{anyRef}(A)$ (this $\texttt{anyRef}(A)$ assignment creates several transitions, to different states).

3.3 Permutations of States

Here is a definition which specifies what happens to a concrete state of the system $\mathcal{M}$ when we permute the processes.

Definition 1. *Let π be a permutation over $\{1, \ldots, n\}$ which is extended to $\{0, \ldots, n\}$ by setting $\pi(0) = 0$.*

We first extend π to $\mathcal{V} \cup \mathcal{R}$: for $v \in W \cup S$ we set $\pi(v) = v$, and for $v \in V \cup P \cup R$ and $1 \le i \le n$, we set $\pi((v, i)) = (v, \pi(i))$.

Next, for a concrete state $s = (q, \nu, \rho) \in \mathcal{S}$ of $\mathcal{M}$, we define

$$\pi(s) = (q \circ \pi, \nu \circ \pi, \pi^{-1} \circ \rho \circ \pi).$$

Intuitively, applying the permutation π to the state s results in a new state $\pi(s)$ where the process at position i in $\pi(s)$ is the process at position $\pi(i)$ in s.

Our system consists of instances of a few proctypes. For instance, we may have only two proctypes, e.g. client and server. Then we can write $\{1,\dots,n\}$ as a disjoint union $A \dot\cup B$ where A and B correspond to the indices of the client and server instances. Since all clients (resp. servers) share the same code, they are modeled with the same initial states and the same transitions: $I_i = I_A$ and $T_i = T_A$ for all $i \in A$ and similarly for the servers. Also, we restrict the `anyRef` assignment to use sets of indices corresponding to some proctype. In the example above, only `anyRef`(A) and `anyRef`(B) are allowed.

It is only meaningful to permute processes of the same type, hence we will only consider permutations that preserve proctypes. Formally, we say that a permutation π of $\{1,\dots,n\}$ preserves proctypes if for any proctype, the set of indices corresponding to the instances of this proctype is invariant under the permutation π. In the example above, this means that $\pi(A) = A$ and $\pi(B) = B$. Note that, if the permutation π preserves proctypes then we have $I_{\pi(i)} = I_i$ and $T_{\pi(i)} = T_i$ for all $i \in \{1,\dots,n\}$. We are going to show that such permutations do not have any effect on the transitions which can be triggered.

Lemma 1. *Let $s, s' \in \mathcal{S}$ be two concrete states of $\mathcal{M}$ and let π be a permutation of $\{1,\dots,n\}$ which preserves proctypes. If $(s,s') \in \mathcal{T}$ then $(\pi(s),\pi(s')) \in \mathcal{T}$.*

Proof. Let $s = (q,\nu,\rho)$ and $s' = (q',\nu',\rho')$ be such that $(s,s') \in \mathcal{T}$. Let $i \in \{1,\dots,n\}$ be such that $(s,s') \in \mathcal{T}_i$ and let $(p,g,a,p') \in T_i$ be the associated transition. Let π be a permutation of $\{1,\dots,n\}$ preserving proctypes.

We want to prove that $(\pi(s),\pi(s')) \in \mathcal{T}_j$ with $j = \pi^{-1}(i)$ and the same associated transition $(p,g,a,p') \in T_j = T_{\pi(j)} = T_i$. We let $\pi(s) = (\tilde q,\tilde\nu,\tilde\rho)$ and $\pi(s') = (\tilde q',\tilde\nu',\tilde\rho')$.

$$\begin{array}{ccc} s = (q,\nu,\rho) & \xrightarrow{\quad\mathcal{T}_i\quad} & s' = (q',\nu',\rho') \\ \Big\downarrow \pi & & \Big\downarrow \pi \\ \pi(s) = (\tilde q,\tilde\nu,\tilde\rho) & \xrightarrow{\quad\mathcal{T}_j\quad} & \pi(s') = (\tilde q',\tilde\nu',\tilde\rho') \end{array}$$

Claim.
(1) For all $r \in X$, we have $\pi(r_{j,\tilde\rho}) = r_{i,\rho}$.
(2) For all $v \in Y$, we have $\pi(v_{j,\tilde\rho}) = v_{i,\rho}$.

Proof of (1). Let $r \in X = \bigcup_k X_k$. We proceed by induction on k.
– If $r \in S$, then $r_{j,\tilde\rho} = r = r_{i,\rho} = \pi(r)$.
– If $r \in P \cup R$, then $\pi(r_{j,\tilde\rho}) = \pi((r,j)) = (r,i) = r_{i,\rho}$.
– Assume the result holds for some $k \ge 0$ and let $r = (s,t) \in R \times X_k$. We have

$$\pi(r_{j,\tilde\rho}) = \pi((s,\tilde\rho(t_{j,\tilde\rho}))) = \pi((s,\pi^{-1}\circ\rho\circ\pi(t_{j,\tilde\rho}))) = (s,\rho(t_{i,\rho})) = r_{i,\rho}.$$

Proof of (2).
– If $v \in W$, then $v_{j,\tilde\rho} = v = v_{i,\rho} = \pi(v)$.

– If $v = (w, t) \in V \times X$. We have

$$\pi(v_{j,\tilde{\rho}}) = \pi((w, \tilde{\rho}(t_{j,\tilde{\rho}}))) = \pi((w, \pi^{-1} \circ \rho \circ \pi(t_{j,\tilde{\rho}}))) = (w, \rho(t_{i,\rho})) = v_{i,\rho}.$$

We deduce first that the evaluation of the guard g at $(j, \tilde{\nu}, \tilde{\rho})$ equals the evaluation of the guard g at (i, ν, ρ). Indeed, the arguments are the same since for all $u \in Y$ we have $\tilde{\nu}(u_{j,\tilde{\rho}}) = \nu \circ \pi(u_{j,\tilde{\rho}}) = \nu(u_{i,\rho})$; and for all $r \in X$ we have $\tilde{\rho}(r_{j,\tilde{\rho}}) = \pi^{-1} \circ \rho \circ \pi(r_{j,\tilde{\rho}}) = \pi^{-1} \circ \rho(r_{i,\rho})$. Hence,

- $\tilde{\rho}(r_{j,\tilde{\rho}}) = \tilde{\rho}(s_{j,\tilde{\rho}})$ if and only if $\rho(r_{i,\rho}) = \rho(s_{i,\rho})$,
- $\tilde{\rho}(r_{j,\tilde{\rho}}) = 0$ if and only if $\rho(r_{i,\rho}) = 0$, and
- $\tilde{\rho}(r_{j,\tilde{\rho}}) = j$ if and only if $\rho(r_{i,\rho}) = i$.

It remains to show that executing from $\pi(s)$ the transition associated with $(p, g, a, p') \in T_i = T_j$ in the j-th automaton yields the state $\pi(s')$. We give the proof when the action a consists of a single assignment. The generalization to a sequence of assignments is easy.

- $\tilde{q}'_j = q'_{\pi(j)} = q'_i = p'$ and for $k \neq j$, we have $\pi(k) \neq i$ and therefore $\tilde{q}'_k = q'_{\pi(k)} = q_{\pi(k)} = \tilde{q}_k$.
- If the assignment is (v, f) with $v \in Y$ then we have $\tilde{\nu}'(v_{j,\tilde{\rho}}) = \nu' \circ \pi(v_{j,\tilde{\rho}}) = \nu'(v_{i,\rho}) = f((\nu(u_{i,\rho}))_{u \in Y}) = f((\nu \circ \pi \circ \pi^{-1}(u_{i,\rho}))_{u \in Y}) = f((\tilde{\nu}(u_{j,\tilde{\rho}}))_{u \in Y})$.
- If the assignment is (r, h) with $r \in X$ then we have
  - If $h \in X$ then $\tilde{\rho}'(r_{j,\tilde{\rho}}) = \pi^{-1} \circ \rho' \circ \pi(r_{j,\tilde{\rho}}) = \pi^{-1} \circ \rho'(r_{i,\rho}) = \pi^{-1} \circ \rho(h_{i,\rho}) = \pi^{-1} \circ \rho \circ \pi(h_{j,\tilde{\rho}}) = \tilde{\rho}(h_{j,\tilde{\rho}})$.
  - If $h =$ `undef` then $\tilde{\rho}'(r_{j,\tilde{\rho}}) = \pi^{-1} \circ \rho'(r_{i,\rho}) = \pi^{-1}(0) = 0$.
  - If $h =$ `lid` then $\tilde{\rho}'(r_{j,\tilde{\rho}}) = \pi^{-1} \circ \rho'(r_{i,\rho}) = \pi^{-1}(i) = j$.
  - If $h =$ `anyRef`(A) then $\tilde{\rho}'(r_{j,\tilde{\rho}}) = \pi^{-1} \circ \rho'(r_{i,\rho}) \in A$ since $\rho'(r_{i,\rho}) \in A$ and π is a permutation preserving proctypes.

Therefore, we have shown that $(\pi(s), \pi(s')) \in \mathcal{T}_j$.

3.4 Abstraction

An abstraction consists of representing several states by a single state. We will be using an equivalence relation $\mathbf{R}$. Then an element will be chosen in each equivalence class to represent all the class.

Using the permutations on $\mathcal{M}$ we can define the equivalence relation $\mathbf{R}$ between concrete states of $\mathcal{M}$: we say that two concrete states s and s' of $\mathcal{M}$ are $\mathbf{R}$-equivalent if $\pi(s) = s'$ for some permutation π of $\{1, \ldots, n\}$.

Here is now the main property which validates our abstraction.

Theorem 1. *The equivalence relation* $\mathbf{R}$ *is a bisimulation on* $\mathcal{M}$.

Proof. Let s, s', s_1 be three states of $\mathcal{M}$ such that $(s, s') \in \mathcal{T}$ and $s \,\mathbf{R}\, s_1$. We have to prove that there exists s'_1 such that $(s_1, s'_1) \in \mathcal{T}$ and $s' \,\mathbf{R}\, s'_1$. Note that since $\mathbf{R}$ is an equivalence relation, we do not have to prove the converse.

By definition of $\mathbf{R}$, there exists a permutation π of $\{1, \ldots, n\}$ such that $\pi(s) = s_1$. Let $s'_1 = \pi(s')$, hence we have $s' \,\mathbf{R}\, s'_1$. By Lemma 1, we know that $(s_1, s'_1) \in \mathcal{T}$, which proves the theorem.

The quotient of $\mathcal{M}$ by the equivalence $\mathbf{R}$ is the transition system $\bar{\mathcal{M}} = (\bar{\mathcal{S}}, \bar{\mathcal{T}}, \bar{\mathcal{I}})$ where: $\bar{\mathcal{S}} = \{\bar{s} \mid s \in \mathcal{S}\}$, $\bar{\mathcal{I}} = \{\bar{s} \mid s \in \mathcal{I}\}$ and $\bar{\mathcal{T}} = \{(\bar{s}, \bar{s'}) | (s, s') \in \mathcal{T}\}$. Now it is well known that when an equivalence relation $\mathbf{R}$ on a transition system $\mathcal{M}$ is a bisimulation, then the quotient $\bar{\mathcal{M}}$ is bisimilar to $\mathcal{M}$.

3.5 Pragmatic Abstraction

In order to apply the above reduction, one has to compute the quotient $\bar{\mathcal{M}}$. A possibility is to choose a canonical representative in each class and to use a function $f : S \to S$ mapping each state to its canonical representative. Such a mapping f satisfies the two properties

1. for all $s \in S$, $f(s)\,\mathbf{R}\,s$,
2. for all $s, s' \in S$, $s\,\mathbf{R}\,s'$ implies $f(s) = f(s')$ (canonical representative).

Actually, our mapping f is given by the pseudo-sorting algorithm presented in Section 4.1. It maps each state s to a permuted state $f(s)$ hence it satisfies (1) but it does not necessarily satisfy (2). Therefore, our reduced system may not be isomorphic to the quotient $\bar{\mathcal{M}}$ and we need to prove that it is still bisimilar to the original system.

Proposition 1. *Let $\mathcal{M} = (\mathcal{S}, \mathcal{I}, \mathcal{T})$ be a transition system and let $\mathbf{R}$ be an equivalence relation on $\mathcal{S}$ which is a bisimulation. Let $f : \mathcal{S} \to \mathcal{S}$ be a mapping satisfying $f(s)\,\mathbf{R}\,s$ for all $s \in \mathcal{S}$. We define the reduced system $\mathcal{M}' = (\mathcal{S}, \mathcal{I}, \mathcal{T}')$ by $\mathcal{T}' = \{(s, f(s')) \mid (s, s') \in \mathcal{T}\}$.*

The relation $\mathbf{R}$ defines a bisimulation between $\mathcal{M}$ and $\mathcal{M}'$.

Proof. let $s, s', s_1 \in \mathcal{S}$ be such that $(s, s') \in \mathcal{T}$ and $s\,\mathbf{R}\,s_1$. There exists $s'_1 \in \mathcal{S}$ such that $(s_1, s'_1) \in \mathcal{T}$ and $s'\,\mathbf{R}\,s'_1$. We deduce that $(s_1, f(s'_1)) \in \mathcal{T}'$ and $s'\,\mathbf{R}\,s'_1\,\mathbf{R}\,f(s'_1)$.

Let $s_1, s'_1 \in \mathcal{S}$ be such that $(s_1, s'_1) \in \mathcal{T}'$. There exists s''_1 such that $(s_1, s''_1) \in \mathcal{T}$ and $s'_1 = f(s''_1)$. Now let s be such that $s\,\mathbf{R}\,s_1$. There exists $s' \in \mathcal{S}$ such that $(s, s') \in \mathcal{T}$ and $s'\,\mathbf{R}\,s''_1$. We are done since $s'\,\mathbf{R}\,s''_1\,\mathbf{R}\,s'_1$.

3.6 Permutable Specification

Now the question is to determine which properties can be verified on the reduced system $\mathcal{M}'$ (or $\bar{\mathcal{M}}$). Since the original system is bisimilar to the reduced one, we can verify any CTL* property provided the state formulas are invariant under permutation. This includes a lot of interesting properties. For instance, in a mutual exclusion algorithm we want to check whether there exists an accessible state such that two processes are in the critical section. Such a property is invariant under permutation.

On the contrary, consider the following response property: "whenever process 1 requests a resource, process 1 will eventually be granted the resource". Such a property cannot be checked directly on our reduced system. If we write literraly this property, then Spin may erroneously detect that it holds or that it does not hold. For instance, if the process at position 1 requests the resource and only gets

the resource when being in some other position, Spin may erroneously say that the property does not hold. Also, if whenever some process at position 1 requests the resource then in the next step some other process that has the resource takes position 1, then Spin may erroneously say that the property holds.

Therefore, in order to use our reduction method, one has to be careful in writing the specification to be checked. Explicit references to process positions should be avoided. It would be very interesting to define a specification language which guarantees that the specifications are invariant under permutations. Another interesting direction is to permute the specification when permuting the state vector after each step of the computation.

4 Computation of the Reduced System

The state vector consists of a tuple giving for each variable of the system its present value. This state vector has however some structure. The global variables occur only once and the local variables declared in each proctype are replicated for each instance of this proctype. Hence, each process running in the system has an associated tuple of variables in the state vector. Permuting the processes of some proctype A means permuting the associated tuples of variables in the state vector and modifying accordingly the values of the reference variables to this proctype as explained in Section 2.

4.1 The Pseudo-Sorting Algorithm

The mapping f transforming each state into an equivalent one is implemented by a sorting algorithm. More precisely, we consider the lexicographic order on the tuples of variables associated with the processes of some proctype A and we sort these tuples according to this lexicographic order. Indeed, we perform this sorting operation for each proctype. We use a classical sorting algorithm, like quick sort [7] for instance. We consider the sort procedure as a procedure which takes three arguments: the list to sort, the procedure to exchange two arguments in the list, the function to compare two arguments. The algorithm is presented in Figure 2.

Actually this algorithm can lead to state vectors that are still not sorted. This is why we call it pseudo-sorting. The final vector might not be sorted because when we exchange two processes we change reference values afterwards, and this might change the relative ordering. For instance, consider a system with only 3 processes of some proctype A defined by

```
active [3] proctype A ()
{ ref A r;
  public byte v;
  /* A body */
}
```

Note that there is always an implicit `_state` variable that is declared last. Hence the tuple of variables associated with a process is `(r,v,_state)`. A possible state

procedure $Exch_i(k,l)$
  exchanges data in state vector at the positions of
    k-th and l-th processes of type i.
  For all references in the state vector to processes of type i
    if the reference is k change it to l
    else if the reference is l change it to k
Function $Compare_i(k,l)$
  returns if k-th process of type i
    is smaller or equal to l-th process of type i.
procedure $pseudoSortStateVector()$
  For each type of process i
    Let l_i be the list of processes of type i.
    Sort(l_i,$Exch_i$,$Compare_i$)

Fig. 2. Pseudo sorting algorithm

vector is $3,1,1|0,3,4|1,2,1$. It is not sorted. Swapping the first two processes gives the state $0,3,4|3,1,1|2,2,1$. Note that the reference value of the third process has changed. This state vector is still not sorted. Now swapping the last two processes yields the state $0,3,4|3,2,1|2,1,1$ which again is not sorted. Therefore no permutations of the processes yield a sorted state vector.

4.2 Efficiency Depends on the Variable Declaration Order

In the tuple of variables associated with some proctype A, the order of the variables is the declaration order. If we change the declaration order then the lexicographic order is changed accordingly and the reduction that we get may be completely different.

Let us consider a very simple example where our system consists only of 3 clients and 3 servers where each server can be connected with a single client. We assume that each client (respectively server) keeps a reference to the server (respectively client) it is connected to. In addition, clients and servers have a public variable.

```
active [3] proctype Client ()
{ ref Server r;
  public byte v;
  /* Client body */
}
```

```
active [3] proctype Server ()
{ ref Client r;
  public byte v;
  /* Server body */
}
```

Note that there is always an implicit `_state` variable that is declared last. Hence the tuple of variables associated with a client or a server is `(r,v,_state)`. An example is shown on Figure 3 for the first value r of each instance in the state vector.

In this example, the state vector is $2,1,2|3,2,3|1,3,4||3,4,3|1,5,2|2,6,1$ where the vertical bars separate the process tuples and the double bar separates the clients from the servers. Applying the sorting algorithm on the clients

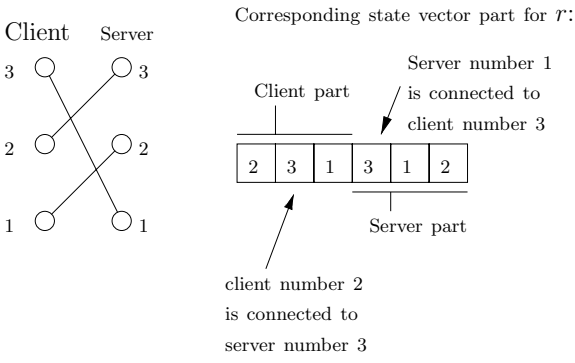


Fig. 3. An example of configuration

gives the state vector $1,3,4|2,1,2|3,2,3||1,4,3|2,5,2|3,6,1$. Note that the client references in the server tuples have been changed accordingly. Now the state vector is also sorted according to the server tuples. As shown on Figure 4, from any initial connection graph we get the most simple configuration possible. Of course this dramatically reduces the state space.

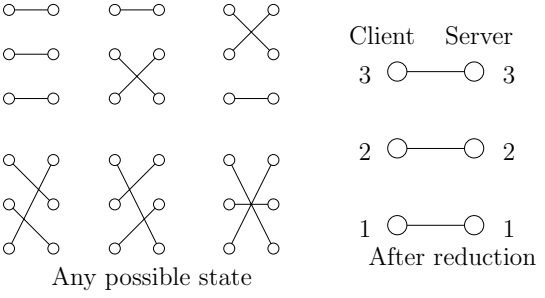


Fig. 4. Before and after a reduction

Now, assume that the declaration order between r and v is reversed as in

```
active [3] proctype Client ()
{ public byte v;
  ref Server r;
  /* Client body */
}
```

```
active [3] proctype Server ()
{ public byte v;
  ref Client r;
  /* Server body */
}
```

Then the state vector associated with the same configuration is $1,2,2|2,3,3|3,1,4\ \ ||4,3,3|5,1,2|6,2,1$. Since this vector is already sorted, no reduction will occur.

This very simple example illustrates why the order in which variables are declared has a great influence on the reduction obtained. As stated above, there is always an implicit variable `_state` which is declared last. If we want to change the position in the tuple of this variable we could declare it explicitly as in

```
active [3] proctype Client ()
{ byte _state;
  public byte v;
  ref Server r;
  /* Client body */
}
```

The results of a more realistic experiment is shown in Figure 5. Here we consider a Client-Server system whose complete description is given in the appendix (Figure 9). We denote by n the number of clients in the system, n is also the number of servers. A client can be idle or can try to connect to a server. The connection will be granted if the server is not busy. The first (respectively second) column of Figure 5 corresponds to the reduced system when the variable `serv_id` of proctype client is declared first (respectively last). The third column corresponds to the system without reduction. This experiment shows that our reduction is quite efficient and that it depends noticeably on the order in which the variables are declared. Hence, in order to get the best benefit from our method, one must have an intuition of which order will be good. This is difficult to guess, and one need to have a good understanding of the system being modeled.

n	ref. first	ref. last	no reduction
2	27	40	63
3	108	227	918
4	405	928	16 929
5	1 458	3 518	375 678

Fig. 5. Number of states for different orders and different sizes

4.3 Layered Model

The pseudo sorting method will work well on layered models. Here is an informal description of a layered model. We have ℓ type of processes (like client, server ...), named $t_1, \ldots, t_\ell$. Each process of type t_i has references to some other process of type t_{i+1} or has an undefined reference. We assume all defined references to process of type t_i are different. This is shown on Figure 6 We assume that the reference to layer $i+1$ comes first in the lexicographic order for an instance of type t_i. If we sort according to proctype $t_{\ell-1}$, then $t_{\ell-2}$ until t_1, then from any element of the equivalence class of $\mathbf{R}$ we will get the same element.

