FI MU

Faculty of Informatics
Masaryk University

TOOLS DAY

Affiliated to CONCUR 2002

Proceedings

by

Ivana Černá
(Ed.)

FI MU Report Series **FIMU-RS-2002-05**

 August 2002

Foreword

These are proceedings of the *Tools Day 2002* held in Brno, Czech Republic, on August 24, 2002, as a satellite event of CONCUR'02, the 13th International Conference on Concurrency Theory.

Formal verification provides an elegant approach to validating the correctness of software and hardware systems behaviour. Verification tools have enjoyed a substantial and growing use over the last few years, showing ability to discover subtle flaws. While until recently these tools were viewed as of academic interest only, they are now routinely used in industrial applications. The aim of the Tools Day was not only to present recent development in the area, but also to give tools developers and users the opportunity to discuss future trends and needs.

Eight regular papers, together with keynote presentations by Ziyad Hanna and Kim G. Larsen, form the content of these proceedings.

I would like to thank the organizing committee members for their support in composing the Tools Day, and the CONCUR'02 Organizing Committee for arranging all local affairs.

Brno, August 2002 — Ivana Černá

Contents

Invited talks

Tools Presentations

Formal Verification Framework and Applications for the Massive Usage at Intel

Ziyad Hanna

Intel, Israel

Abstract. The continuous development and innovations of formal verification technologies during the last decade; put lots of hope and significantly raised the confidence level of the Industrial design teams to apply formal methods for improving the overall chip design verification cycle. In particular, we at Intel have been developing formal verification technologies and tools and employing them for the benefits of the majority of Pentium and Itanium projects. In this talk, I will present our experience in this domain, the usage model, overview our Intel Formal verification framework (FORTE), and discuss the major applications (FEV, FPV) for verifying the design correctness and design implementation. During the talk, I will convey the main challenges and the future of formal verification that we see at Intel.

Uppaal Implementation Secrets

Gerd Behrmann[2] Johan Bengtsson[1] Alexandre David[1] Kim G. Larsen[2]
Paul Pettersson[1] Wang Yi[1]

[1] Department of Information Technology, Uppsala University, Sweden,
[johanb,adavid,paupet,yi]@docs.uu.se.
[2] Basic Research in Computer Science, Aalborg University, Denmark,
[behrmann,kgl]@cs.auc.dk.

Abstract In this paper we present the continuous and on-going development of datastructures and algorithms underlying the verification engine of the tool Uppaal. In particular, we review the datastructures of Difference Bounded Matrices, Minimal Constraint Representation and Clock Difference Diagrams used in symbolic state-space representation and -analysis for real-time systems.
In addition we report on distributed versions of the tool, and outline the design and experimental results for new internal datastructures to be used in the next generation of Uppaal.
Finally, we mention work on complementing methods involving acceleration, abstraction and compositionality.

1 Introduction

Uppaal [LPY97] is a tool for modeling, simulation and verification of real-time systems, developed jointly by BRICS at Aalborg University and the Department of Computer Systems at Uppsala University. The tool is appropriate for systems that can be modeled as a collection of non-deterministic processes with finite control structure and real-valued clocks, communicating through channels or shared variables. Typical application areas include real-time controllers and communication protocols.

Since the first release of Uppaal in 1995, the tool has been under constant development by the teams in Aalborg and Uppsala. The tool has consistently gained in performance over the years, which may be ascribed both to the development of new datastructures and algorithms as well as constant optimizations of their actual implementations. By now (and since long) Uppaal has reached a state, where it is mature for application on real industrial development of real-time systems as witnessed by a number of already carried out case-studies[1].

Tables 1 and 2 show the variations of time and space consumption for three different versions of Uppaal applied to five examples from the literature: Fischer's mutual exclusion protocol with five processes [Lam87], Philips audio-control protocol with bus-collision detection [BGK$^+$96], a Power-Down Controller [HLS99], a TDMA start-up algorithm with three nodes [LP97], and a

[1] See www.uppaal.com for detailed list.

	1998	2000	DBM	Min	Ctrl	Act	PWL	State	2002
Fischer 5	126.30	13.50	4.79	6.02	3.98	2.13	3.83	12.66	0.19
Audio	-	2.23	1.50	1.79	1.45	0.50	1.57	2.28	0.45
Power Down	*	407.82	207.76	233.63	217.62	53.00	125.25	364.87	13.26
Collision Detection	128.64	17.40	7.75	8.50	7.43	7.94	7.04	19.16	6.92
TDMA	108.70	14.36	9.15	9.84	9.38	6.01	9.33	16.96	6.01

Table1. Time requirements (in seconds) for three different UPPAAL versions.

CSMA/CD protocol with eight nodes [BDM+98]. In the column "1998" and "2000" we give the run-time data of UPPAAL versions dated January 1998 and January 2000 respectively. In addition, we report the data of the current version dated June 2002. The numbers in column "DBM" were measured without any optimisations, "Min" with Minimal Constraints Representation, "Ctrl" with Control Structure Reduction [LPY95], "Act" with Active Clock Reduction [DT98], "PWL" with the Passed and Waiting List Unification, "State" with Compact Representation of States, and finally "2002" with the best combination of options available in the current version of UPPAAL. The different versions have been compiled with a recent version of `gcc` and were run on the same Sun Enterprise 450 computer equipped with four 400 MHz processors and 4 Gb or physical memory. In the diagrams we use "-" to indicate that the input model was not accepted due to compability issues, and "*" to indicate that the verification did not terminate within one hour. We notice that both the time and space performance has improved significantly over the years. For the previous period December 1996 to September 1998 a report on the run-time and space improvements may be found in [Pet99]. Similar diagrams for the time period November 1998 to Januari 2001 are reported in [ABB+01].

Despite this success improvement in performance, the state-explosion problem is a still a reality[2] which prevents the tool from ever[3] being able to provide fully automatic verification of arbitrarily large and complex systems. Thus, to truely scale up, automatic verification should be complemented by other methods. Such methods investigated in the context of UPPAAL include that of *acceleration* [HL02] and *abstractions* and *compositionality* [JLS00].

The outline of the paper is as follows: Section 2 summaries the definition of timed automata, the semantics, and the basic timed automaton reachability algorithm. In section 3 we present the three main symbolic datastructures applied in UPPAAL: Difference Bounded Matrices, Minimal Constraint Representation and Clock Difference Diagrams and in section 4 we review various schemes for compact representations for symbolic states. Section 5 introduces a new exloration algorithm based on a unification of Passed and Waiting list datastructures and Section 6 reviews our considerable effort in parallel and distributed reach-

[2] Model-checking is either EXPTIME- or PSPACE-complete depending on the expressiveness of the logic considered.
[3] unless we succeed in showing P=PSPACE

	1998	2000	DBM	Min	Ctrl	Act	PWL	State	2002
Fischer 5	8.86	8.14	9.72	6.97	6.40	6.35	6.74	4.83	3.21
Audio	-	3.02	5.58	5.53	5.58	4.33	4.75	3.06	3.06
Power Down	*	218.90	162.18	161.17	132.75	44.32	18.58	117.73	8.99
Collision Detection	17.00	12.78	25.75	21.94	25.75	25.75	10.38	13.70	10.38
TDMA	8.42	8.00	11.29	8.09	11.29	11.29	4.82	6.58	4.82

Table2. Space requirements (in Mb) of for different UPPAAL versions.

ability checking. Section 7 presents recent work on acceleration techniques and section 8 reviews work on abstraction and compositionality. Finally, we conclude by stating what we consider open problems for future research.

2 Preliminaries

In this section we summaries the basic definition of timed automata, their concrete and symbolic semantics and the reachability algorithm underlying the currently distributed version of UPPAAL.

Definition 1 (Timed Automaton). *Let C be the set of clocks. Let $B(C)$ be the set of conjunctions over simple conditions on the forms $x \bowtie c$ and $x - y \bowtie c$, where $x, y \in C$, $\bowtie \in \{<, \leq, =, \geq, >\}$ and c is a natural number. A timed automaton over C is a tuple (L, l_0, E, g, r, I), where L is a set of locations, $l_0 \in L$ is the initial location, $E \in L \times L$ is a set of edges, $g : E \to B(C)$ assigns guards to edges, $r : E \to 2^C$ assigns clocks to be reset to edges, and $I : L \to B(C)$ assigns invariants to locations.*

Intuitively, a timed automaton is a graph annotated with conditions and resets of non-negative real valued clocks.

Definition 2 (TA Semantics). *A clock valuation is a function $u : C \to \mathbb{R}_{\geq 0}$ from the set of clocks to the non-negative reals. Let $\mathbb{R}^C$ be the set of all clock valuations. Let $u_0(x) = 0$ for all $x \in C$. We will abuse the notation by considering guards and invariants as sets of clock valuations.*

The semantics of a timed automaton (L, l_0, E, g, r, I) over C is defined as a transition system $(S, s_0, \to)$, where $S = L \times \mathbb{R}^C$ is the set of states, $s_0 = (l_0, u_0)$ is the initial state, and $\to \subseteq S \times S$ is the transition relation such that:

- $(l, u) \to (l, u + d)$ *if* $u \in I(l)$ *and* $u + d \in I(l)$
- $(l, u) \to (l', u')$ *if there exists* $e = (l, l') \in E$ *s.t.* $u \in g(e)$, $u' = [r(e) \mapsto 0]u$, *and* $u' \in I(l)$

where for $d \in \mathbb{R}$, $u + d$ maps each clock x in C to the value $u(x) + d$, and $[r \mapsto 0]u$ denotes the clock valuation which maps each clock in r to the value 0 and agrees with u over $C \setminus r$.

The semantics of timed automata results in an uncountable transition system. It is a well known-fact that there exists a exact finite state abstraction based on convex polyhedra in $\mathbb{R}^C$ called zones (a zone can be represented by a conjunction in $B(C)$). This abstraction leads to the following symbolic semantics.

Definition 3 (Symbolic TA Semantics). *Let $Z_0 = \bigwedge_{x \in C} x \geq 0$ be the initial zone. The symbolic semantics of a timed automaton (L, l_0, E, g, r, I) over C is defined as a transition system $(S, s_0, \Rightarrow)$ called the* simulation graph, *where $S = L \times B(C)$ is the set of symbolic states, $s_0 = (l_0, Z_0 \wedge I(l_0))$ is the initial state, $\Rightarrow = \{(s, u) \in S \times S \mid \exists e, t : s \overset{e}{\Rightarrow} t \overset{\delta}{\Rightarrow} u\}$: is the transition relation, and:*

– $(l, Z) \overset{\delta}{\Rightarrow} (l, norm(M, (Z \wedge I(l))^{\uparrow} \wedge I(l)))$
– $(l, Z) \overset{e}{\Rightarrow} (l', r_e(g(e) \wedge Z \wedge I(l)) \wedge I(l'))$ *if* $e = (l, l') \in E$.

where $Z^{\uparrow} = \{u + d \mid u \in Z \wedge d \in \mathbb{R}_{\geq 0}\}$ (the future *operation), and $r_e(Z) = \{[r(e) \mapsto 0]u \mid u \in Z\}$ (the* reset *operation). The function $norm : \mathbf{N} \times B(C) \to B(C)$ normalises the clock constraints with respect to the maximum constant M of the timed automaton.*

The relation $\overset{\delta}{\Rightarrow}$ contains the delay transitions and $\overset{e}{\Rightarrow}$ the edge transitions. Given the symbolic semantics it is straight forward to construct the reachability algorithm, shown in Figure 1. The symbolic semantics can be extended to cover networks of communicating timed automata (resulting in a location vector to be used instead of a location), timed automata with data variables (resulting in the addition of a variable vector).

3 Symbolic Datastructures

To utilize the above symbolic semantics algorithmically, as for example in the reachability algorithm of Figure 1, it is important to design efficient data structures and algorithms for the representation and manipulation of clock constraints. In this section, we present three such datastructures: Diffence Bounded Matrices, Minimal Constraint Representation and Clock Difference Diagrams.

Difference Bounded Matrices

Difference Bounded Matrices (DBM, see [Bel57,Dil89]) is well–known data structure which offers a canonical representation for constraint systems. A DBM representation of a constraint system Z is simply a weighted, directed graph, where the vertices correspond to the clocks of C and an additional zero–vertex 0. The graph has an edge from x to y with weight m provided $x - y \leq m$ is a constraint of Z. Similarly, there is an edge from 0 to x (from x to 0) with weight m, whenever $x \leq m$ ($x \geq -m$) is a constraint of Z[4]. As an example, consider the constraint system E over $\{x_0, x_1, x_2, x_3\}$ being a conjunction of the atomic constraints $x_0 - x_1 \leq 3$, $x_3 - x_0 \leq 5$, $x_3 - x_1 \leq 2$, $x_2 - x_3 \leq 2$, $x_2 - x_1 \leq 10$, and $x_1 - x_2 \leq -4$. The graph representing E is given in Figure 2 (a).

[4] We assume that Z has been simplified to contain at most one upper and lower bound for each clock and clock–difference.

```
W = {(l_0, Z_0 ∧ I(l_0))}
P = ∅
while W ≠ ∅ do
  (l, Z) = W.popstate()
  if testProperty(l, Z) then return true
  if ∀(l, Y) ∈ P : Z ⊄ Y then
    P = P ∪ {(l, Z)}
    ∀(l', Z') : (l, Z) ⇒ (l', Z') do
      if ∀(l', Y') ∈ W : Z' ⊄ Y' then
        W = W ∪ {(l', Z')}
      endif
    done
  endif
done
return false
```

Figure1. The timed automaton reachability algorithm, with P being the passed-list containing all explored symbolic states, and W being the waiting-list containing encountered symbolic states waiting to be explored. The function $testProperty$ evaluates the state property that is being checked for satisfiability. The while loop is refered to as the exploration loop.

In general, the same set of clock assignments may be described by several constraint systems (and hence graphs). To test for inclusion between constraint systems Z and Z'[5], which we recall is essential for the termination of the reachability algorithm of Figure 1, it is advantageous, that Z is *closed under entailment* in the sense that no constraint of Z can be strengthened without reducing the solution set. In particular, for Z a closed constraint system, $Z \subseteq Z'$ holds if and only if for any constraint in Z' there is a constraint in Z at least as tight; i.e. whenever $(x - y \leq m) \in Z'$ then $(x - y \leq m') \in Z$ for some $m' \leq m$. Thus, closedness provides a canonical representation, as two closed constraint systems describe the same solution set precisely when they are identical. To close a constraint system Z simply amounts to derive the shortest–path closure for its graph and can thus be computed in time $\mathcal{O}(n^3)$, where n is the number of clocks of Z . The graph representation of the closure of the constraint system E from Figure 2 (a) is given in Figure 2 (b). The emptiness-check of a constraint system Z simply amounts to checking for negative–weight cycles in its graph representation. Finally, given a closed constraint system Z the operations $Z^{\uparrow}$ and $r(Z)$ may be performed in time $\mathcal{O}(n)$. For more detailed information on how to efficiently implement these and other operations on DBM's we refer the reader to [Ben02,Rok93].

[5] To be precise, it is the inclusion between the *solution sets* for Z and Z'.

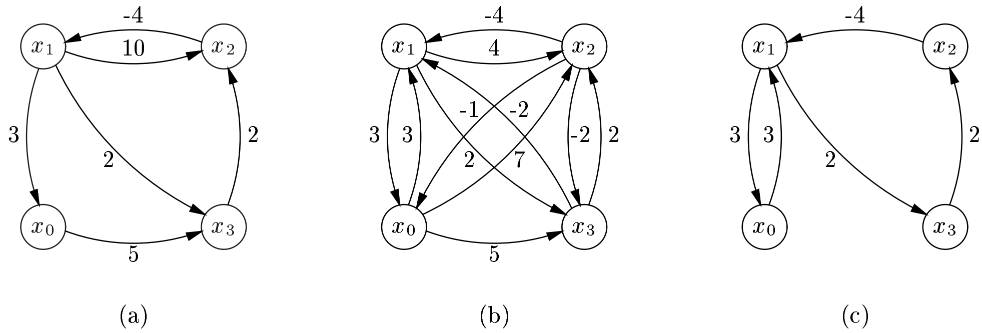


Figure2. Graph for E (a), its shortest–path closure (b), and shortest–path reduction (c).

Minimal Constraint Representation

For the reasons stated above a matrix representation of constraint systems in closed form is an attractive data structure, which has been successfully employed by a number of real–time verification tools, e.g. UPPAAL [BLL+96] and KRONOS [DY95]. As it gives an explicit (tightest) bound for the difference between each pair of clocks (and each individual clock), its space–usage is of the order $\mathcal{O}(n^2)$. However, in practice it often turns out that most of these bounds are redundant, and the reachability algorithm of Figure 1 is consequently hampered in two ways by this representation. Firstly, the main–data structure P (the passed list) will in many cases store all the reachable symbolic states of the automaton. Thus, it is desirable, that when saving a symbolic state in the passed list, we save a representation of the constraint–system with as few constraints as possible. Secondly, a constraint system Z added to the passedlist is subsequently only used in checking inclusions of the form $Z' \subseteq Z$. Recalling the method for inclusion–check from the previous section, we note that (given Z' is closed) the time–complexity of the inclusion–check is linear in the number of constraints of Z. Thus, again it is advantageous for Z to have as few constraints as possible.

In [LLPY97,LLPY02] we have presented an $\mathcal{O}(n^3)$ algorithm, which given a constraint system constructs an equivalent reduced system with the minimal number of constraints. The reduced constraint system is canonical in the sense that two constrain systems with the same solution set give rise to identical reduced systems. The algorithm is essentially a minimization algorithm for weighted directed graphs. Given a weighted, directed graph with n vertices, it constructs in time $\mathcal{O}(n^3)$ a reduced graph with the minimal number of edges having the same shortest path closure as the original graph. Figure 2 (c) shows the minimal graph of the graphs in Figure 2 (a) and (b), which is computed by the algorithm.

The key to reduce a graph is obviously to remove *redundant edges*, i.e. edges for which there exist alternative paths whose (accumulated) weight does not exceed the weight of the edgesthemselves. E.g. in the graph of Figure 2 (a) the edge (x_1, x_2) is clearly redundant as the accumulated weight of the path $(x_1, x_3, (x_3, x_2)$ has a weight (4) not exceeding the weight of the edge itself (10). Being redundant, the edge (x_1, x_2) may be removed without changing the shortest-path closure (and hence the solution-set of the corresponding constraint system). In this manner both the edges (x_1, x_2) and (x_2, x_3) of Figure 2 (b) are found to be redundant. However, thought redundant, we cannot just remove the two edges as removal of one clearly requires the presence of the other. In fact, all edges between the vertices x_1, x_2 and x_3 are redundant, but obviously we cannot remove them all simultaneously without affecting the solution-set. The key explanation of this phenomena is that x_1, x_2 and x_3 constitute a zero-cycle. In fact, for zero-cycle free graphs simulataneous removal of redundant edges leads to a canonical shortest-path reduction form. For general graphs the reduction is based on a partitioning of the vertices according to membership of zero-cycles.

Our experimental results demonstrated significant space-reductions compared with traditional DBMimplmentation: on a number of benchmark and industrial examples the space saving was between 75% and 94%. Additionally, time-performance was improved.

Clock Difference Diagrams

Difference Bound Matrices (DBM's) as the standard representation for time zones in analysis of Timed Automata have a well-known shortcoming: they are not closed under set-union. This comes from the fact that a set represented by a DBM is convex, while the union of two convex sets is not necessarily convex.

Within the symbolic computation for the reachability analysis of UPPAAL, set-union however is a crucial operation which occurs in every symbolic step. The shortcoming of DBM's leads to a situation, where symbolic states which could be treated as one in theory have to be handled as a collection of several different symbolic states in practice. This leads to trade-offs in memory and time consumption, as more symbolic states have to be stored and visited during in the algorithm.

DBM's represent a zone as a conjunction of constraints on the differences between each pair of clocks of the timed automata (including a fictitious clock representing the value 0). The major idea of CDD's (Clock Difference Diagrams) is to store a zone as a decision tree of clock differences, generalizing the ideas of BDD's (Binary Decision Diagrams, see [Bry86]) and IDD's (Integer Decision Diagrams, see [ST98])

The nodes of the decision tree represent clock differences. Nodes on the same level of the tree represent the same clock difference. The order of the clock differences is fixed a-priori, all CDD's have to agree on the same ordering. The leaves of the decision tree are two nodes representing true and false, as in the case of BDD's.

Each node can have several outgoing edges. Edges are labeled with integral intervals: open, half-closed and closed intervals with integer values as the borders. A node representing the clock difference $X - Y$ together with an outgoing edge

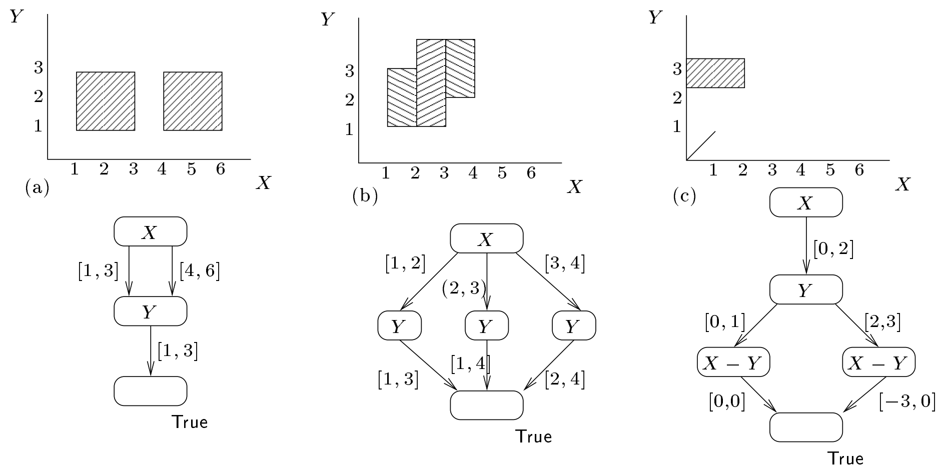


Figure3. Three example CDD's. Intervals not shown lead implicitly to False.

with interval I represents the constraint "$X - Y$ within I". The leafs represent the global constraints true and false respectively.

A path in a CDD from a node down to a leaf represents the set of clock values with fulfill the conjunction of constraints found along the path. Remember that a constraint is found from the pair node and outgoing edge. Paths going to false thus always represent the empty set, and thus only paths leading to the true node need to be stored in the CDD. A CDD itself represents the set given by the union of all sets represented by the paths going from the root to the true node. From this clearly CDD's are closed under set-union. Figure 3 gives three examples of two-dimensional zones and their representation as CDDs. Note that the same zone can have different CDD representations.

All operations on DBM's can be lifted straightforward to CDD's. Care has to be taken when the canonical form of the DBM is involved in the operation, as there is no direct equivalent to the (unique) canonical form of DBM's for CDD's.

CDD's generalize IDD's, where the nodes represent clock values instead of clock differences. As clock differences, in contrast to clock values, are not independent of each other, operations on CDD's are much more elaborated than the same operations on IDD's. CDD's can be implemented space-efficient by using the standard BDD's technique of sharing common substructure. This sharing can also take place between different CDD's.

Experimental results have shown that using CDD's instead of DBM's can lead to space savings of up to 99%. However, in some cases a moderate increase in run time (up to 20%) has to be paid. This comes from the fact that operations involving the canonical form are much more complicated in the case of CDD's compared to DBM's. More on CDD's can be found in [LWYP99] and [BLP$^+$99]. A similar datastructure is that of DDD's presented in [MLAH99a,MLAH99b].

4 Compact Representation of States

Symbolic states are the core objects of state space search and one of the key issues in implementing a verifier is how to represent them. In the earlier versions of Uppaal each entity in a state (i.e. an element in the location vector, the value of an integer variable or a bound in the DBM) is mapped on a machine word. The reason for this is simplicity and speed. However the number of possible values for each entity is usually small, and using a machine word for each of them is often a waste of space.

To conquer this problem two additional, more compact, state representations have been added. In both of them the discrete part of each state is encoded as a number, using a multiply and add scheme. This encoding is much like looking at the discrete part as a number, where each digit is an entity in the discrete state and the base varies with the number of different digits.

In the first packing scheme, the DBM is encoded using the same technique as the discrete part of the state. This gives a very space efficient but computationally expensive representation, where each state takes a minimum amount of memory but where a number of bignum division operations have to be performed to check inclusion between two DBMs.

In the second packing scheme, some of the space performance is sacrificed to allow a more efficient inclusion check. Here each bound in the DBM is encoded as a bit string long enough to represent all the possible values of this bound plus one *test bit*, i.e. if a bound can have 10 possible values then five bits are used to represent the bound. This allows cheap inclusion checking based on ideas of Paul and Simon [PS80] on comparing vectors using subtraction of long bit strings.

In experiments we have seen that the space performance of these representations are both substantially better than the traditional representation, with space savings of between 25% and 70%. As we expect, the performance of the first packing scheme, with an expensive inclusion check, is somewhat better, space-wise, than the packing scheme with the cheap inclusion check.

Considering the time performance for the packed state representations we have found that the price for using the encoding with expensive inclusion check is a slowdown of 2 – 12 times, while using the other encoding sometimes is even faster than the traditional representation. For more detailed information on this we refer the interested reader to [Ben02].

5 Passed and Waiting List Unification

The standard reachability algorithm currently applied in Uppaal is based on two lists: the passed and the waiting lists. These lists are used in the exploration loop that pops states to be explored from the waiting list, explores them, and keeps track of already explored states with the passed list. The first algorithm of Figure 4 shows this algorithm based on two distinct lists.

We have unified these structures to a *PWList* and a queue. The queue has only references to states in PWList and is a trivial queue structure: it stores nothing by itself. The PWList acts semantically as a buffer that eliminates duplicate states, i.e. if the same state is added to the buffer several times it can only be retrieved once, even when the state was retrieved before the state is inserted a second time. To achieve this effect the PWList must keep a record of the states seen and thus it provides the functionality of both the passed list and the waiting list.

Definition 4 (PWList). *Formally, a PWList can be described as a pair* $(P, W) \in 2^S \times 2^S$*, where* S *is the set of symbolic states, and the two functions* $put : 2^S \times 2^S \times S \to 2^S \times 2^S$ *and* $get : 2^S \times 2^S \to 2^S \times 2^S \times S$*, such that:*

- $put(P, W, (l, Z)) = (P \cup \{(l, Z)\}, W')$ *where*

$$W' = \begin{cases} W \cup \{(l, Z)\} & \textit{if } (l, Z) \notin P \\ W & \textit{otherwise} \end{cases}$$

- $get(P, W) = (P, W \setminus \{(l, Z)\}, (l, Z))$ *for some* $(l, Z) \in W$.

Here P and W play the role of the passed list and waiting list, respectively, but as we will see this definition provides room for alternative implementations. It is possible to loosen the elimination requirement such that some states can be returned several times while still ensuring termination, thus reducing the memory requirements [LLPY97].

The reachability algorithm can then be simplified as shows in Figure 4. The main difference with the former algorithm shows when a state is pushed to PWList: it is pushed conceptually to the passed and the waiting lists at the same time. States to be explored are considered already explored for the inclusion checking of new generated states. This greedy behaviour improves performance.

The reference implementation uses a hash table based on the discrete part of the states to find them. Every state entry has its symbolic part represented as a zone union (single linked list of zones). The queue is a simple linked list with references to the discrete and symbolic parts. Only one hash computation and one inclusion checking are necessary for every state inserted into this structure, compared to two with the former passed and waiting lists. Furthermore we gather states with a common discrete part. The former representation did not have this zone union structure. This zone union structure is particularly well-suited for other union representations of zones such as CDDs [BLP+99,LWYP99].

A number of options are realisable via different implementations of the PWList to approximate the representation of the state-space such as *bitstate hashing* [Hol87], or choose a particular order for state-space exploration such as *breadth first*, *depth first*, *best first* or *random* [BHV00,BFH+01]. The ordering is orthogonal to the storage structure and can be combined with any data representation.

This implementation is built on top of the storage structure that is in charge of storing raw data. The PWList uses *keys* as references to these data. This storage structure is orthogonal to a particular choice of data representation, in particular, algorithms aimed at reducing the memory footprint such as *convex hull approximation* [WT95] or *minimal constraint representation* [LLPY97] are

```
W = {(l0, Z0 ∧ I(l0))}
P = ∅
while W ≠ ∅ do
  (l, Z) = W.popState()
  if testProperty(l, Z)
  then return true
  if ∀(l, Y) ∈ P : Z ⊈ Y
  then
    P = P ∪ {(l, Z)}
    ∀(l′, Z′) : (l, Z) ⇒ (l′, Z′) do
      if ∀(l′, Y′) ∈ W : Z′ ⊈ Y′
      then
        W = W ∪ {(l′, Z′)}
      endif
    done
  endif
done
return false
```

→

```
Q = PW = {(l0, Z0 ∧ I(l0))}
while Q ≠ ∅ do
  (l, Z) = Q.popState()
  if testProperty(l, Z)
  then return true
  ∀(l′, Z′) : (l, Z) ⇒ (l′, Z′) do
    if ∀(l′, Y′) ∈ PW : Z′ ⊈ Y′
    then
      PW = PW ∪ {(l′, Z′)}
      Q.append(l′, Z′)
    endif
  done
done
return false
```

Figure4. Reachability algorithm with classical passed (P) and waiting (W) lists adapted to a the unified list (Q and PW).

possible implementations. We have implemented two variants of this storage, namely one with simple copy and the other one with data sharing.