Figure 7 shows the result for a three tier model [8] (client/server/data base).

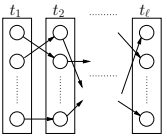


Fig. 6. A layered model

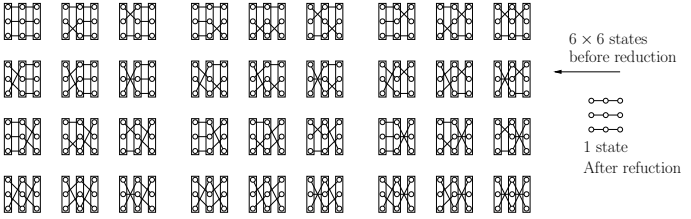


Fig. 7. Three tier model

5 Conclusions

We have introduced two notions which enables abstraction for distributed systems: references to other processes, and a "public" scope for local variables.

These notions need to be used carefully, in a way that all processes of the same type can be exchanged. We have clearly stated in Section 2.3 what are those constraints. They can be easily checked when the Promela model is compiled.

In the future we should specify how to easily write LTL formula which are unchanged by permutations. It would also be interesting to write pluggable modules, to implement other heuristics than the pseudo sorting algorithm to get canonical representatives of equivalence classes of **R**.

Appendix

This appendix shows a practical implementation on Figure 9. Figure 8 shows how our abstraction has been implemented.

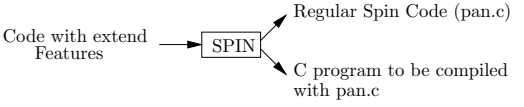


Fig. 8. Software architecture

Code with extented features

```
mtype = {ack,nack,rel,req};

active [3] proctype client () {
  ref server serv_id;
  public chan cliChan =
                [4] of {mtype};
idle :
  serv_id=anyRef(server);
  server[serv_id].serChan!req(_lid);
  if
  :: cliChan?ack; goto work
  :: cliChan?nack; goto idle
  fi;
work:
  server[serv_id].serChan!rel,0;
  goto idle
}

active [2] proctype server () {
  public chan serChan =
          [4] of {mtype,ref server};
  byte inUse;
  ref client cli;

  do
  :: serChan?req(cli) ->
    if
    :: inUse==0 ->  inUse=1;
      client[cli].cliChan!ack,_lid
    :: inUse==1 ->
      client[cli].cliChan!nack,0
    fi;
  :: serChan?rel,_ -> inUse=0
  od
}
```

Generated Promela code

```
mtype = {ack,nack,rel,req};

chan cliChan[3]=
  [4] of {mtype};
chan serChan[3]=
  [4] of {mtype,byte};

#define _lid _pid-0
active [3] proctype client () {
  byte serv_id;
idle :
  if
  :: serv_id=0
  :: serv_id=1
  fi;
  serChan[serv_id]!req(_lid);
  if
  :: cliChan[_lid]?ack; goto work
  :: cliChan[_lid]?nack; goto idle
  fi;
work:
  serChan[serv_id]!rel,0;
  goto idle
}
#undef _lid

#define _lid _pid-3
active [2] proctype server () {
  byte inUse;
  byte cli;
  do
  :: serChan[_lid]?req(cli) ->
    if
    :: inUse==0 -> inUse=1;
      cliChan[cli]!ack,_lid
    :: inUse==1 ->
      cliChan[cli]!nack,0
    fi
  :: serChan[_lid]?rel,_ ->
      inUse=0
  od
}
#undef _lid
```

Fig. 9. Example of new keywords

We start from the Promela code with our extended features. Then our modified version of Spin generates a regular model checking code (`pan.c`), together with a file which will be compiled with `pan.c`.

References

1. C. N. Ip and D. L. Dill. Better Verification through Symmetry. *International Conference on Computer Hardware Description Languages*, pages 87–100, April 1993.
2. C. N. Ip and D. L. Dill. Efficient Verification of Symmetric Concurrent Systems. In *IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 230–234. IEEE Computer Society, 1993.
3. C. Norris Ip and D. L. Dill. Verifying Systems with Replicated Components in Murφ. *Formal Methods in System Design*, 1997.
4. G. J. Holzmann. The Spin Model Checker. *IEEE Trans. on Software Engineering*, 23(5):279–295, May 1997.
5. D. Bošnački, D. Dams, and L. Holenderski. Symmetric spin. In *Proceedings of the 7th SPIN Workshop*, volume 1885 of *Lecture Notes in Computer Science*, pages 1–19. Springer Verlag, 2000.
6. E. A. Emerson and A. P. Sistla. Symmetry and Modelchecking. *Formal Methods in System Design*, 9(1):105–130, 1996.
7. D. E. Knuth. *The Art of Computer Programming*, volume 3, chapter 5, pages 114–123. Addison Wesley, 1973.
8. N. G. Depledge, W. A. Turner, and A. Woog. An open, distributable, three-tier client-server architecture with transaction semantics. *Digital Technical Journal*, 7(1), 1995.

A SPIN-Based Model Checker for Telecommunication Protocols*

Vivek K. Shanbhag and K. Gopinath

CSA Dept, Indian Institute of Science, Bangalore, 560 012 INDIA
{vivek, gopi}@csa.iisc.ernet.in

Abstract. Telecommunication protocol standards have in the past and typically still use both an English description of the protocol (sometimes also followed with a behavioural SDL model) and an ASN.1 specification of the data-model, thus likely making the specification incomplete. ASN.1 is an ITU/ISO data definition language which has been developed to describe abstractly the values protocol data units can assume; this is of considerable interest for model checking as subtyping in ASN.1 can be used to constrain/construct the state space of the protocol accurately. However, with current practice, any change to the English description cannot easily be checked for consistency while protocols are being developed. In this work, we have developed a SPIN-based tool called EASN (Enhanced ASN.1) where the behaviour can be formally specified through a language based upon Promela for control structures but with data models from ASN.1. An attempt is also made to use international standards (X/Open std on ASN.1/C++ translation) as available so that the tool can be realised with pluggable components. One major design criterion is to enable incremental computation wherever possible (for example: hash values, consistency between alternate representations of state). We have used EASN to validate a simplified model of RLC (Radio Link Control) in the W-CDMA stack that imports datatypes from its associated ASN.1 model. In this paper, we discuss the motivation and design of the EASN language, the architecture and implementation of the verification tool for EASN and some preliminary performance indicators.

1 Introduction

Next generation protocols for mobile devices have become very complex and it is becoming increasingly difficult for standards bodies to be sure of the correctness of protocols during the standardization process. This has become an impediment in defining new standards. What one needs is a way of specifying a protocol and have some confidence that, at a certain level of abstraction, the protocol is consistent in spite of modifications.

* Thanks are due to Nokia Research Center, Helsinki for funding this work under SID project 99033. We thank Ari Ahtiainen and Markku Turunen of NRC for their initial project formulation and some key ideas in the software engineering aspects, Dinesh Shanbhag for helping us understand the ASN.1/C++ standard, and Matti Luukkainen, University of Helsinki, for many suggestions and criticisms.

M.B. Dwyer (Ed.): SPIN 2001, LNCS 2057, pp. 252–271, 2001.

There are languages like Promela that can be used, but their data structuring capabilities do not match those that are used in telecommunication protocols. ASN.1[6] (Abstract Syntax Notation One) is a widely used data definition language in telecommunication protocol specification. It will help the standardization process if a model checker could be augmented with ASN.1 data modelling capabilities to check correctness of interim versions of a protocol before establishing a standard. Inspite of prototypes that are built, they often cannot exercise all aspects of a protocol, especially those that are evolving.

In addition, due to the presence of the subtyping mechanism in ASN.1, model checking can be more effective as unreachable parts of the state space that could be introduced in simpler data models in other languages need not be considered.

Hence, we have designed an Enhanced ASN.1 language, EASN, that combines the control structures of Promela with the data definition capabilities of ASN.1. We present our verification tool for EASN and its architecture. We derive our implementation from SPIN, to benefit from its many capabilities.

1.1 Why ASN.1?

ASN.1 separates data modelling into abstract and transfer syntax. The abstract syntax only specifies the universe of abstract values that can be assumed by variables in the model without any concern for how they are mapped to a particular machine, compiler, OS, etc. Hence from the point of view of model checking, an abstract syntax constrains the state space as much as possible *if* there is a mechanism by which a system state vector can be encoded with exactly only the possible values of its constituent substates. The latter is a chief feature of the state compaction infrastructure that has been developed for EASN. This is equivalent to model checking with abstract data models that does not require examining unreachable parts of the system state space introduced due to lack of subtyping, etc. We primarily exploit ASN.1's subtyping feature which is a well developed notation for expressing constraints. Note that data here actually means the control data in the protocols and hence our concerns are different from those approaches that exploit symmetry, etc. While TTCN, the test language in ISO/ITU communities, uses ASN.1, our attempt is to marry ASN.1 with a well known model checker such as SPIN.

1.2 Why SPIN?

SPIN is an effective model checking tool for asynchronous systems, especially designed for communication protocols. The design of control constructs in Promela has been based upon those in SDL, a language that has been used to specify communication protocols since '70s. Nondeterminism and guarded commands in Promela makes it convenient to express behavior of communicating protocol entities. The model checking system SPIN[1], which uses Promela, has many capabilities like deadlock detection, validating invariants or assertions, detecting non-progress cycles and livelocks, and establishing LTL properties. Algorithms that effect substantial space and time savings, like bit-state hashing, on-the-fly

model-checking and partial-order reduction have been incorporated into SPIN. Hence, modifying the SPIN system to handle ASN.1 has been a design goal.

SPIN has a **simulator** that randomly checks only a portion of the state space and also a (generated) **validator** that can attempt to exhaustively check the state space of the system or can use techniques like bit-state hashing to check a substantial portion of the state space with a fairly high level of assurance. Our EASN system also has these components.

One major design criterion for EASN is to enable incremental computation wherever possible (for example: hash values, consistency between alternate representations of state). SPIN, however, does not do such incremental computation but is still faster for other reasons (see section 5).

1.3 EASN Language

ASN.1 can be used to define the datatypes and constant values in an application. Promela, however, is a complete language with a set of basic data types and typedef construct to help users compose datatypes, and a set of control constructs that can be used to define the behaviour of protocol entities.

The EASN Language ***replaces*** all the datatyping capabilities of Promela with ASN.1. Hence, none of the data types of Promela are retained in EASN, except the *chan* construct. Thus basic datatypes of Promela, namely, *bit*, *bool, char*, *short* and *int,* as well as related constructs, *unsigned*, *bitfields, typedef* and the *mtype* declaration are not part of EASN. Channel definition syntax and the capability of defining arrays of channels is retained as there is no similar construct in ASN.1. Defining arrays of other types through the same syntax is, however, disallowed in EASN (as the sequence-of construct in ASN.1 can be used). The subset of the ASN.1 Language that is incorporated into EASN is detailed in [12].

As ASN.1 has richer and more expressive datatypes compared to Promela, EASN needs to overload the semantics of many of the operators of Promela, so as to support a natural set of operations on data. In addition, we have also augmented the set of operators as necessary. In the first version of the language and implementation, only such operator overloadings and new operators have been included as are necessary for functional completeness. Briefly[12], therefore,

EASN = Promela - {mtype, typedef, bit, byte, bool, short, int, unsigned}
+ ASN.1 + appropriately overloaded semantics of the existing operators
+ few new operators.

1.4 Related Work

One interesting work relating to language design and protocol verification using the SPIN infrastructure is that of Promela++ at Cornell University[10]. Additions to the Promela language were made to make the resulting language suitable for expressing user level network protocols for high performance computing. If Promela++ is compiled with the verification option, it can do model checking. If

compiled with the code option, just like YACC, it produces protocol code using 'actions'. However, the 'actions' are not subject to verification through the SPIN system as they are in C.

SPIN does various kinds of state compaction and, in EASN, we have a comparable mechanism for most of them that perform at least as well in space. But some are unnecessary in EASN. Geldenhuys and Villiers[9] also attempt state compression in SPIN along similar lines as ours but by adding a simple construct to the Promela Language in an *ad hoc* fashion with restrictions. For example, different orders of process activation along different execution paths are forbidden in their approach. Also, their variable's ranges must start at zero. We do not have such restrictions.

1.5 Outline of Paper

In section 2, we give a brief overview of our EASN system. Later sections present more details of each subsystem. The section 4 discusses the relevant aspects of the SPIN implementation necessary to understand our modifications and then discusses the EASN implementation in some detail. The last section presents example runs in both EASN and SPIN and performance indicators. Finally, we end with conclusions and future work.

2 EASN, the Verification Tool

2.1 Encoding State Efficiently

SPIN represents state quite efficiently but, for reasons of alignment, etc, allows padding and other extraneous matter in the state vector. Since our system uses ASN.1 data models, we can require that all variables be as constrained as possible in the space of values that they can take through the use of subtyping. For example, if an integer variable takes values from 8..15 only, we can represent the state of that variable in 3 bits. Further, if there are only two variables that are constrained to be between, say, 5..7 and 3..7, there are only 15 possibilities and both can be represented in only 4 bits instead of either 2+3 (5 bits) or worse 3+3 (6 bits)[1]. Similarly, if an ASN.1 datatype assumes only integer values 5, 7, 11 and 13, only 2 bits of space is needed in the linearised state-vector for objects of this type.

Our design for EASN, therefore, has a critical facility called the state compaction infrastructure that views the state space of the system as a multi-dimensional array (with one dimension for every component of the state of EASN), and consequently, every state of the system, as a point in this multi-dimensional space. We use a column-major linearisation that additionally enables incremental computation of state hash values (see section 4.4). Note that a row-major linearisation is not helpful in this regard as the number of components that comprise the system keeps going up and down as the system evolves.

[1] Experienced ASN.1 users may note that such an encoding is even better than the often very compact PER encoding.

2.2 Outline of EASN System Design

Given the EASN language, one can now specify all the aspects of a protocol **formally** (instead of a mixture of English and ASN.1): the structure of the protocol data units (PDUs) and the behavior of the protocol entities. This enables us to to identify inconsistencies and incorrect, unintended and unwanted behaviors in the system if we have a verification tool for EASN with capabilities similar to that in SPIN.

Given that the EASN language is derived from Promela, can one realise a verification tool for EASN through modifications to the SPIN system? However, we need then to worry about modifiability, maintainability, upgradability, or just even maturing over a period of time, given that ASN.1, Promela and SPIN are all live, evolving pieces of work, and have matured over long periods of time.

We see three main issues: how to avoid the cost of processing of a very large language like ASN.1, current non-availability of a complete toolset for ASN.1 and how to insulate oneself from the evolution of SPIN. We discuss them below.
Processing ASN.1: SPIN is open source. We intend EASN to be open source too. Parsing ASN.1 is not easy and we wanted to avoid doing it ourselves, if we could. Nokia Research Center (NRC) has a translator, BEX, conformant with the ASN.1/C++ draft standard[11] that we could use. We wanted to use BEX while not compromising the open source goal. We have, therefore, used the NMF std[7] to architect the tool so as to enable other users besides us and NRC to realise it by plugging in any compliant ASN.1/C++ translator into the system (Figure 1).Since BEX is not completely conformant to the accepted standard, there is a need to adapt the BEX-generated C++ to requirements of the parser and simulator modules. This requirement is encapsulated into a thin layer of software that enables these modules to use BEX generated C++ source.

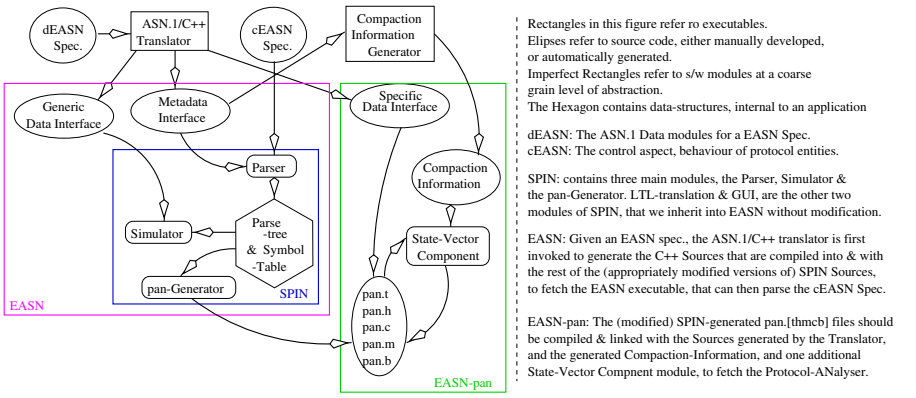


Fig. 1. The EASN system

Changes in SPIN: All our modifications to the SPIN sources are encapsulated in pre-processor conditional compilation flags. This enables us to incorporate any incremental change / bug-fix to SPIN quite easily and fast into EASN.

2.3 EASN Modules

An EASN system specification (to be simulated/verified) consists of two *compilation units*. One contains all the ASN.1 modules (the dEASN spec.) that is parsed by the translator to generate the C++ source. The other compilation unit contains the *behavioural* specification of the protocol entities (the cEASN spec.) that is parsed by the EASN parser (a modified Promela parser, derived from SPIN). It is the variable declarations in the cEASN spec that ties it to the dEASN spec as their types are defined in the ASN.1 modules. The EASN parser *imports* all the relevant information regarding a type, from the generated C++ source, by querying its meta-data interface into the internal symbol-tables of EASN.

The EASN Simulator: The EASN simulator (a modified Promela simulator, derived from SPIN, section 1.2), besides requiring the information generated by the parser, requires to access data values and modify them through permitted operations. However, since the simulator engine has no knowledge of the specific ASN.1 types that might be used in different EASN specifications, these data operations must be carried out using the ASN.1/C++ generic data interface that supports operations on objects of types *a priori* unknown. The ASN.1/C++ translator exports such a complete functional interface to access the values held in objects of ASN.1 types.

The EASN Validator: SPIN, and EASN too, generates a set of files `pan.[chmtb]` that are compiled together into a model checking executable (section 1.2). Some of these files, for example .h file, define structures corresponding to the various *proctypes* and *queues* that comprise the system, and the *state* structure. Components of these structures, together, form the state of the system being analysed. We shall refer to the state of the system as the state of SPIN to differentiate it from what we shall later call the state of EASN.

The state of SPIN is kept in one place in memory, but two sections of code in the generated `pan` files view it differently. The code in `pan.[mb]` corresponds to the transitions that take the system from one state to another, in the forward and backward directions respectively. This code views groups of components in a structured manner, either as some *process* or *queue* structure, or some *global-variable*. However, the code in `pan.c` that has to do with constructing, modifying, maintaining, storing state (into the hash table / or on stack) and comparing for equality views the very same state of SPIN as a block of memory without any further structure. This makes for a highly optimised implementation of the SPIN validator but, in the context of EASN, where most of the components of the state of EASN are C++ objects, this two-views-of-the-same-memory is problematic.

The state of EASN, therefore, is organised differently. We *encapsulate* every *actual* component of the state inside an object of type MSVComponent (Minimal-State-Vector-Component, a C++ template type). The state of EASN, then, is simply an array of such encapsulated objects. This representation of the state of EASN is useful for code in `pan.[mb]`. Since the state manipulating code in `pan.c` needs to view the state as a contiguous chunk of memory, we also maintain a ***consistent,*** *linearised* representation of the state of EASN. The consistency is guaranteed by the functionality of the *encapsulating* class.

In order to play its role, the encapsulating class needs to know some information regarding the type of the object that it encapsulates. This information, for every type, is made available through a function call interface in the state compaction information module that is automatically generated by a compaction information generator (figure 1). Further details are available in Section 4.3.

2.4 EASN System from an User Perspective

The user first uses an available ASN.1/C++ translator that conforms to the NMF standards to generate C++ source corresponding to all the ASN.1 modules that together form the dEASN spec. This generated C++ source, containing both the generic/specific data interfaces and the metadata interface, is then compiled to create (say) asn1 link modules. This C++ source comprises of header (include) files, their implementation, and some tool-provided run-time support. The header files are included in the EASN source which is compiled to generate its set of link modules. These link modules are then linked with the asn1 link modules to generate the EASN (executable) tool.

Generated C++ source is *compiled into* the executable that processes the cEASN spec corresponding to the dEASN spec used to generate it. The EASN system then parses the cEASN spec and uses the metadata interface to validate the types of variables instantiated, their usage in expressions, their compatibility with various operators, and such, to ultimately generate the parse tree and symbol table data structures. This completes the role of the parser. The GUI can be used to choose to either simulate the system or to generate *pan* (the protocol-analyser).

The simulator module makes calls to the generic interface and some components of the specific interface (only those that export the data access services corresponding to the basic data types of ASN.1). If the user chooses to generate the validator instead, the validator-generator takes control and generates C++ source in the `pan.[hcmbt]` files, similar to SPIN. An additional intermediate step requires that another program called the Compaction Information Generator (*cigen*, for short), using the metadata interface generated by the ASN.1/C++ translator, generates the Compaction Information that has to be linked with the `pan` files. Finally, all this generated source has to be compiled and linked with the asn1 link modules, and with the new (l)svcomp ((light-) State Vector Component) module, to generate the *pan*.

3 ASN.1 to C++ Translation

NMF (Network Management Forum 1998) has an "ASN.1/C++ Application Programming Interface"[7] set of standards that give C++ bindings to ASN.1 types. The standard is briefly discussed here. It has two parts, namely, Part 1: Base Classes and Specific Interface and Part 2: Generic Interface.

The C++ Data Model for ASN.1

Corresponding to every ASN.1 module definition, a C++ namespace is instantiated containing a C++ class definition generated for every type in that module. The public interface for the predefined C++ Classes corresponding to all the basic types of ASN.1 is defined to be in the 'ASN1' namespace. All predefined and generated C++ classes are derived from a fundamental Abstract Class called **ASN1::AbstractData**. Also, for each ASN.1 constant, a C++ object of that type is instantiated. The syntactic name-mangling rules to generate the appropriate C++ names for the corresponding ASN.1 names are simple.

In addition, for each ASN.1 type, a C++ class is also created to export (though its public interface) meta-data regarding the properties of that type. Such class definitions corresponding to the basic types already exist in the ASN1 namespace. These classes and all classes that are defined corresponding to the various user-defined ASN.1 types are public subclasses of an abstract class called **ASN1::AbstractType.**

Part 1: Base Classes and Specific Interface

This part defines the C++ classes corresponding to all the basic datatypes in ASN.1 and also the rules followed to generate C++ code for user-defined types in the ASN.1 module(s) using compositional constructs of ASN.1 and the subtyping mechanism. In the generated C++ code, public access methods for every component of the SEQUENCE, CHOICE, and SET type are included. The names of these methods are chosen by name mangling those of the corresponding components in the ASN.1 module specification. This interface, called the specific data interface, is exported by the generated classes for the corresponding SEQUENCE, SET and CHOICE type definitions given in the ASN.1 module.

Part 2: Generic Interface

This part defines the metadata interface as well as the generic data interface. For instance, a C++ interface for querying the metadata of ASN.1 types, methods to get the (names of the) enumerators of an ENUMERATED type or those to get the (names of) components of a SEQUENCE type. Querying through this interface could retrieve all the necessary information to enable parsing EASN code that defines variables of ASN.1 types and operates upon them.

The definition of the Generic Data Interface involves exporting a functionally complete interface that can be used to access the values held in objects of

arbitrary type confirming to the data model. This interface, for instance, defines C++ classes like CHOICE and StructuredData which provide generic methods to access the components of a CHOICE, SEQUENCE or SET object. These methods take the component name as the (string) parameter to provide access to the corresponding component. The EASN Simulator module uses this generic data interface, along with the metadata interface, to discover the structure of datatypes and operate on objects of these types.

4 EASN System: Implementation

We shall now move over to discussing the implementation of the EASN System. We shall begin by discussing the implementation of the SPIN System as a set of sub-systems: here we use the word sub-system to identify either a set of C-modules, or sometimes a single C-module, or even just a set of functions / functionality within a C-module. We then talk about the details of our implementation, with respect to the SPIN description below.

4.1 SPIN and Its Subsystems[8]

SPIN is implemented, mostly, in ANSI C. It, therefore, comprises of various include files and link modules. Apart from C, there is (only) the GUI module that is entirely implemented in the Tcl/Tk scripting language. One of our primary tasks, before attempting any modifications to SPIN, was to come up with an abstract view of the SPIN sources that would make it possible to break down our implementation effort into a set of well-defined subtasks, that we could then attempt systematically. Recall from section 2.2, that our implementation of EASN from SPIN was to retain all the good features of SPIN, and also be able to upgrade to future releases of SPIN, with minimal effort[2]. Below, we list the main components of the SPIN system, from the perspective of a designer wishing to modify SPIN.

GUI: This module is implemented in Tcl/Tk entirely, and requires only cosmetic changes when having to function in the context of EASN.

Parser: This module comprises of a handcrafted lexer and a YACC-generated Promela parser, along with the necessary supporting functionality in associated link modules. Like for all languages, the parser parses the input specification, checks for syntactic correctness, and creates internal data structures that capture all the relevant information from the specification to enable either simulation of the system specified, or generation of a validator for it. This was the starting point for our modifications, since we had a clear set of requirements in the form of an EASN language spec, drawn out as a set of modifications to Promela.

[2] A good indication that we have met this design goal is that while we began making our modifications to SPIN version 2.5.4, by the time we could get EASN to its current state, we had gone through 4 'upgrading's and at the time of making this release, we are in sync with its version 3.4.1.

Parse tree and symbol table: This module includes the various data structures defined in the `spin.h` file, and the supporting functionality provided in various other link modules. The definitions for some of the data structures involved were slightly modified to reflect the change in the data model of EASN with respect to that of Promela.

Simulator: This module includes all the code that operates upon the fully constructed parse tree and the completely populated symbol table. Its role is to effect a random walk on the reachable state space of the specified system, starting at the initial state. One of the key functions is the recursive `eval` function. Given a node of the parse tree, it returns an integer value on evaluating the expression tree rooted at it, since any basic type in Promela is representable in a 32-bit integer, a key design decision in the SPIN implementation. (Similarly, all objects of higher-order types are also represented, passed as parameters to processes (at their initialisation), or passed into and out of message channels, as a collection of the required number of integer objects). In EASN, our major modification to this module stems from our need to change the function call interface of `eval`.

Validator-Generator: The starting point for this subsystem, similar to the simulator described above, is the information in the parse-tree and the associated symbol-table. It comprises of all the files that begin with the `pangen` prefix. This module generates the *pan* (protocol analyser) files in ANSI C. The generated files (described below) are then compiled together to form the executable protocol-analyser that actually analyses the Promela spec.

Protocol-Analyser: This program conducts a search over the reachable state space of the system described in the Promela spec, using a *user-configured* combination of algorithmic components. The basic state space search algorithmic framework remains broadly the same; however, intermediate steps, like whether to use partial-order reduction methods, whether to use complete state storage or bit-state hashing, whether to incorporate fairness, safety properties, acceptance cycles, etc. are user provided inputs that are used to select the code fragments that compose the complete search loop. The entire program is one single C compilation-unit, with the rest of the files mentioned below, being `#include`d into the `pan.c` file.

`pan.h`: This file declares forward prototypes of certain functions that are defined in the `pan.c` files, but may be used before they are defined. More importantly, the structures corresponding to the state of the system (including those of various proctypes and channels defined in the Promela spec) are also defined. For instance, when the Promela spec has a (system-) global or a (proctype-)local variable declaration, the generated `pan.h` file has a member variable (with the same name) of the corresponding C type in the state structure, or in the proctype structure. In EASN, the structure corresponding to these various proctypes, channels, and the state of the system are defined differently. Also, the prototypes of certain functionality had to be changed.

`pan.m`: This file contains, as a single list of case options, C statements corresponding to every state transition (**m**ove) in the input Promela spec. Transitions in this file take the system *forward* in its search. All of this file is `#include`d

as the body of a C switch statement. For instance, an original Promela statement that, say, increments a certain (proctype-)local variable is translated into a `case` entry in this file that is a C expression incrementing the value of the corresponding `proctype` member variable. However, just before incrementing, the value of the variable is copied into a *stack trace*. (The next item illustrates the use of the stack-trace). In EASN, the contents of this generated file continue to remain the same. However, since we *import* the support for operating with ASN.1/C++ Objects from the C++ bindings, where any access to some component of a structured object requires us to make a function call, it looks slightly different.

`pan.b`: Similar to the `pan.m` file, this file is also a single list of case options with one `case` listed here for every `case` in the file above. Each case body expects, at the top of the stack trace, the information that it needs to *restore* the state of the system to the point earlier, *before* the corresponding `case` took it forward. This file, therefore, contains all the code that takes the system *backward* from where it is to where it was, before it chose any particular forward move from the `pan.m` file. The stack trace, then, is the information that is generated by the `case`s in the `pan.m` file, and consumed by the code in the `pan.b` file. In EASN, just like for the previous file, the corresponding `pan.b` file needs to work with C++ objects imported from the ASN.1/C++ translation and is therefore different to that extent.

`pan.t`: The code in this file contains the functionality that generates the transition matrix that encodes the behavioral semantics of the system. These transition objects refer to the case labels that identify the cases in the `pan.[mb]` files above. The various members of the transition structure are used by the policy code (below). EASN does not modify either the definition of the transition or the way transition objects are created and populated by this `pan.t` code.

State-of-SPIN: We refer to the `now` object of type `State` (whose structure is defined in the `pan.h` file), with all the process and queue objects overlaid onto it, as the State-of-SPIN. Some members of the State structure are book keeping information that is used by the policy code in `pan.c` to make decisions when conducting the search. As and when processes and queues are created in the system, space is allocated for them so as to overlay them on the unused trailing portion of `now`. Also, as and when queues become inaccessible or processes reach the end of their computation and therefore can be flushed out of the system, the space that they occupy is recovered but in a strict last-in-first-out fashion. The various functions defined in the `pan.c` file that compress (or compact) `now`, put it into the hash-table (or stack), check for it in the hash-table (or stack), compute various kinds of hash values for it, compare two representations (whether compressed or hash-compacted, or hashed, ...) of `now` for equality, etc. are termed as *mechanisms,* below. The code that makes calls to these mechanism functions and uses the information returned to make decisions, such as whether to move up the search tree or down, is referred to as the *policy* code.

`pan.c`–Policy: The main loop of the generated protocol-analyser (or validator) is the policy-code. This main loop requires a hash-store, where reached states are

stored. It also maintains a stack, that contains some representation of the various states that have been visited along the path from the initial state of the system up until the current state. This stack is used to detect acceptance cycles, or non-progress cycles. The information in the hash-store is used to avoid re-exploring previously explored states. This policy code encapsulates the core of the SPIN-algorithms, including partial-order reduction, and LTL-property conformance. To inherit all these facilities offered by SPIN into EASN, therefore, requires us to carefully avoid tampering with this policy code.

`pan.c`–**Mechanisms:** This code includes the functionality that actually stores the reached-state into the hash-store and/or the stack and checks to see if a particular reached-state has been visited in the past. Based upon various options selectable when generating the validator, and some additional options selectable when compiling and executing it, SPIN offers many choices for compaction / compression algorithms including bit-state hashing. All these functions are tightly tied to the particular linearised representation of the State-of-SPIN. In EASN, where we have a different representation for the State-of-EASN, we have to reimplement this functionality. The correctness of our implementation, therefore, hinges upon the correctness of our management of the various representations of the state of the system.

4.2 EASN and Its Subsystems

EASN has all the modules of SPIN listed above and a few more (see figure 1). Those SPIN modules that are modified in EASN are described only to the extent to which they differ. Sub-systems of SPIN that are not modified in EASN are not included below, for example, the GUI part, and the `pan.t` file.

Except for our choice to change the representation of the state of the generated validator, the major change in EASN from SPIN is the type system. Hence, in the implementation of EASN, this should translate into modifications to SPIN wherever it deals with variables (and expressions involving variables). Therefore, entire pieces of code that have to do with the handling the semantics of the control constructs of Promela need not be modified at all.

Parser: The lexer has been modified to modify the set of keywords in the language, to recognise the ASN.1 value-notation for constant values and to identify ASN.1 type-references and value-references. On encountering an ASN.1 type-name, the parser queries the ASN.1 metadata interface for the complete set of attributes associated with that type and updates its symbol table accordingly. Notice, therefore, that only those types from the ASN.1 module that are actually referred in the cEASN spec, are actually imported into the symbol table.

Parse tree and symbol table: The definitions of the structures of some of the types that build the symbol table and the expression tree in the parse tree have been modified. For instance, the optionality and the default value attributes of the component of structured types need to be represented in the `Symbol` structure. Similarly, in order to enable the compatibility check for operators and operands, the `Lextok` structure has been enhanced.

Simulator: One of our modifications to this subsystem is to the `eval` function that evaluates an expression in the given state of the system. In EASN, this function returns a dynamically allocated object of type `AbstractData`. This makes it convenient to implement the operand type upgrade associated with some of the overloaded operators. Another modification has to do with passing parameters to process invocations, or passing messages through channels using `send` and `recv` operators. In EASN, since the C++ types that are referred through their ASN.1 names are already compiled into the (simulator) application, entire C++ objects can be handled without, as in SPIN, having to compute the number of integer objects required to represent types, whose objects need to be passed as parameters (while invoking process instances), or through channels.

Validator-Generator: EASN also generates all the files that SPIN does, and at the level of abstraction in the previous section, the contents of the EASN generated pan files are similar too. We discuss the details below.

Protocol-Analyser: The SPIN generated pan files are complete in so far as they do not need to link with anything more than the standard C Library. EASN generated files, however, need to link with run time support, ASN.1 support, state vector component support, GNU multi-precision arithmetic, and the compaction information. All of these are linked together to produce *pan*.

State-of-EASN: The *now* object in SPIN generated validators encapsulates the State-of-SPIN. As presented in the section on SPIN, the code in the `pan.[mb]` files deals with the structured objects that overlay onto `now` (and the other members of the `State` structure that correspond to global variables in the Promela spec). Whereas the *policy* code from `pan.c` uses the book-keeping information to store the state of the search, the *mechanisms* code views *now* merely as a sequence of contiguous bytes. This 'multiple-views-of-a-single-chunk-of-memory' tends to become unmanageable in the context of EASN, where the objects that comprise the state of the system are C++ objects that can have virtual-table pointers, RTTI, and other inaccessible private components. The State-of-EASN, therefore has two representations: one which lends itself to be used by structured users such as the code in the `pan.[mb]` files and the *policy* code, and the other, the linearised version, which is intended for use by the *mechanisms* code. By ensuring that these two representations are mutually consistent but only at points where the control changes from the structured-view code to the linearised-view code, we benefit by implementing *incrementally* both the consistency updates as well as the computation of the hash-values for the linearised versions. Consequently, the *mechanisms* code in the EASN generated analysers is simpler than for SPIN.

svcomp-Module: In EASN, the consistency between the two representations discussed above is achieved by *encapsulating* every component of the State-of-EASN into (a publicly derived subclass of) the MSVComponent (Minimal-State-Vector-Component) template class whose functionality *incrementally ensures* consistency. We call this facility the *compaction infrastructure*. In order for the MSVComponent class to execute its responsibility, it requires to know, for every type that it encapsulates: the cardinality c of the value-set represented by

the type, and a mapping from this value-set onto integers in the range: 0..*c*. This information is required to be available through a function-call interface. If the value of *c* for every type that is imported from ASN.1 is representable using 32-bit integers, then a more efficient implementation of this module called *lsvcomp* (light-svcomp) can be used instead. The call interface is: `MP_INT * EASN_GetCardinality (Type *)`, that returns the cardinality, and `MP_INT * EASN_GetIndex (Type * object)`, that returns the index of the value in the value-set. This module uses the **G**NU **M**ulti-**P**recision (GMP) package. The `MP_INT` type in the prototypes above is exported by this GMP library. In case of the *lsvcomp* module, the return types for the above functions is `mp_limb_t`.

cigen: This sub-system generates the implementation for the two functions described above for every type in the ASN.1 module. In our current implementation, it has been implemented as a stand-alone tool that uses the metadata interface to generate these functions for all the types in the ASN.1 module definition, but it could also be implemented as an additional link module of the EASN system, thereby generating the *compaction information* only for those types from the ASN.1 module that are imported into the cEASN Spec.

Compaction-information: This is the set of functions that are generated by the cigen utility above needs to link into *pan*.

panrts-module: This module is required because of the automatic type upgrading semantics of some of the operands of EASN. Also, there is a gap between the set of operations that are supported by the C++ classes corresponding to the basic datatypes of ASN.1 and the operators that we support on them in our EASN Language definition. This module bridges the gap.

`pan.h`: The structures defined in this file are similar to those of SPIN, except that members of the structures that actually correspond to variables in the cEASN spec are encapsulated into the MSVComponent class (or one of its subclasses). For instance, if the finite state automaton corresponding to a proctype has 19 local states, in the SPIN generated `pan.h` file, the corresponding structure has a member, which identifies the state a particular process, of type `char` to store this information. EASN generates, for the same purpose, a member with type `iSVComponent<20>`. Similarly, an cEASN variable of type `asn::Integer` causes the generated `pan.h` file to have the type of the corresponding member variable to be `SVComponent<asn::Integer>`.

`pan.c`-mechanisms: The entire set of these mechanisms that work with the linearised view of the state of the system have been re-implemented in EASN. For instance, the default compression algorithm used in SPIN `pan` is unnecessary for linearised versions of the State-of-EASN. As another example, consider the implementation of the hash compact version of the compression routine in SPIN `pan`. It computes the compressed version of every process and queue state in the system separately from the global state of the system and individually hashes these to generate much smaller id's which are then used to compose a new (far shorter) representation of the reached-state, which is stored into the hash table. This two-level hashing is implemented incrementally by recording the necessary book-keeping information in the *compaction infrastructure*, when installing and

uninstalling components of the State-of-EASN. Contiguous subsets of these system components are demarcated as belonging to either the global state, or of a particular process or queue and separate linearisations for these sets of components are also maintained, incrementally and consistently. These representations are then used to directly compose the next level representation of the reached state that is put into the hash-table.

4.3 Ensuring Consistency - Incrementally

The *compaction infrastructure*, equipped with the necessary *compaction information*, views the state space of the system as a multi-dimensional array (with one dimension for every component of the system), and consequently, every state as a point in this multi-dimensional space. Another representation used is a column-major linearisation of this multi-dimensional array.

Since, as also is the case with the State-of-SPIN, the number of components that compose the system-state can increase (if new processes or queues are added) and decrease (if the last process reaches the end of its computation), the compaction infrastructure has to also recompute its mapping from the multi-dimensional array to a linearised representation. Only the column-major linearisation is useful as the row-major linearisation cannot handle varying numbers of processes. The compaction algorithm, therefore, associates a weight along with every component of the state-of-EASN at the point of installing it and uses this weight to appropriately increase or decrease the impact of the change in the value of this component on the linearised representation. We assume here that the system comprises of components numbered from 0 through n, and we use the prime notation to denote the *new* value of any entity.

c_j**:** *cardinality* of value-set associated with the type of system component j.
i_j**:** *index* of current value of system component j into its associated value-set.
$w_0 = 1$**:** The *weight* of the first component of the system is 1.
$w_j = \prod_{k=0}^{j-1} c_k$**:** *weight* of system component j is the product of the *cardinalities* of the previous $j-1$ system components.
$L = \sum_{j=0}^{n} w_j i_j$**:** The column-major linearisation is the sum of the products of the *weight* of the component and its *index*, over *all* the system components.
$\delta L = w_j(i_j^{'} - i_j)$**:** The impact of a change in the value of system component j is the product of its *weight* and the difference between its new & old *index*.

Whenever the system moves from one state to another, only those few state components that are responsible for the change determine the update on the linearised representation. The compaction infrastructure updates the representation incrementally which is then stored into the hash-store when an exhaustive search is conducted.

4.4 Incremental Hashing

There are functions in SPIN-generated `pan.c`, called `d_hash`, `r_hash` and `s_hash`, that generate (1 or 2) 32-bit hash values by using the complete octet-string (representing a part or the full state) that has been passed to them. SPIN generated

validators use these functions to compute hash values corresponding to reached states represented by *now*. The user can specify any of a set of 32 hash constants to be used by these algorithms. Typically, only a few of the components of the system are responsible for its change in state at any given point in its evolution. Hence, an incremental computation of the hash value can improve performance which is done through our *compaction infrastructure*.

SPIN uses polynomial arithmetic to compute a 32-bit intermediate quantity (that is further used to generate the hash-values), by performing a fast division operation on the *now* state vector using the chosen (32-bit) hash constant as the divisor. In EASN, we use integer division on the linearised representation of the reached state, again using the 32-bit hash constant. The 32-bit remainder, thus generated, is used to generate the hash values, just as they are in SPIN.

We introduce another attribute maintained for every system component: $r_j = w_j \bmod H$: *remainder* after dividing its *weight* by the hash constant, H.

Below, we discuss the incremental computation to generate this hash-value R' for a new state that resulted with a change in the system component j, given that the hash-value for the old system state was R.

$$
\begin{aligned}
& R' \\
&= L' \bmod H \\
&= (L + \delta L) \bmod H \\
&= ((L \bmod H) + (\delta L \bmod H)) \bmod H \\
&= (R + w_j(i_j' - i_j) \bmod H)) \bmod H \\
&= (R + r_j \delta i_j) \bmod H
\end{aligned}
$$

Notice that in the context of the *lsvcomp* implementation, all the four quantities on the RHS of the equation above, namely, $R, r_j, \delta i_j$, and H, are all 32-bit operands. This allows for an efficient implementation of this incremental hash-value computation.

More interestingly, under certain combination of SPIN options (-DBITSTATE & -DSAFETY) while compiling the validator, it turns out that the linearised version of the state is neither stored on the stack nor in the hash-table, which reduces the burden of generating, computing and maintaining it, since we can generate the hash values corresponding to the reached state incrementally, directly available from the *compaction infrastructure*.

However, this incremental hash-value computation scheme requires the value H to be identified at compile time itself, since it is used by the constructors in the *compaction infrastructure*, which could be called before main. This is in contrast with SPIN generated validators, which can be compiled once and then executed many times for various values of H.