Depending on the careful options given to Uppaal our new implementation has been experimentally show to give improvements of up to 80% in memory and improves speed significantly. The memory gain is expected due to the showed sharing property of data. The speed gain (in spite of the overheads) comes from only having a single hash table and from the zone union structure: the discrete test is done only once, then comes only inclusion checks on all the zones in one union. This is showed by the results of the simple copy version. For more information we refer the interested reader to [DBLY].

6 Parallel and Distributed Reachability Checking

Parallel and distributed reachability analysis has become quite popular during recent years. Most work is based on the same explicit state exploration algorithm: The state space is partitioned over a number of nodes using a hash function. Each node is responsible for storing and exploring those states assigned to it by the hash function. The successors of a state are transfered to the owning nodes according to the hash function. Given that all nodes agree on the hash function to use and that the hash function maps states uniformly to the nodes, this results in a very effective distributed algorithm where both memory and CPU usage are distributed uniformly among all nodes.

In [BHV00] we reported on a version of Uppaal using the variation in Figure 5 of the above algorithm on a parallel computer (thus providing efficient

$W_A = \{(l_0, Z_0 \wedge I(l_0)) \mid h(l_0) = A\}$
$P_A = \varnothing$
while $\neg terminated$ **do**
  $(l, Z) = W_A.popState()$
  if $\forall (l, Y) \in P_A : Z \not\subseteq Y$ **then**
    $P_A = P_A \cup \{(l, Z)\}$
    $\forall (l', Z') : (l, Z) \Rightarrow (l', Z')$ **do**
      $d = h(l', Z')$
      if $\forall (l', Y') \in W_d : Z' \not\subseteq Y'$ **then**
        $W_d = W_d \cup \{(l', Z')\}$
      endif
    done
  endif
done

Figure5. The distributed timed automaton reachability algorithm parameterised on node A. The waiting list W and the passed list P is partitioned over the nodes using a function h. States are popped of the local waiting list and added to the local passed list. Successors are mapped to a destination node d.

interprocess communication). The algorithm would only hash on the discrete part of a symbolic state such that states with the same discrete part would map to the same nodes, thus keeping the inclusion checking on the waiting list and passed list. Due to the symbolic nature of the reachability algorithm, the number of states explored depends on the search order. One noticeable side effect of the distribution was an altered search order which most of the time would increase the number of states explored. Replacing the waiting list with a priority queue always returning the state with the smallest distance to the initial state solved the problem.

More recently [Beh] we have ported the algorithm to a multi-threaded version and a version running on a Linux Beowulf Cluster using the new PWList structure. Surprisingly, initial experiments on the cluster showed severe load balancing problems, despite the fact that the hash function distributed states uniformly. The problem turned out to be that the exploration rate of each node depends on the load of the node [6] (due to the inclusion checking). Slight load variations will thus result in slight variations of the exploration rate of each node. A node with a high load will have a lower exploration rate, and thus the load rapidly becomes even higher. This is an unstable system. On the parallel machine used in [BHV00] this is not a problem for most input systems (probably due to the fast interprocess communication which reduces the load variations). Increasing the size of the hash table used for the waiting list and/or using the new PWList structure reduces this effect. Even with these modifications, some input systems cause load balancing problems, e.g. Fischer protocol for mutual exclusion. Most remaining load balancing problems can be eliminated by an ex-

[6] The load of a node is defined as the length of its waiting list.

plicit load balancing layer which uses a proportional controller that redirects states from nodes with a high load to nodes with a low load.

The multi-threaded version uses a different approach to ensure that all threads are equally balanced. All threads share the same PWList, or more precisely, the hash table underlying the PWList is shared but the list of states needed to be explored is thread local. Thus, if a thread inserts a state it will be retrieved by the same thread. With this approach we avoid that the threads need to access the same queue. Each bucket in the hash table is protected by a semaphore. If the hash table has much more buckets than we have threads, then the risk of multiple simultaneous accesses is low. By default, each thread keeps all successors on the same thread (since the hash table is shared it does not matter to which thread a state is mapped). When the system is unbalanced some states are redirected to other threads. Experiments show that this results in very high locality.

Experiments with the parallel version are very encouraging, showing excellent speedups (in the range of 80-100% of optimal on a 4 processor machine). The distributed version is implemented using MPI[7] over TCP/IP over Fast Ethernet. This results in high processing overhead of communication causing low speedups in the range of 50-60% of optimal at 14 nodes. Future work will focus on combining the two approaches such that nodes located on the same physical machine can share the PWList. Also, experiments with alternatives to MPI over TCP/IP will be evaluated, such as VIA.[8] Finally, it is unclear if the sharing of sub-elements of a state introduced in the previous section will scale to the distributed case.

7 Accelerating Cycles

An important problem concerning symbolic model checking of timed automata, is encountered when the timed automata in a model use different *time scales*. This, for example, is often the case for models of reactive programs with their environment. Typically, the automata that model the reactive programs are based on microseconds whereas the automata of the environment function in the order of seconds. This difference can give rise to an unnecessary fragmentation of the symbolic state space. As a result, the time and memory consumption of the model check process increases.

The fragmentation problem has already been encountered and described by Hune and Iversen et al during the verification of LEGO Mindstorms programs using UPPAAL [Hun00,IKL+00]. The symbolic state space is severely fragmented by the busy-waiting behaviour of the control program automata. Other examples were the phenomena of fragmantation is likely to show up include reactive programs, and polling real-time systems, e.g., programmable logic controllers [Die99]. The validation of communication protocols will probably also suffer

[7] The Message Passing Interface.
[8] The Virtual Interface Architecture.

from the fragmentation problem when the context of the protocol is taken into account.

In [HL02] we have proposed an acceleration technique for a subset of timed automata, namely those that contain special cycles, that addresses the fragmentation problem. The technique consists of a syntactical adjustment that can easily be computed from the timed automaton itself. It is proven that the syntactical adjusment is exact with repsect to reachability properties, and it is experimentally validated that the technique effectively speed-up the symbolic reachability analysis.

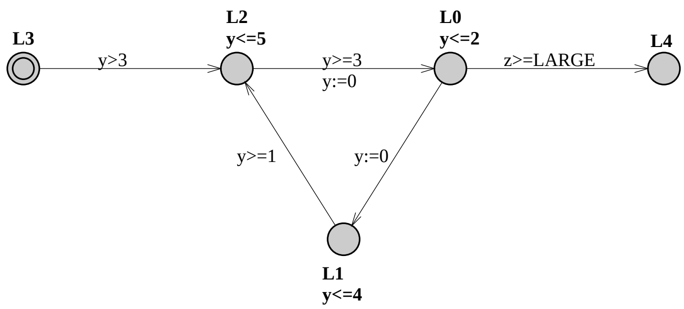


Figure6. Timed automaton P.

The timed automaton of figure 6 offers a simplified modeling of a control program combined with an environment. The cycle `L0`, `L1`, `L2` corresponds to cyclic execution of a control program consisting of three atomic instructions with the invariants and guards on the clock `y` providing execution time information. Whenever the control cycle is in location `L0`, the enviroment (modelled by the clock `z`) is consulted potentially leading to an exit of the control cycle. The size of the threshold constant `LARGE` determines how slow the environment is relative to the execution time of control program instructions: the larger the constant the slower. Depending on the value of `LARGE` the cycle in automaton P must be executed a certain (large) number of times before the edge to location `L4` is enabled. In a symbolic forward exploration the cycle must similarly be explored a large number of times with a fragmentation of the symbolic states involving location `L0` as a consequence.

The acceleration technique proposed in [HL02] eliminates the fragmentation that is due to special cycles. The subset of cycles we can accelerate may use only a single clock y in the invariants, guards and resets. Though this might seem like a strong restriction, this kind of cycles often occur in control graphs of single-processor polling real-time systems. To be acceleratable all ingoing edges to the first location of the cycle C should reset the clock y. This guarantees that C has a *window* $[a, b]$, in the sense that any execution of C has accumulated delay between a and b, and, conversely, for any delay d between a and b any execution of C can be 'adjusted' to have accumulated delay d. Now, the acceleration of such a cycle C is given by addition of a simple unfolding of C, where the invariant of the (copy of the) intial location is removed. Figure 7 illustrates the result of

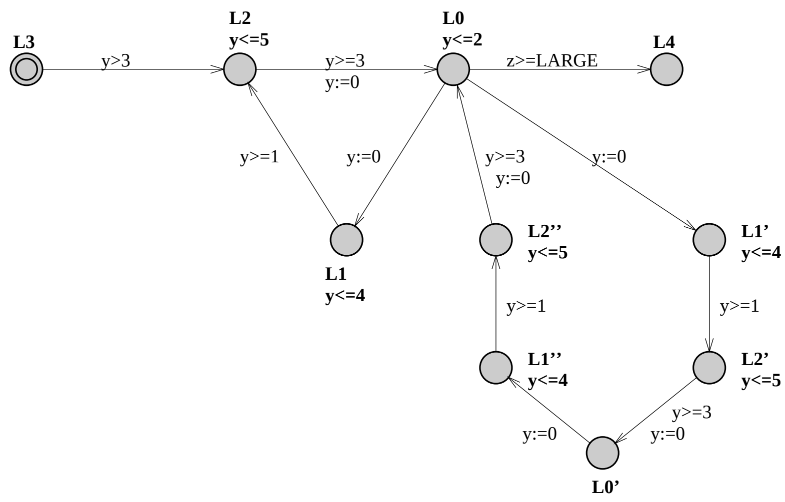


Figure7. The accelerated version of P.

adding the unfolded cycle to the model. Provided $3a \leq 2b$ it can be proved that in terms of rechability (of original locations) the two models are equivalent. Thus, the acceleration is *exact*. In case $(n+1)b \leq na$ a similar result holds provided the cycle is unfolded n times. If moreover the clock y is reset on the first edge of C, all reachable states may be obtained by a *single* execution of the unfolded cycle. Consequently, a symbolic breadth-first analysis of the accelerated version of P in Figure 7 experimentally proves to be insensitive to the value of `LARGE`.

In [HL02] and [Hen02] the proposed acceleration technique has been succesfully applied to analysis of models of LEGO Mindstorm byte code. In particular, the acceleration technique allowed UPPAAL to establish (at the byte code level) several properties of the Production Cell which could not otherwise be analysed.

8 Abstraction and Compositionality

Despite the vast improvement in performance of UPPAAL due to the development improved datastructures and algorithms, the state-explosion is a reality. Thus, in order for the application of a verification tools to truely scale up it is imperative that they are complemented by other methods.

One such method is that of *abstraction*. Assume that SYS is a model of some considered real-time system, and assume that we want some property φ to be established, i.e. SYS $\models \varphi$. Now, the model, SYS, may be too complex for our tools to settle this verification problem automatically (despite all of our algorithmic efforts). The goal of abstraction is to replace the problem with another, hopefully tractable problem ABS $\models \varphi$, where ABS is an abstraction of SYS being smaller in size and less complex. This method requires the user not only to supply the abstraction but also to argue that the abstraction is *safe* in the sense that all relevant properties established for ABS also hold for SYS; i.e. it should be established that SYS $\leq$ ABS, for some property-preserving relationship $\leq$ between models[9]. Unfortunately, this brings the problem of state-explosion right back in the picture because establishing SYS $\leq$ ABS may be as computationally difficult as the original verification problem SYS $\models \varphi$.

To alleviate the above problem, the method of abstraction may be combined with that of *compositionality*. Here, compositionality refers to principles allowing properties of composite systems to be inferred from properties of their components. In particular we want to establish the safe abstraction condition, SYS $\leq$ ABS, in a compositional way, that is, assuming that SYS is a composite system of the form $\mathrm{SYS}_1 \parallel \mathrm{SYS}_2$, we may hope to find simple abstractions ABS_1 and ABS_2 such that:

$$\mathrm{SYS}_1 \leq \mathrm{ABS}_1 \quad \text{and} \quad \mathrm{SYS}_2 \leq \mathrm{ABS}_2$$

Provided the relation $\leq$ is a precongruence with respect to the composition operator $\parallel$, we may now complete the proof of the safe abstraction condition by establishing:

$$\mathrm{ABS}_1 \parallel \mathrm{ABS}_2 \leq \mathrm{ABS}$$

This approach nicely factors the original problem into the smaller problems and, and may be applied recursively until problems small enough to be handled by automatic means are reached.

The method of abstraction and compositionality is an old-fashion recipe with roots going back to the original, foundational work on concurrency theory [Mil89,Hoa78,OG76,Jon83,CM88]. In [JLS00] we have instantiated the method to Uppaal, where real-time systems are modelled as networks of timed automata communicating over (urgent) channels and shared discrete (e.g. integer) variables. A fundamental relationship between timed automata preserving safety properties — and hence useful in establishing safe abstraction properties — is that of timed simulation. However, in the presence of urgent communication and shared variables, this relationship fails to be a precongruence, and hence does not support compositionality. In [JLS00] we identify a notion of timed ready simulation supporting both abstraction and compositionality for Uppaal models. In addition, a method for automatically *testing* for the existence of timed ready simulation between timed automata using reachability analysis is presented (see

[9] i.e. $A \leq B$ and $B \models \phi$ should imply that $A \models \phi$.

also [ABL98]). Thus UPPAAL itself may be applied for such tests. The usefulness of the developed method is demonstrated by application to the verification of an industrial design: a system for audio/video power control developed by the company Bang & Olufsen. The size of the full protocol model is of such complexity that UPPAAL immediately encounters the state-explosion problem in a direct verification. However by application of the compositionality result and testing theory we were able to carry through a verification of the full protocol model. In [SS01] a similar approach is applied to the verification of the IEEE 1394a Root contentin Protocol using UPPAAL.

9 Conclusion

In addition to the techniques described in the previous sections, UPPAAL offers a range of other verification options including active clock reduction and approximate analysis based on convex-hull, supertrace and hash compaction. We refer the reader to `www.uppaal.com` for information on this.

The long effort effort spend on developing and implementing efficient datastructures and algorithms for analysing timed systems has succesfully payed off in terms of tools mature for industrial real-time applications. However, there is still room and need for improvements. Below we give an incomplete list of what could be some of the main algorithmic challanges for future research in the area:

– Continued search for appropriate BDD-like datastructures allowing for efficient representation and analysis of real-timed systems. CDDs and DDDs may be seen as promissing first attempts.
– Partial order reduction for timed systems, and more generally, methods for exploiting structure (e.g. hierarchicies) and (in)dependencies.
– Exploitation of symmetries to reduction explored and stored state-space.
– Extension of distributed and parallel reachability algorithm towards full TCTL model checking.
– Development of techniques allowing efficient use of disk (secondary memory) for storing explored state-spaces.
– Extension of acceleration technique to allow for more general cycles (e.g. involving more than one clock).
– Application of abstract interpretation in particular for dealing with models where the discrete part plays a major role (which is increasingly the case).

References

[ABB+01] Tobias Amnell, Gerd Behrmann, Johan Bengtsson, Pedro R. D'Argenio, Alexandre David, Ansgar Fehnker, Thomas Hune, Bertrand Jeannet, Kim G. Larsen, M. Oliver Möller, Paul Pettersson, Carsten Weise, and Wang Yi. UPPAAL - Now, Next, and Future. In F. Cassez, C. Jard, B. Rozoy, and M. Ryan, editors, *Modelling and Verification of Parallel Processes*, number 2067 in Lecture Notes in Computer Science, pages 100–125. Springer–Verlag, 2001.

[ABL98] Luca Aceto, Augusto Burgueno, and Kim G. Larsen. Model checking via reachability testing for timed automata. In Bernhard Steffen, editor, *Proc. 4th Int. Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'98)*, volume 1384 of *Lecture Notes in Computer Science*, pages 263–280. Springer, 1998.

[BDM+98] Marius Bozga, Conrado Daws, Oded Maler, Alfredo Olivero, Stavros Tripakis, and Sergio Yovine. Kronos: A model-Checking Tool for Real-Time Systems. In *Proc. of the 10th Int. Conf. on Computer Aided Verification*, number 1427 in Lecture Notes in Computer Science, pages 546–550. Springer–Verlag, 1998.

[Beh] Gerd Behrmann. A performance study of distributed timed automata reachability analysis. Submitted.

[Bel57] Richard Bellman. *Dynamic Programming*. Princeton University Press, 1957.

[Ben02] Johan Bengtsson. *Clocks, DBMs and STates in Timed Systems*. PhD thesis, Faculty of Science and Technology, Uppsala University, 2002.

[BFH+01] Gerd Behrmann, Ansgar Fehnker, Thomas S. Hune, Kim Larsen, Paul Petterson, and Judi Romijn. Efficient guiding towards cost-optimality in uppaal. In *Proc. of TACAS'2001*, Lecture Notes in Computer Science. Springer–Verlag, 2001.

[BGK+96] Johan Bengtsson, W.O. David Griffioen, Kåre J. Kristoffersen, Kim G. Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. Verification of an Audio Protocol with Bus Collision Using UPPAAL. In Rajeev Alur and Thomas A. Henzinger, editors, *Proc. of the 8th Int. Conf. on Computer Aided Verification*, number 1102 in Lecture Notes in Computer Science, pages 244–256. Springer–Verlag, July 1996.

[BHV00] Gerd Behrmann, Thomas Hune, and Frits Vaandrager. Distributed timed model checking - How the search order matters. In *Proc. of 12th International Conference on Computer Aided Verification*, Lecture Notes in Computer Science, Chicago, Juli 2000. Springer-Verlag.

[BLL+96] Johan Bengtsson, Kim G. Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. UPPAAL in 1995. In *Proc. of the 2*nd *Workshop on Tools and Algorithms for the Construction and Analysis of Systems*, number 1055 in Lecture Notes in Computer Science, pages 431–434. Springer–Verlag, March 1996.

[BLP+99] Gerd Behrmann, Kim G. Larsen, Justin Pearson, Carsten Weise, and Wang Yi. Efficient Timed Reachability Analysis Using Clock Difference Diagrams. In *Proc. of the 11th Int. Conf. on Computer Aided Verification*, number 1633 in Lecture Notes in Computer Science. Springer–Verlag, 1999.

[Bry86] Randal E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Trans. on Computers*, 1986.

[CM88] K.M. Chandy and J. Misra. *Parallel Program Design: A Foundation*. Addison Wesley, 1988.

[DBLY] Alexandre David, Gerd Behrmann, Kim G. Larsen, and Wang Yi. The next generation of uppaal. Submitted.

[Die99] H. Dierks. *Specification and Verification of Polling Real-Time Systems*. PhD thesis, Carl von Ossietzky Universität Oldenburg, July 1999.

[Dil89] David Dill. Timing Assumptions and Verification of Finite-State Concurrent Systems. In J. Sifakis, editor, *Proc. of Automatic Verification Methods for Finite State Systems*, number 407 in Lecture Notes in Computer Science, pages 197–212. Springer–Verlag, 1989.

[DT98] Conrado Daws and Stavros Tripakis. Model checking of real-time reachability properties using abstractions. In Bernard Steffen, editor, *Proc. of the 4th Workshop on Tools and Algorithms for the Construction and Analysis of Systems*, number 1384 in Lecture Notes in Computer Science, pages 313–329. Springer–Verlag, 1998.

[DY95] C. Daws and S. Yovine. Two examples of verification of multirate timed automata with KRONOS. In *Proc. of the 16th IEEE Real-Time Systems Symposium*, pages 66–75. IEEE Computer Society Press, December 1995.

[Hen02] Martijn Hendriks. Devlopment of reactive programs using uppaal. Master's thesis, KUN, Nijmegen University, 2002.

[HL02] Martin Hndriks and Kim G. Larsen. Exact acceleration of real-time model checking. In *Theory and Practice of Timed Systems*, volume 65 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers, 2002.

[HLS99] Klaus Havelund, Kim G. Larsen, and Arne Skou. Formal verification of a power controller using the real-time model checker UPPAAL. In *Proceedings of AMST 1999*, volume 1601 of *Lecture Notes in Computer Science*, pages 277–298, 1999.

[Hoa78] C.A.R. Hoare. Communicating Sequential Processes. *Communications of the ACM*, 21(8):666–677, 1978.

[Hol87] Gerard J. Holzmann. On limits and possibilities of automated protocol analysis. In *Proc. 7th IFIP WG 6.1 Int. Workshop on Protocol Specification, Testing, and Verification*, pages 137–161, 1987.

[Hun00] Thomas S. Hune. Modeling a language for embedded systems in timed automata. Technical Report RS-00-17, BRICS, Basic Research in computer Science, August 2000. 26 pp. Earlier version entitled *Modelling a Real-Time Language* appeared in FMICS99, pages 259–282.

[IKL+00] Torsten K. Iversen, Kåre J. Kristoffersen, Kim G. Larsen, Morten Laursen, Rune G. Madsen, Steffen K. Mortensen, Paul Pettersson, and Chris B. Thomasen. Model-Checking Real-Time Control Programs — Verifying LEGO Mindstorms Systems Using UPPAAL. In *Proc. of 12th Euromicro Conference on Real-Time Systems*, pages 147–155. IEEE Computer Society Press, June 2000.

[JLS00] Henrik Ejersbo Jensen, Kim G. Larsen, and Arne Skou. Scaling up Uppaal - automatic verification of real-time systems using compositionality and abstraction. In *Proceedings of FTRTFT 2000*, volume 1926 of *Lecture Notes in Computer Science*, pages 19–30, 2000.

[Jon83] C. Jones. Tentative steps toward a development method for interfering programs. *ACM Transactions on Programming Languages and Systems*, 5(4):596–620, 1983.

[Lam87] Leslie Lamport. A Fast Mutual Exclusion Algorithm. *ACM Trans. on Computer Systems*, 5(1):1–11, February 1987. Also appeared as SRC Research Report 7.

[LLPY97] Kim G. Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. Efficient Verification of Real-Time Systems: Compact Data Structures and State-Space Reduction. In *Proc. of the 18th IEEE Real-Time Systems Symposium*, pages 14–24. IEEE Computer Society Press, December 1997.

[LLPY02] Kim G. Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. Compact data structure and state-space reduction for model-checking real-time systems. *Real-Time Systems - the International Journal of Time-Critical Computing Systems*, 2002. To appear – accepted for publication.

[LP97] Henrik Lönn and Paul Pettersson. Formal Verification of a TDMA Protocol Startup Mechanism. In *Proc. of the Pacific Rim Int. Symp. on Fault-Tolerant Systems*, pages 235–242, December 1997.

[LPY95] Kim G. Larsen, Paul Pettersson, and Wang Yi. Compositional and Symbolic Model-Checking of Real-Time Systems. In *Proc. of the 16*th *IEEE Real-Time Systems Symposium*, pages 76–87. IEEE Computer Society Press, December 1995.

[LPY97] Kim G. Larsen, Paul Pettersson, and Wang Yi. Uppaal in a Nutshell. *Int. Journal on Software Tools for Technology Transfer*, 1(1–2):134–152, October 1997.

[LWYP99] Kim G. Larsen, Carsten Weise, Wang Yi, and Justin Pearson. Clock Difference Diagrams. *Nordic Journal of Computing*, 6(3):271–298, 1999.

[Mil89] R. Milner. *Communication and Concurrency.* Prentice Hall, Englewood Cliffs, 1989.

[MLAH99a] J. Møller, J. Lichtenberg, H. R. Andersen, and H. Hulgaard. Difference decision diagrams. In *Proceedings 13th International Conference on Computer Science Logic*, volume 1683 of *Lecture Notes in Computer Science*, pages 111–125, Madrid, Spain, September 1999.

[MLAH99b] J. Møller, J. Lichtenberg, H. R. Andersen, and H. Hulgaard. Fully symbolic model checking of timed systems using difference decision diagrams. In *Proceedings First International Workshop on Symbolic Model Checking*, volume 23-2 of *Electronic Notes in Theoretical Computer Science*, Trento, Italy, July 1999.

[OG76] S. Owicki and D. Gries. An Axiomatic Proof Technique for Parallel Programs I. *Acta Informatica*, 6(4):319–340, 1976.

[Pet99] Paul Pettersson. *Modelling and Analysis of Real-Time Systems Using Timed Automata: Theory and Practice.* PhD thesis, Department of Computer Systems, Uppsala University, February 1999.

[PS80] Wolfgang J. Paul and Janos Simon. Decision Trees and Random Access Machines. In *Logic and Algorithmic*, volume 30 of *Monographie de L'Enseignement Mathématique*, pages 331–340. L'Enseignement Mathématique, Université de Genève, 1980.

[Rok93] Tomas Gerhard Rokicki. *Representing and Modeling Digital Circuits.* PhD thesis, Stanford University, 1993.

[SS01] D.P.L. Simons and M.I.A. Stoelinga. Mechanical verification of the IEEE 1394a root contention protocol using Uppaal2k. *Springer International Journal of Software Tools for Technology Transfer*, 2001.

[ST98] Karsten Strehl and Lothar Thiele. Symbolic Model Checking of Process Networks Using Interval Diagram Techniques. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD-98)*, pages 686–692, 1998.

[WT95] Howard Wong-Toi. *Symbolic Approximations for Verifying Real-Time Systems.* PhD thesis, Standford University, 1995.

The Model-Checking Kit*

Claus Schröter, Stefan Schwoon and Javier Esparza

Laboratory for Foundations of Computer Science,
University of Edinburgh,
email: {clau0603,schw1201,jav}@dcs.ed.ac.uk

Abstract. The Model-Checking Kit [8] is a collection of programs which allow to model finite state systems using a variety of modelling languages, and verify them using a variety of checkers, including deadlock-checkers, reachability-checkers, and model-checkers for the temporal logics CTL and LTL [7].

1 Introduction

Research on automatic verification has shown that no single model-checking technique has the edge over all others in any application area. Moreover, it is very difficult to determine a priori which technique is the most suitable for a given model. It is thus sensible to apply different techniques to the same model. However, this is a very tedious and time-consuming task, since each algorithm uses its own description language. The Model-Checking Kit [8] has been designed to provide a solution to this problem in an academic setting, with potential applications to industrial settings.

There exist many different models for concurrent systems. Within the Kit we chose 1-safe Place/Transition nets as the basic model for the following two reasons: (i) They are a very simple model with nearly no variants. In contrast most other models have many different variants. For instance, communicating automata can be synchronous or asynchronous, and communication can be formalised in different ways. Process algebras have a wealth of different operators and semantics, and there also exist many different high-level net models. (ii) Many different verification techniques which can deal with 1-safe P/T nets are available. Since 1-safe P/T nets have a well-defined partial order semantics, partial order techniques like stubborn sets [21] and net unfoldings [18] can be applied (as a matter of fact, these techniques were originally introduced for Petri nets). Since a marking of a 1-safe P/T net is just a vector of booleans, symbolic techniques based on BDDs [5], like those implemented in SMV [17], can also be used. And, of course, the standard interleaving semantics of Petri nets allows to apply explicit state exploration algorithms, like those of SPIN [13].

For systems modelled in a language with a 1-safe net semantics, all the techniques listed above are in principle applicable. Since each of these techniques has both strengths and weaknesses, it would be highly desirable to apply them

* http://www7.in.tum.de/gruppen/theorie/KIT/

all and to compare the results. However, a user who wishes to employ two or more verification packages does not have an easy task. In particular, the packages have different input formats, and so the user is forced to enter input data multiple times, a rather tedious task and one prone to introduce errors and inconsistencies.

To amend this situation the Kit provides a shell which allows the user to specify input data (i.e. systems and properties) in a variety of input languages. Once a system and a property have been specified, the user can choose any of the model-checkers available in the shell to verify the property. The user is not required to be familiar with the different ways in which 1-safe P/T nets are represented to the different checkers.

The paper is structured as follows. In Section 2 we introduce a small example and use it to show how the Kit works. In Section 3 we present the modelling languages and verification techniques which are supported by the Kit. Section 4 gives a brief overview of the Kit's available options and their use. In Section 5 we present some experimental results which show performance differences of the individual verification techniques. Finally, we close with some conclusions in Section 6.

2 An Example

We show how the Kit works by means of a small example. We modelled Peterson's mutual exclusion algorithm [19] in $B(PN)^2$ as depicted in Figure 1 (a). $B(PN)^2$ [3], originally well-known from the PEP-tool [10], is a parallel programming language and one of the Kit's input languages. The notation is mostly self-explanatory, but for the sake of clarity we would like to point out two things: (i) $\langle$`t'=1`$\rangle$ means value assignment, whereas $\langle$`t=1`$\rangle$ means test of equality; (ii) `incs1:` denotes a label which can be used in formulae to mark a program point between two actions.