4.5 Correctness of Implementation *vis-a-vis* SPIN

In deriving an EASN implementation from SPIN sources (given the above indicated modifications), we identified the following invariant that could be a necessary and sufficient condition to convince oneself that neither the simulator engine of EASN, nor the state-space exploration engine of the generated validator gives different results than SPIN:

Given a Promela spec. s and a cEASN spec. e, derived from s by changing all its variable types to equivalent ASN.1 types (defined in an associated ASN.1 module, appropriately imported into EASN): A. Simulation runs of SPIN over s and of EASN over e should show identical selection sequence of state-transitions, for the same seed value; B. The sequence in which the reachable states of the system are visited by the generated validators (by SPIN for s and by EASN for e) must be identical (for exhaustive state-space searches), with/without partial-order reduction, never-claims, and irrespective of other switches like safety, fairness, state-compaction mechanisms, etc.

EASN preserves this invariant for all the test cases we have tried (as per Test/README.tests). This gives us reason to believe that in the process of crafting an EASN system from SPIN, the most critical components of it are reasonably sane and stable. To that extent we do well in inheriting the time-tested aspect of SPIN.

```
1  /* mtype = { msg0, msg1, ack0, ack1 }; */
2
3  chan sender   = [1] of { asn::MtypeAbp };
4  chan receiver = [1] of { asn::MtypeAbp };
5
6  inline recv(cur_msg, cur_ack, lst_msg, lst_ack) {
7      do
8      :: receiver?cur_msg -> sender!cur_ack; break
9      :: receiver?lst_msg -> sender!lst_ack
10     od;
11 }
12
13 inline phase(msg, good_ack, bad_ack) {
14     do
15     :: sender?good_ack -> break
16     :: sender?bad_ack
17     :: timeout ->
18         if
19         :: receiver!msg;
20         :: skip /* lose message */
21         fi;
22     od
23 }
24
25 active proctype Sender() {
26     do
27     :: phase(msg1, ack1, ack0);
28        phase(msg0, ack0, ack1)
29     od
30 }
31 active proctype Receiver() {
32     do
33     :: recv(msg1, ack1, msg0, ack0);
34        recv(msg0, ack0, msg1, ack1)
35     od                                   (1)
36 }
```

```
1  Easn DEFINITIONS ::=
2  BEGIN
3
4  ...
5  MtypeAbp ::= ENUMERATED {
6      msg0, msg1, ack0, ack1
7  }
8                                        (2)
9  END
```

```
...
State-vector 24 byte, depth reached 9, errors: 0
     12 states, stored
      3 states, matched
     15 transitions (= stored+matched)
      0 atomic steps
hash conflicts: 0 (resolved)
(max size 2^18 states)                   (3)
...
```

```
...
State-vector 12 byte, depth reached 9, errors: 0
     12 states, stored
      3 states, matched
     15 transitions (= stored+matched)
      0 atomic steps
hash conflicts: 0 (resolved)
(max size 2^18 states)                   (4)
...
```

1: The cEASN Spec. 2: The dEASN Spec. The original Promela Spec. can be recovered from
3: The SPIN-pan output. 4: The EASN-pan output. the cEASN Spec, by uncommenting line # 1.

Fig. 2. ABP in SPIN and EASN

5 Results and Conclusions

We have used EASN to validate a simplified RLC protocol in the W-CDMA stack. It has a smaller state than SPIN due to the use of the subtyping information by the state compaction infrastructure. Further details of the performance of EASN will be submitted to the FMICS workshop. Due to its length, we present a much simpler ABP protocol in figure 2. Note that the state vector for EASN is half the size of SPIN's.

5.1 Performance of EASN vs SPIN

We present some preliminary performance comparisons of EASN-generated validators for specs derived from some Promela specs (snoopy and leader election) in the SPIN/Test database, with those generated by SPIN (table 1). We compare the runs of the validators generated by both SPIN and EASN, under compilations with three sets of options, SAFETY, SAFETY and NOREDUCE, SAFETY, NOREDUCE and BITSTATE. Time is in seconds, memory in megabytes, the State(-vector) size in bytes and the last column lists the size of pan in Kilobytes.

Table 1. EASN vs SPIN: time and memory performance

M	Options	Validator	Time	Mem.	States	Transitions	State	Depth	Size
S	SAFETY	SPIN	0.33	3.065	13380	18550	112	2380	89
N		EASN	0.85	2.758	13380	18550	64	2380	4046
O	(plus)	SPIN	2.85	9.334	81013	273782	112	11080	79
O	NOREDUCE	EASN	8.35	7.388	81013	273782	64	11080	4040
P	(plus)	SPIN	1.74	1.129	80264	271302	112	10580	77
Y	BITSTATE	EASN	4.87	1.129	80983	273679	0	11114	4038
L	SAFETY	SPIN	0.04	1.493	97	97	196	108	54
E		EASN	0.06	1.493	97	97	112	108	3998
A	(plus)	SPIN	0.93	4.014	15779	58181	196	108	47
D	NOREDUCE	EASN	2.09	3.279	15779	58181	112	108	3995
E	(plus)	SPIN	0.40	0.929	15779	58181	196	108	46
R	BITSTATE	EASN	1.17	0.929	15778	58177	0	108	3993

Since we have made only the minimal modifications to a Promela spec to derive the corresponding cEASN spec, we expect to see (except for the bitstate runs) the number of states detected and the number of transitions explored must be the same for both the SPIN generated validator and the EASN-generated ones. Note that this is indeed true.

Bytes required to store the linearised State representation are lesser for EASN, than for SPIN, also as expected, thereby reducing run-time memory usage, again except for the bitstate runs, where both the validators use the same number of bits to store a single state. (The reduction in memory is much lower than expected, since, we believe we still have some unplugged memory leaks in our generated validators). The Table reads 0 for EASN bitstate runs indicating incremental hash-value computation, as described in section 4.4.

Also as expected, our run-time is higher than for SPIN-generated validators. However, the increase is many-fold, being much higher than what we anticipated. Profiling has revealed the following major contributing factors.

C++: In SPIN, an integer variable in the Promela spec translates to an integer member of a C structure, and access to it is not protected, unlike in EASN, where that compares to an `asn::Integer` object, that (typically) encapsulates the actual integer (that holds the value of interest), as a private/protected member. Therefore, any Promela expression that involves use of this integer object, translates into a C++ member function call (that may or may not be inlined).

C++ Constructors and Destructors: We observe a substantial portion of run-time spent initialising and destroying temporary C++ objects on the stack. Since we wanted to prevent having to incur time penalties related to allocating and deallocating objects on the heap, our generated code uses compiler temporaries, but even then one cannot avoid the constructor/destructor cost that is implied by the ASN.1/C++ run-time support system.
Integer Arithmetic, instead of Polynomial Arithmetic: Time is also consumed in routines that have to do multi-precision arithmetic operations, like addition, multiplication and modulus. SPIN scores a big plus on this aspect since it uses cheaper polynomial division operations in its hash functions. Although our choice of the integer arithmetic enables us to implement the hashing algorithm incrementally, it would be much faster if we could find another lower cost mapping from our full-blown C++ version of the State-of-EASN to some linearised representation that also enables us to compute the latter from the former incrementally, like we can now.
Huge Executables: The generated validators usually make very few system calls, but due to the very large (compared to the size of the SPIN-generated validators) size of our validators, we see much higher system activity in our runs, as compared to that in the case of SPIN validators.

We have also attempted to see how EASN compares with SPIN with a change in the problem size (Table 2).Note that as the size of the system being validated is increased, the memory benefit of EASN's more compact state-vector begins to show, and also, as the SPIN state-vector size grows larger, the run-time cost of handling the same begins to reduce the gap between the two validators' performance. The sort program from the Test database has N instances of the `middle` proctype in the system, each reading from and writing into its left and right channels respectively. The system also has N channels. Increasing N, therefore, increases the size of the system being analysed.

Table 2. EASN vs SPIN: Scaling

Model	Options	Comparison Factor	N=5	N=6	N=7	N=8	N=9[3]
Sort	SAFETY	Mem (EASN / SPIN)	1	0.88	0.825	0.821	0.937
	NOREDUCE	Time (EASN / SPIN)	3.05	3.18	2.86	2.81	2.42
	(plus) BITSTATE	Time (EASN / SPIN)	3.36	3.57	3.35	2.95	2.55

5.2 Future Work

There are mainly two lines of activity, we believe, that can be pursued to get better results than are possible by further cleaning up our implementation of any wasteful handling of either memory and/or time, when executing the validator.

[3] In this case neither SPIN-pan nor EASN-pan completed the exhaustive search for want of memory (we used a 500MHz, 256MB Linux for all the tests in this paper). However, in the memory that they used, the EASN-pan stored 15.6% more states, and explored 18% more transitions than the SPIN-pan.

- Since only the simulator requires the generic interface capability, generating C-source for the validator would better its performance. Since the ASN.1/C binding is lighter, we would still get the benefit of having type information available from ASN.1, while reducing of our run-time cost. The subtype information would continue to be available. The EASN language definition would still be the starting point for the tool which continues to work with the ASN.1/C++ bindings, but uses the ASN.1/C binding for the validator.
- Study ways of reducing the cost of multi-precision arithmetic.

References

1. Holzmann, Gerald J., Doron Peled, "The state of SPIN", CAV '96.
2. Rob Gerth, Eindhoven University, "Concise Promela Reference", August 1997, Soft-copy available with SPIN.
3. G. Gerth, D. Peled, M. Y. Vardi, P. Wolper, "Simple On-the-fly Automatic Verification of Linear Temporal Logic", PSTV94.
4. Holzmann, G.J., Design and Validation of Computer Protocols, Prentice Hall, 1992.
5. Patrice Godefroid, "Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State-Explosion Problem", PhD Thesis, University of Liege, Computer Science Department, Nov. '94.
6. Information Technology - Abstract Syntax Notation One (ASN.1): Specification of Basic Notation, (Technical Corr. 1, Amd. 1:Rules of extensibility), ITU-T Rec. X.680 (1994), Corr.1 (1995), Amd. 1 (1995); Information Object Specification (Amd. 1: Rules of Extensibility), ITU-T Rec. X.681 (1994), Amd. 1 (1995); Constraint Specification, ITU-T Rec. X.682 (1994); Parameterization of ASN.1 specifications, ITU-T Rec. X.683 (1994)
7. ASN.1/C++ Application Programming Interface, Part 1: Base Classes & Specific Interface, & Part 2: Generic Interface, NMF 040- 1 & 2, Issue 1.0, Feb. 1998
8. Holzmann, G.J., SPIN Sources, Version 3.4.1, 15th August 2000; "Basic Spin Manual", available with SPIN.
9. J.Geldenhuys, PJA de Villiers, 'Runtime Efficient State Compaction in SPIN,' The 5th Intl SPIN Workshop on Theoretical Aspects of Model Checking.
10. Anindya Basu, 'A Language-based Approach to Protocol Construction', PhD Dissertation, Cornell Univ., Aug. '97
11. ASN.1/C++ Application Programming Interface, Issue 1.0 Draft 10a - Submission to X/Open August 21, 1996
12. Appendix A: The ASN.1 language, and Appendix B:The EASN Language, are only in the full paper; available at http://144.16.67.13/~ gopi/spin01/easn.ps.gz.

Modeling and Verifying a Price Model for Congestion Control in Computer Networks Using Promela/Spin

Clement Yuen and Wei Tjioe

Department of Computer Science
University of Toronto
10 King's College Road,
Toronto, Ontario M5H 3G4, Canada
{clhyuen,wtjioe}@cs.toronto.edu
http://www.cs.toronto.edu/~wtjioe/pricing/index.html

Abstract. Congestion control is an important research area in computer networks. Using PROMELA/SPIN, we verified that priority pricing schemes can be used to effectively control congestion. Under the simulation/verification framework provided by SPIN, we verified the propositions that the enforcement of priority pricing on a network link (i) results in an equilibrium state in packet allocation, and (ii) effectively controls congestion level when pricing is being dynamically adjusted. Furthermore, we have extended these propositions to demonstrate the convergence property of equilibrium in packet allocation. This particular result would be difficult to verify with existing network analysis tools.

1 Introduction

PROMELA/SPIN is a versatile tool in simulation and verification of software systems [5]. A common application of PROMELA/SPIN is in modeling and verifying communication protocols [6]. In the area of mobile communication, part of the MCNet architecture has been simulated and verified in SPIN [7]. With Java PathFinder [8], it is now possible to translate programs written in JAVA version 1.0 to PROMELA. In addition, PROMELA/SPIN has been shown to be useful in modeling business rules [9]. Inspired by these novel applications, we examine the applicability of SPIN for a nontrivial pricing model in relation to congestion control in computer networks.

As demand for Internet services increases, so does the importance of congestion control in computer networks. Conventionally, congestion control can be accomplished by (i) sending control packets from an overloaded node to some or all of its sources, (ii) providing delay information to influence network routing decisions, or (iii) making use of end-to-end probe packets and timestamps to measure the delay between two end nodes. One drawback of these approaches is that more control packets must be added to the network before congestion can be relieved.

Putting network users in a more active decision-making position is a novel idea emerged from recent congestion control researches [2,3]. Priority pricing scheme is one realization of this idea. The quality of packet transmissions is associated with the price a user is willing to pay. The establishment of prioritized service classes coupled

M.B. Dwyer (Ed.): SPIN 2001, LNCS 2057, pp. 272-287, 2001.

with pricing enables network providers to adjust network volume through pricing. However, ideas based on priority pricing remain difficult to verify due to the complexity of their scopes. We discuss two such ideas below.

Gupta [4] presented a robust model using priority pricing in the dynamic management of network bandwidth. In this model, a network node may return an estimated price and waiting time to a user in response to the quality of service requested. After collecting the current pricing and predicted waiting times from all servers, the user calculates the total expected cost in using the network. If the estimated cost is higher than the benefit of using the network, the user may decide to delay transmission until a more agreeable price becomes available. Otherwise, the user may begin releasing packets into the network. Note that the cost and waiting time can increase when traffic volume becomes high. In response to increased cost, the user will reduce the volume of packets released into the network. However, network delays are common and random enough that updated pricing information may not reach the user in time to become useful. Therefore, this pricing scheme can be difficult to implement and verify.

Marbach [1] described a simpler model that may become more effective in practice. In this model, each priority transmission level is associated with a "fixed" price (on a fast time-scale). Among this hierarchy of pricing choices, two consecutive price levels are of great significance. In this paper, we refer to the higher price as the "premium price", and the lower one the "best effort price". When packets are released into the network with premium priority, they are guaranteed to transmit successfully. Packets released with best effort priority are always associated with a certain non-zero probability of packet drop. All submissions at a price higher than the premium are guaranteed to transmit, while submissions at a price below the best effort priority are bound to be dropped. Users are expected to dynamically adjust the quantity of submission (on a fast time-scale) to maximize their economic benefits. Marbach's mathematical model showed that an equilibrium in packet allocation from network users can be established under priority pricing scheme. This equilibrium can potentially be manipulated to control congestion.

While there are many excellent tools available for network simulation [10-13], they do not provide any verification capability. When a model is introduced, its correctness must be analytically proven before it can be used as the basis for further discussion. Simulation is an alternative to a formal proof when analytical verification becomes too difficult. However, simulation cannot provide as strong an assertion on any important results. In this regard, PROMELA/SPIN is a valuable alternative in asserting properties that may otherwise be difficult to verify.

In this work, we examine the applicability of PROMELA/SPIN as a simulation/verification tool for nontrivial congestion control modeling, based on Marbach's priority pricing model [1]. Our objective is to explore whether SPIN can be used to provide effective assertion to important properties of the underlying model. In particular, we apply SPIN to simulate the correctness properties of Marbach's priority pricing model and provide verification of the convergence property in LTL. This result has not been analytically proven and would be very hard to verify with existing tools for network simulation. We extend this model to show that allocation equilibrium can be maintained and manipulated under the dynamic pricing scheme suggested by Gupta [4]. We have chosen SPIN for this work due to its inherent power and flexibility in modeling complex software systems. We hope this work will expand the scope of SPIN to areas relating to computer networks, economics or mathematical modeling.

The remainder of this paper is organized as follows. Section 2 provides an overview of Marbach's priority pricing model. Section 3 presents the design and architecture of two proposed models for verification. Section 4 discusses details of simulation and verification in PROMELA/SPIN. Section 5 discusses results and limitations. We conclude this paper in Section 6.

2 Overall Design of the Pricing Model

This section introduces Marbach's priority pricing model with highlights on the pertinent details and terminologies that are fundamental to discussions throughout the paper. Propositions of Marbach's model are then presented. We designed two models in PROMELA/SPIN to verify his propositions. Model 1 verifies the proposition that an equilibrium in packet allocation exists. Furthermore, it verifies the convergence of this equilibrium. Model 2 is a natural extension to Model 1. It consists of an additional process for manipulating transmission prices. In effect, this process allows us to verify whether congestion control can be achieved through dynamic priority pricing.

2.1 The Mathematical Priority Pricing Model

A computer network that implements priority pricing is modeled in [1] based on the following framework:

- A discrete-time, single-link bufferless *channel* with a fixed capacity **C**.
- There are *R users* sharing the channel, $\mathbf{R} = \{1, \ldots, R\}$.
- Network services are provided in *N* different *priority classes*, $\mathbf{N} = \{1, \ldots, N\}$. Class *i* is at a higher priority than class *j* if $i > j$, $i, j \in \mathbf{N}$. Packet traffic from a higher priority class receives preferential treatment over that from a lower priority class.
- Associated with each user $r \in \mathbf{R}$ are:
  - allocation quantities denoted by $d_r(i)$ which are the amount of packets allocated by the user at priority $i \in \mathbf{N}$ in each time slot; and
  - a private *utility function* U_r known to the user. A utility function is a well-known concept from economics that in this context reflects a user's economic benefit as a result of using the network. It is assumed to be a strictly increasing function of the perceived throughput, denoted by x_r.
- Associated with each priority class $i \in \mathbf{N}$ are:
  - a *transmission probability* $P_{tr}(i)$ defined as

$$P_{tr}(i) = \begin{cases} 1 & \text{if } \mathbf{C} > \sum_{k=i}^{N} d(k) \\ \dfrac{\mathbf{C} - \sum_{k=i+1}^{N} d(k)}{d(i)} & \text{if } \sum_{k=i}^{N} d(k) \geq \mathbf{C} \geq \sum_{k=i+1}^{N} d(k), \\ 0 & \text{otherwise} \end{cases} \quad (1)$$

where $d(i) = \sum_r d_r(i)$ is the aggregated allocation of packets over all users at priority *i*. This is the probability that packets allocated (submitted) at priority *i* are successfully transmitted; and

– a *price* u_i . In a priority pricing scheme, $0 < u_i < u_j$ whenever $i < j$, $i, j \in \mathbf{N}$. Submitting packets at a higher priority is more expensive.

It can be seen from the definition of transmission probability (1) that a *best effort* class i_0 exists such that (a) $0 < P_{tr}(i_0) \le 1$, (b) $P_{tr}(i) = 1$ for all $i > i_0$ and (c) $P_{tr}(i) = 0$ for all $i < i_0$. In other words, packets in best effort class are transmitted with a non-zero probability; packets from higher priority classes are guaranteed to be transmitted; packets from lower priority classes are all dropped. In this paper the class i_0+1 is referred to as the *premium* class. u_{i_0} is referred to as the *best effort price*, and u_{i_0+1} the *premium price*.

In a discrete-time formulation, the following interactions take place in every time slot. The network channel provides priority services according to the probability distribution in (1). Network user determines an optimal allocation of packets (i.e. $d_r(i)$) for the next time slot according to the same transmission probabilities. The optimality arises from the fact that the quantity of packets allocated to the network should maximize the net economic gain of the user. In mathematical terms, the determined allocation of user r is the solution of the maximization problem $\max_{d_r(i)\ge 0} U_r(x_r) - \sum_i d_r(i)\cdot u_i$, representing the tradeoff between a high utility and the associated cost. If we let $G_r(x_r)$ (the marginal utility function) be the derivative of $U_r(x_r)$, and G_r^{-1} its inverse function, it can be shown analytically that when best effort cost is lower than premium cost, allocation is optimal with $d_r(i_0) = \frac{1}{P_{tr}(i_0)} G_r^{-1}(u_{i_0} / P_{tr}(i_0)), d_r(i) = 0$ for all $i \ne i_0$. This means all packets should be submitted to the best effort class to perceive the best benefit. When best effort cost is higher than premium cost, allocation is optimal with $d_r(i_0+1) = G_r^{-1}(u_{i_0+1}), d_r(i) = 0$ for all $i \ne i_0$+1. This means all packets should be submitted to the premium class to perceive the best benefit. Finally, when best effort cost equals premium cost, user receives identical benefit from submitting packets to either class. Hence both allocation strategies are optimal. A network user generally makes use of the above strategy to determine their allocation of packets for the next time slot.

Under this model, Marbach [1] provided an analytically proof for the following two propositions.

Propositions:

I. Under a fixed pricing scheme, an *equilibrium* condition exists in packet allocations of channel users.
II. Priority pricing is effective in congestion control.

At equilibrium, each user's allocation maximizes its own net benefit. Therefore, no user has an incentive to change its allocation. *Equilibrium* is thus defined to be the

state where no user allocation will deviate from its current value. In the model, this condition can be inferred when all users' allocations remain unchanged over any two consecutive time slots.

Note that congestion level of the network channel can be reflected in the total (aggregate) number of packets allocated by all users. With a fixed transmission capacity, the channel is considered congested when packet submission exceeds its capacity. This in reality will result in packet drops and poor quality of service. Congestion control is observed through proposition II.

In order to verify proposition I and II, we formulate the following models in PROMELA/SPIN and show their relationships with Marbach's analytical model [1] in terms of the main components and the objectives of their propositions.

2.2 Model 1 (Demand Equilibrium Model)

Model 1 is designed to capture the major ingredients of the mathematical model. It is readily translated into PROMELA and verified in SPIN. It also serves as the basis for Model 2 where congestion control is verified. Model 1 is made up of the following main constituents.

- A channel process with a fixed capacity
- R user processes running in parallel with the channel process (sharing the channel)

A user process probes the channel for the transmission probabilities on a frequent but fair basis. It derives an optimal allocation of packets for the next time slot using the strategy described in Section 2.1. A channel process computes the transmission probabilities of the network according to (1). These values are made available to all users of the network.

The objective of Model 1 is to examine the validity of proposition I through the simulation and verification framework provided by PROMELA/SPIN. In addition proposition I is enhanced to a stronger postulation, which we now restate as proposition 1.

> ***Proposition 1***
>
> Under a fixed pricing scheme, an equilibrium condition in packet allocations of channel users exists and *converges* from some initial allocations.

Since PROMELA/SPIN only work with discrete values, an equilibrium condition in Model 1 is considered attained when the differences between consecutive allocations for all users are bounded by a small integer. In this model we choose the bound to be 1.

It should be noted that in general convergence properties are difficult to prove analytically. In particular an analytical proof of the convergence property in proposition 1 has not been given in [1]. However with an automated verification tool such as SPIN, we have the capability to mechanically verify the correctness of such proposition.

2.3 Model 2 (Dynamic Priority Pricing Model)

In order to explore the effect of dynamic price adjustments and verify proposition II, we extend Model 1 to Model 2 by adding an administrator process to the system.

The channel and user processes operate and interact in the same manner as in Model 1. The administrator process monitors the aggregated packet allocation at regular intervals and adjusts the best effort price. In response to a price change, user processes adopt the same allocation strategy as described in Section 2.1 based on the new price given by the administrator.

The objective of Model 2 is to examine the validity of proposition II using PROMELA/SPIN. Proposition II is restated as proposition 2 to more specifically reveal the mechanism in which priority pricing controls congestion.

> ***Proposition 2***
>
> Priority pricing can effectively change the aggregated equilibrium level of packet allocation.

3 Design and Architecture

This section describes the requirements, overall architecture, implementation decisions and the various core components of the two aforementioned models.

The following requirements apply to both Models:

- Users are expected to closely monitor the level of congestion as reflected by the transmission probabilities of the network, and react promptly by computing new quantities of packets to submit for the next time slot.
- The channel process is expected to return up-to-date transmission statistics to its users in an equally timely fashion. As a result, both user and channel processes should proceed on a fast time-scale.
- Pricing for each priority class should be maintained at a steady level to allow sufficient time for the system to reach equilibrium (if exists). Since this falls under the administrator's responsibility, the administrator process should proceed on a slow time-scale.

The main implementation choices applicable to both Models are:

- The time-scale would be of discrete-value.
- The channel process should be invoked following every change in user allocation. This models timely transmission probabilities updates.
- A user process should proceed immediately after each update of the network transmission probabilities. This models prompt reaction of user to congestion and changes in price levels.
- In order to guarantee that users submit packets at a constant and deterministic rate, a particular user is required to wait (block) until all other users have updated their

allocations for the next time slot. This mechanism is necessary to enforce strong fairness.

An additional implementation choice for Model 2 is:

- The administrator process should be invoked on occasion to dynamically adjust the best effort transmission price if the network channel is currently in the state of equilibrium.

The overall architecture of Model 1 comprises of a channel process and user processes, which form the environment and interact with the channel dynamically. The successful operation of Model 1 requires user processes to submit packets to the channel process, and the channel process to return the current transmission probabilities of the network to all users. Figure 1 depicts the overall architecture of Model 1.

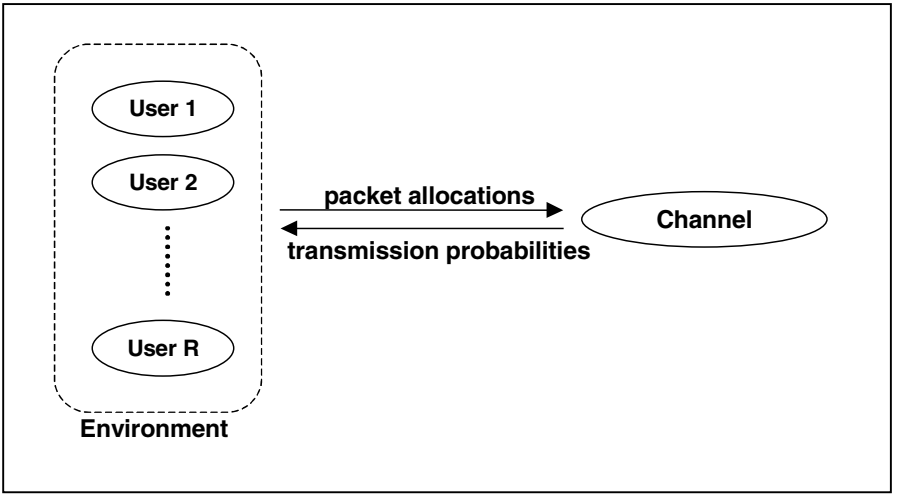


Fig. 1. Overview of Model 1 Architecture

Model 2 consists of a channel, an administrator, and user processes. In this architecture, the users communicate dynamically with both the network channel and the administrator. Given the quantities of packets submitted by users, the channel process returns the current transmission probabilities to users. The administrator process communicates new pricing information to users at regular time intervals. Figure 2 describes the overall architecture of Model 2 containing the user, administrator and channel processes.

To effectively model the priority pricing scheme, we require the corresponding implementation of data structures:

- A parameter N representing the number of priority classes.
- Parameters associated with each user process:
  - a data structure of allocation quantities, representing packet allocations to each class; and
  - a utility function defined for the user. This should be a simple function with the desired (strictly increasing) property.
- A data structure representing rational transmission probabilities.
- A data structure representing prices associated with priority classes.

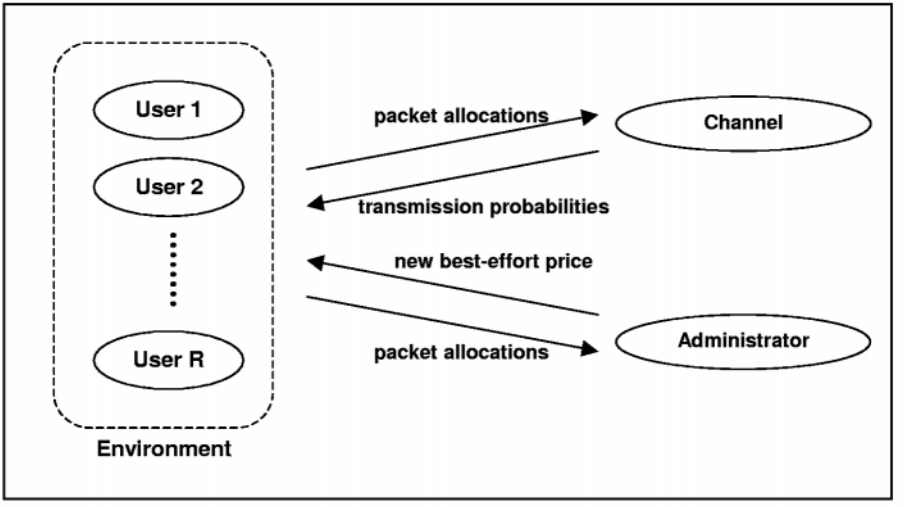


Fig. 2. Overview of Model 2 Architecture

The condition whereby equilibrium is detected is also of great importance. It was not clear whether the limitation of PROMELA/SPIN as a discrete-value modeling tool would adversely impact the outcome of simulation. In particular, it is unlikely to observe convergence of packet allocation to a single value in every case. On the other hand, packet allocation may also diverge. Through experimentation, we found in general that the model did converge to a very confined range, although the actual behavior depended on scalable parameters such as the number of users. To facilitate verification of convergence, we need to effectively express such confinement by means of a bound condition "`#define BOUND(A,B,C) (A - B <= C) && (A - B >= -C)`". It can be used to maintain the previous (`B`) and current (`A`) values of interest and compare their distance with the bound (`C`).

Another area requiring special attention is the representation of rational numbers, or fractions in PROMELA. It is obvious that fractions such as transmission probabilities cannot be stored in integer variables, as they will be truncated. As a result, a `fraction` data type is devised to represent fractions as pairs of integers (numerator, denominator). In this way, fraction multiplications can be translated to integer multiplications and divisions.

We outline in the following two sections the strategies associated with the different process types in accordance with the cited requirements. These were translated into PROMELA codes of the processes during implementation. Interested readers may refer to our web site for the complete PROMELA codes.

3.1 Channel, User Processes, and Strategies

The channel process reviews the transmission probabilities of the network link in a timely fashion. It also coordinates the execution of user processes and administrator process. Since a communication channel usually assumes full capacity when it first

becomes available, the transmission probabilities of all classes are initialized to one. The channel process is also the first to be instantiated. This ensures that a communication infrastructure is available before user processes can possibly interact. For each priority class, starting from the highest priority, the channel process proceeds by computing the available transmission capacity of the network link and determining the total demands from users of the network. If the network capacity exceeds the total user demands, then the transmission probability of the class is set to one. Otherwise, it is set to be equal to the available capacity of the link divided by the total user demands. Note that transmission probabilities updates are interleaved between user process executions, and stored as global variables. This arrangement effectively simulates the situation where users of a network probe the channel for current congestion information.

A user process computes its next allocation of packets according to the current congestion information. The quantity of packet allocation is determined according to the net benefit each individual user receives from using the network. User benefit can be represented by a utility function, chosen to be $U_r(x_r) = A_r\sqrt{x_r}$ in this implementation. This definition satisfies the general requirement that a utility function increases with strictly decreasing marginal values. The scaling constant A_r is assumed to be uniquely related to a user's utility characterization. For simplicity, we assume a homogeneous user population, adopting the same scaling constant $A_r = A$.

In general, user strategy in packet allocation follows the description in Section 2.1. A user process is enabled only after the channel process has updated the transmission probabilities. All users in the network are guaranteed a fair chance to allocate their packets. In this implementation, zero packets are allocated to all low priority classes up to but not including the best effort class. If the condition $(u_{best-effort} / P_{best-effort} \geq u_{premium})$ is true, user submits $A^2 / 4u^2_{premium}$ packets to the premium class. When condition $(u_{best-effort} / P_{best-effort} \leq u_{premium})$ becomes true, user submits $A^2 P_{best-effort} / 4u^2_{best-effort}$ packets to the best effort class for transmission. Here $u_{best-effort}$ refers to the best effort price, $u_{premium}$ refers to the premium price, and $P_{best-effort}$ refers to the transmission probability of the best effort class. Zero packets are allotted to all remaining high priority classes. Note that when equality holds in the above governing condition, the user process non-deterministically selects either class to allot.

3.2 Administrator Process and Strategy

The objective of administrator process is to introduce pricing perturbations into the system at designated time intervals. It first calculates the aggregated user demand and determines if equilibrium has been established under the bound condition. It also asserts an inversely proportional relationship between the total demand and the best effort price at equilibrium point. When equilibrium is reached, the administrator process non-deterministically picks the new best effort price from the set $\{ u_{init} - 1, u_{init}, u_{init} + 1 \}$ where u_{init} represents the initial price of the best effort class. Users are forced to adapt their packet allocations in order to optimize their economic benefits under the new price. As a result, the transmission probabilities of the network can be dynamically influenced at run time.

4 Simulation and Verification

Simulations and verifications were performed in order to observe convergence of packet allocations, and its stability thereafter. Since Model 1 and 2 are supposed to mimic the continuous operation of a network channel, their simulations do not normally terminate. There is no final state in both Model 1 and 2. Data was therefore collected through simulations of 5-second duration. Approximately 1500 data points can be obtained from each trial. Simulation and data collection for both Model 1 and 2 followed the same procedure, however, each simulation was conducted independently of the others. For data analysis, simulation of Model 1 was performed three times, and the averaged data set was taken in the construction of Figure 3. Simulation of Model 2 was performed numerous times, however only data from one trial is shown in Figure 4. Each simulation of Model 2 mimics a self-contained network system, with an administrator process arbitrarily adjusting prices. Therefore, there is little coherence between different executions of the model to justify averaging of the data.

In this implementation we adopt 5 priority classes and 10 network users. All simulations and verifications were performed using SPIN Version 3.4.3 on a Sun UltraSPARC server with 4 400MHz processors and 4GB of physical memory.

4.1 Simulation of Channel, User, and Administrator Processes

Table 1 tabulates the simulation results of Model 1. It shows the data values from the first 20 cycles of the interaction between the user processes and the channel process. In particular, rows 1 to 10 show user packet allocations for the first ten cycles. Rows 11 to 20 show subsequent packet allocations for the same users. At Time 1, User 7 allocates 2500 packets under priority class 1. The aggregated demand for the entire channel at this time is only 2500 packets. Other users have not yet allotted their packets. At Time 2, User 0 allots 2000 packets to the network. The aggregated demand at this instance becomes the sum of allotments of both User 7 and User 0. Before User 3 releases its packets to the network at Time 3, it has computed that submitting to priority class 2 carries more advantage than to a lower class. Therefore, a shift in allotment to higher priority classes can be observed in subsequent cycles. After all 10 users have released their packets to the network, the allocation cycle repeats again. The order in which users release their packets to the network in the new cycle is determined randomly. This feature is important in simulating a working network where the arrivals of packets are commonly considered as random and independent events.

The aggregated demand represents the current allocation of all users of the network. For example, at Time 1, User 7 allocates 2500 packets for the first time slot. At Time 13, User 7 updates its allocation to 277 packets. Hence, at Time 13, `aggregated demand = aggregated demand + 227 – 2500.`

The aggregated allocation (demand) from all network users is presented over time in Figure 3. The data was averaged over three simulation runs. Before network becomes saturated, all users upload as many packets as desired under the assumption that transmission probabilities equal one. Guided by the results from benefit maximization, users realize that submitting as desired is not economically optimal. Total demand from users therefore begins to shift towards a value that would optimize

everyone's economic benefit. It settles over a plateau around iteration (time) 20 and remains relatively stable thereafter. This remarkable result indicates that the system has reached an equilibrium condition in packet allocation. Each user, in optimizing its own economic benefit, participates in a non-cooperative game with the outcome of effectively drifting the network traffic to a steady level. This essentially supports Proposition I that an equilibrium is indeed possible under priority pricing scheme.

To simulate Model 2, an administrator process is added. Data values between every price change follow the trend observed in the simulation of Model 1. Model 2 can thus be considered as a sequence of Model 1 simulations, each with a different best effort price maintained by the network administrator. Figure 4 illustrates the simulation of Model 2.

Table 1. Aggregated User Demand

		Priority N					
Time	User	1	2	3	4	5	Aggregated Demand
1	7	2500	0	0	0	0	2500
2	0	2000	0	0	0	0	4500
3	4	0	625	0	0	0	5125
4	9	0	625	0	0	0	5750
5	2	0	625	0	0	0	6375
6	1	0	625	0	0	0	7000
7	5	0	500	0	0	0	7500
8	6	0	416	0	0	0	7916
9	8	0	0	277	0	0	8193
10	3	0	0	277	0	0	8470
11	9	0	0	277	0	0	8122
12	8	0	0	277	0	0	8122
13	7	0	0	277	0	0	5899
14	2	0	0	277	0	0	5551
15	1	0	0	277	0	0	5203
16	3	0	0	277	0	0	5203
17	5	0	0	277	0	0	4980
18	0	0	0	277	0	0	3257
19	6	0	0	250	0	0	3091
20	4	0	0	225	0	0	2691

The upper curve in Figure 4 represents the current best effort price determined by the network administrator. The lower curve is the aggregated allocation for all users in the network along simulation time. To illustrate the effect of dynamic price adjustment, we focus on the interval between Time 100 and Time 200. The best effort price at Time 100 starts with a value of 5. The aggregated allocation stabilizes to an equilibrium of about 2800 packets at Time 114 and remains stable at this level. Note that the network capacity is only 2000 packets per time units. The probability of transmission is clearly below 1 at the current best effort price. Having observed that network traffic is maintained at a steady level, the administrator process increases the best effort price to 7. Facing a new higher cost in using the network, users begin to reduce the amount of packet allocation, resulting in a downward incline in the aggregated allocation curve. This phenomenon continues up to Time 148, where the total allocation drops to 1975 packets, which is below the total link capacity. As a result, a transmission probability of one is returned to the next user. This user allocates as many packets as desired under the false assumption that transmission for all packets

submitted would be successful. Subsequent results from benefit maximization remind users that network bandwidth is not as abundant as anticipated. Each user then reduces its allocation to maximize the economical benefit under the new price. As a result, a new equilibrium at a lower aggregated allocation level is observed.

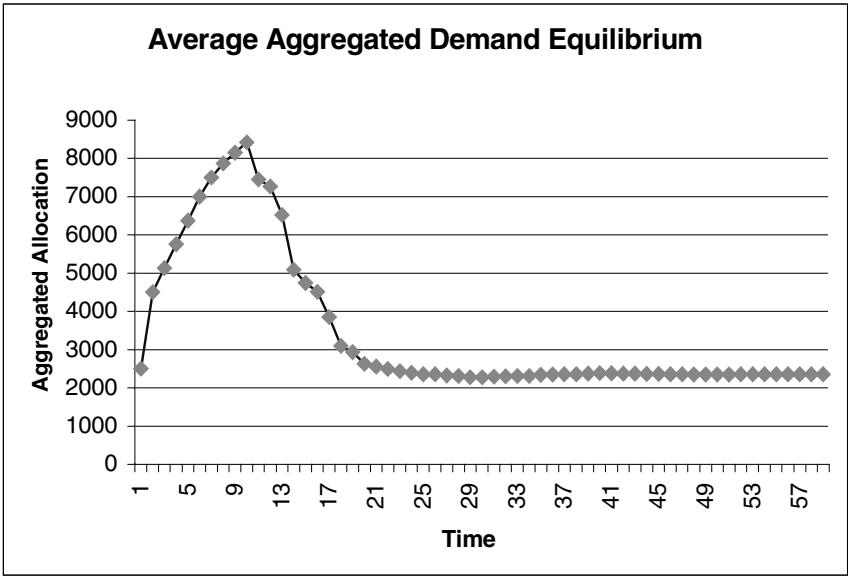


Fig. 3. Equilibrium in Pricing Model

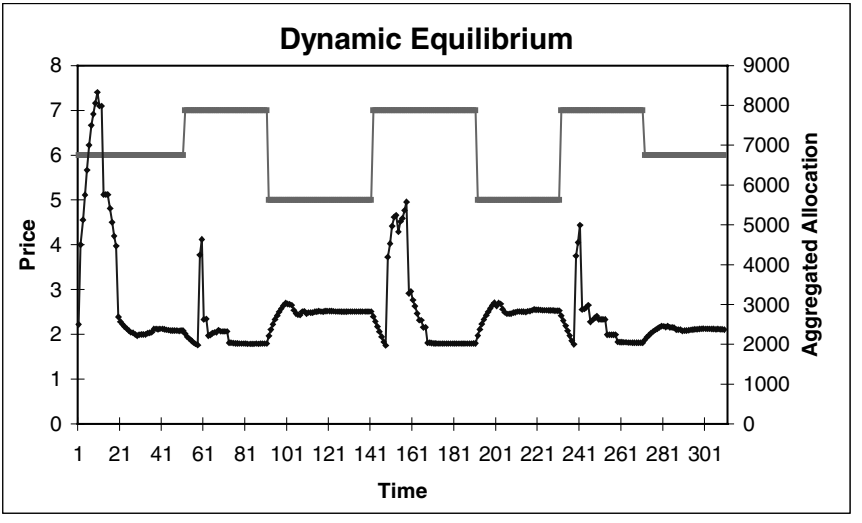


Fig. 4. Change of equilibrium with price

4.2 Verification of Convergence to Equilibrium (Proposition 1)

To verify that an equilibrium exists and converges, we formulate Proposition 1 as an LTL property and search for an acceptance cycle in the intersection of the Büchi automata for the corresponding never claim and Model 1. With fixed price levels in the absence of the administrator process, the stability nature of convergence could be verified.

We have specified the following LTL property for one user (user 0) and submitted to SPIN:

$$\Diamond\Box\left(curr\_cl = prev\_cl\right)\wedge\left(d_0(curr\_cl)-d_0(prev\_cl)\le 1\right)\wedge\left(d_0(curr\_cl)-d_0(prev\_cl)\ge -1\right)$$

This property reads "eventually user 0 will allocate all its packets to a single priority class and that allocation will remain constant. Alternatively, user 0 has reached an equilibrium in its packet allocation". For R number of users in the network, there will be R such LTL properties one for each user. Due to homogeneity, we can anticipate consistent results from all users. The combined effect of the LTL "always" ($\Box$) and "eventually" ($\Diamond$) operators verifies the existence *and* convergence of equilibrium in packet allocation.