Suppose we want to check a mutual exclusion property, i.e. whether there exists a global system state in which both processes enter their critical sections simultaneously. The Kit allows this property to be expressed as `"incs1"` & `"incs2"`. The Kit takes the formula and the $B(PN)^2$ description and translates them into a 1-safe P/T net and a corresponding formula (e.g. `P9 & P14`, where P9, P14 are place names of the net). Then the Kit invokes the model-checker chosen by the user (which would usually be one of the available reachability-checkers in this case). It checks whether there exists a reachable marking with tokens on both P9 and P14 simultaneously, and returns the result to the Kit. In case (a) the answer is 'no' which means that the mutex property holds. But what happens in case of an error? For that let us suppose that the user made a typo within the specification and wrote $\langle$`i1'=0 or t=1`$\rangle$ in Process 2 instead of $\langle$`i1=0 or t=1`$\rangle$. This causes a violation of the mutex property and the model-checker returns a transition sequence leading to the state which puts tokens simultaneously onto the places which represent the critical regions of the processes. As shown in Figure 1 (b) the Kit interprets this transition sequence at the

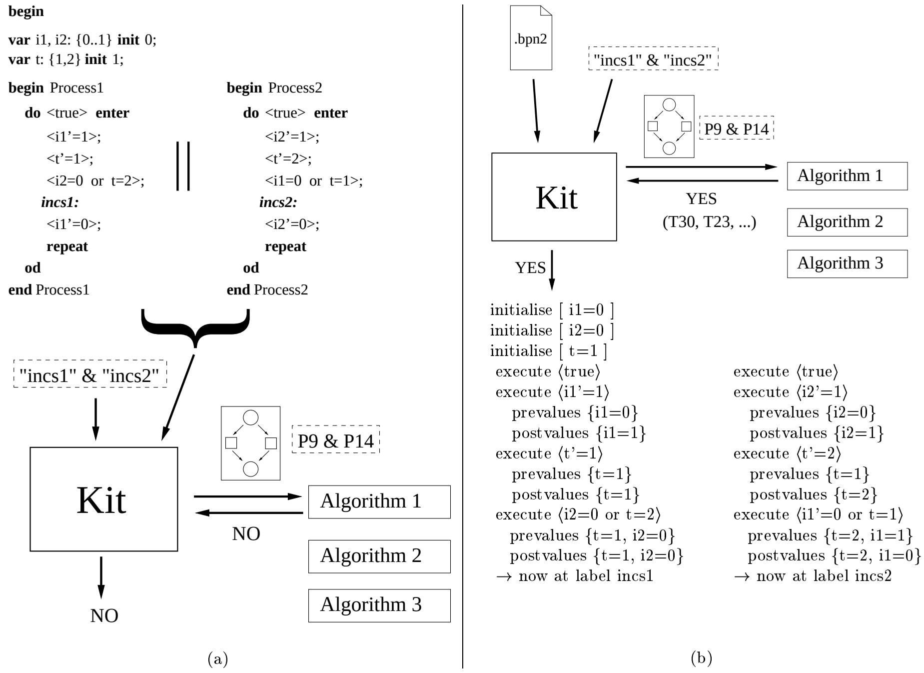


Fig. 1. Peterson's mutex algorithm in $\mathrm{B(PN)}^2$

level of the chosen input language (i.e. $\mathrm{B(PN)}^2$) and outputs the result suitably formatted to the user.

3 Modelling languages and verification techniques

In this section we briefly introduce the different modelling languages and verification techniques supported by the Kit. Furthermore, we give a quick overview of how to describe properties.

3.1 Modelling a system

The Kit offers several languages for modelling a system. These languages can be divided into so-called net languages and high-level languages which abstract from net details.

- **High-Level Languages**
  Loosely speaking, these description languages abstract from structural net concepts like places, transitions, and arcs. The Kit currently offers three such languages called $\mathbf{B(PN)^2}$, **CFA**, and **IF**. $\mathrm{B(PN)}^2$ [3] (Basic Petri Net Programming Notation) is a structured parallel programming language offering features such as loops, blocks, and procedures. It is well-known from

the PEP-tool [10]. CFA [9, 8] (Communicating Finite Automata) is a language for the description of finite automata which communicate via shared variables or channels of finite length. It offers very flexible communication mechanisms and is also one of the modelling languages of the PEP-tool [10]. Finally, IF [4] (Interchange Format) is a language proposed in order to model asynchronous communicating real-time systems. It is the common model description language of the European ADVANCE [1] (Advanced Validation Techniques for Telecommunication Protocols) project.

- **Net languages**
  The Kit supports two net languages, **PEP** and **SENIL**. In these languages one has to define places, transitions, and arcs explicitly. PEP [2] is the low level net language of the PEP-tool [10]. It is supported by the Kit mostly because some tools can automatically export models into this format. SENIL [8] (Simple Extensible Net Input Language) is designed to make it easy to specify small P/T nets by hand; it is suitable for small nets with at most a few dozens of nodes, but not for larger projects.

3.2 Describing properties

The Kit can be used to check several types of properties, e.g. deadlock-freeness, reachability, safety and liveness properties. Except for deadlock-freeness these properties will be expressed as formulae.

Reachability properties In our framework a reachability property is a statement about states of the system. For example, the mutual exclusion property of critical regions can be understood as a reachability property. It amounts to the question whether there exists a reachable global state of the system in which two processes enter their critical regions simultaneously. These properties can be expressed with so-called state formulae. A state formula is a propositional logic formula consisting of atomic propositions and logical operators.

Safety and liveness properties Safety and liveness properties are expressed as formulae of temporal logics like CTL and LTL. They are the most popular temporal logics, and together they can express all common safety and liveness properties. Here we give just a brief introduction to LTL and CTL according to [7].

- **LTL** means Linear-Time Temporal Logic. The underlying structure of time is a totally ordered set. Under the assumption that the time corresponds to $(\mathbf{N}, <)$, the time is discrete, has an initial moment with no predecessors and is infinite into the future. LTL formulae consist of atomic propositions, boolean connectives and temporal operators. Temporal operators are $\mathbf{G}p$ ("always p", "henceforth p"), $\mathbf{F}p$ ("sometime p", "eventually p"), $\mathbf{X}p$ ("nexttime p") and $p\,\mathbf{U}\,q$ ("p until q"). The Kit supports only the *next-free* fragment of LTL.

- **CTL**, meaning Computation Tree Logic, is a branching time logic. The underlying structure of time is assumed to have a branching tree-like nature. It corresponds to an infinite tree where each node may have finitely many successors and must have at least one successor. These trees have a natural correspondence with the computations of concurrent systems or nondeterministic programs. A CTL formula consists of a path quantifier [**A** (all paths), **E** (there exists a path)] followed by an arbitrary linear-time formula, allowing boolean combinations and nestings of linear-time operators (**G**, **F**, **X**, **U**).

3.3 Verification techniques

As mentioned in the introduction many different verification techniques for 1-safe Petri nets are available. These techniques include among others the explicit construction of the state space, stubborn sets [21], BDDs [5], and net unfoldings [18]. The explicit construction of the state space is the classical approach, and still adequate in cases where the state space explosion is not very acute. Stubborn sets are used to avoid constructing part of the state space. They exploit information about the concurrency of actions. Using symbolic techniques (e.g. BDDs) one can succinctly represent large sets of states. They can reach spectacular compactification ratios for regularly structured state spaces. Approaches which are based on unfolding techniques make use of an explicitly constructed partial-order semantics of the system. It contains information not only on the reachability relation, but also on causality and concurrency. This technique is adequate for systems exhibiting a high degree of concurrency.

When planning the Kit we intended to integrate various checkers such that each of the verification approaches mentioned above is represented by at least one checker. This has led to the following selection:

- The **PEP**-tool [10] (Programming Environment based on Petri nets) is a programming and verification environment for parallel programs written in $B(PN)^2$ or CFA. Programs can be formally analysed using methods which are based on the unfolding technique [18]. The PEP-tool is distributed by the Theory group (subgroup Parallel Systems) of the University of Oldenburg. PEP contributes to the Kit a deadlock-checker, a reachability-checker, and a model-checker for LTL.
- **PROD** [22] is an advanced tool for efficient reachability analysis. It implements different advanced reachability techniques for palliating the state explosion problem, including partial-order techniques like stubborn sets [21], and techniques which exploit symmetries. PROD is distributed by the Formal Methods Group of the Laboratory for Theoretical Computer Science at the Helsinki University of Technology. PROD contributes to the Kit a deadlock-checker, a reachability-checker, a CTL- and LTL-checker.
- The **SMV** system [17] is a tool for checking finite state systems against specifications in the temporal logic CTL. The input language of SMV is designed to allow the description of finite state systems that range from

completely synchronous to completely asynchronous, and from the detailed to the abstract. SMV is distributed by the Carnegie Mellon University. Its verification algorithms are based on BDDs [5] and it contributes to the Kit a deadlock-checker and a CTL-checker.

- **SPIN** [13] is a widely distributed software package that supports the formal verification of distributed systems. It can be used as a full LTL model-checking system, but it can also be used as an efficient on-the-fly verifier for more basic safety and liveness properties. Many of the latter properties can be expressed and verified without the use of LTL. SPIN uses explicit construction of the state space. It is distributed by the Formal Methods and Verification Group of Bell Labs. SPIN contributes to the Kit an LTL-checker.
- The tool **MCSMODELS** [12] is a model-checker for finite complete prefixes (i.e. net unfoldings [18]). It currently uses the PEP-tool [10] to generate the prefixes. These prefixes are then translated into logic programs with stable model semantics, and the integrated Smodels solver is used to solve the generated problems. MCSMODELS is distributed by the Formal Methods and Logic Groups of the Laboratory for Theoretical Computer Science at the Helsinki University of Technology. MCSMODELS contributes to the Kit a deadlock-checker and a reachability-checker.
- **CLP** [14] is a linear-programming model-checker. It uses net unfoldings [18] and can check among others deadlock-freeness, reachability, and coverability of a marking. CLP is distributed by the Parallelism Research Group of the University of Newcastle upon Tyne and contributes to the Kit a deadlock-checker.

4 How to use the Kit

In this section we show how to use the Kit for verification tasks. The Kit is a command-line oriented tool (called check) without any graphical user interface. The available options are listed by calling check without any arguments. Figure 2 shows an overview of its current version.

For a correct program call one has to type

```
check [options] <input>:<checker> <modelfile> <formulafile>
```

where

- <input> is a place holder for one of the available modelling languages, i.e. cfa, bpn2, if, pep, senil.
  Note: <input> may be omitted; in this case check guesses the input language by looking at the extension of <modelfile> (which should be .cfa, .bpn2, .if, .ll_net, or .senil, in the order of the languages mentioned above). The user is free to use arbitrary extensions, but then the language has to be specified explicitly.
- <checker> should be replaced by one of the available algorithms, see the list at the bottom of Figure 2.

```
Usage: check [options] <input>:<checker> <modelfile> <formulafile>
   or: check -r <name> [options] <checker> <formulafile>

     Options:
      -s <name>         save intermediate results under <name>
      -r <name>         resume from intermediate results saved under <name>
      -t <dir>          place temporary files in <dir> (default is '.')
      -v                run in verbose mode

     Available input formats:
      cfa               concurrent finite automata
      bpn2              B(PN)^2 language
      if                IF language
      pep               PEP low level net format
      senil             SENIL net format

     Available algorithms:
      CTL         : prod-ctl, smv-ctl
      LTL         : prod-ltl, pep-ltl, spin-ltl
      Deadlock    : prod-dl, smv-dl, pep-dl, mcs-dl, clp-dl
      Reachability: prod-reach, pep-reach, mcs-reach
```

Fig. 2. The Kit's available options

- `<modelfile>` is the name of the file containing the system specification.
- `<formulafile>` is the name of the file containing the formula to be checked. Note: For deadlock-checking no formula file is needed.
- `[options]` are as follows:
  - `-s <name>`
    Temporary files representing intermediate results will be saved in a tar-archive `mckit_save_<name>.tar`. Some algorithms profit from the reuse of intermediate results. For example, if one uses a method based on the unfolding technique for verifying many properties on the same system, it is sensible to calculate the unfolding only once and not for every property over and over again. So the unfolding can be saved with this option for reuse (see option `-r`).
  - `-r <name>`
    With this option one can reuse intermediate results saved before with option `-s <name>`. This is sensible if one wants to check many properties on the same system. Then the translation from the modelling language into the correct input format for the checker should be done only once and not for every single property. When using this option one should omit the modelling language and the modelfile. Then the correct program call is:
    `check -r <name> [options] <checker> <formulafile>`
    The selected checker can then take advantage of the files saved in the file `mckit_save_<name>.tar`.

	prod-dl	smv-dl	pep-dl	mcs-dl	clp-dl
peterson	7.04 (0.09)	0.24	0.04	0.05	0.03
plate(5)	46.68 (1.38)	*mem*	4.80	0.53	0.54
client/server	61.06 (0.79)	111.80	0.76	0.54	0.55
key(4)	37.63 (0.20)	*mem*	*mem*	*mem*	*mem*
fifo(30)	36.74 (0.72)	*mem*	*mem*	*mem*	*mem*

Fig. 3. Results for Deadlock-Checking

5 Experimental results

In this section we compare the performances of the algorithms by means of experimental results on several systems. The results demonstrate the point we made in the beginning, namely that no single method has the edge over all others. We present results for checking deadlock-freeness and some safety properties.

All experiments were performed on a Linux PC with 64 MByte of RAM and a 230 MHz Intel Pentium II CPU. The times are measured in seconds. The systems we used are as follows:

- peterson: Mutual exclusion algorithm [19].
- plate(5): Production cell which handles 5 plates [11, 15].
- client/server: Client/Server system with 2 clients and 1 server [1].
- key(4): Manages keyboard/screen interaction in a window manager for 4 customer tasks [6].
- fifo(30): 1-bit-FIFO with depth 30 [16, 20].

The systems are modelled in different languages. Peterson's mutual exclusion algorithm is modelled in $B(PN)^2$, and the client/server system in IF. All other examples are modelled in PEP's low-level net format.

Figure 3 shows the results for deadlock-checking. We split PROD's verification times for the following reason: At first, PROD reads the net description file and produces a corresponding C file. Then this C file is compiled and linked to an executable reachability graph generator program. Finally, the actual verification task is done by performing this executable program. Since PROD spends most of the time for the generation of the executable file, the pure verification times are quoted in parentheses.

Peterson's mutual exclusion algorithm is a small example, and all verification techniques behave well. But a look at the systems plate(5) and client/server already shows a big difference in the performances. The unfolding based techniques outperform PROD and SMV here. Actually, SMV runs out of memory during the verification of the production cell (signified by '*mem*'). On the contrary, the systems key(4) and fifo(30) are examples in which PROD beats the other tools.

The results for safety properties are depicted in Figure 4. The properties were expressed as LTL-formulas, and they were checked using LTL checkers. For the production cell we checked a mutual exclusion property: exactly one of

	prod-ltl		pep-ltl	spin-ltl
plate(5)	67.22	(2.52)	1.20	*mem*
client/server	257.57	(30.83)	2.51	20.74
key(4)	4206.00	(4152.66)	*mem*	11.53

Fig. 4. Results for LTL-Checking

three places of the net carries a token. For the client/server system we checked that a buffer overflow can not occur. For the key(4) system we checked a mutual exclusion property for two places. A look at the results confirms that the checkers behave quite differently here as well. We are able to verify the property for the production cell with PROD and PEP, but not with SPIN. On the contrary, for the client/server system SPIN checks the formula much faster than PROD. The key(4) system is an example which can be verified quickly with SPIN, but not with PEP.

It is important to notice that the performance of each of the tools may degrade a lot when used as part of the Kit. For instance, it is easy to model a system in PROMELA such that checking some easy reachability property with SPIN takes virtually no time; if the same system is modelled in, say **CFA**, and then checked again with SPIN, the verification can run out of memory. The reason is that the Kit transforms the initial CFA model into a 1-safe Petri net, and this net into PROMELA. If the model is data intensive, the net (and with it the PROMELA model) can easily blow-up. Another reason for a degraded performance is that the user of the Kit does not have access to all the flags of each of the checkers (future versions of the Kit should include this feature).

6 Conclusions

The Model-Checking Kit is a collection of programs which allow to model a finite-state system using a variety of modelling languages, and (attempt to) verify it using a variety of checkers. It has been successfully applied in a lab course on automatic verification. Special care has been taken to design it in a modular way and to make it easy to use and easy to install. Experiments with beta-testers have shown that a moderately skilled user can install the tool and verify the first property of a small system within half an hour. Furthermore, the Kit has a high degree of portability since all programs are written in plain C and we do not offer any graphical user interface. The Kit can be used for comparing the performances of different verification methods. However, it must be emphasised that, since each of the Kit's checkers has been optimised for its own modelling language, the Kit's internal language conversions can lead to important losses in performance. Finally, the Kit is an open library. Due to its modular design it is easy to add new description languages or checkers, and to replace old versions of checkers by new ones. Anyone who is interested in adding new languages and/or tools is cordially invited to contact the authors.

References

1. ADVANCE - Advanced Validation Techniques for Telecommunication Protocols. http://verif.liafa.jussieu.fr/~haberm/ADVANCE/main.html.
2. E. Best and B. Grahlmann. PEP Documentation and User Guide 1.8. Universität Oldenburg, 1998.
3. E. Best and R. P. Hopkins. $B(PN)^2$ - a Basic Petri Net Programming Notation. In *PARLE'93*, LNCS 694, pages 379 – 390. Springer-Verlag, 1993.
4. M. Bozga, J.-C. Fernandez, L. Ghirvu, S. Graf, L. Mounier, J. P. Krimm, and J. Sifakis. The Intermediate Representation IF. Technical Report. Vérimag, 1998.
5. R. E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, C-35(8):677 – 691, Aug. 1986.
6. J. C. Corbett. Evaluating Deadlock Detection Methods, 1994.
7. E. A. Emerson. Temporal and Modal Logic. In *Handbook of Theoretical Computer Science*, volume B, pages 997 – 1067. Elsevier Science Publishers B. V., 1990.
8. J. Esparza, C. Schröter, and S. Schwoon. The Model-Checking Kit. http://www7.in.tum.de/gruppen/theorie/KIT/.
9. B. Grahlmann, M. Möller, and U. Anhalt. A new Interface for the PEP-tool - Parallel Finite Automata. 2nd Workshop of Algorithms and Tools for Petri nets. Oldenburg, 1995.
10. B. Grahlmann, S. Römer, T. Thielke, B. Graves, M. Damm, R. Riemann, L. Jenner, S. Melzer, and A. Gronewold. PEP: Programming Environment Based on Petri Nets. *Hildesheimer Informatik Berichte*, (14), May 1995. Universität Hildesheim.
11. M. Heiner and P. Deusen. Petri net based qualitative analysis - A case study. Technical report I-08/1995. Brandenburg Technische Universität Cottbus, 1995.
12. K. Heljanko. *Combining Symbolic and Partial Order Methods for Model Checking 1-Safe Petri Nets*. PhD thesis, Helsinki University of Technology, 2002.
13. G. J. Holzmann. *Design and Validation of Computer Protocols*. Prentice Hall, 1991.
14. V. Khomenko. CLP. http://www.cs.ncl.ac.uk/people/victor.khomenko/home.formal/tools/tools.html.
15. C. Lewerentz and T. Lindner. Formal Development of Reactive Systems: Case Study Production Cell. LNCS 891. Springer-Verlag, 1995.
16. A. J. Martin. Self-timed FIFO: An exercise in compiling programs into VLSI circuits. In *From HDL Descriptions to Guruanteed Correct Circuit Designs*, pages 133 – 153. Elsevier Science Publishers, 1986.
17. K. L. McMillan. *Symbolic Model Checking - An approach to the state explosion problem*. PhD thesis, Carnegie Mellon University, 1992.
18. K. L. McMillan. Using Unfoldings to Avoid the State Explosion Problem in the Verification of Asynchronous Circuits. In *CAV'92*, LNCS 663, pages 164 – 174. Springer-Verlag, 1992.
19. M. Raynal. Algorithms For Mutual Exclusion, 1986.
20. O. Roig, J. Cortadella, and E. Pastor. Verification of Asynchronous Circuits by BDD-based Model Checking of Petri Nets. In *ATPN'95*, LNCS 935, pages 374 – 391. Springer-Verlag, 1995.
21. A. Valmari. On-the-Fly Verification with Stubborn Sets. In *CAV'93*, LNCS 697, pages 397 – 408. Springer-Verlag, 1993.
22. K. Varpaaniemi, J. Halme, K. Hiekkanen, and T. Pyssysalo. PROD Reference Manual. *Technical Reports*, B(13):1 – 56, Aug. 1995.

An Overview of CADP 2001

Hubert Garavel, Frédéric Lang, and Radu Mateescu

INRIA Rhône-Alpes – VASY – 655, avenue de l'Europe, Montbonnot Saint Martin – F-38 334 Saint Ismier Cedex, France
{Hubert.Garavel,Frederic.Lang,Radu.Mateescu}@inrialpes.fr

Abstract. CADP is a toolbox for specifying and verifying asynchronous finite-state systems described using process algebraic languages. It offers a wide range of state-of-the-art functionalities assisting the user throughout the design process: compilation, rapid prototyping, interactive and guided simulation, verification by equivalence/preorder checking and temporal logic model-checking, and test generation. The languages, models, and verification techniques used in CADP have a broad application domain, allowing to deal with communication protocols, distributed systems, embedded software, mobile telephony, asynchronous hardware, cryptography, security, human-computer interaction, etc. CADP is currently used both in industrial companies and academic institutions for research and teaching purposes. During the last years, over 50 applications and case-studies performed using CADP have been reported.

1 Introduction

CADP (the CÆSAR/ALDÉBARAN Development Package[1]) is a toolbox for protocol engineering, which offers a wide range of functionalities, from interactive simulation to the most recent formal verification techniques. CADP is dedicated to the efficient compilation, simulation, formal verification, and testing of descriptions written in the ISO language LOTOS [19], a value passing process algebra. It also accepts other input languages such as finite state machines and networks of communicating finite state machines.

July 2001 has seen the release of the new version CADP 2001 "Ottawa"[2]. Among many other features, CADP provides several tools for computing bisimulations (minimizations and comparisons), several model-checkers for various temporal logics and μ-calculus, and several verification algorithms including exhaustive verification, on-the-fly verification, symbolic verification using Binary Decision Diagrams, and compositional verification based on refinement. It contains many improvements and five new tools.

The architecture of CADP 2001 is displayed in Figure 1 and explained throughout the paper, which is organized as follows. Section 2 introduces the different

[1] Detailed information is available at http://www.inrialpes.fr/vasy/cadp.

[2] CADP 2001 "Ottawa" was named in honor of Professor Luigi Logrippo and his research team at the University of Ottawa, who are actively promoting formal methods, especially LOTOS, in the telecommunication industry.

languages and models used by the CADP tools. Section 3 describes the main tools of CADP. Section 4 presents the new tools contained in CADP 2001. Finally, Section 5 gives concluding remarks.

2 Description languages and intermediate models

The CADP toolbox accepts three different input formalisms, materialized by the three grey boxes in Figure 1:

- high-level protocol descriptions written in the ISO language LOTOS [19]: CADP contains two compilers (CÆSAR and CÆSAR.ADT) which translate LOTOS descriptions into C code that can be used for simulation, verification, and testing purposes;
- low-level protocol descriptions specified as Labeled Transition Systems (LTSs, for short), i.e., finite state machines the transitions of which are labeled by action names;
- intermediate-level networks of communicating LTSs, i.e., finite state machines running in parallel and synchronizing together by means of rendezvous; these networks can be expressed in two different formats: EXP (LTSs combined together using LOTOS parallel composition and hiding operators) and FC2 (LTSs combined together using a synchronization product).

The latest releases of the CADP toolbox devote a growing importance to the concept of intermediate formats and programming interfaces. In the sequel of this section, we present the OPEN/CÆSAR environment, which allows the CADP tools to be applied to protocol descriptions written in other languages than LOTOS (e.g., μCRL, SDL, UML/RT), and the BCG environment, which provides a compact LTS description format together with efficient and useful tools and libraries.

2.1 The OPEN/CAESAR environment

OPEN/CÆSAR [13] is an extensible, language-independent Application Programming Interface (API) that allows user-defined programs for simulation, execution, verification (partial, on-the-fly, etc.), and test generation to be developed in a simple and modular way. Various modules have already been written in the OPEN/CÆSAR framework, including: EVALUATOR, an on-the-fly model-checker (Section 4.2), OCIS, an interactive simulator with X-window interface (Section 4.3), TGV, a tool for the generation of conformance test suites based on verification technology (Section 4.5), and many other tools for random execution, deadlock detection, reachability analysis, sequence searching, abstraction of an LTS w.r.t. an interface, etc.

Basically, the OPEN/CÆSAR environment offers primitives to transform a system description into an LTS represented *implicitly* by its initial state and its successor function. Then, every tool connected to the OPEN/CÆSAR environment can take the resulting implicit LTS as input. Three languages have access

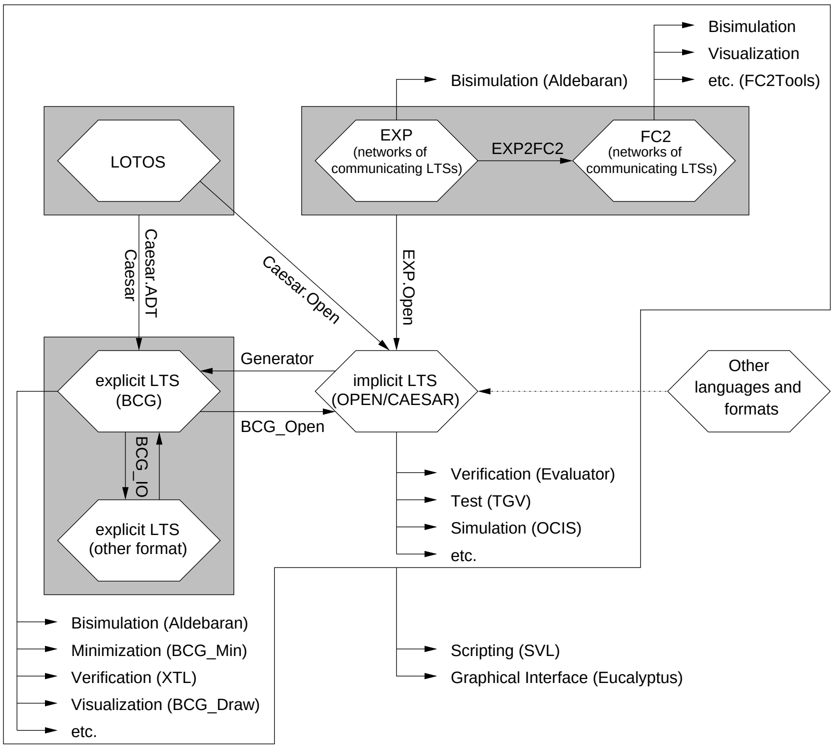


Fig. 1. Architecture of CADP 2001 "Ottawa"

to the OPEN/CÆSAR environment, namely the BCG graph format (Section 2.2), the LOTOS language, and networks of communicating automata in the EXP format. Since OPEN/CÆSAR is open and well-documented, users can easily extend the environment by adding their own modules or connect their own languages to fit specific needs.

2.2 The BCG graph format and libraries

BCG (*Binary-Coded Graphs*) [11] is both a format for the representation of explicit LTSs and a collection of libraries and programs dealing with this format. Compared to ASCII-based formats for LTSs, the BCG format uses a binary representation with compression techniques resulting in much smaller (up to 20 times) files. BCG is independent from any source language but keeps track of the objects (types, functions, variables) defined in the source programs.

The BCG format supports tools for drawing BCG graphs with an automatic layout of states and transitions, editing the display of BCG graphs interactively, providing information about BCG graphs such as the size of the graph, its number of states and transitions or the list of its labels, performing conversions between the BCG format and a dozen of other formats, hiding and renaming the labels of a graph according to regular expressions, generating dynamic libraries for BCG graphs, minimizing graphs according to strong or branching bisimulation (see the BCG_MIN tool, Section 4.1), etc.

Simple application programming interfaces are available to read and to produce a BCG graph. Moreover, the BCG_OPEN tool establishes a gateway between the BCG format (explicit LTSs) and the OPEN/CÆSAR environment (implicit LTSs).

3 Main tools of CADP

The CADP toolbox contains several tools. In the sequel, we describe the most significant of these tools.

3.1 The ALDÉBARAN tool for computing bisimulations

Jointly developed by the VASY team and the VERIMAG laboratory, ALDÉBARAN [6] is a tool for verifying communicating systems, represented by LTSs. It allows the reduction of LTSs modulo various equivalence relations (such as strong bisimulation, observational equivalence, $\tau^* \cdot a$ bisimulation, branching bisimulation, safety equivalence, etc.). It also allows to perform comparison according to strong bisimulation preorder, $\tau^* \cdot a$ preorder, or safety preorder.

The verification algorithms used in ALDÉBARAN are based either on the Paige-Tarjan algorithm for computing the relational coarsest partition [26], on the "on-the-fly" techniques proposed by Fernandez-Mounier [7], or on symbolic LTS representations using Binary Decision Diagrams (BDDs) [3]. ALDÉBARAN has diagnosis capabilities that provide the user with explanations (counter-example sequences) when two LTSs are found to be not equivalent.

3.2 The CÆSAR compiler

CÆSAR [9] is a compiler that translates the behavioral part of a LOTOS specification into either a C program (to be executed or simulated) or into an LTS.

CÆSAR translation algorithms proceed in several steps. First the LOTOS description is translated into a simplified process algebra called SUBLOTOS. Then an intermediate Petri Net model is generated, which provides a compact, structured and user-readable representation of both the control and data flow. Eventually the LTS is produced by performing reachability analysis on the Petri Net.

CÆSAR accepts full LOTOS with very slight contextual restrictions as regards process recursion. Despite these restrictions, the subset of LOTOS handled by CÆSAR is large and usually sufficient for real-life needs.