We configured SPIN to use the maximum amount of available memory, 4096MB, and allowed a maximum search depth of up to 30,000,000. We also adopted bit-state hashing[1] with 26 bits in our verification. A typical run resulted in a 668-byte state vector, with 3.87348×10^7 states and 8.25047×10^7 transitions. In general verification of proposition 1 lasts for four hours with the given search depth.

4.3 Verification of Dynamic Priority Pricing (Proposition 2)

With the addition of an administrator process, Proposition 2 can be verified using Model 2 by means of an assertion statement embedded into the PROMELA code. To verify this proposition, the administrator process is introduced to perform pricing adjustments on the best effort class. Specifically we would like to verify an inverse relationship between price and allocation at equilibrium. The following assert statement is embedded into the administrator process where equilibrium is detected:

$$\texttt{assert(BOUND(}\, d_{all}(curr\_cl)\ \texttt{* Price(}\, curr\_cl\,\texttt{),K,20));}$$

This statement asserts "the product of total demand (allocation) and price of the current allocated class of any user is equal to K", modeled using the bound condition.

We configured SPIN to use the same resources as in the case of proposition 1. A typical run resulted in a 692-byte state vector, with 3.93016×10^7 states and 8.03291×10^7 transitions. In general verification of proposition 2 lasts for two and a half hours with the given maximum search depth.

[1] Exhaustive searches were also performed in both verifications and no counterexample was generated in either case for the given search depth.

5 Discussion

This section briefly discusses the role of PROMELA/SPIN in achieving the objectives of Model 1 and 2. It also discusses some limitations that we have experienced during the course of design and simulation.

5.1 Contribution of PROMELA/SPIN in Model 1

Model 1 was constructed based on the mathematical framework of Marbach's Model [1]. It is, however, distinguished from Marbach's model by the additional verification of the convergence property of equilibrium in packet allocations. This provision considerably strengthens the result of equilibrium existence and facilitates simulation and verification of congestion control in Model 2.

5.2 Contribution of PROMELA/SPIN in Model 2

With the addition of an administrator process, PROMELA/SPIN provide an important means to simulate runtime dynamic price adjustments in the priority pricing scheme. Model 2 distinguishes itself from earlier work on dynamic price adjustments [4] in two important ways. Using the simulation/verification capability of PROMELA/SPIN and the results obtained from Model 1, individual price-change intervals in Figure 4 can be considered as cascaded simulations of Model 1. Therefore, the existence of an equilibrium within each interval and its convergence can be formally verified. In addition, data collected from the simulation of Model 2 also indicates inverse proportionality between price and total packets allotted by network users. As illustrated in Figure 4, when price is high, the aggregated demand rests on a lower volume in equilibrium. This result obtained from PROMELA/SPIN is arguably stronger than similar results based on simulation alone. Further investigation and verification of this inverse price-volume relationship can be readily supported by PROMELA/SPIN. In this regard, congestion control studies based on priority pricing modeled by PROMELA/SPIN clearly present a unique advantage over other simulation tools.

5.3 Limitations

In Model 1 we have performed verification using only one set of initial conditions, namely all user processes begin with a zero allocation to all priority classes. Given another set of non-zero initial allocations, it would be desirable to observe an identical equilibrium condition. The ultimate goal is to verify the more general case where proposition 1 and 2 remain affirmative for all sets of initial conditions. Though little evidence is observed in this respect to suggest inherent limitations in PROMELA/SPIN, it is considered impractical, if not impossible, to model this situation by non-deterministic selection over all possible initial allocations. In spite of this, the simulation and verification framework provided by PROMELA/SPIN is still more powerful and convincing than merely simulation based on currently available network tools.

We observe that there are cases in which the approximating bound condition could pose a false detection of an equilibrium state during verification. This may happen in ill-behaved models where the system exhibits a steady drift in the perceived output value and never ends up in an equilibrium state. However such a change in output value over any two consecutive time slots (current – previous) can be so small (in particular, smaller than the bound) that the system is considered by the bound condition to have reached an equilibrium state, which of course is untrue. Further efforts are therefore needed to either enhance the bound condition to deal with this situation or devise different mechanism to detect equilibrium conditions.

We have compared the applicability of SPIN versus SMV in modeling priority pricing. In SMV, it is possible to model all users updating packet allocations in one single time slot. With modeling in SPIN, we are forced to simulate the interaction between users and the network asynchronously. In each time slot, only one process operates. In this regard, SMV's ability to model synchronous systems seems to make it a more logical choice for this work. After experimenting with both automated verification tools, we have however chosen PROMELA/SPIN for PROMELA's expressiveness and SPIN's verification capability. We are especially impressed with the rich language constructs and programming flexibility provided by PROMELA.

6 Conclusion

Throughout this work, we have been impressed with the expressiveness of PROMELA and interpretative strength of SPIN as a simulation/verification tool. We have been equally impressed with the strength of automated verification tools in general in providing modeling and verifying capability to complicated scenarios. By following the assumptions made in [1], we successfully developed three process types in SPIN, interacting with one another asynchronously in simulating a network link in action. Not only have we verified in SPIN that the postulated equilibrium indeed existed, we have shown in Model 1 that convergence to equilibrium can be achieved. Furthermore, we demonstrated in Model 2 the unique result that effective congestion control can be achieved through dynamic price adjustments. The choice of using PROMELA/SPIN for this work had arisen through necessity. The strength of the combined simulation and verification framework provided by PROMELA/SPIN would be difficult to replicate using another tool.

Acknowledgement. We would like to express our gratitude to Professor Marsha Chechik for her invaluable suggestions and patient guidance throughout this work. We are especially indebted to her tireless effort that had led us through numerous revisions of this manuscript. We would like to thank Professor Peter Marbach for his guidance in the understanding of the priority pricing model for congestion control.

References

1. P. Marbach, "*Pricing Priority Classes in Differentiated Services Networks,*" 37th Annual Allerton Conference on Communication, Control, and Computing, Monticello, IL, September 1999

2. A. Gupta, D. O. Stahl, and A. B. Whinston, "*A Priority Pricing Approach to Manage Multi-Service Class Networks in Real-Time,*". Presented at MIT Workshop on Internet Economics, March 1995
3. A. Gupta, D. O. Stahl, and A. B. Whinston, "*The Economics of Network Management,*" Communications of the ACM, 42(9): 57-63, September 1999
4. A. Gupta, D. O. Stahl, and A. B. Whinston, "*A Stochastic Equilibrium Model of Internet Pricing,*", Journal of Economics Dynamics and Control, 21:697-722, 1997
5. G.J. Holzmann, "*The Model Checker SPIN,*" IEEE Transactions on Software Engineering, 23(5): 1-17, May 1997
6. E. Fersman and B. Jonsson, "*Abstraction of Communication Channels in PROMELA: A Case Study,*" In SPIN Model Checking and Software Verification: Proc. 7th Int. SPIN Workshop, volume 1885 of Lecture Notes in Computer Science, pages 187-204, Stanford, CA, 2000. Springer Verlag
7. Theo C. Ruys and Rom Langerak, "*Validation of Bosch' Mobile Communication Network Architecture with SPIN,*" In Proceedings of SPIN97, the Third International Workshop on SPIN, University of Twente, Enschede, The Netherlands, April 1997
8. Klaus Havelund, Thomas Pressburger, "*Model Checking JAVA Programs using JAVA PathFinder*," STTT 2(4): 366-381 (2000)
9. Wil Janssen, Radu Mateescu, Sjouke Mauw, Peter Fennema, Petra van der Stappen, "*Model Checking for Managers,*" SPIN 1999: 92-107
10. Lee Breslau, Deborah Estrin, Kevin Fall, Sally Floyd, John Heidemann, Ahmed Helmy, Polly Huang, Steven McCanne, Kannan Varadhan, Ya Xu, and Haobo Yu, "*Advances in Network Simulation,*" IEEE Computer, 33 (5), pp. 59-67, May, 2000
11. Sandeep Bajaj, Lee Breslau, Deborah Estrin, Kevin Fall, Sally Floyd, Padma Haldar, Mark Handley, Ahmed Helmy, John Heidemann, Polly Huang, Satish Kumar, Steven McCanne, Reza Rejaie, Puneet Sharma, Kannan Varadhan, Ya Xu, Haobo Yu, and Daniel Zappala, "*Improving Simulation for Network Research,*" Technical Report 99-702, University of Southern California, March, 1999
12. Deborah Estrin, Mark Handley, John Heidemann, Steven McCanne, Ya Xu, and Haobo Yu, "*Network Visualization with the VINT Network Animator Nam*," Technical Report 99-703, University of Southern California, March, 1999
13. "*The Network Simulator – ns-2*," http://www.isi.edu/nsnam/ns/

A Model Checking Project at Philips Research

Leszek Holenderski

Philips Research, Eindhoven, The Netherlands
leszek.holenderski@philips.com

Abstract. A new verification project has recently started at Philips Research. It concerns verification of C-like concurrent programs used to specify functionality of hardware-software systems. The problem is to make the verification practical, in the sense that it must be mechanized, and applicable to thousands of lines of C-like code. We present the goals of the project and some combination of existing solutions we want to try.

1 Introduction

In what follows, we are only concerned with functional verification of hardware-software systems (as opposed to manufacture testing, for example). We distinguish between formal verification, as understood in academia, and verification in general, as practised in industry. In other words, we use the term verification in its broader sense, including testing of final products and simulation of models.

Verification of hardware-software systems is a serious effort consuming non-negligible part of the time devoted to their design and implementation. Due to the ever increasing complexity of the systems, the time needed to verify them will increase even more, finally reaching unacceptable levels. In addition, more and more high-tech products become part of safety-critical systems and thus must be extremely reliable, making the verification effort even more time consuming. These potential problems have caused some industries involved in the design and manufacturing of high-tech products to turn their attention to more systematic methods of verification (say, formal verification via model checking) which are promised by their proponents to shorten the verification time and/or increase reliability of final products.

Philips is far from facing any verification crises on the scale of several spectacular hardware/software failures well publicized in recent years. This is due to the fact that Philips is not that often involved in the production of safety-critical devices.[1] However, in anticipation of the potential problem of how the increased time spent on verification can affect time-to-market, Philips Research has already been experimenting for several years with incorporating some formal verification methods into design flows.

[1] For example, how much damage could be caused by a remote TV set controller that would deadlock after a very unusual combination of 10 buttons was pressed accidentally? Especially, if it is enough to switch the TV set off and on again, in order to resolve the problem?

M.B. Dwyer (Ed.): SPIN 2001, LNCS 2057, pp. 288–295, 2001.

Historically, formal verification started at Philips with hardware (i.e., circuits) since most devices produced by Philips were not software-intensive in the past.[2] Currently, formal equivalence checks between various levels of hardware descriptions are performed routinely, in most production projects. As far as formal verification of functional properties (say, via symbolic model checking) is concerned, the available tools still do not scale up well enough to apply them to anything bigger than designs with several hundred latches. For this reason, the only viable option is to apply model checking to relatively simple sub-components only. However, even in this restricted setting formal verification of properties is considered useful, since it is supposed to play some role in a bigger effort of making reusability of components easier.

Formal verification of software via model checking is currently not considered a serious option, due to the very limited usability of available tools. The main obstacle is a relatively small size of models/programs that can be analyzed automatically. Decomposition cannot help much here, for at least two reasons. First, it takes a substantial effort to isolate those software components of a final hardware-software system for which the properties expected by the rest of the system could easily be formalized. Second, the software components that could be isolated are usually orders of magnitude bigger than what current tools can cope with. On the other hand, software model checking has found its niche at Philips, but only in the area where it has already proven to be successful from the early beginnings, namely protocol verification. Several control-intensive protocols have been verified, mostly with SPIN. However, tricky protocols are not that common in daily practice, so this kind of application is not considered to be important enough to justify the substantial effort of incorporating model checking into design flows.

In what follows we present a new project that aims to change this negative perception of model checking, and formal methods in general.

2 Relevance of the Project

In many design flows currently used in Philips, the first precise descriptions of the functionality of a hardware-software system are given as programs in languages close to RTL (Register Transfer Level). For this reason, most of the verification time is currently spent on RTL specifications. However, more recent design flows, such as COSY [1], start at a much higher level.

In COSY, one starts with describing the functionality of a design in the YAPI language [10]. In short, YAPI is just concurrent C++ based on a version of Kahn networks: processes communicate via bounded FIFOs (using *read* and *write* primitives) and can non-deterministically choose between several possible communications (using *select* primitive). This fairly standard model of distributed concurrent systems has a relatively simple formal semantics, and this opens possibilities of some formal analysis of YAPI programs.

[2] With the exception of Philips Medical Systems. Their various scanning devices rely heavily on complex software, like image processing.

A design is specified as a YAPI program, in the form of a process network. This network is accompanied with a description of a final hardware architecture consisting of programmable circuits (microprocessors) and custom designed circuits (ASICs). The YAPI network is mapped on the hardware architecture. The processes mapped on microprocessors are called software components while those implemented as ASICs are called hardware components. The main goal of the COSY design flow is to start performance analysis as early as possible. To this end various models are prepared, and then simulated in order to gather some performance metrics. If the initial design does not satisfy assumed performance requirements it is modified by either repartitioning the required functionality into a new process network or changing the hardware architecture, or both. After a new re-mapping, the performance of a modified design is re-evaluated. The cycle may be repeated several times.

Since YAPI specifications can be simulated, functional verification can already start during a system design phase in order to expose some errors as early as possible, and thus making them easier/cheaper to correct. So far, such functional errors were exposed much later, usually during a system integration phase.

Currently, functional verification is done via simulation only, and there is no much support for the automation of the verification effort (the simulation scenarios must be prepared manually). The COSY design flow could be improved if some aspects of the verification of YAPI programs were mechanized. In this setting new verification challenges arise, which we will address in this project.

We distinguish between two kinds of verification tasks: property checking, where a design is verified to satisfy some kind of behavioral constraint, and equivalence checking, where two models of a design are checked to be equivalent (for instance, YAPI versus RTL, RTL versus gate, or gate versus gate).

Several verification tasks can be identified in the COSY flow:

(1) Property checking of YAPI networks. Here we can distinguish between generic properties (such as absence of deadlocks) and specific properties (such as invariants or assertion violations).
(2) Equivalence checking of two YAPI networks. Often, a YAPI process has to be split up into a network of processes, in order to facilitate a better software/hardware repartitioning. This is done manually. The question is whether the original YAPI process is equivalent to the resulting decomposed network.
(3) Equivalence checking of a YAPI process against its RTL implementation (in case of hardware components).
(4) Equivalence checking of a YAPI process against its software implementation (in case of software components). Often, the C++ code that works fine when simulated, will not suffice as DSP code, for instance. It has to be tweaked manually. Thus, it becomes important to check equivalence (at some abstraction level) of the original against the specialized code.

In order to simplify our job, we have decided to concentrate initially on verification tasks (1) and (2). The goal is to investigate methods that would

make a verification effort easier. The methods must be practical, in the sense that they must be mechanized and applicable to thousands of lines of YAPI code. We propose to investigate both model checking and directed testing.

3 Proposed Solutions

Formal verification will be used to find errors, and not to establish correctness. This makes an important difference since methods for finding errors usually scale up much better than those for establishing correctness, as far as the size of YAPI programs is concerned. The reason is quite obvious, and has to do with reliability of tools. A tool for establishing correctness is expected to be very reliable: it should be both sound and complete. First, the answer "yes" should always mean "correct" (soundness). Second, for every correct property the answer should be "yes" (completeness). A tool for finding errors does not need to be that exact. In fact, it needs to be neither sound nor complete. First, the "error" answer need not necessarily mean a real error (unsoundness) since a user can make his own judgement relatively easily, by simulating an erroneous trace. Second, it is not expected to find all errors (incompleteness).

We will investigate two approaches:

- How to use the model checker SPIN [8] to find errors in YAPI programs.
- How to use the testing tool TorX [12] for directed testing of YAPI programs.

There is a link between our project and the CdR (Côte-de-Resyste) project [2] which Philips Research is involved in. We want to use their TorX tool to automate both property checking and equivalence checking, by directed testing. In the CdR approach to testing, two objects are compared, using TorX. The first object (called an implementation) is checked to conform with the other object (called specification). Both implementation and specification are perceived by TorX as input-output state transition systems, i.e., labelled state transition systems whose labels are of two kinds: input and output actions. TorX explores the specification and chooses some input actions offered by the specification, to stimulate the implementation. The implementation performs a chosen input action and after some computation produces an output action in return, and afterwards waits for another input action. The output action produced by the implementation is then checked to conform with the specification, and if it conforms, another input action is chosen for the next stimulation. The testing process is supported by a formal theory, in the sense that the *ioco* (input-output conformance) relation between the implementation and specification is formally defined.

TorX has a very flexible architecture, and is rather a framework than a concrete tool. It consists of several modules that can relatively easily be changed to adapt TorX to various testing scenarios. First, both specification and implementation can be given in any language, by providing wrappers that allow TorX to perceive the two objects as input-output state transition systems. Second, a specific exploration strategy can relatively easily be programmed as one of the

modules. For example, in order to test YAPI process networks for deadlocks, the exploration strategy can consistently prefer YAPI read operations over YAPI write operations (or vice versa), trying to make FIFOs empty (or full), since these are the typical situations that cause deadlocks in YAPI networks. In this way, the generation of test cases is driven by a property being tested and thus the name "directed testing".

It seems worthwhile to try both SPIN and TorX to verify YAPI programs since they lead to different tradeoffs as far as the exactness in finding errors versus scalability is concerned. SPIN allows for exhaustive testing (in the expense of the limitations imposed on the size of the verification model) while TorX should in principle scale up better (in the expense of being non-exhaustive). Another reason for investigating TorX is given in Section 4.

Successful application of both tools relies on solving two fundamental problems:

- How to obtain a verification model from a YAPI program (the assumption here is that YAPI programs are too complex/informal to be analyzed directly).
- How to make the model small enough (to cope with the inherent limitations of most fully automated verification tools).

There are essentially two techniques to cope with the "size" problem: decomposition and abstraction. Compositional verification is hard to mechanize so we prefer to tackle the "size" problem by aggressive abstractions. We are going to try abstraction via omission and abstraction via projection, as realized in two existing tools. Details are given below.

3.1 Extracting a Promela Program from a YAPI Program

We are going to solve both the "model" and "size" problems simultaneously, by creating a tool similar to the AX tool [9]. AX extracts a Promela program from a C program. The extracted program can then be fed to both SPIN and TorX (it happens that Promela is one of many input languages of TorX). The term "extracts" is used instead of "translates to" since the translation from C to Promela is combined with abstracting, so the Promela program is not necessarily equivalent to the C program. (Note that this does not pose theoretical problems since we are only finding errors and not establishing correctness.) The abstracting process is driven by a table that specifies what to do with some basic statements (i.e., declarations, assignments, guards and procedure calls) that appear in the C program. The table must be prepared by a user, and this is (hopefully) the only place where a human intervention is needed in extracting the model.

The table driven extraction is very flexible. On one hand, it allows for fully automated abstractions (via built-in general abstraction rules). On the other hand, it allows to fine tune abstractions as much as needed (via more specific rules provided by a user). The flexibility is crucial for the successful usage of SPIN and TorX on real-life complex YAPI programs since it allows to fine tune

a verification effort relatively easily (to trade the exactness of results for the time that can be spent on verification).

Since YAPI is based on C++, and not on C, extracting a Promela model from YAPI seems unfeasible (manipulations on objects, including the memory management for object allocation and deallocation, would have to be simulated in Promela and this would lead to explosion of state space). Fortunately, C++ classes are only used in YAPI to describe the structure of a YAPI process network. The behaviour of processes is specified in C, to avoid potential problems with mapping them later as hardware components.

As far as the specification of properties is concerned, the generic properties (like deadlock) don't need to be specified. The specification of behavioral properties poses some problems, but only if we want to be very general. In principle, behavioral properties can be grouped into safety and liveness properties. The safety properties pose no problems since they can easily be specified as observers or assertion violations, either in YAPI (from which Promela is then extracted) or directly in Promela. For the liveness properties, one can either revert to LTL (linear temporal logic) or, in case this is perceived as too difficult for a user, one can design a less general yet simpler language on top of LTL.

3.2 Extracting a Verification Model Directly from a YAPI Program, on-the-Fly

In order to avoid the potentially tedious task of preparing an abstraction table for the AX tool, we will also try the approach used in the VeriSoft tool [5]. In this approach, the whole state space of a C program (which is large, due to the very fine granularity of C statements) is projected on a much smaller space consisting of so-called observable states. The observable states are defined by break points, as in debugging, and the transitions are the computations between break points.

As in the first solution, the "model" and "size" problems are solved simultaneously, but this time a relatively small verification model is obtained directly, just by executing a concurrent C program. The observable states are extracted on-the-fly, by executing a program in parallel with the VeriSoft model checker: whenever the program reaches a break point, the model checker gets control and analyzes the current observable state. In fact, the VeriSoft model checker is provided as a library, and the parallel composition of the program and the model checker is realized simply by linking the program with the library. Since the observable states visited during model checking are too big to be stored, the stateless model checking technique [5] is used to overcome this problem.

Gluing YAPI with VeriSoft is easy. First, break points are naturally defined by the YAPI communication primitives (read, write and select). Second, YAPI is not implemented as an extension to C++, in the sense that no special version of C++ compiler is needed. Instead, it is implemented as several C++ libraries that implement the YAPI communication primitives on a particular platform (one library for each combination of a processor and an operating system that YAPI supports). Since VeriSoft library already provides implementation of bounded queues, in order to glue YAPI with VeriSoft it is enough to write

yet another YAPI library that simply delegates the YAPI communication primitives to VeriSoft communication primitives. A verification experiment consists in linking an unmodified YAPI program with two libraries (YAPI and VeriSoft) and executing it.