The current version of CÆSAR allows the generation of large LTSs (some million states) within a reasonable lapse of time. Moreover, the efficient compiling algorithms of CÆSAR can also be exploited in the framework of the OPEN/CÆSAR environment.

The most recent version of the CÆSAR compiler provides a functionality called EXEC/CÆSAR [15] for C code generation. This C code interfaces with the real world, and can be embedded in applications. This allows rapid prototyping directly from the LOTOS specification.

3.3 The CÆSAR.ADT compiler

CÆSAR.ADT [10] is a compiler that translates the data part of LOTOS specifications into libraries of C types and functions. Each LOTOS sort is translated into an equivalent C type and each LOTOS operation is translated into an equivalent C function (or macro-definition). CÆSAR.ADT also generates C functions for comparing and printing abstract data types values, as well as iterators for the sorts of finite domain.

CÆSAR.ADT accepts full LOTOS with the following (quite natural) restriction, as regards the data part: constructor operations must be identified; equations are oriented; there is a decreasing priority between equations; equations between constructors are not allowed. Also, parameterized types are not compiled (yet).

CÆSAR.ADT is fast: translation of large programs (several thousands of lines) is usually achieved in a few seconds. CÆSAR.ADT can be used in conjunction with CÆSAR, but it can also be used separately to compile and execute efficiently large abstract data types descriptions.

3.4 The XTL model-checker

XTL (*eXecutable Temporal Language*) [23] is a functional-like programming language designed to allow an easy, compact implementation of various temporal logic operators. These operators are evaluated over an LTS encoded in the BCG format. Besides the usual predefined types (booleans, integers, etc.), the XTL language defines special types, such as sets of states, transitions, and labels of the LTS. It offers primitives to access the informations contained in states and labels, to obtain the initial state, and to compute the successors and predecessors of states and transitions. The temporal operators can be easily implemented using these functions together with recursive user-defined functions working with sets of states and/or transitions of the LTS. A compiler for XTL has been developed, and several temporal logics like HML [18], CTL [4], ACTL [25], and LTAC [28] have been easily implemented in XTL.

3.5 The EUCALYPTUS graphical user interface

EUCALYPTUS [12] is a graphical user interface written in Tcl/Tk that integrates the CADP tools in a unified, user-friendly interface. This interface has the name of the project within which it was developed: the Euro-Canadian project "EUCALYPTUS".

Additionally, EUCALYPTUS integrates complementary software such as the APERO data type pre-processor for LOTOS [27], the ELUDO simulator of LOTOS descriptions [31], or the FC2 tools [1], together with the graphical editor AUTOGRAPH [29].

4 New tools of CADP 2001

The new release of CADP contains five new tools: BCG_MIN, EVALUATOR 3.0, OCIS, SVL, and TGV.

4.1 The new BCG_MIN tool for computing bisimulations

Jointly developed by the VASY team and Holger Hermanns (University of Twente), BCG_MIN implements various minimization algorithms for graphs encoded in the BCG format. It can be used to minimize "standard" LTSs, as well as "probabilistic" and "stochastic" LTSs, which may carry respectively probabilistic and stochastic labels and generalize many theoretical models published in the literature (for instance the Discrete Time Markov Chains and the Continuous Time Markov Chains).

Compared to former LTS minimization tools (including ALDÉBARAN and FC2 tools), BCG_MIN only implements two equivalences, namely strong and branching bisimulation. For these two equivalences, however, it offers compelling advantages:

- BCG_MIN can handle larger LTSs, at least larger by an order of magnitude; for instance, the largest LTS reduced so far by BCG_MIN has more than 7 million states and 40 million transitions; according to Prof. Jan Friso Groote [17], BCG_MIN is "*the best implementation of the standard (i.e., Groote & Vaandrager [16]) algorithm for branching bisimulation*";
- BCG_MIN uses BCG as its native format, thus leading to speed improvement, because BCG occupies much less disk space than most graph formats;

– BCG_MIN is able to print state equivalence classes in a user-friendly way, by relating the state numbers of the minimized graph to the state numbers of the original graph; in the case of branching equivalence, the τ-cycles are properly displayed.

4.2 The new EVALUATOR 3.0 model-checker

EVALUATOR 3.0 [24] is a new version of the EVALUATOR tool, which performs on-the-fly verification of regular alternation-free μ-calculus formulas on LTSs represented implicitly according to the OPEN/CÆSAR API. Compared to the previous version 2.0 [8], EVALUATOR 3.0 brings major improvements:

– The input specification language of EVALUATOR 3.0 is more powerful than the one of EVALUATOR 2.0:
  • action formulas can contain any combination of boolean operators and basic predicates over transition labels (which can be now given also as UNIX regular expressions over character strings);
  • regular transition sequences can be succinctly described using regular formulas built from action formulas and the usual regular expression operators;
  • it is also possible to define macro operators parameterized by formulas and to group them into separate libraries that may be included in the main specification.
– The model-checking algorithm of EVALUATOR 3.0 uses a new on-the-fly boolean resolution algorithm, which has a much better average complexity than the algorithm used in EVALUATOR 2.0. It explores less states before deciding the truth value of the formula, which leads sometimes to dramatic reductions (several orders of magnitude) of the execution time. Moreover, EVALUATOR 3.0 has been optimized in order to work more efficiently when verifying temporal formulas on explicit LTSs encoded as BCG files. Due to these optimizations, the memory consumption and the execution time of EVALUATOR 3.0 have been reduced by up to 5% and 20%, respectively.
– The diagnostics generated by EVALUATOR 3.0 are improved [22]. Diagnostics are portions of LTSs explaining either the satisfaction or the refutation of a formula: if the formula is false, a diagnostic is a counter-example; if the formula is true, a diagnostic is an example. In particular, the diagnostics obtained for derived "pure" branching-time logics like CTL and ACTL fully explain the semantics of their operators. EVALUATOR 3.0 may also serve to search regular execution sequences in the LTS, by asking for diagnostics of regular modalities.

Three libraries are also available that encode the operators of the ACTL temporal logic as well as a set of generic temporal property patterns defined by Prof. Matthew Dwyer from Kansas State University [5].

4.3 The new OCIS interactive graphical simulator

A new interactive, graphical simulator named OCIS (OPEN/CÆSAR *Interactive Simulator*) was added to CADP. Designed to replace the venerable XSIMULATOR, OCIS enables visualization and error detection during the design phase of systems containing parallelism and asynchronous communication between tasks. Its main features are:

- visualization of simulation scenarios as execution traces, trees, or Message Sequence Charts (MSCs),
- manipulation of simulation scenarios, which can be edited, saved as BCG graphs, and loaded again during another simulation session,
- manual (step by step) and automatic (pattern-guided) navigation in the system under simulation,
- source-level debugging, with access to parallel tasks, state variables, etc.
- possibility to modify the source code and to re-compile it without leaving the current simulation session.

OCIS was designed to be as much as possible language-independent and should therefore be usable for any specification language or formalism interfaced with the OPEN/CÆSAR API.

4.4 The new SVL scripting language

A new tool named SVL (*Script Verification Language*) [14] was added to CADP. The SVL language and its associated compiler target at simplifying and automating the verification of LOTOS programs. SVL behaves as a tool-independent coordination language on top of the CADP and FC2 tools, in the same way as EUCALYPTUS is a tool-independent graphical user interface.

SVL offers high-level operators for generation, parallel composition, minimization, label hiding, label renaming, abstraction, comparison, and model-checking of LTSs. It supports several methods of verification (e.g., enumerative, compositional, and on-the-fly), which can be easily combined together.

A compiler for SVL has been developed, which translates an SVL verification scenario into a Bourne shell script, which will perform all the operations needed to execute the verification scenario, e.g., invoking verification tools with appropriate options and parameters, generating intermediate files, etc.

SVL has been used in several case-studies: most of the CADP demo examples (19 demos over a total of 29) take advantage of SVL readability and conciseness. In most cases, SVL allows the user to get rid of Makefiles and shell-scripts as well as many auxiliary files which are generated automatically from a simple SVL script. Because of its expressiveness and robustness, SVL subsumes totally the DES2AUT tool [21] used in previous versions of CADP.

4.5 The new TGV test generator

The latest version of the TGV (*Test Generation based on Verification technology*) [20] tool, jointly developed by the PAMPA team of INRIA Rennes/IRISA and the VERIMAG laboratory, has been integrated into the CADP toolbox. TGV is a tool for the automatic generation of test suites from formal specifications. These test suites are used to assess the conformance of a protocol implementation with respect to the formal specification of this protocol. TGV takes two main inputs:

- a specification of the protocol's behavior, defined as an implicit LTS using the OPEN/CÆSAR API (this API allows TGV to be used for various languages: LOTOS, SDL, UML/RT, etc.),
- a test purpose, which selects the subset of the protocol's behavior to be tested; the test purpose is defined as an explicit LTS, the states of which are either normal states, accepting states (i.e., final states characterizing parts of the protocol satisfying the test purpose), or refusing states (i.e., final states characterizing parts of the behavior that are irrelevant to the test purpose).

To produce conformance test suites automatically, TGV applies algorithms coming from verification technology. Test generation is done "on-the-fly" on the synchronous product of the specification with the test purpose; this product allows to avoid state explosion by exploring only the subset of the protocol specification permitted by the test purpose. The test cases generated by TGV are LTSs, the transitions of which carry test verdicts such as "pass", "fail", and "inconclusive".

5 Conclusion

CADP contains a lot of tools and offers a wide variety of functionalities. CADP 2001 "Ottawa" provides several of the best algorithms for simulation and verification. It supports libraries that make the addition of new tools and the connection to new languages and description formats extremely modular. It also contains a graphical user interface and a scripting language that make its use easier for both expert and non-expert users.

Moreover, CADP is supported, maintained and constantly improved. It is available on LINUX, SOLARIS, and WINDOWS platforms. It is widely distributed: in June 2001, it had been licensed to 239 sites and during year 2000, licenses were granted for 770 machines around the world; from January 1st, to June 26th, 2001, licenses were granted for 797 machines. Additionally, the CADP tools are integrated into the Web-based, open communication platform ETI (Electronic Tool Integration Platform) [30, 2].

During the last years, over 50 applications and case-studies performed using CADP have been published[3] and 10 research tools based upon the OPEN/CÆSAR and BCG environments of CADP have been developed[4].

[3] Information is available at http://www.inrialpes.fr/vasy/cadp/case-studies.

[4] Information is available at http://www.inrialpes.fr/vasy/cadp/software.

Acknowledgements

We would like to thank all the people who contributed to the development of CADP:

- Moez Cherif, Hubert Garavel, Marc Herbert, Bruno Hondelatte, Pierre Kessler, Frédéric Lang, Stéphane Martin, Radu Mateescu, Aldo Mazilli, Frédéric Perret, Mihaela Sighireanu, and Irina Smarandache of the VASY project at INRIA Rhône-Alpes (Grenoble, France),
- Laurent Mounier and Aline Sénart of the VERIMAG laboratory (Grenoble, France),
- Thierry Jéron, Pierre Morel, and Séverine Simon of the PAMPA project at INRIA/IRISA (Rennes, France),
- Holger Hermanns at the University of Twente (The Netherlands).

We are also extremely grateful to all the scientists who contributed to the development of CADP in the past and who provided us with valuable feedback and advices about the use of CADP.

At last, we thank Solofo Ramangalahy for his useful comments about this paper.

References

1. Amar Bouali, Annie Ressouche, Valérie Roy, and Robert de Simone. The Fc2Tools set: a Toolset for the Verification of Concurrent Systems. 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*. Springer Verlag, August 1996.
2. Volker Braun, Jürgen Kreileder, Tiziana Margaria, and Bernhard Steffen. The ETI Online Service in Action. In Rance Cleaveland, editor, *Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS 1999)*, volume 1579 of *Lecture Notes in Computer Science*, pages 439–443, Amsterdam (The Netherlands), 1999.
3. R. E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, C-35(8), 1986.
4. E. Clarke, E. A. Emerson, and A. P. Sistla. Automatic Verification of Finite-State Concurrent Systems using Temporal Logic. In *10th Annual Symposium on Principles of Programming Languages*. ACM, 1983.
5. Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Patterns in Property Specifications for Finite-State Verification. In *Proceedings of the 21st International Conference on Software Engineering ICSE'99 (Los Angeles, CA, USA)*, May 1999.
6. Jean-Claude Fernandez. An Implementation of an Efficient Algorithm for Bisimulation Equivalence. *Science of Computer Programming*, 13(2–3):219–236, May 1990.
7. Jean-Claude Fernandez and Laurent Mounier. "On the Fly" Verification of Behavioural Equivalences and Preorders. In K. G. Larsen and A. Skou, editors, *Proceedings of the 3rd Workshop on Computer-Aided Verification (Aalborg, Denmark)*, volume 575 of *Lecture Notes in Computer Science*, Berlin, July 1991. Springer Verlag.

8. Jean-Claude Fernandez and Laurent Mounier. A Local Checking Algorithm for Boolean Equation Systems. Rapport SPECTRE 95-07, VERIMAG, Grenoble, March 1995.
9. Hubert Garavel. *Compilation et vérification de programmes LOTOS*. Thèse de Doctorat, Université Joseph Fourier (Grenoble), November 1989.
10. Hubert Garavel. Compilation of LOTOS Abstract Data Types. In Son T. Vuong, editor, *Proceedings of the 2nd International Conference on Formal Description Techniques FORTE'89 (Vancouver B.C., Canada)*, pages 147–162. North-Holland, December 1989.
11. Hubert Garavel. Binary Coded Graphs: Definition of the BCG Format. Rapport SPECTRE C28, Laboratoire de Génie Informatique — Institut IMAG, Grenoble, January 1991.
12. Hubert Garavel. An Overview of the Eucalyptus Toolbox. In Z. Brezočnik and T. Kapus, editors, *Proceedings of the COST 247 International Workshop on Applied Formal Methods in System Design (Maribor, Slovenia)*, pages 76–88. University of Maribor, Slovenia, June 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 Frédéric Lang. SVL: a Scripting Language for Compositional Verification. In Myungchul Kim, Byoungmoon Chin, Sungwon Kang, and Danhyung Lee, editors, *Proceedings of the 21st IFIP WG 6.1 International Conference on Formal Techniques for Networked and Distributed Systems FORTE'2001 (Cheju Island, Korea)*, pages 377–392. IFIP, Kluwer Academic Publishers, August 2001. Full version available as INRIA Research Report RR-4223.
15. Hubert Garavel, César Viho, and Massimo Zendri. System Design of a CC-NUMA Multiprocessor Architecture using Formal Specification, Model-Checking, Co-Simulation, and Test Generation. *Springer International Journal on Software Tools for Technology Transfer (STTT)*, 3(3):314–331, July 2001. Also available as INRIA Research Report RR-4041.
16. Jan Friso Groote and Frits Vaandrager. An Efficient Algorithm for Branching Bisimulation and Stuttering Equivalence. In M. S. Patterson, editor, *Proceedings of the 17th ICALP (Warwick)*, volume 443 of *Lecture Notes in Computer Science*, pages 626–638. Springer Verlag, 1990.
17. J.F. Groote and J.C. van Pol. State space reduction using partial tau-confluence. Research Report SEN-R0008, CWI, Amsterdam, The Netherlands, 2000.
18. M. Hennessy and R. Milner. Algebraic laws for nondeterminism and concurrency. *Journal of the ACM*, 32:137–161, 1985.
19. 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.
20. T. Jéron and P. Morel. Test generation derived from model-checking. In N. Halbwachs and D. Peled, editors, *Proceedings of the Conference on Computer-Aided Verification CAV'99 (Trento, Italy)*, volume 1633 of *Lecture Notes in Computer Science*, pages 108–122. Springer Verlag, July 1999.

21. Jean-Pierre Krimm and Laurent Mounier. Compositional State Space Generation from LOTOS Programs. In Ed Brinksma, editor, *Proceedings of TACAS'97 Tools and Algorithms for the Construction and Analysis of Systems (University of Twente, Enschede, The Netherlands)*, volume 1217 of *Lecture Notes in Computer Science*, Berlin, April 1997. Springer Verlag. Extended version with proofs available as Research Report VERIMAG RR97-01.
22. Radu Mateescu. Efficient Diagnostic Generation for Boolean Equation Systems. In Susanne Graf and Michael Schwartzbach, editors, *Proceedings of 6th International Conference on Tools and Algorithms for the Construction and Analysis of Systems TACAS'2000 (Berlin, Germany)*, volume 1785 of *Lecture Notes in Computer Science*, pages 251–265. Springer Verlag, March 2000. Full version available as INRIA Research Report RR-3861.
23. Radu Mateescu and Hubert Garavel. XTL: A Meta-Language and Tool for Temporal Logic Model-Checking. In Tiziana Margaria, editor, *Proceedings of the International Workshop on Software Tools for Technology Transfer STTT'98 (Aalborg, Denmark)*, pages 33–42. BRICS, July 1998.
24. 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.
25. R. De Nicola and F. W. Vaandrager. *Action versus State based Logics for Transition Systems.* In Irène Guessarian, editor, *Semantics of Systems of Concurrent Processes*, volume 469 of *Lecture Notes in Computer Science*, pages 407–419. Springer Verlag, April 1990.
26. Robert Paige and Robert E. Tarjan. Three Partition Refinement Algorithms. *SIAM Journal of Computing*, 16(6):973–989, December 1987.
27. Charles Pecheur. *Improving the Specification of Data Types in* LOTOS. Doctorate thesis, University of Liège, November 1996. Collection of Publications of the Faculty of Applied Sciences, Nr 171.
28. Jean-Pierre Queille and Joseph Sifakis. Fairness and Related Properties in Transition Systems — A Temporal Logic to Deal with Fairness. *Acta Informatica*, 19:195–220, 1983.
29. Valérie Roy and Robert de Simone. Auto/Autograph. In R. P. Kurshan and E. M. Clarke, editors, *Proceedings of the 2nd Workshop on Computer-Aided Verification (Rutgers, New Jersey, USA)*, volume 3 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 477–491. AMS-ACM, June 1990.
30. Bernhard Steffen, Tiziana Margaria, and Volker Braun. The Electronic Tool Integration Platform: Concepts and Design. *Springer International Journal on Software Tools for Technology Transfer (STTT)*, 1–2(1):9–30, December 1997.
31. B. Stepien, J. Tourrilhes, and J. Sincennes. ELUDO: The University of Ottawa LOTOS Toolkit. Technical report, University of Ottawa, 1994. Obtainable by FTP on lotos.csi.uottawa.ca.

Simulating Nondeterministic Systems at Multiple Levels of Abstraction

Dilsun Kırlı Kaynar, Anna Chefter, Laura Dean, Stephen J. Garland
Nancy A. Lynch, Toh Ne Win, Antonio Ramírez-Robredo
MIT Laboratory for Computer Science*

Abstract

IOA is a high-level distributed programming language based on the formal I/O automaton model for asynchronous concurrent systems. A suite of software tools, called the IOA toolkit, has been designed and partially implemented to facilitate the analysis and verification of distributed systems using techniques supported by the formal model. An important proof technique for distributed systems defined by a hierarchy of abstractions involves the notion of a simulation relation between pairs of automata at different levels in the hierarchy. The IOA toolkit's simulator tests purported simulation relations by executing the low-level automaton and, given a proposed correspondence between its steps and those of the higher-level automaton, generating and checking an execution of the higher-level automaton. Once checked by the simulator, the simulation relation and the step correspondence can be used in conjunction with the toolkit's proof tools to construct a formal proof that the low-level automaton implements the higher-level one. This paper presents a case study that illustrates this use of the IOA toolkit to prove correct an algorithm for mutual exclusion. The case study shows how tools like the IOA simulator can play an important role in proving distributed systems correct.

1 Introduction

The input/output (I/O) automaton model [LT89, Lyn96] is a labeled transition system model suitable for describing systems with asynchronously interacting components. In this model, a component is represented as an I/O automaton which is a nondeterministic, possibly infinite-state, state machine. The external behavior of each automaton is defined by a simple mathematical object called a trace.

The I/O automata model supports viewing systems at multiple levels of abstraction. A system can be described first at a high-level of abstraction, capturing only the essential requirements about its behavior, and then be refined successively until the desired level of detail is reached. The model defines what it means for an automaton to implement another (in terms of trace inclusion), and it introduces the notion of a simulation relation as a sufficient condition to prove an implementation relation between two automata. A parallel composition operator, also included in the model, allows one to decompose the description, analysis and verification of large and complex systems.

IOA [GL00, GL98] is a formal language for describing I/O automata. It can be regarded as a high-level distributed programming language. Its design was driven by a motivation to support both simulation [Che98, RR00, Dea01] and verification [Bog01]. The IOA toolkit is a partially

*Corresponding address: 200 Technology Square, Cambridge, MA 02139, USA, dilsun@theory.lcs.mit.edu. Currently, Chefter is employed by Merill Lynch, Dean is employed by Oryxa, and Ramírez is in the PhD program in mathematics at Stanford University.

implemented set of software tools that support the design, analysis, and development of systems within the I/O automaton framework. The toolkit contains a front-end that checks whether system descriptions (IOA programs) comply with IOA's syntax and static semantics, and that produces an intermediate representation of the code for use by the back-end tools (a simulator, interfaces to a number of existing theorem provers, model checkers, and an automatic code generator).

A key feature of the I/O automaton model is nondeterminism. Nondeterminism allows systems to be described in their most general forms and to be verified considering all possible behaviors without being tied to a particular implementation of a system design. The results obtained for a nondeterministic system carry over to different implementations of the same system. Nondeterminism also makes it easier to prove correctness in the absence of extraneous, unnecessary restrictions. A key challenge in the design of IOA has been to provide support for both simulation and verification in a unified framework. Nondeterminism in IOA assists verification in the ways noted above. On the other hand, nondeterminism complicates simulation, which must choose particular executions. Therefore, simulation requires mechanisms for resolving nondeterminism. The IOA language and toolkit provide such mechanisms. Moreover, these mechanisms turn out to be useful not just for simulation, but for verification as well.

In this paper, we describe by means of a case study how the IOA toolkit can be used for simulating and subsequently verifying distributed algorithms. We focus on the capability of the IOA simulator to simulate pairs of I/O automata at different levels of abstraction. Users present the paired simulator with descriptions of two automata, a candidate simulation relation, and a mapping, called a step correspondence, from the actions of the lower-level automaton to sequences of actions of the higher-level one. The simulator simulates the low-level automaton, checks whether the trace of the high-level automaton induced by the step correspondence is identical to that of the low-level automaton, and checks whether the candidate simulation relation holds throughout the simulated executions.

In our case study we present an algorithm for mutual exclusion and use the paired simulator to obtain evidence that this algorithm satisfies the mutual exclusion property. We then verify that the algorithm satisfies this property with LP [GG91]. The toolkit facilitates the automatic translation of the algorithm and the candidate simulation relation into the language of LP.

Related work Other toolkits such as AsmL [GSV01] tools, Mocha [dAAG^{+}00], the SMV system [McM], and TLC [LY01] support simulation or verification of concurrent and distributed systems. The IOA toolkit differs from these in that it combines paired simulation capability with theorem-proving based verification. AsmL facilitates simulating systems at different levels of abstraction, checking step by step whether a system satisfies its specification, but it does not support using paired simulation in conjunction with proof tools. The verification components of Mocha, SMV, and TLC use model checking and hence are limited to exploring finite state spaces; the proof tools in the IOA toolkit apply to finite and infinite systems alike. Another feature that distinguishes the IOA toolkit from other tools is the connection of its simulator to a program analysis tool [ECGN01] for automatic invariant discovery.

2 I/O automata and the IOA Language

This section includes a brief introduction to the I/O automaton model and the IOA Language. We refer the reader to [Lyn96, GL98] for an in-depth introduction.

2.1 Theoretical background

An I/O automaton is a simple type of state machine in which the transitions between states are associated with named *actions* π. The actions are classified as either *input, output*, or *internal.* The input actions are assumed not to be under the automaton's control, whereas the automaton itself controls which output and internal actions should be performed. An I/O automaton consists of a *signature*, which lists its actions, a set of *states*, some of which are distinguished as start states, a *state-transition relation*, which contains triples of the form (state, action, state), and an optional set of *tasks*. We do not consider automata with tasks in this paper.

An action π is said to be enabled in a state s if there is another state s' such that (s, π, s') is a transition of the automaton. Input actions are enabled in every state. The operation of an I/O automaton is described by its *executions* $s_0, \pi_1, s_1, \ldots$, which are alternating sequences of states and actions, and its *traces*, which are the externally visible behavior occurring in executions. One automaton is said to *implement* another if all its traces are also traces of the other. The *parallel composition* operator allows an output action of one automaton to be identified with input actions in other automata; this operator respects the trace semantics.

The I/O automaton model provides support for system descriptions at multiple levels of abstraction. The process moving through a series of abstractions, from higher to lower levels, is called *successive refinement.* To prove that one automaton implements another, one needs to show that for any execution of the lower level automaton there is a corresponding execution of the higher level automaton. The notion of a *simulation relation* proves useful in constructing proofs of implementation relations.

Definition 2.1 (Forward simulation). A forward simulation from automaton A to automaton B is a relation f on $states(A) \times states(B)$ with the following properties:

1. For every start state a of A, there exists a start state b of B such that $f(a, b)$.
2. If a is a reachable state of A, b is a reachable state of B such that $f(a, b)$, and $a \xrightarrow{\pi} a'$, then there exists a state b' of B and an execution fragment β of B such that $b \xrightarrow{\beta} b'$, $f(a', b')$ holds, and $trace(\pi) = trace(\beta)$.

Theorem 2.1. If there is a forward simulation relation from A to B, then every trace of A is a trace of B. (See [Lyn96] for a proof.)

2.2 The IOA language

In the IOA language, the description of an I/O automaton has four main parts: the action signature, the states, the transitions, and the tasks of the automaton. States are represented by collections of typed variables. The transition relation is usually given in precondition-effect style, which groups together all transitions that involve a particular action into a single piece of code. Each definition has a precondition (indicated by the keyword **pre**), which describes a condition on the state that should be true before the transition can be executed, and an effect (indicated by the keyword **eff**) which describes how the state changes when the transition is executed. The entire piece of code in the effect of a transition is executed indivisibly. If **pre** is not specified, then it is assumed to always hold.

The code may be written either in an imperative style, as a sequence of assignment, conditional, and looping instructions, or in declarative style, as a predicate relating state variables in the pre- and post-states, transition parameters, and nondeterministic parameters. It is also possible to use a combination of these two styles.

Nondeterminism appears in IOA in two ways: *explicitly* in the form of **choose** constructs in state variable initializations and the effects of the transition definitions, and *implicitly*, in the form of action scheduling uncertainty. We present examples for both forms of nondeterminism later in the paper and describe how they are resolved by the IOA simulator.

2.3 Example: Specification of mutual exclusion

We present a sample IOA program to illustrate some of the language constructs discussed above and to introduce the mutual exclusion problem that constitutes the basis of our case study. We build on this example gradually as we discuss simulation and proof techniques based on simulation relations.

The mutual exclusion problem involves the allocation of a single, indivisible, non-shareable resource among n processes. The resource could be an output device that requires exclusive access to produce sensible output or a data structure that requires exclusive access in order to avoid interference among the operations of different processes. A process with access to the resource is modeled as being in a *critical region*, which is a designated subset of its states. When a process is not involved in any way with the resource, it is said to be in the *remainder region*. In order to gain admittance to its critical region, a process executes a *trying protocol*; after it is done with the resource, it executes an *exit protocol*. This procedure can be repeated so that each process follows a cycle, moving from its remainder region to its trying region and arriving back at the remainder region after going through critical and exit regions.

We consider mutual exclusion within the shared memory model explained in [Lyn96]. The shared memory system contains n processes, numbered $1, \ldots, n$. The $try, crit, exit$, and rem actions are the only external actions of a process. Input actions consist of try_i, which models a request for access to the resource by process i, and $exit_i$, which models an announcement that process i is done with the resource. Output actions consist of $crit_i$, which models the granting of access to process i, and rem_i, which tells process i that it can continue with the remainder of its work. Formally, we define a sequence of $try_i, crit_i, exit_i$, and rem_i actions to be well-formed for process i if it is a prefix of the cyclically ordered sequence $try_i, crit_i, exit_i, rem_i, try_{i'}, \ldots$ The automaton `Mutex` (Figure 1) is an IOA specification of mutual exclusion for three processes in which the well-formedness of interaction with the environment is guaranteed.

The state variable `regionMap` maps process indices to regions and keeps track of the current region of each process. The initialization of `regionMap` to `constant(rem)` defines the start state. The transition definitions are mostly self-explanatory. Each action updates the variable `regionMap` to record the region entered upon its execution. The transition definition for `crit` imposes the mutual exclusion condition: a process in a trying region is allowed to enter its critical region only if there is no other process that is in region `crit`.

2.4 Example: An algorithm for mutual exclusion

Figure 2 contains IOA code for an algorithm for mutual exclusion. Comments in the code indicate items that will be of particular interest when we discuss the mechanism for resolving nondeterminism to enable simulation. We start, however, by explaining the algorithm briefly, pointing at the sources of nondeterminism.