Gluing YAPI with TorX does not pose fundamental problems as well, due to the very modular architecture of the TorX toolset. Again, yet another version of YAPI library will do the automatic projection on observable states.

4 Another Reason to Investigate TorX

When verifying complex system (say, specified by thousands of lines of YAPI code), the only hope of successfully using the model checking technology together with abstractions (as advocated in Section 3.1 and 3.2) is to verify relatively simple properties. By a relatively simple property we mean a property that depends on a small part of the system only (in other words, that only concerns a particular aspect of a system). Only in this way one will be able to abstract the whole system to a model small enough to be analyzed. Observe that equivalence checking is not a relatively simple property in this sense since it involves the whole system (in fact, two of them). Thus, SPIN seems to be hopeless for equivalence checking of complex YAPI programs.

Since one of the goals of our project is to support equivalence checking, we have decided to use TorX for this purpose. TorX establishes a kind of containment relation between implementation and specification (formally defined in [12]). To check equivalence of a pair of YAPI programs, TorX can be run twice. In the first run, the first program plays the role of specification while the other is treated as implementation. In the second run, the roles are reversed. Such a reversal of roles is possible since both the implementation and specification are perceived by TorX as objects of the same kind (namely, input-output state transition systems).

5 Related Work

There has been a significant amount of research in the past years to adapt model checking to program checking. Beside AX and VeriSoft, tools like JCAT [4], Java PathFinder [7] and Bandera [3] also caught our attention. In our opinion, the approach taken in JCAT and early versions of Java PathFinder (i.e., a direct translation of Java to Promela) is unlikely to enable tractable model checking for non-trivial programs. Although a newer version of Java PathFinder [11] employs a dedicated model checker that works directly on Java bytecode, it remains to be seen how well this new approach scales up. On the other hand, Bandera seems to be quite a promising tool for our purpose of verifying JAPI programs, due to its support for slicing and abstraction. We have not yet considered it in our project since we envision much bigger problem in adapting Bandera to YAPI than the AX and VeriSoft tools.

Another promising approach to bridge the gap between model checking and program checking may emerge from a relatively recent research on runtime verification. In runtime verification, a program is executed once, and various kinds of information is gathered during this particular run. This information can then be used to predict whether other different runs may violate some properties of interest. Of particular interest to us is the deadlock detection algorithm proposed in [6] which we hope to adapt to YAPI in the future.

6 Conclusions

In order to make the verification of YAPI programs practical, we propose to use model checking and directed testing combined with abstraction via omission and projection. The combination can be obtained relatively easy, by combining four existing tools: SPIN, TorX, AX and VeriSoft. This will allow to asses relatively quickly how well the proposed methods perform.

References

1. J.-Y. Brunel et al, *COSY: a Methodology for System Design Based on Reusable Hardware & Software IP's*, EMMSEC'98, 709–716, Bordeaux France, Sept. 1998.
2. Côte de Resyste, http://fmt.cs.utwente.nl/projects/CdR-html
3. J. Corbett et al, *Bandera: Extracting Finite-state Models from Java Source Code*, The 22nd Int. Conference on Software Engineering, Limerich, Ireland, June 2000, ACM Press.
4. C. Demartini, R. Iosif and R. Sisto, *A deadlock detection tool for concurrent Java programs*, Software Practice and Experience, 29(7):577–603, July 1999.
5. P. Godefroid, *Model Checking for Programming Languages using VeriSoft*, POPL'1997 (The 24th ACM Symposium on Principles of Programming Languages), 174–186, Paris, Jan. 1997.
6. K. Havelund, *Using Runtime Analysis to Guide Model Checking of Java Programs*, SPIN'2000 (The 7th SPIN Workshop), Stanford University, USA, LNCS 1885, Springer Verlag, 245–264, Sept. 2000.
7. K. Havelund and T. Pressburger, *Model Checking Java Programs Using Java PathFinder*, STTT (Int. Journal on Software Tools for Technology Transfer), 2(4):366–381, April 2000.
8. G. Holzmann, *The Model Checker Spin*, IEEE Trans. on Software Engineering, 23(5):279–295, May 1997.
9. G. Holzmann, *Logic Verification of ANSI-C code with SPIN*, SPIN'2000 (The 7th SPIN Workshop), Stanford University, USA, LNCS 1885, Springer Verlag, 131–147, Sept. 2000.
10. E.A. de Kock et al, *YAPI: Application Modelling for Signal Processing Systems*, DAC'2000 (The 37th Design Automation Conference), Los Angeles, 2000.
11. W. Visser et al, *Java PathFinder - Second Generation of a Java Model Checker*, Post-CAV Workshop on Advances in Verification, Chicago, July 2000.
12. R. de Vries et al, *Côte de Resyste in PROGRESS*, PROGRESS'2000 Workshop on Embedded Systems, Utrecht, The Netherlands, 141–148, Oct. 2000. http://fmt.cs.utwente.nl/publications/cdr.pap.html

Applications of Model Checking at Honeywell Laboratories*

Darren Cofer, Eric Engstrom, Robert Goldman,
David Musliner, and Steve Vestal

Honeywell Laboratories, Minneapolis MN 55418, USA
darren.cofer@honeywell.com

Abstract. This paper provides a brief overview of five projects in which Honeywell has successfully used or developed model checking methods in the verification and synthesis of safety-critical systems.

1 Introduction

Embedded software in control and communication systems is becoming increasingly complex. Verification of important safety or mission-critical properties by traditional methods of test and design review will soon be impossible or prohibitively expensive. The only way developers of complex systems will be able to manage life cycle costs and yet still field systems with the functionality that the market demands is to rely on mathematical models and analyses as the basis for our designs.

For several years Honeywell has been investigating the use of model checking techniques to analyze the behavior and correctness of a variety of safety and mission-critical systems. This paper provides a brief overview of five projects in which we have successfully used or developed model checking tools and methods.

2 Automatic Synthesis of Real-Time Controllers

Unmanned Aerial Vehicles (UAVs) under development by the military and deep space probes being developed by NASA require autonomous, flexible control systems to support mission-critical functions. These applications require hybrid real-time control systems, capable of effectively managing both discrete and continuous controllable parameters to maintain system safety and achieve system goals.

We have developed a novel technique for automatically synthesizing hard real-time reactive controllers for these and other similar applications that is based on model-checking verification. Our algorithm builds a controller incrementally, using a timed automaton model to check each partial controller for

* This material is based in part upon work supported by Rome Labs (contract F30602-00-C-0017), AFOSR (contract F49620-97-C-0008), and NASA (cooperative agreement NCC-1-399)

M.B. Dwyer (Ed.): SPIN 2001, LNCS 2057, pp. 296–303, 2001.

correctness. The verification model captures both the controller design and the semantics of its execution environment. If the controller is found to be incorrect, information from the verification system is used to direct the search for improvements. Using the CIRCA architecture for adaptive real-time control systems [6], these controllers are synthesized automatically and dynamically, on-line, while the platform is operating. Unlike many other intelligent control systems, CIRCA's automatically-generated control plans have strong temporal semantics and provide safety guarantees, ensuring that the controlled system will avoid all forms of mission-critical failure.

CIRCA uses model-checking techniques for timed automata [11] as an integral part of its controller synthesis algorithm. CIRCA's *Controller Synthesis Module* (CSM) incrementally builds a hard real time reactive controller from a description of the processes in its environment, the control actions available, and a set of goal states. To do this, the CSM must build a model of the controller it is constructing that is faithful to its execution semantics and use this model to verify that the controller will function safely in its environment.

CIRCA employs two strategies to manage this complex task. First, its mission planner decomposes the mission into more manageable subtasks that can be planned in detail. Second, CIRCA itself is decomposed into two concurrently-operating subsystems (see Figure 1): an *AI Subsystem* including the CSM reasons about high-level problems that require powerful but potentially unbounded computation, while a separate *real-time subsystem* (RTS) reactively executes the generated plans and enforces guaranteed response times.

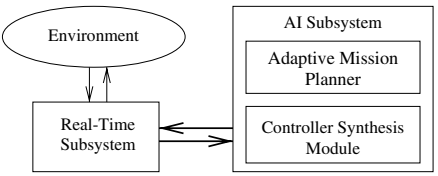


Fig. 1. Basic CIRCA architecture.

The CIRCA CSM builds reactive discrete controllers that observe the system state and some features of its environment and take appropriate control actions. In constructing such a controller, the CSM takes a description of the processes in the system's environment, represented as a set of transitions that modify world features and that have worst case time characteristics. From this description, CIRCA incrementally constructs a set of reactions and checks them for correctness using a timed automaton verifier.

The real-time controllers that CIRCA builds sense features of the system's state (both internal and external), and execute reactions based on the current state. That is, the CIRCA RTS runs a memoryless reactive controller.

Given the above limitation on the form of the controller, the controller synthesis problem can be posed as choosing a control action for each reachable

state (feature-value assignment) of the system. This problem is not as simple as it sounds, because the set of reachable states is not a given — by the choice of control actions, the CSM can render some states (un)reachable.

Indeed, since the CSM focuses on generating safe controllers, a critical issue is making failure states unreachable. In controller synthesis, this is done by the process we refer to as preemption. A transition t is preempted in a state s iff some other transition t' from s must occur before t could possibly occur.

Note that the question of whether a transition is preempted is not a question that can be answered based on local information: preemption of a transition t in a state s is a property of the controller as a whole, not of the individual state. It is this non-local aspect of the controller synthesis problem that has led us to use automatic verification.

3 Real-Time Scheduler of the MetaH Executive

MetaH is an emerging SAE standard language for specifying real-time fault-tolerant high assurance software and hardware architectures[8]. Users specify how software and hardware components are combined to form an overall system architecture. This specification includes information about one or more configurations of tasks, their message and event connections, information about how these objects are mapped onto a specified hardware architecture, and information about timing behaviors and requirements, fault and error behaviors and requirements, and partitioning and safety behaviors and requirements.

The MetaH executive supports a reasonably complex tasking model using preemptive fixed priority scheduling theory [1,2]. It includes features such as period-enforced aperiodic tasks, real-time semaphores, mechanisms for tasks to initialize themselves and to recover from internal faults, and the ability to enforce execution time limits on all these features (time partitioning).

Traditional real-time task models cannot easily handle variability and uncertainty in clock and computation and communication times, synchronizations (rendezvous) between tasks, remote procedure calls, anomalous scheduling in distributed systems, dynamic reconfiguration and reallocation, end-to-end deadlines, and timeouts and other error handling behaviors. One of the goals of this project was to use dense time linear hybrid automata models to analyze the schedulability of real-time systems that cannot be easily modeled using traditional scheduling theory.

Figure 2 shows an example of a simple hybrid automata model for a preemptively scheduled, periodically dispatched task. A task is initially waiting for dispatch but may at various times also be executing or preempted. The variable t is used as a timer to control dispatching and to measure deadlines. The variable t is set to 0 at each dispatch (each transition out of the waiting state), and a subsequent dispatch will occur when t reaches 1000. The assertion $t \leq 750$ each time a task transitions from executing to waiting (each time a task completes) models a task deadline of 750 time units. The variable c records accumulated compute time, it is reset at each dispatch and increases only when the task is

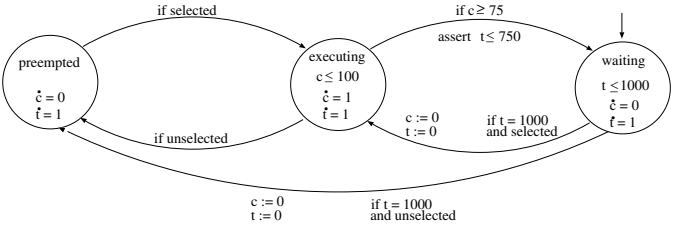


Fig. 2. Hybrid Automata Model of a Preemptively Scheduled Task.

in the computing state. The invariant $c \leq 100$ in the computing state means the task must complete before it receives more than 100 time units of processor service, the guard $c \geq 75$ on the completion transition means the task may complete after it has received 75 time units of processor service (i.e. the task compute time is uncertain and/or variable but always falls in the interval $[75, 100]$).

In this example the edge guards `selected` and `unselected` represent scheduling decisions made at scheduling events (called scheduling points in the real-time literature). These decisions depend on the available resources (processors, busses, etc.) being shared by the tasks.

We began our work using an existing linear hybrid automata analysis tool, HyTech [4], but found ourselves limited to very small models. We developed and implemented a new reachability method that was significantly faster, more numerically robust, and used less memory. However, our prototype tool allows only constant rates (not rate ranges) and does not provide parametric analysis.

Using this new reachability procedure we were able to accomplish one of our goals: the modeling and verification of a piece of real-time software. We developed a hybrid automata model for that portion of the MetaH real-time executive that implements uniprocessor task scheduling, time partitioning and error handling. Results of this work are presented in [9].

Our hybrid automata model was not developed using a separate modeling language. Instead, statements were added to the code to generate pieces of the model as subprograms were unit tested. When unit testing of all subprograms was completed, the complete system model was then subjected to reachability analysis. This provided a high degree of traceability between code and model.

The conditions we checked during reachability analysis were that all deadlines were met whenever the schedulability analyzer said an application was schedulable; no accessed variables were unconstrained (undefined) and no invariants were violated on entry to a region; and no two tasks were ever in a semaphore locking state simultaneously. Assertion checks appearing in the code were also captured and verified in the model.

We discovered nine defects in the course of our verification exercise. In our judgement, three of these would have been almost impossible to detect by testing due to the multiple carefully timed events required to produce erroneous behavior.

There are limits on the degree of assurance that can be provided, but in our judgement the approach may be significantly more thorough and significantly less expensive that traditional testing methods. This result suggests the technology has reached the threshold of practical utility for the verification of small amounts of software of a particular type.

4 Fault-Tolerant Ethernet Protocol

Large industrial control systems require highly reliable communcation services, especially in chemical processing applications. We have developed a Fault-Tolerant Ethernet (FTE) communication network for industrial control applications. The network is composed of dual redundant LANs implemented with standard commercially available hardware and software drivers. It is transparent to control applications, both in terms of application coding and communication latency.

The original FTE protocols performed failure detection and recovery for single point of network failure. We used the Spin model checker [5] to produce a simple model of the dual LAN fault detection algorithm and verify the correctness of the initial version of the protocols. We found one significant error and some minor ambiguities and potential design errors that were addressed in the final design and implementation of the system.

We first constructed a single node model to verify that faults are correctly detected by the fault detection algorithm, and that single message losses are tolerated. The model consisted of four processes: the two LANs, a state broadcast process (sends pairs "I'm alive" messages on the two LANs), and the fault detection algorithm.

Faults were injected by permitting single message losses in each of the two LANs. Messages could be lost at any time, as long as two consecutive messages on a LAN were never lost. In this situation, the fault detection algorithm should never enter its error state.

Verification of this model identified an execution in which the error state could be (incorrectly) entered, thus exposing an error in the algorithm. The problem occured because the algorithm assumed that a lost message always results in the arrival of two consecutive messages from the same LAN. However, message ordering is not sufficient to detect a missing message.

The fault detection algorithm was revised to add a short sequence number to each "I'm alive" message. The maximum sequence number must be larger than the number of consecutive lost messages that are to be tolerated, so 2 or 3 bits is sufficient. Each message in a pair is given the same sequence number so pairs of messages can be identified by matching sequence numbers. However, Spin identified an further counterexample for the revised design in which alternating messages are lost on each LAN. There are never two messages lost in a row on either LAN, but a complete pair of messages is never received.

Two possibilities were considered to deal with this situation:

1. Revise the robustness requirement to exclude the message loss scenario identified above.

2. Relax the fault detection algorithm to tolerate the message loss scenario. This could be done by clearing the missing message counters upon receipt of the missing message, even if its sequence number does not match.

The second approach was selected. This is reasonable since messages are in fact being received from both LANs, so there is no reason to declare a LAN failure.

5 Time Partitioning in Integrated Modular Avionics

The Digital Engine Operation System (DEOS) was developed by Honeywell for use in our Primus Epic avionics product line. DEOS supports flexible Integrated Modular Avionics applications by providing both space partitioning at the process level and time partitioning at the thread level. Space partitioning ensures that no process can modify the memory of another process without authorization, while time partitioning ensures that a thread's access to its CPU time budget cannot be impaired by the actions of any other thread.

The DEOS scheduler enforces time partitioning using a Rate Monotonic Analysis (RMA) scheduling policy. Using this policy, threads run periodically at specified periods and they are given per-period CPU time budgets which are constrained so that the system cannot be overutilized [3].

Honeywell engineers and researchers at NASA Ames collaborated to produce a model for use with the Spin model checker [7]. The model was translated from a core "slice" of the DEOS scheduler. This model was then checked for violations of a global time partitioning invariant using Spin's automated state space exploration techniques. We successfully verified the time partitioning invariant over a restricted range of thread types. We also introduced into the model a subtle scheduling error; the model checker quickly detected that the error produced a violation of the time partitioning invariant.

We attempted to verify the following liveness property, which is necessary (but not sufficient) for time partitioning to hold: If the CPU is not scheduled at 100% utilization, then the idle thread should run during every longest period. When verification was attempted with two user threads and dynamic thread creation and deletion enabled, Spin reported a violation. The error scenario results when one of the user threads deletes itself and its unused budget is immediately returned to the main thread (instead of waiting until the next period). This bug was, in fact, one which had been previously discovered by Honeywell during code inspections (but intentionally not disclosed to the NASA researchers performing the verification). Therefore, it would seem that model checking can provide a systematic and automated method for discovering subtle design errors.

Our current time partitioning model does not incorporate several important time-related features of DEOS. These include:

– The existence of multiple processes, which serve as (among other things) budget pools for dynamically creating and deleting threads. Time partitioning must be verified at a process level as well as a thread level.

- Several types of thread synchronization primitives provided by DEOS, including counting semaphores, events, and mutexes. These allow threads to suspend themselves or be suspended in ways not accounted for by the current model.
- The existence of aperiodically running threads, used to service aperiodic hardware interrupts.

We plan to integrate these features into the model and verify that time partitioning still holds with these features present. The principal challenge here will be keeping the state space size manageable while increasing the complexity of the model by incorporating these new features. The current model has already approached the bounds of exhaustive verifiability on currently available computer systems, although subsequent optimizations have reduced the size of the model somewhat. Furthermore, the current model has only been tested on a small range of possible thread budgets and periods.

6 Synchronization Protocol for Avionics Communication Bus

ASCB-D (Avionics Standard Communications Bus, rev. D) is a bus structure designed for real-time, fault-tolerant periodic communications between Honeywell avionics modules. The algorithm we modeled is used to synchronize the clocks of communicating modules to allow periodic transmission. The algorithm is sufficiently complex to test the limits of currently available modeling tools. Working from its specification, we modeled the synchronization algorithm and verified its main correctness property using Spin [10].

The ASCB-D synchronization algorithm is run by each of a number of *NICs* (Network Interface Cards) which communicate via a set of buses. For each side of the aircraft there are two buses, a primary and a backup bus. Each NIC can listen to, and transmit messages on, both of the buses on its own side. It can also listen to, but not transmit on, the primary bus on the other side.

The operating system running on the NICs produces *frame ticks* every 12.5 msec which trigger threads to run. In order for periodic communication to operate, all NICs' frame ticks must be synchronized within a certain tolerance. The purpose of the synchronization algorithm is to enable that synchronization to occur and to be maintained, within certain performance bounds, over a wide range of faulty and non-faulty system conditions.

The synchronization algorithm works by transmitting special timing messages between the NICs. Upon initial startup, these messages are used to designate the clock of one NIC as a "reference" to which the other NICs synchronize; after synchronization is achieved, the messages are used to maintain synchronization by correcting for the NICs' clock drift relative to each other. The algorithm is required to achieve synchronization within 200 msec of initial startup regardless of the order in which the NICs start.

The synchronization algorithm must also meet the 200 msec deadline in the presence of malfunctioning NICs or buses. For example, any one of the NICs might be unable to transmit on, or unable to listen to, one or more of the buses;

or it might babble on one of the buses, sending gibberish which prevents other messages from being transmitted; or one of the buses might fail completely at startup, or fail intermittently during operation.

The introduction of an explicit numerical time model, and the combination of that time-modeling capability and the message-transmission capability in the same "environment" process, allowed us to produce a tractable four-NIC model that includes most of the important features of the synchronization algorithm.

The environment process encapsulates all those parts of the system that provide input to the algorithm we wish to model (frame ticks, buffers, and buses), while the NIC process encapsulates the algorithm itself. The interface between the two is simple and localized. It allows faults to be injected and complicated hardware interactions to be added with no change required to the NIC code. Complicated tick orderings produced by frames of different lengths are explicitly and accurately represented in the model. Because the interface between environment and NIC includes all the data that must be shared between them, there is no need for global data structures. This allows Spin's compression techniques to reduce the memory required to store each state.

With this model we were able to verify the key system property as an assertion in the environment process that states that all NICs should be in sync within 200 msec of the startup time.