The algorithm described by the automaton `DijkstraInt` is a simplified version of a mutual exclusion algorithm by Dijkstra presented in [Lyn96]. It abstracts away those parts in the original algorithm dedicated to dealing with liveness. The suffix "Int" in the automaton name indicates that we consider it to be an intermediate level algorithm: not at as high a level as the specification,

```
type Index =enumeration of p1, p2, p3
type Region =enumeration of rem, try, crit, exit

automaton Mutex
  signature output try(p: Index), crit(p: Index), exit(p: Index), rem(p: Index)
  states regionMap: Array[Index, Region] := constant(rem)
  transitions
    output try(p)
      pre regionMap[p] =rem
      eff regionMap[p] := try
    output crit(p)
      pre regionMap[p] =try ∧∀u: Index (p ≠u ⇒regionMap[u] ≠crit)
      eff regionMap[p] := crit
    output exit(p)
      pre regionMap[p] =crit
      eff regionMap[p] := exit
    output rem(p)
      pre regionMap[p] =exit
      eff regionMap[p] := rem
```

Figure 1: Specification of mutual exclusion

yet less detailed than the original algorithm of Dijkstra.

The automaton `DijkstraInt` uses two types, `PcValue` and `Stage`, in addition to those in Figure 1. Values of type `PcValue` represent possible program counter values for a process, while values of type `Stage` represent stages of the algorithm. The automaton has four external and four internal actions. The external actions have the same names as those of `Mutex` in Figure 1. This is no coincidence, as our ultimate aim is to show that `DijkstraInt` implements mutual exclusion as specified by `Mutex`.

The algorithm has two stages. The first, `stage1`, indicates that a process is either inactive or is about to enter the second stage. The second, `stage2`, embodies the crucial steps and determines whether a process is allowed to enter its critical region. A process can enter its critical region only if all other processes are in `stage1`. The transition definition for action `check` details how this works. Each process p uses a set $S[p]$ to keep track of the processes that it has detected as being in `stage1`. The state variables `flag` and `pc` record the stage of the algorithm for each process and control the order of occurrence of the actions mimicking the program counter for a process.

Explicit nondeterminism in this example arises from the **choose** statement in the transition definition for action `check`. When a process p performs the `check` action, it nondeterministically chooses the process u to be checked. The predicate in the **where** clause allows the nondeterministic choice to yield any process that is not already in the set $S[p]$. Implicit nondeterminism also arises in this example, because there may be more than one action enabled at a time. Consider, for example, the very first action to be performed by the automaton. Since the program counters (`pc`) of all processes are initialized to `rem`, all processes are enabled to perform the `try` action. To simulate this automaton, one must select one of these processes to start execution.

3 Simulation and nondeterminism resolution

The simulator runs sample executions of an IOA program, allowing the user to help select the executions. It generates logs of execution traces and displays information upon the user's request. The IOA Language allows users to propose invariants, which the simulator checks in the selected executions.

```
type PcValue =enumeration of rem, setflag1, setflag2, check, leavetry,
                             crit, reset, leaveexit
type Stage =enumeration of stage1, stage2

automaton DijkstraInt
  signature
    output try(p: Index), crit(p: Index), exit(p: Index), rem(p: Index)
    internal setflag1(p: Index), setflag2(p: Index), check(p: Index), reset(p: Index)
  states
    flag: Array[Index, Stage] := constant(stage1),
    pc: Array[Index, PcValue] := constant(rem),
    S: Array[Index, Set[Index]] := constant({}),
    u: Index
  transitions
    output try(p)
      pre pc[p] =rem
      eff pc[p] := setflag1
    internal setflag1(p)
      pre pc[p] =setflag1
      eff flag[p] := stage1; pc[p] := setflag2
    internal setflag2(p)
      pre pc[p] =setflag2
      eff flag[p] := stage2; S[p] := {p}; pc[p] := check
    internal check(p)
      pre pc[p] =check
      eff u := choose x: Index where ¬(x ∈S[p]);
              %% explicit nondeterminism to be resolved for simulation
          if flag[u] =stage2 then S[p] := {}; pc[p] := setflag1
          else S[p] := S[p] ∪{u};
               if ∀ i: Index (i ∈S[p]) then pc[p] := leavetry fi
          fi
    output crit(p)
      pre pc[p] =leavetry
      eff pc[p] := crit
    output exit(p)
      pre pc[p] =crit
      eff pc[p] := reset
    internal reset(p)
      pre pc[p] =reset
      eff flag[p] := stage1; S[p] := {}; pc[p] := leaveexit
    output rem(p)
      pre pc[p] =leaveexit
      eff pc[p] := rem
    %% implicit nondeterminism to be resolved for simulation
```

Figure 2: An algorithm for mutual exclusion

The simulator requires that IOA programs be transformed into a form suitable for simulation. The crucial problem in this transformation is resolving nondeterminism. The nondeterminism resolution approach adopted by the IOA simulator is to assign a program, called an *NDR program*, to each source of nondeterminism in an automaton. There is an NDR program corresponding to every **choose** statement, and an NDR program for scheduling the actions of the automaton. We explain the nondeterminism resolution mechanism of the IOA simulator by referring to the example presented in Section 2.4.

3.1 Resolving explicit nondeterminism

A simple NDR program (**det**erminator), given below, resolves the explicit nondeterminism for the `check` action in the automaton `DijkstraInt`. It **yield**s a process index that is not in $S[p]$. This index is guaranteed to differ from p because p is placed in $S[p]$ before `check` is enabled, and it is guaranteed to exist because `check` is no longer enabled once $S[p]$ contains all indices.

```
det do
      if ¬(p1 ∈S[p]) then yield p1
      elseif ¬(p2 ∈S[p]) then yield p2
      elseif ¬(p3 ∈S[p]) then yield p3
      fi
   od
```

3.2 Resolving implicit nondeterminism

To resolve implicit nondeterminism, users of the IOA simulator must specify a scheduling policy using the language constructs of IOA. We present below a sample schedule block that implements a randomized scheduling policy for three processes. It picks a random integer between 1 and 3 and uses this integer to decide which process will be given the turn to perform an action. It checks the enabling conditions for the randomly chosen process and **fire**s the enabled action. The **while** loop that contains these steps is nonterminating; the IOA simulator prompts the users for the maximum number of steps to simulate and halts the execution automatically when the predetermined step is reached.

```
schedule
  states pick: Int, p: Index
  do while true do
       pick:= randomInt(1,3);
       if pick =1 then p := p1
       elseif  pick =2 then p := p2
       else p := p3
       fi;
       if pc[p] =rem then fire output try(p)
       elseif pc[p] =setflag1 then fire internal setflag1(p)
       elseif pc[p] =setflag2 then fire internal setflag2(p)
       elseif pc[p] =check then fire internal check(p)
       elseif pc[p] =leavetry then fire output crit(p)
       elseif pc[p] =crit then fire output exit(p)
       elseif pc[p] =reset then fire internal reset(p)
       else  fire output rem(p)
       fi
    od
 od
```

3.3 Checking invariants

The IOA simulator checks the validity of invariants proposed by users. We present below several invariants for the automaton `DijkstraInt` that are key lemmas for proving the algorithm correct. In Section 5 we take up the question of how the user discovers such lemmas.

Each process p uses a set $S[p]$ to keep track of successfully checked processes, that is, of processes that are not contending with p to enter the critical region. The first assertion states that two processes cannot both be executing the second stage of the algorithm and be in each other's set. The second states that whenever the `pc` value for a process is `leavetry` or `crit`, its set contains all of the processes. These two assertions express the key ideas we will use in our proof: if the `pc` values for two processes were `crit` at the same time, it would be impossible for `assertion1` and `assertion2` to be both true.

```
invariant assertion1 of DijkstraInt:
  ∀ i: Index ∀j: Index ¬(i ≠j ∧flag[i] =stage2 ∧flag[j] =stage2
                          ∧ i ∈S[j] ∧j ∈S[i])

invariant assertion2 of DijkstraInt:
  ∀ i: Index (  (pc[i] =leavetry ⇒∀j:Index (j ∈S[i]))
             ∧ (pc[i] =crit    ⇒∀ j:Index (j ∈S[i])) )
```

3.4 Simulator output

The automaton `DijkstraInt` from Figure 2 can be simulated with the IOA simulator after inserting the NDR programs specified in Section 3 in the indicated places. The invariants to be checked need to be appended to the code.

Some output of the simulator for running `DijkstraInt` is shown below. It displays the step involving the first entry to the critical region (step 21) in a simulation for 200 steps. The simulator reports errors if any of the invariants fail at a simulated step, if an NDR program attempts to fire a transition that is not enabled, or if it attempts to yield a value that does not satisfy the **where** clause of the corresponding **choose** statement.

```
[[[[ Begin step 21 [[[[
     transition: output crit(p1) in automaton DijkstraInt
%%%% Modified state variables:
     pc --> (ArraySort (ConstantValue rem) (p1 crit) (p2 setflag2) (p3 rem))
]]]] End step 21 ]]]]
```

4 Paired Simulation

In this section, we describe how the simulator simulates execution of a pair of automata related by a simulation relation as defined in Section 2. The key problem here is that simulation relation, being merely a predicate that relates the states of two automata, does not identify how each step in the implementation automaton corresponds to a sequence of steps in the specification automaton. In general, there might be multiple step correspondences that realize a given valid simulation relation between automata; even if there is only one, it can be difficult to find it. The problem of deriving a specification-level execution from an implementation-level execution is analogous to that of deriving a deterministic execution of a single automaton from a specification that allows nondeterminism.

The design of the paired simulator is based on the observation that it is reasonable and beneficial to require users to specify a step correspondence. In most correctness proofs, determining when a particular action in the specification is performed by the implementation turns out to be the key to the proof. By requiring a user to specify the step correspondence, the simulator actually urges the user to understand the relationship between the two levels. Once the main invariants and the step correspondence is determined, the rest of the proof is likely to involve routine bookkeeping steps.

4.1 Encoding step correspondences

A step correspondence needs to specify, for a given low-level transition, a high-level execution fragment such that execution of both the low-level transition and the high-level fragment preserves the simulation relation. Thus, a step correspondence can be seen as an "attempted proof" of the simulation relation, missing only the reasoning that shows that the simulation relation is preserved. To specify the proposed proof of a simulation relation, the IOA **forward simulation** assertion allows a section called **proof** for specifying the step correspondence. This section contains one entry for each possible transition definition in the low-level automaton; each entry provides an algorithm for producing a high-level execution fragment. In addition to these entries, the **proof** section contains an initialization block, which specifies how to set the variables of the high-level automaton given the initial state of the low-level automaton, and an optional `states` section that declares auxiliary variables used by the step correspondence.

4.2 Example: Forward simulation from `DijkstraInt` to `Mutex`

Figure 3 defines a forward simulation relation in IOA and contains a proof block for that relation. Together with the IOA descriptions of `Mutex` and `DijkstraInt` augmented with the NDR programs from Section 3, this block allows one to use the paired simulator to check whether the relation holds in the simulated executions.

The candidate relation in this example is based on the relation between the values of the state variable `pc` of the low-level automaton and those of the state variable `regionMap` of the specification automaton. The intuition behind this relation is as follows. For each region in the specification of mutual exclusion there are certain actions that can be performed by the low-level automaton. These actions are determined by the `pc` values. The relation states that whenever the program counter of a process at the low-level automaton is set to one of `setflag1`, `setflag2`, `check`, or `leavetry`, the `regionMap` of the specification automaton must show region `try` for the same process. The rest of the relation is defined similarly. The delimiter ";" can be interpreted as conjunction.

In paired simulation, the simulation of the low-level algorithm drives the simulation of the high-level one. For each external action performed by the low-level automaton, the proof block directs the simulator to fire the action with the specified name at the high-level. The internal actions are matched by empty execution fragments indicated by **ignore** statements. The simulator checks whether the proposed simulation relation holds after the actions are performed. The following is a sample output of the paired simulator, displaying the simulation step 17.

```
[[[[ Begin step 17 [[[[
     Executed impl transition: output crit(p1) in automaton DijkstraInt
%%%% Modified state variables for impl automaton:
     pc --> (ArraySort (ConstantValue rem) (p1 crit) (p2 setflag2) (p3 setflag2))
     Executed spec transition: output crit(p1) in automaton Mutex
%%%% Modified state variables for spec automaton:
     regionMap --> (ArraySort (ConstantValue rem) (p1 crit) (p2 try) (p3 try))
]]]] End step 17 ]]]]
```

Note that the simulator gives information about how the states of the two automata change upon the occurrence of an action of the implementation automaton. In this example, each step

```
forward simulation from DijkstraInt to Mutex :
  ∀ i: Index (DijkstraInt.pc[i] =setflag1 ∨DijkstraInt.pc[i] =setflag2 ∨
            DijkstraInt.pc[i] =check ∨DijkstraInt.pc[i] =leavetry
              ⇔ Mutex.regionMap[i] =try);
  ∀ i: Index (DijkstraInt.pc[i] =crit ⇔Mutex.regionMap[i] =crit);
  ∀ i: Index (DijkstraInt.pc[i] =rem ⇔Mutex.regionMap[i] =rem);
  ∀ i: Index (DijkstraInt.pc[i] =reset ∨DijkstraInt.pc[i] =leaveexit
              ⇔ Mutex.regionMap[i] =exit);
proof
    initially Mutex.regionMap := constant(rem)
    for output try(p:Index) do fire output try(p) od
    for output crit(p:Index) do fire output crit(p) od
    for output exit(p:Index) do fire output exit(p) od
    for output rem(p:Index) do fire output rem(p) od
    for internal setflag1(p:Index) ignore
    for internal setflag2(p:Index) ignore
    for internal check(p:Index) ignore
    for internal reset(p:Index) ignore
```

Figure 3: Forward simulation from `DijkstraInt` to `Mutex`

in the low-level execution is matched by either a single step or an empty execution fragment in the specification. The IOA simulator can also handle paired simulations in which this is not the case. It allows execution fragments to be specified by any IOA program consisting of assignments, conditional, while, and fire statements. For example, a step correspondence in which an output action `a` at the low-level is matched by a sequence consisting of an output action `a` that is preceded and followed by an internal action `b` could be encoded as follows:

```
for output a do fire internal b; fire output a; fire internal b od
```

5 Using simulation results to help construct a proof of correctness

In the previous section we introduced a method for simulating pairs of automata at different levels of abstraction with the aid of the IOA toolkit. It is important to note that paired simulation provides only empirical evidence for the correctness of a simulation relation. In most cases it is desirable to complement this evidence with a proof. In this section we describe the support provided by the IOA toolkit for formal verification.

5.1 Method

The IOA toolkit has been designed to support verification of *safety properties*, which specify that a "bad' event never happens. LP is an interactive theorem proving system for multisorted first-order logic and is suitable for reasoning about safety properties expressible in this kind of logic. It admits specifications of theories in the Larch Shared Language (LSL). The IOA toolkit includes a tool called `ioa2lsl` [Bog01], which translates IOA definitions of automata, their invariants, and simulation relations into LSL theories. The tool `ioa2lsl` combines the definition of an automaton with standard LSL definitions of I/O automata to produce axioms in first-order logic that describe the operation of the automaton. These are subsequently used to generate input for LP.

5.2 Example: Proof of forward simulation

We now describe how we proved that a candidate simulation relation, presented in Figure 3 and checked with the paired simulator for selected executions, is actually a forward simulation relation from `DijkstraInt` to `Mutex`. It then follows from Theorem 2.1 that `DijkstraInt` implements mutual exclusion.

We first used `ioa2lsl` to process the file containing the definitions of the two automata, their invariants, and the simulation relation.[1] We then used the LSL Checker to prepare the axioms and proof obligations for LP.

The proof of the simulation relation proceeds by induction. The basis step consists of showing that the relation holds for the start state. The proof of the induction step takes the form of proof by cases. The heart of the proof lies in providing a "witness" for an existential quantifier asserting the existence of a simulating step sequence in the high-level automaton that preserves the simulation relation and has the the same trace as a given step of the low-level automaton. The step sequence already constructed for the paired simulator turns out to be exactly what is needed to provide this witness.

Proofs of the invariants were routine proofs by induction. The proof of `assertion1` gave rise to the need to prove two other simpler invariants:

```
invariant assertion3 of DijkstraInt:
  ∀ i: Index (pc[i] =leavetry ⇒flag[i] =stage2)

invariant assertion4 of DijkstraInt:
  ∀ i: Index (pc[i] =crit ⇒flag[i] =stage2)
```

5.3 Automatic detection of invariants

Finding key invariants is an essential step in proofs of correctness. Any help from automatic tools in finding these invariants would alleviate the burden on the user. For example, if a tool could discover simple invariants such as `assertion3` and `assertion4`, which LP can prove more or less automatically, and if LP could use these to prove the invariants `assertion1` and `assertion2` used in the correctness proof, that proof would become much easier.

We have begun [WE02] developing this kind of automated proof assistance by connecting the IOA simulator to Daikon [ECGN01], a tool for dynamic invariant discovery. The user can instruct the IOA simulator to record the values of state variables upon entry to and exit from each transition in the course of a selected execution. Then Daikon can infer invariants about the pre-state and post-state of each transition by examining these values.

In our preliminary experiments, Daikon was able to infer some potentially useful invariants. For example, Daikon detected that `flag[p]=stage2` in the pre-state of `crit(p)`. The invariant `assertion3` in the previous is just the implication of this invariant by the precondition of the `crit` action. We are continuing to work on the Daikon-IOA connection to detect other useful invariants and to automate the formulation of invariants such as `assertion3`.

6 Overview of the implementation

A preliminary "IOA toolkit distribution" (software package including source and Java executables) is available from the home page of the IOA project (`http://theory.lcs.mit.edu/tds/ioa.html`).

[1]The tool `ioa2lsl` is still under development, and we had to edit its output to correct a number of small errors.

The front-end of the toolkit takes IOA descriptions and LSL specifications as input and outputs an equivalent specification written in an intermediate language. Each back-end tool takes as input the intermediate form of an IOA specification. There is common support for the back-end tools in the form of an intermediate language parser and an internal representation of IOA elements, in the form of a Java class hierarchy.

Data types are defined axiomatically in IOA so as to facilitate their translation into theorem prover input languages. We provide definitions for built-in data types and allow the programmer to define new data types using LSL. However, in order to simulate data type operations, the simulator needs actual code for the specified operations. Each IOA sort is implemented by a Java class, and each operator is implemented by a method on that class. The implementation classes extend the `ioa.runtime.ADT` class, which provides two operators common to all IOA data types. The simulator obtains implementations for sorts and operators by querying a global *implementation registry*.

The simulator shares runtime type libraries with the IOA code generator to ensure similar code behavior and to reduce repeated code [Tsa02].

7 Discussion and conclusions

Formal correctness proofs for distributed systems can be long, hard, or tedious to construct. Simulation can be used as a way of testing system designs before delving into correctness proofs. It either reveals bugs or increases confidence that a system behaves as expected. Simulation can also assist users in constructing correctness proofs. It is this aspect of simulation that we focused on throughout this paper.

We considered nondeterministic systems modeled using the I/O automata formalism and described how these systems can be simulated with the support of the IOA language and the toolkit. Our aim was to draw attention to a useful capability of the IOA simulator – paired simulation – that allows users to check whether two automata at different levels of abstraction are related by a simulation relation for the selected executions. In the I/O automaton model, the notion of a simulation relation between two automata is a useful conceptual tool to prove the correctness of systems. Hence, the ability to propose and check simulation relations with the IOA simulator constitutes a valuable step towards a formal proof based on a simulation relation. The specification of a relation is not the only thing that is required from a user by the paired simulator. A user is also required to specify a step correspondence that will make the simulation relation hold throughout paired simulation. This is particularly useful since finding the right step correspondence is usually the key to the proof of a simulation relation. This indeed happened in our case study.

Another capability of the IOA simulator that helps the construction of proofs is invariant checking. The invariants that are observed to be true for simulated executions constitute candidates for useful lemmas. The invariants that we checked with the paired simulator in our case study were later used as lemmas in the full proof.

The case study in this paper suggests a general methodology for the analysis and verification of distributed systems with the IOA toolkit, using multiple levels of abstraction. The basic steps are to:

1. Write the IOA code for the specification and the implementation automata;
2. For each automaton, resolve nondeterminism and perform simulation to test that the automaton behaves as expected;
3. Formulate a candidate forward simulation relation from the implementation automaton to the specification automaton, specify a step correspondence and perform paired simulation to

check whether the relation holds for the selected executions;

4. Formulate the potentially useful invariants for the proof of the simulation relation and check whether they are true for the selected executions;
5. Use the tool ioa2lsl to translate the IOA code for automata and the forward simulation relation to LSL, and to generate proof obligations for LP; and
6. Prove with LP that the simulation relation holds for all possible executions, making use of the step correspondence and the key invariants.

A current project aims at improving the connection between the program analysis tool Daikon and the IOA simulator. We expect this connection to contribute to this methodology by automating parts of the correctness proofs.

References

[Bog01] Andrej Bogdanov. Formal verification of simulations between I/O automata. Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA, 2001.

[Che98] Anna E. Chefter. A simulator for the IOA language. Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA, May 1998.

[dAAG+00] L. de Alfaro, R. Alur, R. Grosu, T. Henzinger, M. Kang, R. Majumdar, F. Mang, C. Meyer-Kirsch, and B.Y. Wang. *Mocha: Exploiting Modularity in Model Checking.* University of California at Berkeley Department of Electrical Engineering and Computer Sciences, University of Pennsylvania Department of Computer and Information Sciences, 2000. URL `http://www-cad.eecs.berkeley.edu/~mocha/refs.shtml`.

[Dea01] Laura G. Dean. Improved simulation of Input/Output automata. Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA, September 2001.

[ECGN01] Michael Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):1–25, 2001.

[GG91] Stephen Garland and John Guttag. *A guide to LP, the Larch Prover.* Technical report, DEC Systems Reserach Center, 1991. Updated version avaliable at URL `http://nms.lcs.mit.edu/Larch/LP`.

[GL98] Stephen J. Garland and Nancy A. Lynch. The IOA language and toolset: Support for designing, analyzing, and building distributed systems. Technical Report MIT/LCS/TR-762, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA, August 1998. URL `http://theory.lcs.mit.edu/tds/papers/Lynch/IOA-TR-762.ps`.

[GL00] Stephen J. Garland and Nancy A. Lynch. Using I/O automata for developing distributed systems. In Gary T. Leavens and Murali Sitaraman, editors, *Foundations of Component-Based Systems*, chapter 13, pages 285–312. Cambridge University Press, USA, 2000.

[GSV01] Yuri Gurevich, Wolfram Schulte, and Margus Veanes. Toward industrial strength abstract state machines. Technical Report MSR-TR-2001-98, Microsoft Research, 2001. URL for software `http://www.research.microsoft.com/foundations/asml/`.

[LT89] Nancy A. Lynch and Mark R. Tuttle. An introduction to input/output automata. *CWI-Quarterly*, 2(3):219–246, September 1989. Centrum voor Wiskunde en Informatica, Amsterdam, The Netherlands. Technical Memo MIT/LCS/TM-373, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA 02139, November 1988.

[LY01] Leslie Lamport and Yuan Yu. *TLC – The TLA+ Model Checker.* Compaq Systems Research Center, Palo Alto, California, 2001. URL `http://research.microsoft.com/users/lamport/tla/tlc.html`.

[Lyn96] Nancy Lynch. *Distributed Algorithms.* Morgan Kaufmann Publishers, Inc., San Mateo, CA, March 1996.

[McM] K. L McMillan. *The SMV Language.* Cadence Berkeley Labs, 2001 Addison Street, Berkeley, CA 94 704, USA. URL `http://www.cis.ksu.edu/santos/smv-doc/`.

[RR00] J. Antonio Ramirez-Robredo. Paired simulation of I/O automata. Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA, September 2000.

[Tsa02] Michael Tsai. Code generation for the IOA language. Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA, June 2002.

[WE02] Toh Ne Win and Michael Ernst. Verifying distibuted algorithms via dynamic analysis and theorem proving. Technical Report MIT-LCS-TR-841, MIT Laboratory for Computer Science, May 2002.

A Proof Script

The following is the Larch proof script for the simulation relation presented in Section 4. The set of axioms `DijkstraInt2Mutex_Axioms` is generated by the LSL checker by processing the LSL specification of the simulation relation. The tactics that are referred to in the proof are given in Section A.2.

A.1 Main simulation proof

```
execute DijkstraInt2Mutex_Axioms

declare variables u': States[DijkstraInt], act: Actions[DijkstraInt], pi: ActionSeq[Mutex]


set name theorem
prove start(u:States[DijkstraInt]) => \E s:States[Mutex] (start(s:States[Mutex]) /\ F(u, s))
  resume by specializing s:States[Mutex] to [constant(rem)]
    execute tactic_implies
qed

prove
  isStep(u:States[DijkstraInt], act, u') /\ F(u, s)
  /\ assertion1(u) /\ assertion2(u) /\ assertion3(u) /\ assertion4(u)
    => \E pi:ActionSeq[Mutex] (execFrag(s, pi) /\ trace(pi:ActionSeq[Mutex]) = trace(act)
                               /\ first(s, pi) = s /\ F(u', last(s, pi)))
  ..
  resume by induction on act
    % try action
    resume by =>
      resume by specializing pi to try(i1c) * {}
        execute tactic_and4cases
    % crit action
    resume by =>
      resume by specializing pi to crit(i1c) * {}
        resume by /\
```

```
        critical-pairs *Hyp with *Hyp
        execute tactic_4cases
        resume by =>
          resume by contradiction
            critical-pairs *Hyp with *Hyp
  % exit action
  resume by =>
    resume by specializing pi to exit(i1c) * {}
      execute tactic_and4cases
  % rem action
  resume by =>
    resume by specializing pi to rem(i1c) * {}
      resume by /\
        critical-pairs *Hyp with *Hyp
        execute tactic_4cases
  % setflag1 action
  resume by =>
    resume by specializing pi to {}
      critical-pairs *Hyp with *Hyp
      execute tactic_and4cases
  % setflag2 action
  resume by =>
    resume by specializing pi to {}
      critical-pairs *Hyp with *Hyp
      execute tactic_and4cases
  % check action
  resume by =>
    resume by specializing pi to {}
      critical-pairs *Hyp with *Hyp
      resume by /\
        execute tactic_stage2_i2c
        execute tactic_stage2_i2c
        execute tactic_stage2_i2c
        execute tactic_stage2_i2c
  % reset action
  resume by =>
    resume by specializing pi to {}
      critical-pairs *Hyp with *Hyp
      exe tactic_and4cases
qed
quit
```

A.2 Tactics

```
  % tactic_implies
  resume by =>

  % tactic_case
  res by case i1c = i

  % tactic_4cases
  execute tactic_case
  execute tactic_case
  execute tactic_case
  execute tactic_case

  % tactic_and4cases
  resume by /\
```

```
      execute tactic_4cases

% tactic_stage2_i2c.lp
resume by case uc.flag[i2c] = stage2
        execute tactic_case
        resume by case \A i:Index (i = i2c \/ i \in uc.S[i1c])
          execute tactic_case
```

The Parallel PV Model-Checker

Robert Palmer and Ganesh Gopalakrishnan*
{rpalmer,ganesh}@cs.utah.edu,
http://www.cs.utah.edu/formal_verification/

University of Utah, School of Computing

Abstract. Parallel PV is based on the sequential PV model-checker. Sequential PV is an depth-first LTL-X model-checker for an enhanced subset of the Promela language. Parallel PV is a breadth-first safety-only model-checker. It capitalizes on PV's two-phase partial-order reduction algorithm by carrying out partial order reduction steps with no communication, and performs state space distribution at global steps. This helps reduce the number of messages exchanged. Also, based on state ownership information, parallel PV reduces the number of states that are cached. This reduction is in addition to the selective state caching supported by sequential PV. We report encouraging preliminary experimental results drawn from the domain of 'hardware' protocols, as well as software models generated by the Bandera tool. Implementation details of parallel PV and setup information are also provided.

1 Introduction

Parallel processing can help speed-up model-checking by providing many CPUs that can work on the problem, higher aggregate memory capacity, as well as higher memory bandwidth. While the problem remains exponential, the overall performance of the model-checking algorithm can increase by significant factors. In this paper, we focus on enumerative model-checkers which are widely used to verify many classes of models, such as high-level cache coherence protocols and Java software models, in which models containing thousands of variables and involving global dependencies – such as processor IDs passed around in messages – are to be verified. Such models have never been shown to be capable of being efficiently represented or manipulated using BDDs. There are many previous attempts to parallelize enumerative model-checkers such as SPIN [16] and Murφ [10]; see section 2.3 for a brief survey.

Among the scores of methods used to reduce state-explosion, a prominent method – partial order reduction – exploits the fact that whenever a set of independent transitions $\{t_1, \ldots, t_n\}$ arise during exploration, they need not be explored in more than one linear order. Given the widely recognized importance of partial order reduction, efficient methods to realize it in a parallel context must be explored. This topic has, hitherto, not been studied extensively. We first motivate why partial order reduction and parallelism

* Supported by NSF Grants CCR-9987516 and CCR-0081406, and a gift from the Intel Corporation. The authors wish to thank Ratan Nalumasu who wrote the first version of PV, Hemanthkumar Sivaraj for his help with MPI, and Omid Saadati for developing the PV to Bandera link.

are two concepts that are closely related. We then briefly recap our partial order reduction algorithm "Twophase" implemented in our SPIN-like model-checker "PV," (see [22, 28] for a full discussion). In this paper, we show that Twophase extends naturally to a parallel context – more readily than algorithms that use in-stack checking – and also leads to a very natural task-partitioning method. We show preliminary experimental results and describe the Parallel PV tool, interface, and operating environment.

2 Background

2.1 Partial order reduction and parallel processing

The earliest identifiable connection between parallel processing and the central idea behind partial order reduction (long before that term was coined) appears in Lipton's work of 1975 on optimizing P/V programs [21] for the purpose of efficient reasoning. In that work, Lipton identifies *left* and *right* movers – actions that can be postponed without affecting the correctness of reasoning. Additionally, in the parallel compilation literature (e.g., [25] for an example), it has been observed that by identifying computations that "commute," one can schedule these computations for parallel evaluation without interference.

If a partial order reduction algorithm involves a sequentializing step, it can significantly reduce available parallelism. Such a sequentializing step is present in current partial order reduction algorithms such as the one used in SPIN [6, 17]. This is the[1]*in-stack* check used as a sufficient condition to eliminate the *ignoring* problem (a problem where some process P_j may have a transition t enabled everywhere in a cyclic execution path but t is never executed along the path, thus causing missed, "ignored," states – requirement C3 of [6, Page 150]) Using the 'in-stack' check, if the next state generated by moving a process P_i on transition t result in a state in the global DFS stack, an *ample set* (a commuting set of actions) cannot be formed out of the enabled transitions of P_i. The in-stack check is involved in every step of SPIN's partial order reduction algorithm. In a parallel setting, this could translate into considerable communication to locate the processes over which the stack is spread, perform the in-stack check, and resume the search. For this reason, in past work, a "worst case" assumption is used to gain some limited reductions. The assumption is that any successor state held outside the node is assumed to be currently in the search stack. This insures that the ignoring problem is dealt with, but may cause significant loss in reduction [18].

2.2 The Twophase Algorithm

We created Twophase after realizing that SPIN's algorithm for partial order reduction can, due to the in-stack check based proviso, miss out on reduction opportunities. There are also added complications when realizing nested DFS based LTL-x checking [8], as explained in [15] – essentially requiring a 'proviso-bit' to convey information from the outer DFS to the inner DFS. The algorithm of Figure 1 can be suitably modified for

[1] "The" in-stack check is a misnomer - this check is done differently for safety-preserving and liveness-preserving reductions. To simplify things, we ignore such variations.

```
Twophase()
  Vr := φ;  /* Hash table */
  Phase−2(initial_state);
end Twophase
```

```
Phase−1(in)
  local old-s, s, list;
  s := in;
  list := {s};
  for each process Pi do
    while(SAS−I(Pi,s)^2)
      old-s := s;
      s  := t(old-s);
      if s ∉ list
        list :=  list + {s};
      else
         break out of
         while loop
      end if
    end while;
  end for each;
  return(list,s);
end Phase−1
```

```
Phase−2(s)
  local list;
  /* Phase 1 */
  (list,s) := Phase−1(s);
  /* Phase 2: Classic DFS */
  if s∉Vr then
    Vr := Vr + all states in list + {s};
    for each enabled
    transition t do
      if t(s) ∉ Vr then
        Phase−2(t(s));
      end if;
    end for each;
  else
    Vr := Vr + all states in list;
  end if;
end Phase−2
```

Fig. 1. The Sequential Twophase Algorithm

nested DFS based LTL-x model-checking and no proviso bit is required to convey information between outer and inner depth first searches. Full treatment of the sequential version of the algorithm is presented in [22].

The sequential algorithm alternates between a classical depth first search *Phase-2*, and a reduction step *Phase-1*. Phase-1 works as follows. Each process is considered in turn. For a given system state s an ample set is formed for process P_i. If the size of this set is one then the transition is executed resulting in a new state. This is repeated until no singleton ample set can be formed for that process. In Phase-2 $ample(s) = enabled(s)$. In this phase of the algorithm one process having $t \in enabled(P_i, s)$ is allowed to execute t i.e., all enabled moves of all processes are included in $ample(s)$ – no reduction is made in Phase-2. This is similar to a global state expansion in SPIN's algorithm.

To deal with ignoring, references to states generated in Phase-1 are placed in a local list. We then check this list for a successor state that is a re-visitation. Note, we have changed C3 with respect to [6]. We accomplish C3 not by the C3′ condition (the in-stack check) but rather by checking against a local list, shown in Figure 1.

[2] We define SAS$_{-I}$ to be a set of transitions that form an ample set such that $check\_C1 \wedge check\_C2 \wedge |enabled(P_i, s)| = 1$ is true as described in [6, Pages 158-159] with condition $check\_C3$ satisfied separately by checking against the local list. (A Singleton Ample Set (SAS) with no check for ignoring (-I).)

2.3 Related Work

Work in parallel and distributed model-checking can be divided into the categories of *explicit state representation based* and *symbolic state representation based*. Only the explicit state survey is included in this paper. The remainder of this section can be found in [24].

Explicit State Model-Checking

Safety: Most work on distributed model-checking focus on safety model-checking. In [27], Stern and Dill report their study of parallelizing the Murφ Verifier [10]. It originally ran on the Berkeley Network of Workstations (NOW) [1] using the Berkeley Active Messages library. It was subsequently ported to run on the IBM SP2 processor. Murφ is a safety-only explicit state enumeration model-checker. In its parallel incarnation, whenever a state on the breadth-first search queue is expanded, a uniform hashing function is applied to each successor s to determine its "owner" – the node that records the fact that s has been visited, and pursues the expansion of s.

We [26] have recently ported parallel Murφ from Active Messages to the popular MPI [23] library. Despite our relative inattention to performance for reasons of expediency, our speed-up figures for runs on the Testbed are very encouraging [5]. The largest model we ran far exceeds the sizes run by Stern and Dill.

In [19], a distributed implementation of the SPIN [14] model-checker, restricted to perform safety model-checking, and similar to [27], is described. Their first innovation is in state distribution. They exploit the structure of SPIN's state representation and reduce (heuristically) the number of times a state is sent to other nodes. In addition, they employ look-ahead computation to avoid cases where a state is sent elsewhere, but very soon generates a successor that comes back to the original node. Their algorithm is also compatible with partial order reduction, although the reported results to date do not include the effects of this optimization. Their examples are standard ones such as 'Bakery' and 'Philosophers' running on up to four nodes on 300MHz machines with 64M memory. In [3], the algorithm of [27] is adapted to Uppal, a timed automaton model-checker, and applied to many realistic industrial-scale protocols, running on 24, 333MHz Sun Enterprise machines. Several scheduling policies are studied along with speed-up results.

In [11], parallel state space construction for labeled transition systems (LTSs) obtained from languages such as LOTOS is described. They use a cluster of 450MHz machines of up to 10 processors, each with 0.5GB of memory. They use the widely supported `Socket` library. They obtain speedups on most examples (industrial bus protocols) and perform analysis of the effects of communication buffers on overall performance.

In [20], issues relating to software model-checking and state representation are discussed. A large number of load distribution policies are discussed, and preliminary experimental results are reported. Many of these ideas are adaptations of techniques from their original work [19] to work well in the context of a software model such as Java.

LTL-x: Several works go beyond state space reachability and attempt the distributed model-checking of more expressive logics. In [2], the authors build on [19] and create a distributed LTL-x model-checker. The main drawback of their work is that the standard [8, 6] nested depth-first search algorithm employed to detect accepting (violating) Büchi automaton cycles tends to run sequentially in a distributed context, as the postorder enumeration of the "seed" states is still essential. They ameliorate the situation slightly by employing a data-structure called DepS that records how states were transported from processor to processor, and gathering the postorder numbering of the seed states in a distributed manner. However, the seed states still end up in a central queue, and are processed sequentially. A small degree of pipelining parallelism appears possible between the inner depth-first search on the "left half" of the search tree and the outer depth-first search on the "right half" of the search tree. Their paper reports feasibility (without actual examples) on a nine 366MHz Pentium cluster.

In [4], Büchi acceptance is reduced to detecting negative cycles (those that have a negative sum of edge weights) in a weighted directed graph. This reduction is achieved by attaching an edge-weight of '-1' to all outgoing edges out of an accepting state, and a weight of '0' to all other edges.

3 The Parallel PV Tool

3.1 The Parallel Twophase Algorithm

The parallel Twophase algorithm shown in Figure 2 works as follows. Basically, Phase-2 generates all successor states for a non-commuting state. Phase-1 is then applied to each of these successors. Only states generated for process P_i since the last Phase-2 state are inserted into the list. Each thread maintains it's own list. This is not possible in a parallel SPIN implementation because all these threads would still check with respect to the global DFS stack as in [18]. A uniform hashing function is applied to the resultant non-commuting state of Phase-1 and these states are then distributed by placing a state into the search queue of that state's owner.

Phase-1 is the same as in the sequential version. Each node knows which transitions commute so given any state, the next global state can be created by any node.

3.2 Implementation

The Parallel PV tool is based on the sequential DFS based PV model-checker [22]. PV and parallel PV are written entirely in C. Parallel PV replaces the DFS based algorithm shown in Figure 1, used in the sequential version, with the distributed BFS based algorithm shown in Figure 2. It can be executed without partial order reduction to perform exhaustive breadth first search.

By ensuring *Singleton Ample Set*[3]global successor states it becomes un-necessary to save all of the states along the Phase-1 path. Heuristics can be applied to reduce the number of state lookup and insertion operations along the Phase-1 path. We refer to these heuristics as *Selective State Caching*.

[3] As in the sequential version discussed in section 2.2

```
Phase−1(in)
  local old-s, s, list;
  s := in;
  list := {s};
  for each process P do
    while(SAS_−I(P_i,s))
      old-s := s;
      s  := t(old-s);
      if s ∉ list
        list :=  list + {s};
      else
        break out of
        while loop
      end if
    end while;
  end for each;
  return(list,s);
end Phase−1
```

```
Phase−2(s)
  V_r := φ;  /* Hash table */
  local list;
  local queue;
  local s, s';
  i = owner(s);
  enqueue[i](s);
  /* Phase 2: Classic BFS */
  while search not complete
    s = dequeue();
    if s∉V_r then
      for each enabled
      transition t do
        if t(s) ∉ V_r then
          (list, s') = Phase−1(t(s));
          V_r := V_r + all states in list;
          i = owner(s');
          enqueue[i](s');
        end if;
      end for each;
    end if;
  end while;
end Phase−2
```

Fig. 2. The Parallel Twophase Algorithm

In many cases, Phase-1 executes transitions that will cross the boundary of the state partition. Thus at least one, and as many as all of the states entered into the list are not owned by the node performing the computation. Those states not belonging to the computing node can, at the end of Phase-1, be deleted while preserving safety properties. These states are not sent to their owning nodes, they are simply dropped. This technique can be used with all of the selective state caching variants described in [12].

The list used to avoid ignoring, shown in Figures 1 and 2, and the Drop States optimization are both implemented as arrays of pointers into the hash table. A state is marked and placed in one of these lists when that state is created in Phase-1. If that state is revisited, a simple state look-up is necessary. At the end of Phase-1, those states that have been placed in the list are then set to normal hash table entries. Those in the Drop States array are removed from the hash table.

Figure 2 does not indicate termination conditions. The search is terminated in one of two instances. It is possible that the entire reduced graph has been generated. In this case we detect this condition using the Dijkstra-Scholten algorithm for stable condition detection in a diffused computation[9]. A description of our implementation of the Dijkstra-Scholten algorithm can be found in [24].

The other condition for termination is the detection of a safety violation. The node that finds the violation broadcasts a message to other nodes requesting them to wait in assistance of reconstruction of the error trail. When a state s' is entered into the hash table, a memory reference to the hash table location containing the global predecessor

Number of Network Nodes	Without PO Redn	Save All	Save Back Edge	Save All	Save Back Edge
		Save States		Drop States	
6 Processes in the Leader Election Model					
1	221239	47086	33166	47086	33166
2	221239	55694	38482	24696	18578
4	221239	66279	42315	17030	14195
8	221239	73967	44373	14305	12713
7 Processes in the Leader Election Model					
1	1719197	243704	169637	243704	169637
2	1719197	283219	195636	155063	117732
4	1719197	335102	214940	98328	79164
8	1719197	383172	228279	73012	63280
8 Processes in the Leader Election Model					
1	nc	1243666	857554	1243666	857554
2	nc	1426706	984844	789550	576799
4	13365379	1694581	1085897	483187	381980
8	13365379	1917546	1166676	310552	266735
PipeInt Model					
1	nc	nc	349708	nc	349708
2	nc	nc	377620	nc	260848
4	nc	nc	395016	1583563	184624
8	nc	3721420	407238	679869	160936

Fig. 3. Number of states generated.

s of the new state s' is also entered, along with a small constant amount of information about the state s, including the rank of the node that generated s. To construct the error trail, the hash table is traversed from the error state to the initial state, writing the information necessary to reconstruct an error trail to a file. If it is necessary to traverse multiple partitions of the state space the error trail that is contained locally is generated, packed and transmitted to the next node in the trail. This continues until the initial state is reached. We assume a shared file system for the error trail output file. An error trail can then be simulated in a sequential setting.

3.3 Experimental Results

Three of the models used as examples here are simple variants of the Leader Election Protocol as presented in [6, Pages 167-168]. This model does not include the never claim as we are considering safety properties only. The variation on this model is the number of processes that are participating in the protocol. We model-check for six, seven, and eight processes using the same model. The other model reported is generated by the Bandera tool. It is included in the Bandera tutorial distribution and can be recreated using their tutorial and is included in the examples directory of our tool distribution. We report the number of states generated, memory usage, message count,

Number of Network Nodes	Without PO Redn	Save All	Save Back Edge	Save All	Save Back Edge
		Save States		Drop States	
6 Processes in the Leader Election Model					
1	45	22	20	22	20
2	70	40	38	37	37
4	120	76	74	72	71
8	219	148	145	142	142
7 Processes in the Leader Election Model					
1	213	42	35	42	35
2	226	64	55	51	47
4	276	10	92	80	78
8	377	181	165	149	148
8 Processes in the Leader Election Model					
1	nc	166	122	166	122
2	nc	212	162	140	116
4	1590	294	225	157	146
8	1700	422	337	240	235
PipeInt Model					
1	nc	nc	168	nc	168
2	nc	nc	205	nc	157
4	nc	nc	261	749	175
8	nc	1730	366	478	265

Fig. 4. Total memory used in MB.

and total runtime for each model. Figures 3, 4, 5, and 6 present the statistics for the respective reports.

A description of the figures is as follows. The number of network nodes column indicates how many physical machines are used in each model-checking computation. Without PO Redn indicates the statistics of a full breadth first graph search. Save All indicates a search that uses partial order reduction, but no selective state caching heuristic. Save Back Edge indicates a search that uses both partial order reduction and a selective state caching heuristic. Save States indicates that all states generated during Phase-1 are placed permanently into the hash table of the computing node, regardless of ownership. Drop States indicates that those states generated during Phase-1 which do not belong to the computing node are deleted at the end of Phase-1.

Figure entries showing “nc” indicate the computation was Not Complete. In each of these cases there was not enough memory to successfully generate the entire state graph. (We have not resorted to any hash-compaction techniques yet.) Otherwise the states and messages are integer values. The memory used is reported in megabytes. Time is reported in seconds.

Our tool release includes these examples and a script that will recreate our results. All verifications were performed using one Unix process per network node. The experiments were performed on a computational cluster of eight workstations. Each has a stock Red Hat Linux 7.1 operating system, 512MB memory and one 850MHz Intel Pen-

Number of Network Nodes	Without PO Redn	Save All	Save Back Edge	Save All	Save Back Edge
		Save States		Drop States	
6 Processes in the Leader Election Model					
2	538400	15829	18087	20208	20208
4	851143	25212	27162	28737	28737
8	945534	30600	31934	33040	33040
7 Processes in the Leader Election Model					
2	4281970	80016	91325	104421	104421
4	7446183	129721	140332	149947	149947
8	8452930	167161	174620	181173	181173
8 Processes in the Leader Election Model					
2	nc	396464	447182	519832	519832
4	65752911	684746	748652	809919	809919
8	77583833	830234	877647	912861	912861
PipeInt Model					
2	nc	nc	144377	nc	145118
4	nc	nc	220052	449860	232929
8	nc	243402	242062	344934	242743

Fig. 5. Number of messages passed.

tium III CPU. Each Unix process was limited to 450MB of memory. The MPICH[13] implementation of the MPI standard is used for message passing between network nodes. Nodes are connected on a dedicated 100Mbps hub.

3.4 Interface

A TCL/TK graphical user interface is included in the Parallel PV distribution. This makes available sequential DFS, sequential BFS, and parallel BFS based searches. Models can be edited and error trails can also be simulated using the graphical user interface. Parallel PV can also be used from the command line. All output is routed via MPI to the master node. Please see [28] for command line interaction with Parallel PV.

The PipeInt model was generated using the Bandera[7] tool set. The subset of the Promela language supported by PV and Parallel PV is sufficiently expressive to model-check these models. Additionally, the Bandera user interface has been enhanced to access both the sequential and parallel versions of PV. Thus an error that is found using PV or Parallel PV on a Bandera generated model can be simulated within the Bandera framework.

4 Conclusions

We have presented a distributed partial order reduction based safety verification algorithm that is a variant of the sequential Twophase [22] algorithm.

Number of Network Nodes	Without PO Redn	Save All	Save Back Edge	Save All	Save Back Edge
		Save States		Drop States	
6 Processes in the Leader Election Model					
1	54.786	4.006	2.958	4.342	3.256
2	46.061	3.783	2.671	3.535	2.966
4	31.158	3.422	2.847	3.258	2.913
8	20.270	4.871	4.170	4.456	4.226
7 Processes in the Leader Election Model					
1	569.140	21.105	16.005	24.998	17.947
2	432.657	14.080	12.134	17.323	13.744
4	282.142	9.548	8.529	11.102	9.300
8	162.547	8.007	7.730	8.774	8.021
8 Processes in the Leader Election Model					
1	nc	123.522	92.921	149.020	105.825
2	nc	77.744	64.995	97.039	75.528
4	2712.108	47.009	40.765	56.346	45.585
8	1522.699	29.693	25.716	33.266	27.820
PipeInt Model					
1	nc	nc	125.934	nc	142.131
2	nc	nc	71.203	nc	79.407
4	nc	nc	40.135	350.378	44.018
8	nc	72.904	24.268	126.410	25.753

Fig. 6. Total run time for each model.

The parallel Twophase algorithm has several advantages. It avoids the in-stack check allowing distributed partial order reduction. The algorithm can be used with BFS or DFS. It allows natural task partitioning and also reduces communication. It supports selective state caching in conjunction with a “Drop States” optimization.

The parallel Twophase algorithm is implemented in the parallel PV tool which supports selective state caching. The parallel PV tool can be used from the command line, with the included graphical user interface, or with the Bandera tool.

We have shown some preliminary experimental results using two models that generate large state spaces. The results are encouraging for their relative short run-times and efficient use of memory.

Our future work will include application-level check-pointing to be able to suspend and/or rerun crashed parallel model-checks, load balancing, hash compaction, and possibly symmetry reduction. PV and Parallel PV are available from our website [28].

References

1. Thomas E. Anderson, David E. Culler, and David A. Patterson. A case for networks of workstations: Now. *IEEE Micro*, February 1995.
2. Jiri Barnat, Lubos Brim, and Jitka Stribrna. Distributed ltl model-checking in spin. In *Proceedings of the 7th International SPIN Workshop*, pages 200–216, 2001. LNCS 2057.

3. G. Behrmann, T.S. Hune, and F.W. Vaandrager. Distributed timed model checking - how the search order matters. In *Computer Aided Verification (CAV)*, pages 216–231, 2000. LNCS 1855.
4. Lubos Brim, Ivana Cerna, Pavel Krcal, and Radek Pelanek. Distributed ltl model checking based on negative cycle detection. In *Proceedings of the FSTTCS Conference*, 2001. Bangalore, India, December 2001. To appear.
5. Prosenjit Chatterjee, Hemanthkumar Sivaraj, and Ganesh Gopalakrishnan. Shared memory consistency protocol verification against weak memory models: refinement via model-checking. To appear in CAV'02.
6. Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, 2000.
7. James C. Corbett, Matthew B. Dwyer, John Hatcliff, Shawn Laubach, Corina S. Pasareanu, Robby, and Hongjun Zheng. Bandera: extracting finite-state models from java source code. In *International Conference on Software Engineering*, pages 439–448, 2000.
8. C. Courcoubetis, M. Vardi, P. Wolper, and M. Yannakakis. Memory efficient algorithms for the verification of temporal properties. pages 233–242, June 1990.
9. E. Dijkstra and C.S. Scholten. Termination detection for diffusing computations. *Information Processing Letters*, 11(1):1–4, August 1980.
10. David Dill. The murφ verification system. In *Computer Aided Verification (CAV)*, pages 390–3, 1996.
11. Hubert Garavel, Radu Mateescu, and Irina Smarandache. Parallel state space construction for model-checking. In *Proceedings of the 7th International SPIN Workshop*, pages 217–234, 2001. LNCS 2057.
12. Ganesh Gopalakrishnan, Ratan Nalumasu, Robert Palmer, Prosenjit Chaterjee, and Ben Prather. Performance studies of pv: an on-the-fly model-checker for ltl-x featuring selective state caching and partial order reduction. Technical Report UUCS-01-004, The University of Utah, School of Computing, January 2001.
13. W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22(6):789–828, September 1996.
14. G. J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997. Special issue on Formal Methods in Software Practice.
15. G. J. Holzmann and Doron Peled. An improvement in formal verification. In *Proc. Formal Description Techniques, FORTE94*, pages 197–211, Berne, Switzerland, 1994. Chapman & Hall.
16. Gerard Holzmann. *Design and Validation of Computer Protocals*. Prentice Hall, 1991.
17. G.J. Holzmann, P. Godefroid, and D. Pirottin. Coverage preserving reduction strategies for reachability analysis. In *Proc. 12th Int. Conf on Protocol Specification, Testing, and Verification, INWG/IFIP*, Orlando, Fl., June 1992.
18. 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.
19. Flavio Lerda and Riccardo Sisto. Distributed-memory model checking with spin. In *Proceedings of the 5th International SPIN Workshop*, pages 22–39, 1999.
20. Flavio Lerda and Willem Visser. Addressing dynamic issues of program model checking. In *Proceedings of the 7th International SPIN Workshop*, pages 80–102, 2001. LNCS 2057.
21. Richard J. Lipton. Reduction: A method of proving properties of parallel programs. *Communications of the ACM*, 18(12), December 1975.
22. Ratan Nalumasu and Ganesh Gopalakrishnan. An efficent partial order reduction algorithm with an alternative proviso implementation. *Formal Methods in System Design*, 20(3):231–247, May 2002.

23. Peter Pacheco. *Parallel Programming with MPI*. Morgan Kaufmann, 1996. ISBN 1-55860-339-5.
24. Robert Palmer and Ganesh Gopalakrishnan. Partial order reduction assisted parallel model-checking (full version). Technical report, University of Utah, August 2002.
25. Martin Rinard and Pedro Diniz. Commutativity analysis: A new analysis technique for parallelizing compilers. *ACM Transactions on Programming Languages and Systems*, 19(6):1–47, November 1997.
26. Hemanthkumar Sivaraj. MPI port conducted in October 2001. Personal Communication.
27. Ulrich Stern and David Dill. Parallelizing the Murφ verifier. *Formal Methods in System Design*, 18(2):117–129, 2001. (Journal version of their CAV 1997 paper).
28. The Utah Verifier group website. http://www.cs.utah.edu/formal_verification.

New Petri Net Programming Features in PEP*

Cécile Bui Thanh[1] and Christian Stehno[2]

[1] Université Paris 12, LACL, 61 avenue du général de Gaulle, F-94010 Créteil, France. bui@univ-paris12.fr
[2] Department of Computing Science, Carl von Ossietzky Universität, D-26111 Oldenburg, Germany.
stehno@informatik.uni-oldenburg.de

Abstract. We present two new facilities of the high level Petri net editor of the PEP tool. The latest version widens the class of supported Petri nets by a time extension of M-nets. Additionally it features a new operator for asynchronous communication complementing the synchronization operator. We present an example of an ARQ protocol with enhanced acknowledgement handling.

1 Introduction

The PEP tool [5, 10] provides a development environment for a number of different parallel programming and specification languages. Additionally, the high and low level Petri net levels offer another, quite unique, programming interface. Especially the high level nets facilitate design of data handling algorithms and complex programs through their extended token types and powerful set of operators.

The latest extensions of the net editors add major new features with the introduction of a new basic operator tie and a new class of nets, namely the Time Petri net extension of M-nets.

This paper presents the enhancements of the specification and programming process gained by these extensions. The example chosen for this purpose is a simple ARQ (automatic repeat request) protocol with possibly deferred acknowledgements.

The next two sections will introduce the formalisms used for the example, i.e. (Time) M-nets and the tie operator. In Sect. 4 the example is presented, and in Sect. 5 we give a conclusion and highlight some future work.

2 M-nets and Time M-nets

We will only give a short introduction to M-nets and their time extension. More details can be found in the cited papers and related articles.

* This work has been done during a two month visit of one of the authors to Oldenburg, and has been partially supported by the Procope project PORTA (Partial Order Real Time Semantics)

The coloured Petri nets algebra of M-nets (multi labelled nets [2]) has been developed as a flexible semantic model for concurrent programming languages. The annotations of net elements and a set of operations provide a framework for the compositional creation of complex nets. Operations include sequential, parallel and alternative composition, iteration and synchronous communication. Unfolding into plain P/T nets provides ways for formal verification of designed systems by means of well-known algorithms for ordinary Petri nets, e.g. with the PEP tool.

Time M-nets (TM-nets) are a cautious, but still substantial extension of M-nets towards real-time systems. The time extension was originally defined in [4] for a real-time version of the specification language SDL. TM-nets had been supported so far only in the MOBY tool [1] without a possibility to validate the nets. The newly implemented TM-net features of the PEP tool, together with the already presented analysis tools for timed systems [13, 3] increase the importance of TM-nets.

The time extension is done analogously to the Time PBC extension in [8]. Transitions get an additional inscription of an interval in the natural numbers. This interval restricts the ability of the transition to fire (cf. [9]), i.e. the lower (upper) bound defines the earliest (latest) firing time of each transition.

To preserve the properties of M-nets with respect to unfolding into low level nets for TM-nets, firing times are not unique to a transition, but each firing mode (i.e. each binding, which enables the transition) counts time steps on its own, after that mode is enabled. As such, each transition may have a large number of clocks showing different times.

The current implementation in PEP covers just an elementary version of TM-nets. We restrict the interval bounds to natural numbers (plus infinity for the upper bound). This is in contrast to the original definition where arbitrary expressions are allowed, which still have to evaluate to natural numbers according to the chosen binding, though. As such, the restriction is not just syntactic, but the expressiveness of the simple class is still large enough for many real world systems.

3 Tie operator

The tie operator was introduced to M-net algebra to allow asynchronous communications [7] in a compositional way. This extension allows to give a complete semantics of the Basic Petri Net Programming Notation ($B(PN)^2$) [6] in terms of M-nets, and a simpler semantics for FIFO buffers [11, 12].

To use asynchronous communications with M-nets, they are extended by new transition labels called *link labels*. These labels indicate that a transition can export and/or import information from an asynchronous channel.

We consider a set B of *tie symbols* such that each $b \in \mathsf{B}$ has a type $\mathsf{type}(b)$. An *asynchronous link* $b^d(\tilde{a})$, where $b \in \mathsf{B}$, $d \in \{+, -\}$ and $\tilde{a} \in \mathsf{type}(b)$, represents an asynchronous communication action on (the channel represented by) b. The direction of the communication (export or import) is given by d, and $\tilde{a}$ is a set of M-net value terms, representing possible information communicated on b. A link label L is a multi-set of asynchronous links.

An example drawn with the high-level net editor of the PEP tool is given in Fig. 1. On the left hand, we show a net N with asynchronous annotations. The label $\{$`b+(x)`$\}$ on transition T1 means that this transition *can* export $x \in \mathsf{type}(b)$ on a channel b, and $\{$`b-(1),b+(1)`$\}$ on T2 means that T2 expects to import a token labelled "1" and to export it again via b. The result of N tie b is given on the right hand of the figure: The asynchronous links of N containing b were made effective by the operation, i.e. the transitions containing these asynchronous links now export and/or import the described informations when they are fired, according to the directions given by the previous notation. Every exchanged information goes through the created buffer place of b (named P6) with $\mathsf{type}(b)$ as type. Note that applying tie b deletes the link labels concerning b (not the other links) from every transition.

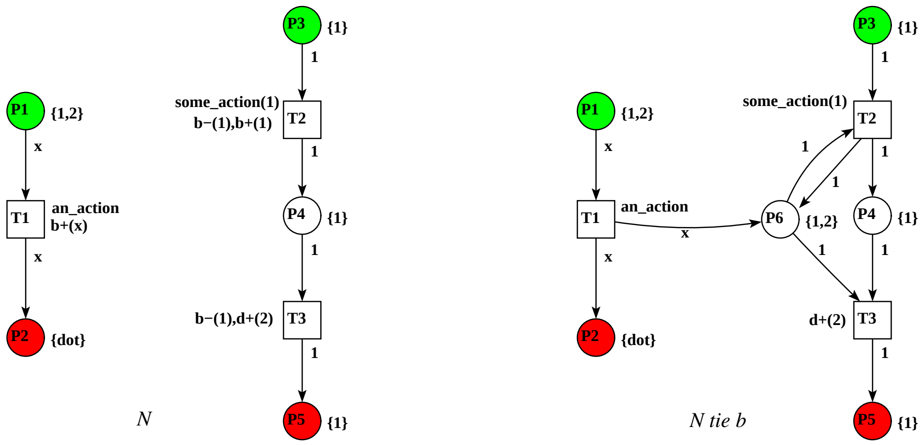


Fig. 1. An example of using tie with the high-level editor. On the left hand a high-level net N containing link labels, on the right hand N after applying tie b

In the high-level editor, declarations (respectively, deletions, modifications) of asynchronous link names and their corresponding types are made through the window of the tie operator (cf. Fig. 2). The syntax for link types is the same as for place types. Link labels on transitions are added the same way as action multisets. After applying the operator once to some link b, this one is taken into account as a brand new link and may be reused afterwards to add new annotations on link labels and to apply $\mathsf{tie}\,b$ again. A new buffer place will be created for b each time the operation is called.