References

1. P. Binns. Scheduling Slack in MetaH. *Real-Time Systems Symposium*, December 1996.
2. P. Binns. Incremental Rate Monotonic Scheduling for Improved Control System Performance. *Real-Time Applications Symposium*, 1997.
3. P. Binns. Design Document for Slack Scheduling in DEOS. Honeywell Technology Center Technical Report SST-R98-009, September 1998.
4. T. Henzinger, P. Ho, H. Wong-Toi. A User Guide to HyTech. University of California at Berkeley, `www.eecs.berkeley.edu/~tah/HyTech`).
5. G. Holzmann. The SPIN Model Checker. *IEEE Transactions on Software Engineering*, vol. 23, no. 5, May 1997, pp. 279-295.
6. D. Musliner, E. Durfee, and K. Shin. CIRCA: a cooperative intelligent real-time control architecture. *IEEE Transactions on Systems, Man and Cybernetics* 23(6):1561-1574.
7. J. Penix, W. Visser, E. Engstrom, A. Larson, and N. Weininger. Verification of time partitioning in the DEOS scheduler kernel. *ICSE 2000.*
8. S. Vestal. An architectural approach for integrating real-time systems. *Workshop on Languages, Compilers and Tools for Real-Time Systems*, June 1997.
9. S. Vestal. Modeling and verification of real-time software using extended linear hybrid automata. *Fifth NASA Langley Formal Methods Workshop*, June 2000 (see `http://atb-www.larc.nasa.gov/fm/Lfm2000/`).
10. N. Weininger, D. Cofer. Modeling the ASCB-D Synchronization Algorithm with Spin: A Case Study. *7th International Spin Workshop*, September 2000.
11. S. Yovine. Kronos: A verification tool for real-time systems. *International Journal of Software Tools for Technology Transfer*, vol. 1, no. 1/2, Oct. 1997.

Coarse-Granular Model Checking in Practice

Bernhard Steffen, Tiziana Margaria, and Volker Braun

LS V, Universität Dortmund, Baroper Str. 301, D-44221 Dortmund (Germany),
{steffen,tiziana,braun}@ls5.cs.uni-dortmund.de
METAFrame Technologies, Borussia Str. 112, D-44149 Dortmund (Germany),
{bsteffen,tmargaria,vbraun}@metaframe.de

Abstract. In this paper, we illustrate the power of 'classical' iterative model checking for verifying quality of service requirements at the coarse granular level. Characteristic for our approach is the library-based modelling of applications in terms of service logic graphs, the incremental formalization of the underlying application domain in terms of temporal quality of service constraints, and the library-based consistency checking allowing continuous verification of application- and purpose-specific properties by means of model checking. This enables application experts to safely configure their own applications without requiring any specific knowledge about the underlying technology. The impact of this approach will be demonstrated in three orthogonal industrial application scenarios.

1 Motivation

Moving large portions of the needed application programming load from programming experts to application experts or even to end users is today a major economic challenge. For application experts, this requires *freeing* activities intrinsic to the development of applications from their current need of *programming expertise*. For end users to take over advanced reshapings of applications, little expertise in the particular *application domain* must suffice.

In this paper, we present our experience with the Agent Building Center, a tool for *formal methods-based, application-specific* software design, which is designed for directly supporting the described division of labour by means of a model-checking-based consistency mechanism. In particular, it supports application experts during their combination of application-specific business objects to service logic graphs (SLG), which explicitly model the intended workflows of the application under development. These graphs are sufficient to automatically generate the application code for the considered platforms.

We will illustrate the impact of our method along the following three application scenarios:

- The *Integrated Test Environment* (ITE), designed to steer the system-level test of computer telephony integrated (CTI) applications (Sect. 3).
- An E-commerce shop with personalization and online adminstration (Sect. 4).

M.B. Dwyer (Ed.): SPIN 2001, LNCS 2057, pp. 304–311, 2001.

– A role-based editorial system supporting the whole evaluation process for international conferences (Sect. 5).

This paper sketches the Agent Building Center in Section 2, before describing the above-mentioned three application scenarios and finally drawing some conclusions in Section 6.

2 The Agent Building Center

Here we provide an overview of the Agent Building Center in the light of the announced *programming-free programming* paradigm of application development.

In the ABC, application development consists of the behaviour-oriented combination of Building Blocks (BBs)[2] on a *coarse* granular level. BBs are here identified on a functional basis, understandable to application experts, and usually encompass a number of 'classical' programming units (be they procedures, classes, modules, or functions). They are organized in application-specific collections. In contrast to (other) component-based approaches, e.g., for object-oriented program development, the ABC focusses on the dynamic behaviour: (complex) functionalities are graphically stuck together to yield flow graph-like structures embodying the application behaviour in terms of control.

Throughout the behaviour-oriented development process, the ABC offers access to mechanisms for the verification of libraries of constraints by means of model checking. The model checker individually checks hundreds of typically very small and application- and purpose-specific constraints over the flow graph structure. This allows concise and comprehensible diagnostic information in the case of a constraint violation, in particular since the feedback is provided at the application rather than at the programming level. These characteristics are the key towards distributing labour according to the various levels of expertise.

Programming Experts: They are responsible for the software infrastructure, the runtime-environment for the compiled services, as well as for programming the BBs.

Constraint Modelling Experts: They classify the BBs, typically according to technical criteria like their version or specific hardware or software requirements, their origin (where they were developed) and, here, most importantly, according to their intent for a given application area. The resulting classification scheme is the basis for the constraint definition in terms of modal formulas.

Application Experts: They develop concrete applications, by graphically combining BBs into coarse-granular flow graphs. These graphs can be immediately executed by means of an interpreter, in order to validate the intended behaviour (rapid prototyping). Model checking ([2]) guarantees the consistency of the constructed graph with respect to the constraint library.

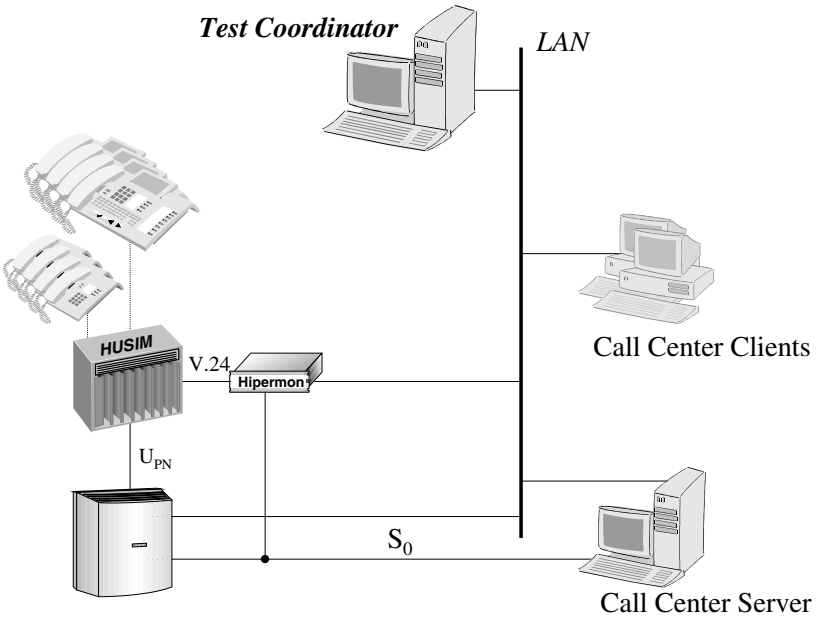


Fig. 1. Example of an Integrated CTI Platform

3 System-Level Testing of Telephony Systems

Traditional approaches fail to enter practice in the scenario we are considering here, because they do not fit the current test design practice. The main obstacle is the absence of any fine granular formal model of the involved systems.

In the approach described in [1] we present a component-based test design on top of a library of elementary but intuitively understandable test case fragments, supported by formal methods-based validation. Together, this establishes a coarse-granular 'meta-level' on which

- test engineers are used to think,
- test cases and test suites can be easily composed,
- test scenarios can be configured and initialized,
- critical consistency requirements including version compatibility and frame conditions for executability are easily *formulated*, and
- consistency is fully automatically enforced via *model checking* and error diagnosis.

As a typical example of an integrated CTI platform, Fig. 1 shows a midrange telephone switch and its environment. The switch is connected to the ISDN telephone network or, more generally, to the public service telephone network (PSTN), and acts as a "normal" telephone switch to the phones. Additionally, it communicates directly via a LAN or indirectly via an application server with CTI applications that are executed on PCs. Like the phones, CTI applications

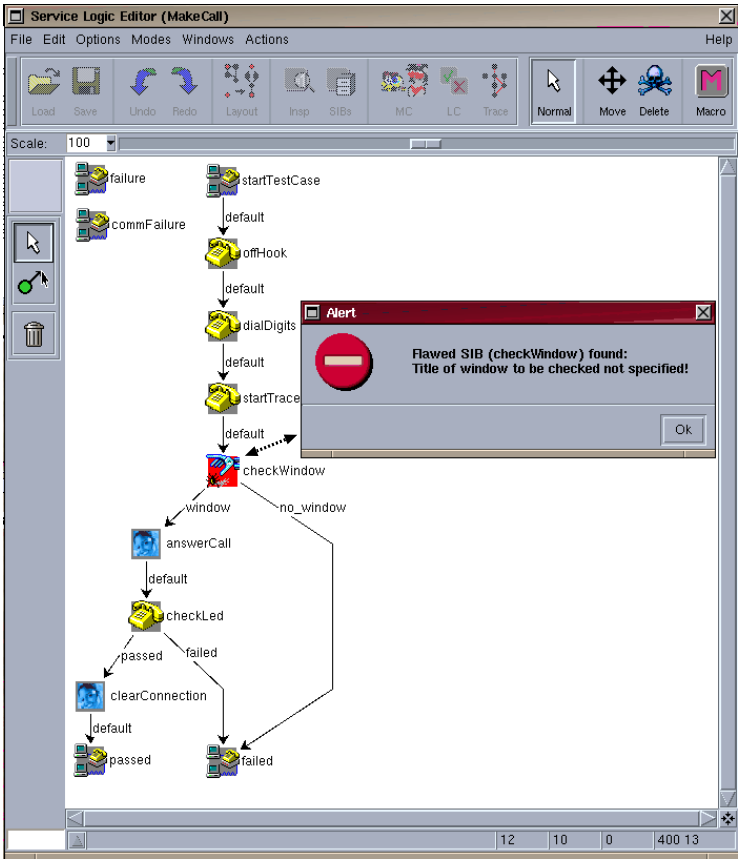


Fig. 2. Test Case Checking in the ITE Environment

are active components: they may stimulate the switch (e.g. initiate calls), and they react to stimuli sent by the switch (e.g. notify incoming calls). In a system level test it is therefore necessary to investigate the interaction between such subsystems. Typically, each participating subsystem requires an individual test tool. Thus in order to test systems composed of several independent subsystems that communicate with each other, one must be able to coordinate a heterogeneous set of test tools in a context of heterogeneous platforms. This task exceeds the capabilities of today's commercial test management tools, which typically cover the needs of specific subsystems and of their immediate periphery.

The effort for instantiating the ABC as required for the CTI application consists of designing some application-specific building blocks, and formulating frame conditions which must be enforced during test case design and test suite design. The building blocks are used by test designers to graphically construct test cases by drag-and-drop on the Test Coordinator canvas. The resulting test

graphs are directly executable for test purposes, and, at the same time, they constitute the models for our verification machinery by means of model checking. Figure 2 shows a typical test graph for illustration.

We distinguish classes of constraints according to the application domains: it depends on the test purpose, which constraints are bound to the test case.

Legal Test Cases define the characteristics of a correct test case, independently of any particular system under test and test purpose. Specifically, testing implies an evaluation of the runs wrt. expected observations done through *verdicts*, represented through the predicates passed and failed. For example, to enable an automated evaluation of results, verdict points should be disposed in a *nonambiguous* and *noncontradictory way* along each path, which is expressed as follows:

$$(\mathsf{passed} \ \vee \mathsf{failed}) \Rightarrow \ \mathbf{next}(\mathbf{generally} \ \neg(\mathsf{passed} \ \vee \mathsf{failed} \ ))$$

POTS Test Cases define the characteristics of correct functioning of Plain Old Telephone Services (POTS), which build the basis of any CTI application behaviour. Specific constraints of this class concern the different signalling and communication channels of a modern phone with an end user: signalling via tones, messaging via display, optic signalling via LEDs, vibration alarm. They must e.g. all convey correct and consistent information.

System Under Test-Specific Test Cases define the correct initialization and functioning of the single units of the system under test (e.g. single CTI applications, or the switch), of the corresponding test tools, and of their interplay.

Our evolutionary approach turned out to be a central feature here: by discovery of a mismatch users strengthened the model and the constraints.

4 A Personalized E-Commerce Shop

In [3] we presented a component-based approach to internet service construction based on the ABC that supports a *user friendly, flexible*, and *reliable* development of personalizable and self-adapting online services. Personalizing modification, extension, and global adaption of the service potential are based on a usage analysis and/or on new requirements by the users.

Our servlet-based system environment enforces a strict separation between data (the content) and control (the service functionality). Together with an integrated caching mechanism and an offline execution of the adaption, this guarantees a fast response time also in the presence of high system load. These features, which are currently being commercially used as development platform for online shops and workflow management services, are here illustrated along the example of the METACatalogue Online Kiosk[1], an online magazine shop, whose customers are individually supported via personalized offers [3]. Fig. 3 shows the initial portion of the Service Logic Graph and a snapshot of the online service.

[1] The METACatalogue Service was presented at the CeBIT in Hannover.

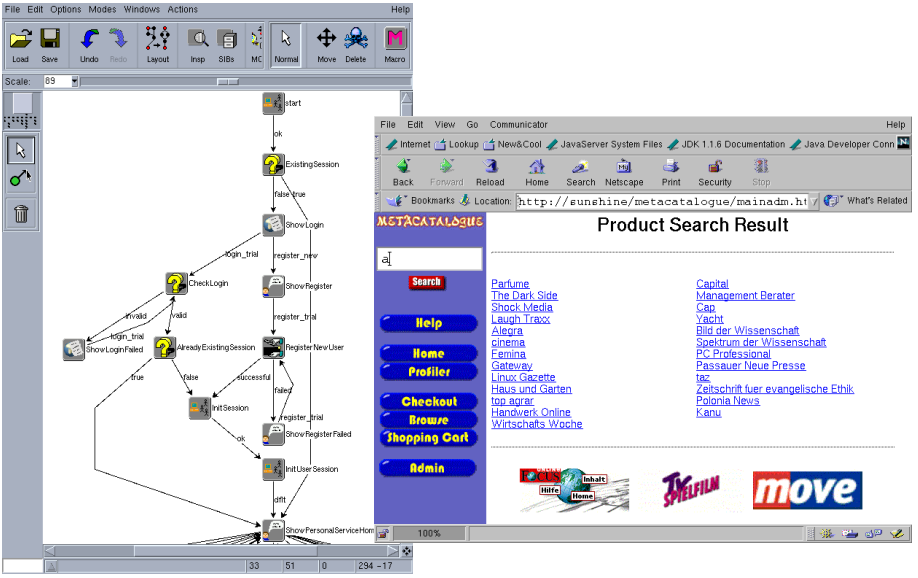


Fig. 3. Personalized Offers in the Magazine Search (r) and Service Logic Graph(l)

A (personalizable) *internet service* organizes therefore the complete potential of information and of service that a service provider wishes to make available to the users over the World Wide Web, and this provides each user with a person-specifically tailored selection of content and support. The navigation through this selection is supported via a *personal agent* that adapts itself to the particular user's behaviour. This adaptation process is steered on the one hand via information provided explicitely by the user, and on the other hand implicitly, through observation (behaviour tracking).

Personalizability and administrability can well be designed and realized together, and best with the help of a development environment for the efficient and structured internet organization offering both

- a largely programming-free, building block oriented, and high-level service development, and
- transparent adaption mechanisms and their steering for a controlled evolution (the learning process) of agent personalization.

This way, not only the concrete content, but also the service workflow can be structured, organized, and personalized.

The basic service logic components of the internet service application domain offer navigation, search, and browsing facilities, and menu-driven selection of content categories and personal profile. On top of that, the individualization functionality includes tracking, profile update, profile dependent advertisement

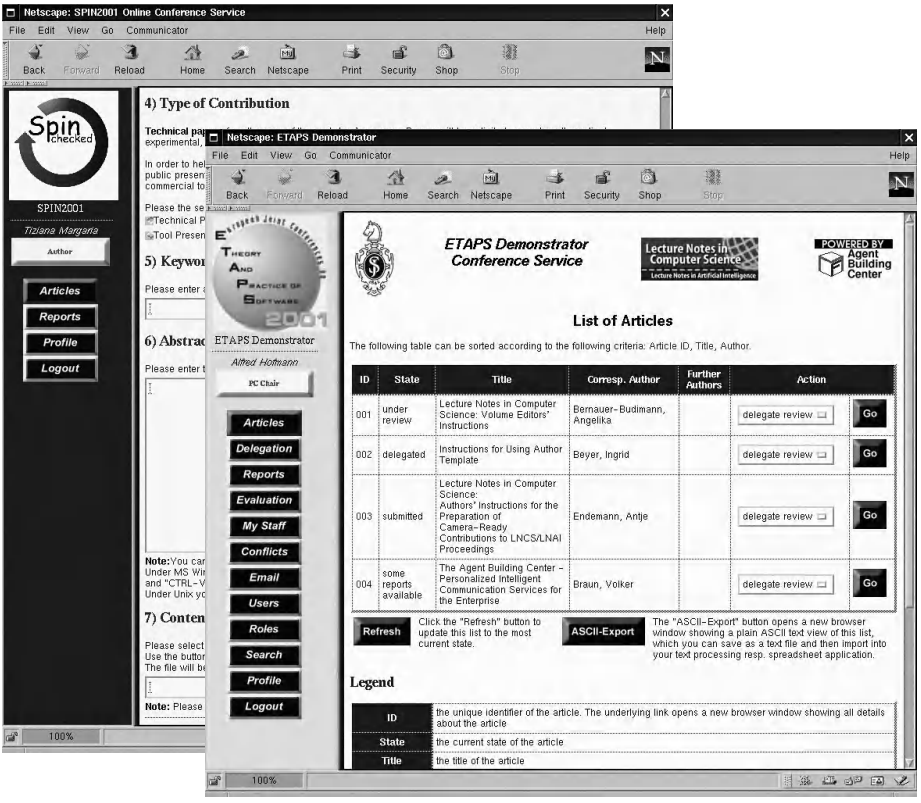


Fig. 4. The SPIN and LNCS Online Conference Services

and offer selection. The administration portion of the service also offers rule administration, including rule parsers and an individualization interface. Of central importance for a shop application is the shopping cart management, which gives rise to several constraints of the kind "*Shopping cart items do not get lost*", expressed as

$$\mathsf{add\text{-}item} \;\Rightarrow\; \neg\mathsf{exit}\ \mathbf{until}\ (\mathsf{pay} \vee \mathsf{empty\text{-}cart})$$

5 A Role-Based Editorial System

This online service customizes a strongly workflow oriented application built with the ABC. It proactively helps authors, Program Committee chairs, Program Committee members, and reviewers to cooperate efficiently during their collaborative handling of the composition of a conference program.

The service provides a timely, transparent, and secure handling of the papers and of the related submission, review, report and decision management tasks.

Several security and confidentiality precautions have been taken, in order to ensure proper handling of privacy and intellectual property sensitive information. In particular,

- the service can be accessed only by registered users,
- users can freely register only for the role Author,
- the roles Reviewer, PC Member, PC Chair are sensitive, and conferred to users by the administrator only,
- users in sensitive roles are granted well-defined access rights to paper information,
- users in sensitive roles agree to treat all data they access within the service as confidential.

These policies inherently define a loose specification of the service at the service logic level, and can be directly formulated in our model checking logic. For example, the access control policy is a primary source of constraints like "*A user can modify the defined roles only after having successfully registered as Administrator*", expressed as

$$\neg(\text{modify-roles})\ \textbf{unless}\ \ \text{user-login}\ [\text{Role=Admin}]$$

The service has been successfully used for several beta-test conferences, including this SPIN Workshop (see Fig. 4), TACAS 2001, and CHARME 2001. The product version, customized for the LNCS proceedings serie, is going to be launched by Springer Verlag in Autumn 2001.

6 Conclusions

We have illustrated the power of 'classical' iterative model checking for verifying quality of service requirements at the *coarse* granular level on the basis of three rather different application scenarios. In all these scenarios the *library-based modelling* of applications in terms of so-called service logic graphs, the *incremental formalization* of the underlying application domain in terms of temporal quality of service constraints, and the *library-based consistency checking* allowing continuous verification of application- and purpose-specific properties by means of model checking, guaranteed a new level of flexibility without sacrificing reliability. We are convinced that this will also lead to a new level of efficiency, a fact, which can only (and will be) be proved in everyday's practice.

References

1. O. Niese, B. Steffen, T. Margaria, A. Hagerer, G. Brune, H.-D. Ide: *Library-based Design and Consistency Checking of System-level Industrial Test Cases*, Proc. FASE 2001, Genova (Italy), April 2001, LNCS 2029, Springer Verlag, 2001.
2. B. Steffen, T. Margaria: *MetaFrame in Practice: Design of Intelligent Network Services*, Lecture Notes in Computer Science 1710, pp. 390-416, 1999.
3. B. Steffen, T. Margaria, V. Braun: *Personalized Electronic Commerce Services*, IFIP WG 7.3 8th Int. Conference on Telecommunication Systems Modeling and Analysis, March 9-12, 2000, Nashville, Tennessee, USA.

Author Index