The tie operator was also defined for PBC (see [7] for details). This is not completely implemented yet, but after applying tie to each link, a net does not

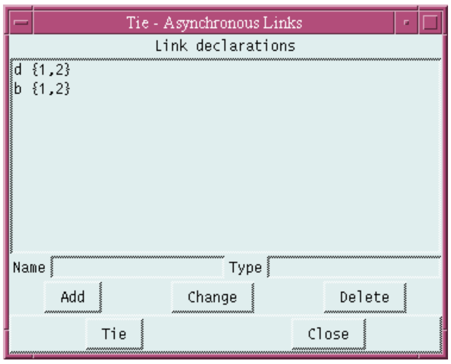


Fig. 2. The window of the tie operator

contain asynchronous notations anymore, so there is a way to unfold and analyse it right away.

4 Example of an ARQ protocol

As an example for the presentation of the newly introduced features we chose a modelisation of an ARQ protocol. There is a sender and a receiver side, where the sender communicates data packets to the receiver, who may answer with positive or negative acknowledgements. Both parts can be found in Fig. 3, with the sender at the top and the receiver at the bottom.

Communication is done in both directions via a buffered channel with some delay, i.e. asynchronous. Moreover the acknowledgements may be deferred, such that not every packet will be acknowledged, but a positive acknowledgement includes all waiting ack's in between. There is no buffering of correctly received data, so negative acknowledgements also clear everything backwards up to the negatively acknowledged packet.

If the sender receives a negative acknowledgement, it sets the next packet pointer to the received number, such that the next packet send will be the requested one. Positively acknowledged packets just increase the counter, with some sanity check done at reception (only shown in Fig. 4 for channel control due to space and readability). Sending data just incorporates the packet number, i.e. data content is abstracted away.

At the receiver side, things are even easier. The received packets are stored in the respective counter place, which serves as a source for generating (positive or negative) acknowledgements. To model the removal of destroyed packets by some negative acknowledge, the received packet counter is reset accordingly. Thus, these packets will not be used by the send ack transition, preventing false acknowledgements.

The two channels (C1 from receiver to sender and C2 vice versa) are simplified versions of the FIFO buffers presented in [11, 12]. Unused parts have been

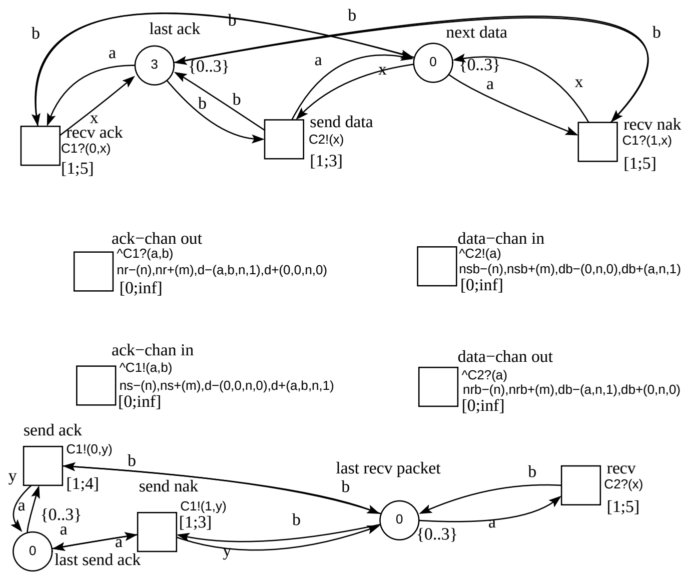


Fig. 3. ARQ model before synchronisation and tie

deleted, the data handling is simplified and the channel use has been extended with timing restrictions. Figure 3 just shows four transitions, which create two separate ring buffers with FIFO semantics and compatible interface after application of the tie operation, one of them is shown in Fig. 4.

The time intervals are attached to each channel operation, specifying some minimal and maximal delay needed for these actions. The channel transitions themself do not contribute to the delay due to their interval of $[0, \infty]$. The delays have been chosen arbitrarily for this example, resulting in a delay for sending a packet between two and 8 time units, a complete round-trip with packet sending and acknowledgement takes up to 17 time units in the case of no data loss in between. The worst case scenario will take infinitely many time units, as we do not take into account fairness.

5 Conclusion and future works

In this paper we have presented the latest features of the PEP tool. These allow verification of Time M-nets and add facilities for modelisation of asynchronous links in a compositional way. Moreover, the enhancements contribute to readability and handiness of the editor. M-nets provide a very powerful specification formalism. The ARQ system includes some interesting features, using just a

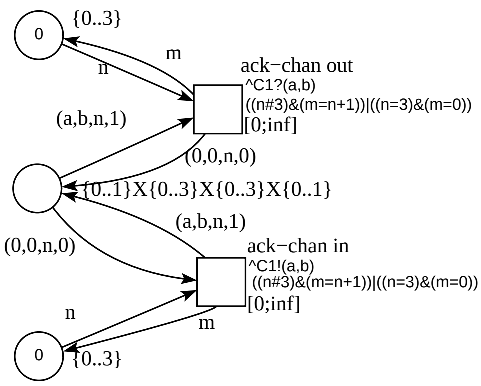


Fig. 4. FIFO channel after tie. New buffer places have been created

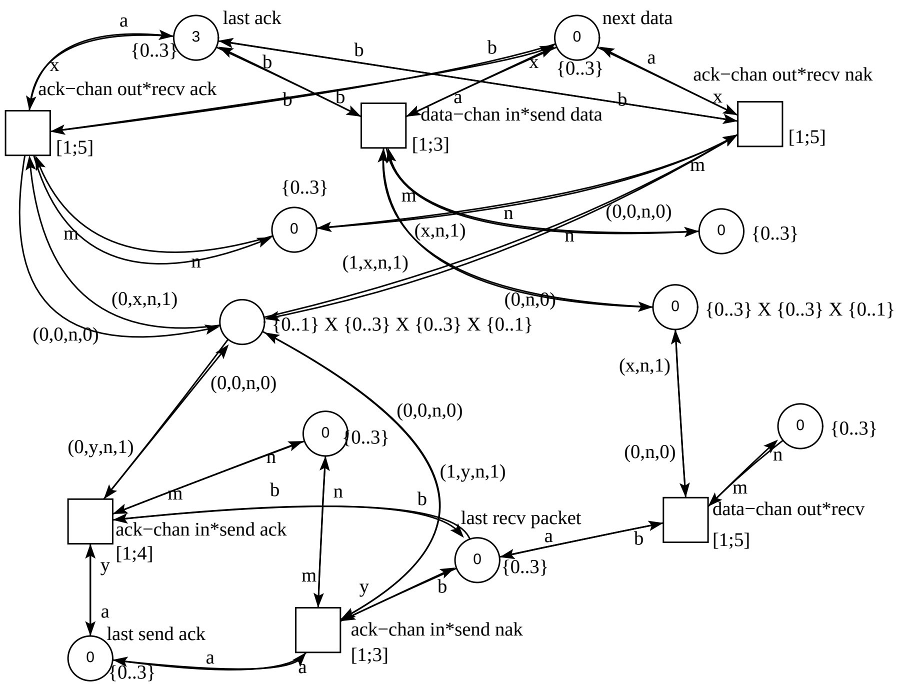


Fig. 5. Complete system after application of tie and scope

small number of net elements. The time extension presented offers a comfortable high-level programming interface for real-time systems.

However, the asynchronous links have not yet been integrated into every part of the tool, which is planned for one of the next versions. This will remove some restrictions currently imposed on the user.

Time Petri nets are still not supported the same way like ordinary Petri nets. Most notably, a timed simulator is missing. Further work will include different time semantics and time support in even higher specification concepts, like SDL. Some further investigation of case studies of real-time systems, including formal verification, is planned.

References

1. Peter Amthor, Hans Fleischhack, and Josef Tapken. MOBY - More than a tool for the verification of SDL-specifications. Technical report, Fachbereich Informatik, Carl von Ossietzky Universität Oldenburg, 1996.
2. E. Best, W. Frączak, R. P. Hopkins, H. Klaudel and E. Pelz. *M-nets: An algebra of high level Petri nets, with an application to the semantics of concurrent programming languages.* Acta Informatica, 35:813–857. Springer, 1998.
3. Hans Fleischhack and Christian Stehno. Computing a Finite Prefix of a Time Petri Net. In *ICATPN*, 2002. To appear.
4. Hans Fleischhack and Josef Tapken. An M-net semantics for a real-time extension of μSDL. In John Fitzgerald, Cliff B. Jones, and Peter Lucas, editors, *FME'97: Industrial Applications and Strengthened Foundations of Formal Methods (Proc. 4th Intl. Symposium of Formal Methods Europe, Graz, Austria, September 1997)*, volume 1313, pages 162–181. Springer-Verlag, 1997.
5. Bernd Grahlmann. *The PEP Tool.* In Orna Grumberg (ed.), Computer Aided Verification, volume 1254 of Lecture Notes in Computer Science, pp. 440–443. Springer, 1997.
6. Hanna Klaudel. *Parameterized M-expression Semantics of Parallel Procedures.* DAPSYS'02, pp. 105–114. Kluwer Academic Publishers, 2002.
7. Hanna Klaudel and Franck Pommereau. *Asynchronous links in the PBC and M-nets.* ASIAN'99, LNCS 1742, pp. 190–200. Springer, 1999.
8. Maciej Koutny. A Compositional Model of Time Petri Nets. In M. Nielsen and D. Simpson, editors, *Application and Theory of Petri Nets 2000*, volume 1825 of *LNCS*, pages 303–322. Springer-Verlag, 2000.
9. P. Merlin and D. Farber. Recoverability of Communication Protocols – Implication of a Theoretical Study. *IEEE Transactions on Software Communications*, 24:1036–1043, 1976.
10. The PEP tool. http://parsys.informatik.uni-oldenburg.de/~pep
11. Franck Pommereau. *FIFO buffers in tie sauce.* Proc. of DAPSYS'00, pp. 95–104. Kluwer Academic Publishers, 2000.
12. Franck Pommereau and Christian Stehno. FIFO buffers in hot tie sauce. Technical Report 2001-04, LACL, Université Paris 12, 61 avenue du général de Gaulle, F-94010 Créteil, France, 2001.
13. Christian Stehno. Real-Time Systems Design with PEP. In Joost-Pieter Katoen and Perdita Stevens, editors, *TACAS*, volume 2280 of *Lecture Notes in Computer Science*, pages 476–480. Springer-Verlag, 2002.

A Markov Chain Model Checker

HOLGER HERMANNS[a*], JOOST-PIETER KATOEN[a],
JOACHIM MEYER-KAYSER[b**], AND MARKUS SIEGLE[b]

[a] *Formal Methods and Tools Group, University of Twente*
P.O. Box 217, 7500 AE Enschede, The Netherlands

[b] *Lehrstuhl für Informatik 7, University of Erlangen-Nürnberg*
Martensstraße 3, 91058 Erlangen, Germany

Abstract. Markov chains are widely used in the context of performance and reliability evaluation of systems of various nature. Model checking of such chains with respect to a given (branching) temporal logic formula has been proposed for both the discrete [5] and the continuous time setting [1, 3]. In this note, we describe the prototype model checker $\mathsf{E} \vdash \mathsf{MC}^2$ for discrete and continuous-time Markov chains, where properties are expressed in appropriate extensions of CTL. We illustrate the general benefits of this approach and discuss the structure of the tool.

Introduction

Markov chains are widely used as adequate models in many diverse areas, ranging from mathematics and computer science to other disciplines such as operations research, industrial engineering, biology and demographics. Markov chains can be used to estimate performance characteristics of various nature, for instance to quantify throughput of manufacturing systems, locate bottlenecks in communication systems, or to estimate reliability in aerospace systems.

Model checking is a very successful technique to establish the *correctness* of systems from similar application domains, usually described in terms of a non-deterministic finite-state model. If non-determinism is replaced by randomized, i.e. probabilistic decisions, the resulting model boils down to a finite-state discrete-time Markov chain (DTMC). For these models, a number of qualitative and quantitative model checking algorithms have been proposed. In a qualitative setting it is checked whether a property holds with probability 0 or 1; in a quantitative setting it is verified whether the probability for a certain property meets a given lower or upper bound. PCTL [5] is a representative of the latter kind. It is an extension of CTL [4], allowing one to specify and verify properties such as *"After a system failure, the probability that the system will not come up again is at most* 10^{-6}*."*

Markov chains are *memoryless*. In the discrete-time setting this is reflected by the fact that probabilistic decisions do not depend on the outcome of decisions taken earlier, only the state currently occupied is decisive to completely

* supported by the Netherlands Organisation for Scientific Research (NWO).
** supported by the German Research Council DFG under HE 1408/6-1.

determine the probability of next transitions. For continuous-time Markov chains (CTMCs), where time ranges over (positive) reals (instead of discrete subsets thereof) the memoryless property further implies that the probabilities of taking next transitions do not depend on the amount of time spent in the current state. DTMCs are mostly applied to strictly synchronous scenarios, while CTMCs have shown to fit in well with (interleaving) asynchronous scenarios. In particular, CTMCs are the underlying semantic model of major high-level performance modelling formalisms such as stochastic Petri nets, stochastic automata networks, stochastic process algebras, Markovian queueing networks, and various extensions thereof.

Recently, the logic CSL has been proposed [1, 3], an extension of both PCTL and CTL tailored to quantitative properties of CTMCs. Apart from CTL and PCTL properties, a selection of typical properties that can be verified using this logic is:

- *"After a system failure, there is at least a 99.99 % chance that the system will come up again within 5 time units."*
- *"On the long run, the probability of the system being unavailable is at most* 10^{-4}*."*
- *"The probability that signal 'ready' will be received within the next* 4 *time units is more than* 0.3*."*

In this short paper we describe the *Erlangen–Twente Markov Chain Checker* ($\mathsf{E} \vdash \mathsf{MC}^2$), to our knowledge the first implementation of a model checker for DTMCs and CTMCs. It uses numerical methods to model check PCTL and CSL-formulas, based on [5, 3, 2]. Apart from standard graph algorithms, model checking CSL involves matrix-vector multiplications, solutions of linear systems of equations, and solutions of systems of Volterra integral equations. Linear systems of equations are solved iteratively by standard numerical methods [12]. Two alternatives to solve systems of integral equations are implemented: One is based on piecewise integration of discretized distribution functions, the other is based on uniformisation [2]. Uniformisation is the default option, because it allows the tool to a priori calculate the computational effort needed to check a given property. This effort depends on the numerical parameters of the current model, on the property to be checked, and on the required numerical precision ε (the latter is a parameter set by the user).

$\mathsf{E} \vdash \mathsf{MC}^2$ is a *global* model checker, i.e. it checks the validity of a formula for all states in the model. It has been developed such that it can easily be linked to a wide range of existing high-level modelling tools based on, for instance, stochastic process algebras, stochastic Petri nets, or queueing networks. A whole variety of such tools exists [6], most of them using dedicated formats to store the transition matrix $\mathbf{R}$ of the Markov chain that is obtained from a high-level specification. This matrix encodes the probabilistic behaviour of the system as time passes. Together with a labelling function L, which associates the states of the Markov chain with sets of atomic propositions, the matrix $\mathbf{R}$ constitutes the interface between the high-level formalism at hand and the model checker. Currently, the tool accepts DTMCs and CTMCs represented in a format generated by the

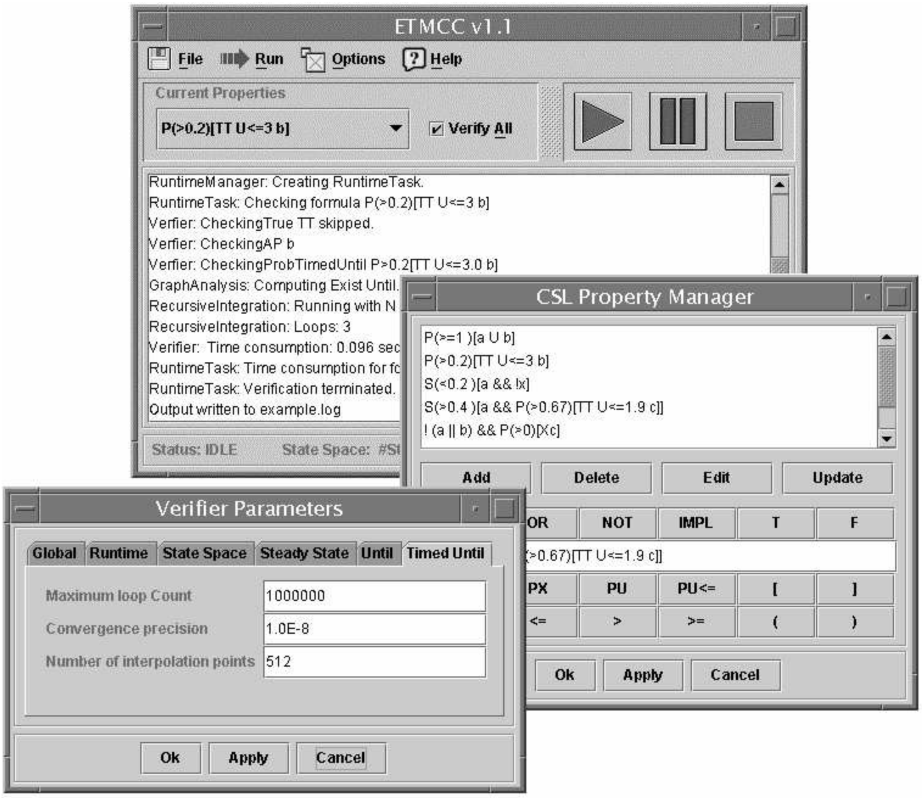


Fig. 1. User interface of $\mathsf{E} \vdash \mathsf{MC}^2$

stochastic process algebra tool TIPPTOOL [11], but the tool is designed in such a way that it can easily bridge to various other input formats.

Tool architecture

The tool has been written entirely in JAVA (version 1.2), in order to provide platform independence and to enable fast and efficient program development. Furthermore, support for the development of graphical user interfaces as well as grammar parsers is at hand. For the sake of simplicity, flexibility and extensibility we abstained from low-level optimizations, such as minimization of object invocations. The design and implementation took approximately 15 man-months, with about 10000 lines of code for the kernel and 1500 lines of code for the GUI implementation, using the SWING library. The tool architecture consists of five components:

Graphical User Interface (cf. Fig. 1) enables the user to load, modify and save verification projects. Each project consists of a model **R**, a labelling

L, and the properties to be checked. The GUI contains the 'CSL Property Manager' which allows the user to construct and edit CSL-formulas. The GUI also prints results and additional logging information on screen or writes them into file. Several verification parameters for the numerical analysis, such as solution method, precision ε, and number of interpolation points for the piecewise integration, can be set by the user.

Tool Driver controls the model checking procedure. It generates the parse tree corresponding to a given CSL property. Subsequent evaluation of the parse tree issues calls to the respective verification objects that encapsulate the verification sub-algorithms. These objects, in turn, use the analysis and/or numerical engine.

Analysis Engine is the engine that supports standard model checking algorithms for CTL-style until-formulas, as well as graph algorithms, for instance to compute the bottom strongly connected components of a Markov chain. The former algorithms are very useful in a pre-processing phase during the checking of probabilistic until-formulas (they may help to avoid many numerical calculations), while the latter is needed when calculating long-run average properties.

Numerical Engine is the numerical analysis engine of the tool. It provides several methods for the numerical solution of linear systems, for numerical integration, and for uniformisation. These are used to solve sytems of linear or integral equations on the basis of parameters provided by the user via the GUI.

State Space Manager represents DTMCs and CTMCs in a uniform way. In fact, it provides an interface between the various checking and analysis components and the way in which DTMCs and CTMCs are actually represented. This eases the use of different, possibly even symbolic (i.e. BDD-based) state space representations. It is designed to support input formats of various kinds, by means of a simple plug-in-functionality (using JAVA's dynamic class loading capability). It maintains information about the validity of atomic propositions and of sub-formulas for each state, encapsulated in a 'Sat' sub-component. After checking a sub-formula, this sub-component stores the results, to be used later. In the current version of the tool, the state space is represented as a sparse matrix [12]. All real values are stored in the IEEE 754 floating point format with double precision (64 bit).

Conclusion

In this short paper we have described the Markov chain model checker $\mathsf{E} \vdash \mathsf{MC}^2$. Even though the tool is still a prototype, it has already been used in a number of nontrivial case studies, including

– validation and performance analysis of a cyclic server polling system [7],
– reliability estimation of the Hubble space telescope [8],
– dependability analysis of a workstationclusters [9], and
– performance and availability assessment of a distributed database server [10].

For more information about $\mathsf{E} \vdash \mathsf{MC}^2$, the reader is invited to consult [7], or `http://www7.informatik.uni-erlangen.de/etmcc/`.

References

1. A. Aziz, K. Sanwal, V. Singhal and R. Brayton. Verifying continuous time Markov chains. In *Computer Aided Verification, CAV 96*, Springer LNCS 1102: 269–276, 1996.
2. C. Baier, B.R. Haverkort, H. Hermanns and J.-P. Katoen. Model checking continuous-time Markov chains by transient analysis. In *Computer Aided Verification, CAV 2000*, Springer LNCS 1855: 358-372, 2000.
3. C. Baier, J.-P. Katoen and H. Hermanns. Approximate symbolic model checking of continuous-time Markov chains. In *CONCUR 99*, Springer LNCS 1664: 146–162, 1999.
4. E.M. Clarke, E.A. Emerson and A.P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Tr. on Progr. Lang. and Sys.*, **8**(2): 244-263, 1986.
5. H. Hansson and B. Jonsson. A logic for reasoning about time and reliability. *Form. Asp. of Comp.*, **6**(5): 512–535, 1994.
6. B.R. Haverkort and I.G. Niemegeers. Performability modelling tools and techniques. *Performance Evaluation* **25**: 17–40, 1996.
7. H. Hermanns, J.P. Katoen, J. Meyer-Kayser and M. Siegle. A Markov chain model checker. In *TACAS 2000*, Springer LNCS 1785: 347–362, 2000.
8. H. Hermanns. Performance and reliability model checking and model construction. In *Formal Methods for Industrial Critical Systems, FMICS 2000*, GMD Report 91, pages 11-28, Berlin, April 2000.
9. B. Haverkort, H. Hermanns, and J.P. Katoen. The Use of Model Checking Techniques for Quantitative Dependability Evaluation.In *IEEE Symposium on Reliable Distributed Systems, SRDS 2000*, IEEE CS Press, October 2000.
10. H. Hermanns, J.P. Katoen, J. Meyer-Kayser, and M. Siegle. Towards Model Checking Stochastic Process Algebra. In *IFM 2000*, Springer LNCS 1945: 420–439, November 2000.
11. H. Hermanns, U. Herzog, U. Klehmet, V. Mertsiotakis and M. Siegle. Compositional performance modelling with the TIPPtool. *Performance Evaluation*, **39**(1-4): 5–35, 2000.
12. W. Stewart. *Introduction to the Numerical Solution of Markov Chains.* Princeton Univ. Press, 1994.

Rapture:
A tool for verifying Markov Decision Processes

Bertrand Jeannet[1], Pedro R. D'Argenio[2], and Kim G. Larsen[3]

[1] IRISA – INRIA, Campus de Beaulieu. F-35042 Rennes Cedex. France
Bertrand.Jeannet@irisa.fr
[2] FaMAF - UNC, Ciudad Universitaria. 5000 - Córdoba. Argentina
dargenio@mate.uncor.edu
[3] BRICS - Aalborg University, Frederik Bajers vej 7-E. DK-9220 Aalborg. Denmark
kgl@cs.auc.dk

Abstract. We present a tool that performs verification of quantified reachability properties over Markov decision processes (or probabilistic transition system). The originality of the tool is to provide two reduction techniques that limit the state space explosion problem: automatic abstraction and refinement algorithms, and a so-called essential states reduction. We present several case-studies to illustrate the usefulness of these techniques.

1 Introduction

Fully automatic verification of a specified transition system with respect to a given temporal logic property is known as *model checking* [22, 5]. For such systems model checkers allow to verify properties such as "the system will never reach an erroneous situation", and the property can be stated true or false. In many cases however, the absolute validity of a formula cannot be determined as wished, because of the nature of the system. For instance, consider a protocol that attempts to access to a lossy medium a bounded number of times after which it aborts. A property like "access will be granted" is obviously false. Nevertheless, to assess quality of service one would like the protocol grants access to the medium "often enough". Therefore, we would like, instead, to verify a *quantified* property like "access will be granted with probability at least 99%".

In this paper, we present Rapture[1], a tool that performs verification of quantified reachability properties over Markov decision processes (or probabilistic transition system). The system to be analysed is described as a parallel composition of finite probabilistic automata extended with finite-state variables. The automata communicate à la CSP [17] via synchronisation on a set of channels.

Other tools that verify quantified properties on (discrete time) Markov decision processes have been developed. For instance, ProbVerus [13] and

[1] Rapture is a loose acronym of *"Reachability Analysis of Probabilistic Transition systems based on REduction strategies"*. Rapture can be freely downloaded from http://www.irisa.fr/prive/bjeannet/prob/prob.html.

PRISM [20] can check the validity of properties specified in the logic PCTL [12] (PROBVERUS is however restricted to Markov chains). The originality of RAPTURE is to provide two reduction techniques that limit the state space explosion problem: automatic abstraction and refinement algorithms, and the so-called essential states reduction [6, 7]. The use of these techniques considerably reduces the high cost of the numerical analysis involved in the computation of the minimum and maximum reachability probabilities for PTSs. The price to pay is that RAPTURE cannot do verification of the full PCTL. Like PROBVERUS and PRISM, RAPTURE uses BDDs and MTBDDs [10, 1] to efficiently store the state space and the transition relation, but unlike them, RAPTURE uses these data structures to perform abstractions and process the refinement steps rather than to perform numerical analysis. Numerical analysis in RAPTURE is indeed performed by two different linear programming solvers: the first one uses sparse matrix on floating point numbers, the second uses dense matrix on exact rational numbers, which enables *exact* computations.

The paper is organized as follows: Section 2 shortly describes RAPTURE modelling language and Section 3, the properties it checks. Section 4 explains the machinery inside RAPTURE. Section 5 reports the performance of the tool on several case studies.

2 The model of systems: probabilistic transition systems

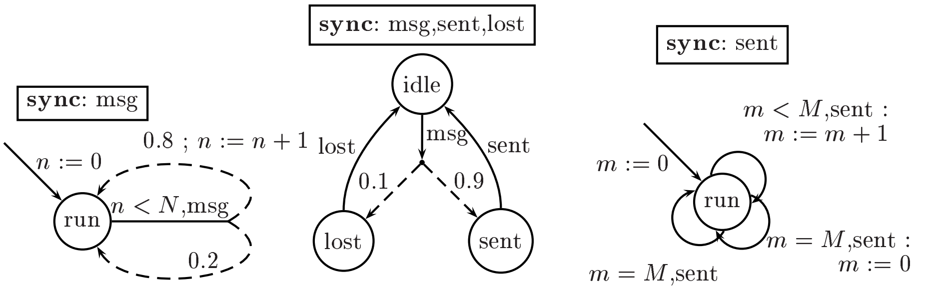
Probabilistic transition systems (PTS for short) generalize the well-known transition systems with probabilistic information. In a PTS, a transition does not lead to a single state but to a distribution over a set of states. The model we define is widely used (see, e.g. [23, 3, 18]) and is also known as Markov decision processes [21].

Fig. 1 depicts three PTSs and the result of their parallel composition. The **Sender** process sends a number N of messages. There is a probability 0.2 that it sends again the same message (here modelled by the absense of an increment of n). The **Line** process represents the transmission process. There is a probability 0.1 that the message is not transmitted to the **Receiver** process. The **Receiver** process just counts the number of received messages up to $M \geq N$. If M messages have been counted, the counter can be undermistically be reseted or maintained at this maximum value. In a "normal" execution, we should have $n = m$ at the end of the execution. The parallel composition operator uses synchronization over channels with a semantic *à la CSP* [17].

Such processes can be defined in RAPTURE with a textual language describing automata extended with finite-state variables, as shown on Fig. 1, where we have $N = M = 7$.

3 RAPTURE verification through reachability properties

Informally, the properties RAPTURE verifies are of the type: *"the probability to reach a set of* final *states from a given* initial *state is lower (or greater) than a*

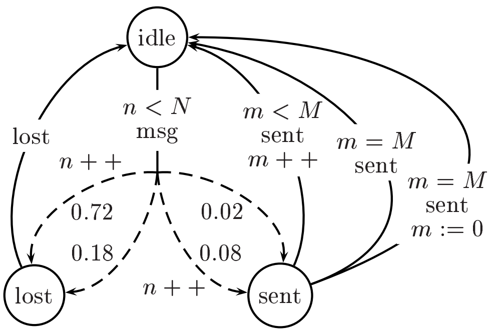


(a) Sender (b) Line (c) Receiver

```
channel msg,sent,lost;

process Sender {
 sync msg;
 var  n : uint(3);
 init #run and n=0;
 loc run:
   when n<7 sync msg goto
     { run 0.8 assign { n:=n+1 };
       run 0.2 }
}
process Receiver {
 sync sent;
 var  m : uint(3);
 init #run and m=0;
 loc run:
   when m<7 sync sent goto run assign { m:=m+1 };
   when m=7 sync sent goto run;
   when m=7 sync sent goto run assign { m:=0 };
}
process Line {
 sync msg,lost,sent;
 init #idle;
 loc idle:
   when true sync msg goto { lost 0.1 ; sent 0.9 };
 loc lost:
   when true sync lost goto idle;
 loc sent:
   when true sync sent goto idle;
}

system Sender,Receiver,Line;
```



(d) Parallel composition

Fig. 1. Example PTS

given bound, for any execution of the system". For instance, in the PTS described in Fig. 1, provided that the system has reached a state with $n = 5$ and $m = 7$, is the probability to reach a state with $n = 7, m = 1$ for any execution greater (or lower) than 0.5? The important point is that this probability depends on the considered execution, as soon as the system exhibits non-determinism. For instance, if the PTS is in the state **sent** with $m = 7$, it can nondeterministically reset m or leave it unchanged.

We specify the set of initial and final states of our reachability property by adding to the textual description of fig 1 the lines

```
initial Sender.n=5 and Receiver.m=7;
final   Sender.n=7 and Receiver.m=1;
```

The probability of the property is specified on the control line of our tool. The property in quote can be checked with the command '`rapture -ratio 200 -goal i0.5 ex.pts`'. Flag '`-goal i0.5`' indicates that we would like to check that the minimum probability of the reachability property is above 0.5. RAPTURE returns the report given in Fig. 2: the minimum probability to reach the final set is 0, and it was proven by discrete fixpoint computation only. Indeed, if m is never reset, it is not possible to ever reach a final state. Therefore the property is false. Alternatively, we can check whether the probability of reaching a final state is always lower than 0.9 using, for instance, the command '`rapture -goal s0.9 ex.pts`'. RAPTURE returns the report given in Fig. 3, which states that the property is true.

```
** computing processes and expressions
** Boolean composition
** Boolean analysis
    Initially: 10 state variables, 1024 states
    relation: 101 nodes
    Reordering...
    Reachability analysis from init: 192 states, 5 nodes
    Reachability analysis from initial: 72 states, 8 nodes
    psup0: 20 states, 12 nodes
    Extending final states
    Second reachability analysis from initial:
                                    64 states, 10 nodes
    Non-sink state space: 48 states, 9 nodes
    Pinf=0: 51 states
    All initial states are in pinf0; pinf = 0.0

Example: (Sender.loc=run)(Receiver.loc=run)
         (Line.loc=sent,lost)
         (Sender.n=5)(Receiver.m=7)
```

Fig. 2. Infimum prob. ≥ 0.5

```
idem
...
** computing global probabilistic transition function
** size of the transition function: 119 nodes, 619 paths, 6 leaves
System build and analysed (Boolean analysis) in 0.06 seconds

Building initial partition in 0 seconds

Step 1, automaton has 6 locations and 10 nails, 5684 bytes
essential automaton has 5 locations and 9 nails
...
Step 3, automaton has 28 locations and 33 nails, 19552 bytes
essential automaton has 15 locations and 20 nails
Computing psup: 13 variables, 20 constraints, 6 equalities;
pinf=-inf  psup=0.828196810136  diff=inf
analysis in 0 seconds
** Success **
After 3 steps and 5 divisions, final automaton has 28 locations
pinf=-inf psup=0.828196810136 diff=inf
Times in seconds:
  building: 0.06
  analysis: 0.03 (numerical computations: 0; refinement: 0.03)
```

Fig. 3. Supremum prob. ≤ 0.9

4 Verification method

The standard verification method for verifying reachability properties is to compute the *minimum* and/or the *maximum* probability of reaching a final state from an initial state, and to use it to deduce the truth of the property. The computation of those *extremum probabilities* is done by solving a system of fixpoint equations involving min and max operators over sets of linear expressions.

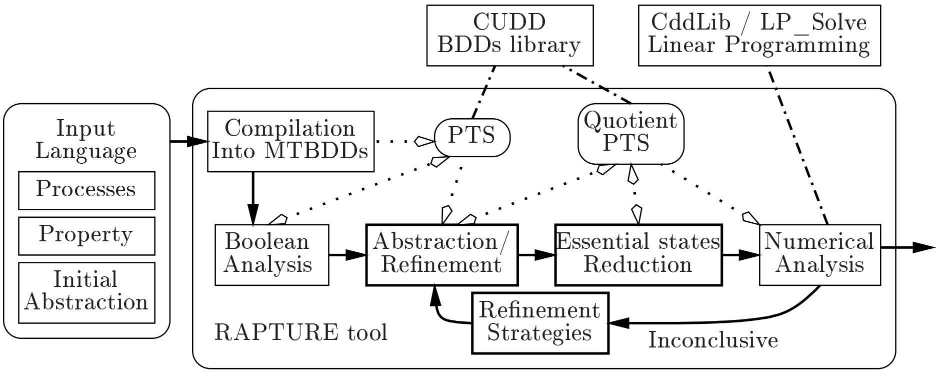


Fig. 4. Architecture of the Rapture tool

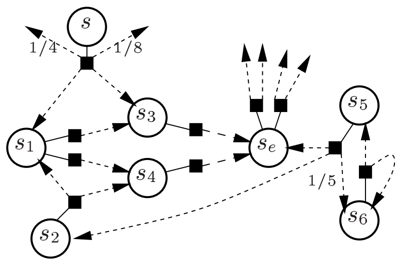
The two solving methods are the *value iteration method*, used together with a symbolic representation in [20], or the *linear programming method.* In both case, the number of unknowns is the number of the states of the analysed PTS. The aim of our tool is to reduce as much as possible the number of unknowns to be considered to compute extremum probabilities as efficiently as possible, possibly in an approximate way. The architecture of Rapture is depicted in Fig. 4.

Representation of Probabilistic Transition Systems. — Following [16, 8], we use BDDs (Binary Decision Diagrams [4]) to represent sets of states. Transition relations are represented with MTBDDs (Multi-Terminal Decision Diagrams), with real numbers as terminal nodes. We refer the reader to [6] for more details about the encoding and the chosen variable ordering in diagrams.

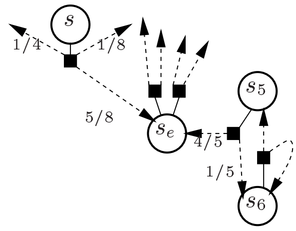
The reduction techniques implemented in the tool. — The purpose of the reduction techniques which have been implemented is to overcome the strong limitation in the size of the systems that can be verified. Three reduction techniques are implemented.

Discrete precomputations. — We use a standard precomputation of certain sets of system states in order to simplify the system before applying linear programming techniques. These sets are: the set of all reachable states *Reach*, and for each $p \in \{0, 1\}$ the set of states with infimum (resp. supremum) probability p of reaching F. These latter sets of states are denoted $\mathsf{P}^{\text{inf}}_{=0}$, $\mathsf{P}^{\text{inf}}_{=1}$, $\mathsf{P}^{\text{sup}}_{=0}$, and $\mathsf{P}^{\text{sup}}_{=1}$, respectively. All of the above sets can be computed using discrete fixpoint analysis [9] on a Boolean abstraction of the system. In our case these analysis are implemented using BDDs. As we restrict our attention to simple reachability properties, we can use these analysis to reduce the state space under consideration, unlike tools that handle more complex probabilistic properties involving nested fixpoints [13, 8, 15].

The Abstraction and successive refinement method. — The main principle of our method is founded on abstraction and successive refinements of the initial abstract model. The idea is to try the verification of the property on a (rough) abstraction of the model, induced by a partition of the state space, and in case of



(a) Initial situation



(b) After reduction

Fig. 5. Example of essential reduction

failure of the verification to refine this abstract model into a finer abstraction, on which better approximations of extremum probabilities can be computed. This process is stopped as soon as a verdict (true or false) to the property can be deduced from the computations.

The initial partition (or abstraction) should at least separate initial states, final states, and others, for correctness reasons [6]. The user can also specify a set of PTSs and of variables that should not be abstracted in the initial partition. This choice usually depends on the property to check, but due to the refinement process a bad choice of these parameters will only slow down the verification of the property. By using a suitable variable ordering in BDDs, the computation of an abstract PTS from a concrete PTS and a partition of the state space of size k can be performed in $\mathcal{O}(k)$ BDD operations, which are in turn linear or quadratic in the number of nodes of the involved BDDs/MTBDDs [6].

If a partition is not detailed enough to deliver a good approximation of the extremum probabilities involved in the property, heuristic strategies are used to refine it into a more detailed partition. They are all based on the stabilization of the partition w.r.t. a transition relation ([7]). Several heuristics are offered to the user, who chooses them when he launches the verification. These strategies do not necessarily stabilize each class w.r.t. the current partition in one step, but to proceed in a more incremental and guided way. They differ on the abstract transition that is used as a basis for the split of a class. The user can also tune the ratio between the refinement and the analysis steps.

Essential states reduction. — We developed an additional reduction technique, the *essential state reduction* [7]. This abstraction, which preserves extremum probabilities, is based on the observation that most transitions in a PTS are non-probabilistic, i.e. they have a unique successor state. To give the intuition of this reduction, suppose that a fragment of a PTS looks like the one depicted in Fig. 5(a). All executions starting from s_1, s_2, s_3, s_4 are leading to the state s_e with probability 1. If we are only interested in probabilities, we can reduce the system by representing all of the above states via the single state s_e, and in addition merge (add) the probabilities for any distribution to enter the states represented by s_e, as shown on Fig. 5(b).

This essential state reduction is applied on the abstract model, just before the generation of the LP problem.

Numerical methods available in the tool. — The previously described reduction techniques are independent from the chosen method for solving the numerical equations. The tool is currently connected to two different LP solver. The first one, LP_SOLVE [2], uses sparse matrix over floating point numbers as its internal representation. Sparse matrices are very useful for our purpose, as the size of the support of the distributions (i.e., the number of successor states of the distributions) is generally very small in our models. However we encountered numerical precision problems with our first case studies, the Bounded Retransmission Protocol [14, 6]. This was not due to proper numerical instability, but to the fact that with IEEE floating point numbers, $1 - 1e20 = 1$!! Very small probabilities indeed appear in this case study, if a great number of retransmissions is allowed. The second one, which is part of the polyhedra library CDDLIB [11], uses dense matrices of rational numbers. It is useful when the above-mentioned problem arises, or in cases where numerical instability is observed and *exact* results are wanted. However, in other case LPSOLVE performs much better: floating point arithmetic is cheap compared to multi-precision rational (integer) arithmetic.

5 Experiments

We have conducted several experiments in order to evaluate our reduction and refinement strategies as well as our implementation.

Bounded Retransmission Protocol. — This protocol, originally studied in [14], is based on the well-known alternating bit protocol but allows for a bounded number of retransmissions of a chunk, i.e., part of a file, only. So, eventual delivery is not guaranteed and the protocol may abort the file transfer. We use the version presented in [6], where probabilities model the possible failures of the two channels used for sending chunks and acknowledgments, respectively. In Table 1 we check the maximum probabilities that the sender does not report a successful transmission. We consider a file composed of either 16 or 64 chunks, and N is the number of allowed retransmissions. We use here the dense matrix based solver with exact arithmetic, because probabilities of very different magnitude order appear in LP problems, which makes the usual floating point arithmetic unstable. The initial partitioning is here performed w.r.t. the explicit control structure of the specification: only variables are abstracted.

The meaning of row labels in the table is the following: `#reach` is the number of states reachable from root states, `#rel` is the number of relevant states after Boolean preprocessing, and `time` is the time needed to build and preprocess the PTS. The three next sets of rows details the refinement process for different upper bounds for $\mathsf{P}^{\mathrm{sup}}$. `#refin` is the number of refinement steps, `#abst` the number of states of the most refined abstract PTS, `#ess` the number of its essential states, `psup` the computed probability, `verd` is the verdict (true or false), and `time(a+r)` gives the time spent in numerical analysis and in refinement process. When the verdict is false, the refinement has gone to the stable partitioning of the PTS and gives the actual $\mathsf{P}^{\mathrm{sup}}$ of the concrete PTS.

Table 1. Results in BRP

		file length 16				file length 64
	MAX	2	4	8	15	15
	#reach.	3908	6060	10364	17896	58024
	#relev.	1014	1790	3342	6058	26362
	time	0.94	1.10	1.59	1.75	5.20
$\leq 10^{-3}$	#refin.	4	5	6	6	6
	#abst.	52	89	161	161	161
	#ess.	24	42	85	85	85
	psup	3.09e-04	4.27e-06	7.89e-06	7.89e-06	7.89e-06
	verd.	T	T	T	T	T
	time(a+r)	0.07+0.88	0.72+0.83	2.96+1.68	2.95+1.72	2.97+2.62
$\leq 10^{-10}$	#refin.	9	10	7	7	7
	#abst.	375	675	242	247	247
	#ess.	152	272	108	115	115
	psup	2.65e-05	2.35e-08	7.01e-12	7.01e-12	7.01e-12
	verd.	F	F	T	T	T
	time(a+r)	0.58+2.56	3.30+5.42	3.15+2.07	3.54+2.33	3.51+3.55
$\leq 10^{-90}$	#refin.	9	10	11	12	16
	#abst.	375	675	1275	2325	9765
	#ess.	152	272	512	932	3908
	psup	2.65e-05	2.35e-08	1.85e-14	3.87e-25	3.87e-25
	verd.	F	F	F	F	F
	time(a+r)	0.58+2.56	3.30+5.42	15.87+11.00	186.58+22.06	1209.72+165.3

Observe the efficiency of Boolean preprocessing and essential states reduction, which gives both a reduction of one third in average. Notice also that it is nearly as easy to prove $\mathsf{P}^{\text{sup}} \leq 10^{-3}$ for big instances of BRP than for small ones: that means that the refinement strategy works well and will not perform too many useless splits. It can also be observed that checking smaller upper bounds can still be performed on very small abstract PTSs, compared to the concrete one, even reduced by preprocessing, and also compared to the stable partitioning (row $\mathsf{P}^{\text{sup}} \leq 10^{-90}$).

The probabilistic dinning philosophers. — In this example, which originates from [19] and has been analysed using PRISM [20], N philosophers are trying to eat. We want to prove a lower bound on the probability for some process to eat after a number of time units specified by value of deadline, with the aditional requirement that a philosopher cannot stay idle for more than K steps. Table 2 shows results for $N = 3$ and different values of K. The chosen deadline corresponds to the smallest one for which the property holds with a probability more than 0.

Here we give in the table not only the number of abstract and essential states, but also in each case the number of abstract distributions. We use the sparse matrix based solver with ordinary floating point arithmetic. The initial partition is chosen to be obtained by abstracting everything but the counter used for the deadline, as it is clear the value of the deadline is of fundamental importance for the studied property. Most of the encouraging observations made for the BRP are still true. The only exception is that essential state reduction does not perform as good as in the BRP. Execution times are much higher, because MTBDDs are much bigger, and the abstract PTSs are much more complex, which results in very big LP problems. Still, refinement remains much cheaper than analysis, and state space reduction between the concrete PTS and the abstract one allowing to prove the property is impressive.

Table 3 compares various refinement options and initial control structures on a particular instance of the system. The first column corresponds to the options that work best and that were used in the previous table: the initial partition detail only the counter for the deadline, and we use n-ary division, giving priority

Table 2. Results in Dining Philosophers with $N = 3$

	K	4	5	6
	deadline	23	27	31
	#reach.	1.00e06	1.97e06	3.40e06
	#relev.	121041	271287	488859
	time	14.4	23.6	34
$\leq \frac{1}{16}$	#refin.	5	7	8
	#abst.	3064/11536	16903/52435	35780/111084
	#ess.	2778/11250	14442/49974	30361/105665
	pinf	0.0625	0.0625	0.0625
	verd.	T	T	T
	time(a+r)	49.6+79.5	2120+590	10353+1462
$\leq \frac{1}{8}$	#refin.	7	8	9
	#abst.	8512/22757	21011/59866	37542/114703
	#ess.	6668/20913	16996/55851	31656/108817
	pinf	0.125	0.125	0.125
	verd.	T	T	T
	time(a+r)	290+220	3683+712	20335+1575

Table 3. Results in Dining Philosophers with $N = K = 3$ and deadline $= 19$

	control option	deadline nary+osl	deadline bin+osl	deadline nary+lso	deadline nary+a	ctrl. struct. nary+osl
	#reach.	408397				
	#relev.	30018				
	time	6.14				
$\leq \frac{1}{4}$	#refin.	2	2	4	2	4
	#abst.	51/87	35/122	882/2880	140/680	5861/12972
	#ess.	51/87	35/122	827/2825	140/680	4109/11196
	pinf	0.25	0.25	0.25	0.25	0.25
	verd.	T	T	T	T	T
	time(a+r)	0.03+3.85	0.02+3.48	5.16+16.79	0.19+5.09	53.6+44.8

to different types of probabilistic transitions. Using binary divisions gives similar results (second column). Column 3 shows that inverting the priority of the different types of split in column 1 gives very bad results: a much more refined system is needed to prove the property. Last column illustrates the importance of a good initial partition. Here, we generated it according to the explicit control structure of the philosopher, and it produces very bad result.

Binary Exponential Backoff Algorithm in the IEEE 802.3. — This protocol is part of the CSMA/CD protocol and is used to state the policy in which machines retry to access the medium after a collision was detected. The protocol works as follows. After a collision the time is divided into slots. Each of the colliding hosts waits 0 or 1 slots (each with probability 1/2) before retrying to access the medium. If collision happens again each host will wait 0, 1, 2, or 3 slots with probability 1/4. Collision may repeat several times. In its ith collision, a host must choose a waiting time between 0 and $2^i - 1$ with probability $1/2^i$ each. After the Kth collision, it will only choose between 0 and $2^K - 1$, and after the Nth unsuccessful attempt ($N \geq K$), the host will gave up. When no collision happens, the only transmitting host seize the line and its message is transmitted. The appendix shows details on the modelling of the binary exponential backoff method and the property under study using the Rapture modelling language.

The property we study is whether one given host gives up with probability less than or equal to p. Results are reported in Table 4 where 3 hosts are considered, the number of attempts to access the channel before giving up is $N = 5$, and the exponential variable grows until $K = 2$. The check probability p varies and is specified in the table. We have done three types of run: selecting two types of initial partition (see the appendix) and using the default initial partition which ammounts to distinguishing states if they belong to different

Table 4. Results in the Binary Exponential Backoff $N = 5$, $K = 2$

#reach. 752170 #relev. 752170 time 33.14						
Technique		nary+osl	nary+lso+init$_1$	nary+lso+init$_2$	nary+osl+init$_2$	nary+a+init$_2$
$\leq 5 \cdot 10^{-2}$	verd.	T	T	T	T	T
	#refin.	8	7	8	8	7
	#abst.	28237	6678	10099	7720	9994
	#ess.	23326	5616	8409	6535	8391
	psup	0.0299973	0.0161254	0.021849	0.0491828	0.0204969
	time(a+r)	2418.47+1265.01	96.3+555.57	235.67+942.24	124+788.51	212.4+804.83
$\leq 1 \cdot 10^{-2}$	verd.	T	T	T	T	T
	#refin.	9	8	9	10	8
	#abst.	56840	13637	20819	30701	20111
	#ess.	47548	11395	17705	25580	17077
	psup	0.00388816	0.00745022	0.00471054	0.000386098	0.00470079
	time(a+r)	10169.7+2023.94	477.85+980.62	1293.17+1599.61	2423.8+2111.9	1137.99+1432.39
$\leq 5 \cdot 10^{-3}$	verd.	T	T	T	T	T
	#refin.	9	9	9	10	8
	#abst.	56840	28143	20819	30701	20111
	#ess.	47548	24147	17705	25580	17077
	psup	0.00388816	0.00143866	0.00471054	0.000386098	0.00470079
	time(a+r)	10455.7+2140.99	2892.53+1748.86	1296.38+1622	2228.21+2017.41	1244.52+1435.45
$\leq 1 \cdot 10^{-3}$	verd.	T	T	T	T	T
	#refin.	10	10	10	10	9
	#abst.	97642	39209	30701	30701	30701
	#ess.	79655	33449	25580	25580	25580
	psup	0.000386098	0.000386098	0.000386098	0.000386098	0.000386098
	time(a+r)	29182.8+4096.43	5745.09+2464.28	2933.94+2349.45	2288.85+2084.56	2767.8+2177.3
$\leq 5 \cdot 10^{-4}$	verd.	T	T	T	T	T
	#refin.	10	10	10	10	9
	#abst.	97642	39209	30701	30701	30701
	#ess.	79655	33449	25580	25580	25580
	psup	0.000386098	0.000386098	0.000386098	0.000386098	0.000386098
	time(a+r)	28484.9+3922.24	5755.49+2469.5	3005.25+2330.12	2036.3+1865.65	2987.38+2348.35
$\leq 1 \cdot 10^{-4}$	verd.	F	F	F	F	F
	#refin.	10	10	10	10	9
	#abst.	97642	39209	30701	30701	30701
	#ess.	79655	33449	25580	25580	25580
	psup	—	—	—	—	—
	time(a+r)	28297.5+3931.48	5762.32+2470.58	2753.41+2345.52	2241.72+2049.34	2835.37+2240.27

control structure. Again, the importance of selecting a good initial partition is evident. We mention that combinations "lso" and "osl+init$_1$" (not shown on the table) showed instability problems[2]. Table 5 shows the same exercise but with $K = 3$. Both in Table 4 and Table 5 we have highlighted the techniques with best performance. Clearly the refinement priority order "lso" has done worst, but the other two techniques have shown rather incomparable results. The case "a", in which refinement is done w.r.t. all transitions, is faster on refining and spends more time per iteration than the case "osl". On average, this last technique seems to do better.

We carried out the experiments using the sparse matrix based solver with ordinary floating point arithmetic. We tried larger settings, but unfortunately many of them suffered the numerical instability problem. When running the dense matrix based solver with exact arithmetic it required a much larger use of memory which could not be allocated on the machine it was running. It would be very useful for such cases to have a solver using both sparse matrices and exact arithmetic. However, such a solver is still to be implemented!

[2] Refinement can be guided by defining a priority order on the type of transition that should be selected first to partition an equivalence class. The types are: "l" for *looping*, i.e. non-probabilistic transitions that loops on the same abstract state; "s" for *single*, i.e. non-probabilistic transitions leading to a different abstract state; "o" for *others*, namely, the probabilistic transitions.

Table 5. Results in the Binary Exponential Backoff $N = 5$, $K = 3$

#reach. $3.40796 \cdot 10^6$	#relev. $3.40796 \cdot 10^6$	time 111.04		
	Technique	nary+lso+init$_2$	nary+osl+init$_2$	nary+a+init$_2$
$\leq 5 \cdot 10^{-2}$	verd.	T	T	T
	#refin.	8	9	6
	#abst.	14877	30657	13509
	#ess.	14088	27191	12939
	psup	0.0168896	0.00637896	0.0190634
	time(a+r)	816.03+1957.2	2893.04+4459.17	563.85+1680.64
$\leq 1 \cdot 10^{-2}$	verd.	T	T	T
	#refin.	9	9	7
	#abst.	32313	30657	27740
	#ess.	27727	27191	23769
	psup	0.00499646	0.00637896	0.00601642
	time(a+r)	3693.23+4046.84	3051.12+4397.42	2687.23+3279.83
$\leq 5 \cdot 10^{-3}$	verd.	T	T	T
	#refin.	9	10	8
	#abst.	32313	61970	55703
	#ess.	27727	53858	48628
	psup	0.00499646	0.00067507	0.00104156
	time(a+r)	3069.24+3585.62	10013.6+6426.98	12976.6+6077.6
$\leq 1 \cdot 10^{-3}$	verd.	T	T	T
	#refin.	10	10	9
	#abst.	66635	61970	97831
	#ess.	58173	53858	85099
	psup	0.000636677	0.00067507	9.60093e-05
	time(a+r)	27220.8+6555.4	10111.1+6754.95	34694.1+10976.3
$\leq 5 \cdot 10^{-4}$	verd.	T	T	T
	#refin.	11	11	9
	#abst.	97831	97831	97831
	#ess.	85099	85099	85099
	psup	9.60093e-05	9.60093e-05	9.60093e-05
	time(a+r)	46629.2+10074	27261.4+9426.13	34293.2+10615.6
$\leq 1 \cdot 10^{-4}$	verd.	T	T	T
	#refin.	11	11	9
	#abst.	97831	97831	97831
	#ess.	85099	85099	85099
	psup	9.60093e-05	9.60093e-05	9.60093e-05
	time(a+r)	45332+9899.48	27216+9408.45	34021.5+10672.9

6 Conclusion

We presented in this paper the probabilistic verification tool Rapture. We described its functionalities and the principles of its verification method. We gave also a set of experimental results that illustrates the behaviour of the implemented techniques and their efficiency.

Acknowledgments We thank the cooperation of Henrik E. Jensen in previous works, helping to develop the foundations of the current tool Rapture.

References

1. R. Bahar, E. Frohm, C. Gaona, G. Hachtel, E. Macii, A. Pardo, and F. Somenzi. Algebraic decision diagrams and their applications. *Formal Methods in System Design*, 10(2/3):171–206, April 1997.
2. M. Berkelaar. LP_Solve: Mixed integer linear program solver. `ftp://ftp.ics.ele.tue.nl/pub/lp_solve`.
3. A. Bianco and L. de Alfaro. Model checking of probabilistic and nondeterministic systems. In *Proceedings of FSTTCS'95*, *LNCS* 1026, 1995.
4. R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–692, 1986.
5. E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM TOPLAS*, 8(2), 1986.

6. P.R. D'Argenio, B. Jeannet, H.E. Jensen, and K.G. Larsen. Reachability analysis of probabilistic systems by successive refinements. In *Proceedings of PAPM-PROBMIV 2001*, *LNCS* 2165, Aachen (Germany), September 2001.
7. P.R. D'Argenio, B. Jeannet, H.E. Jensen, and K.G. Larsen. Reduction and refinement strategies for probabilistic analysis. In *Proceedings of PAPM-PROBMIV 2002*, *LNCS*, Copenhagen (Denmark), July 2002.
8. L. de Alfaro, M. Kwiatkowska, G. Norman, D. Parker, and R. Segala. Symbolic model checking of concurrent probabilistic processes using MTBDDs and the Kronecker representation. In *Proceedings of TACAS'00*, *LNCS* 1785, 2000.
9. Luca de Alfaro. *Formal Verification of Probabilistic Systems*. PhD thesis, Stanford University, 1997.
10. M. Fujita, P.C. McGeer, and J.C.-Y. Yang. Multi-terminal binary decision diagrams: An efficient data structure for matrix representation. *Formal Methods in System Design*, 10(2/3):149–169, April 1997.
11. K. Fukuda. CDDLIB. `ftp://ftp.ifor.math.ethz.ch/pub/fukuda/cdd`.
12. H.A. Hansson and B. Jonsson. A logic for reasoning about time and reliability. *Formal Aspects of Computing*, 6:512–535, 1994.
13. V. Hartonas-Garmhausen and S. Campos. ProbVerus: Probabilistic symbolic model checking. In *Proceedings of ARTS'99*, *LNCS* 1601, 1999.
14. L. Helmink, M. Sellink, and F. Vaandrager. Proof-checking a data link protocol. In *Proc. International Workshop TYPES'93*, *LNCS* 806, 1994.
15. H. Hermanns, J.-P. Katoen, J. Meyer-Kayser, and M. Siegle. A Markov chain model checker. In *Proceedings of TACAS'00*, *LNCS* 1785, 2000.
16. H. Hermanns, J. Meyer-Kayser, and M. Siegle. Multi terminal binary decision diagrams to represent and analyse continuous time Markov chains. In *Procs. of Int. Workshop on the Numerical Solution of Markov Chains.* Prensas Universitarias de Zaragoza, 1999.
17. C.A.R. Hoare. *Communicating Sequential Processes.* Prentice Hall, 1985.
18. B. Jonsson, K.G. Larsen, and W. Yi. Probabilistic extensions in process algebras. In J.A. Bergstra, A. Ponse, and S. Smolka, editors, *Handbook of Process Algebras.* Elsevier, 2001.
19. D. Lehmann and M. Rabin. On the advantages of free choice: A symmetric fully distributed solution to the dining philosophers problem. In *Proc. 8th Symposium on Principles of Programming Languages*, 1981.
20. PRISM Web Page. `http://www.cs.bham.ac.uk/~dxp/prism/`.
21. M.L. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming.* John Wiley & Sons, 1994.
22. J. P. Queille and J. Sifakis. Specification and verification of concurrent systems in CAESAR. In *International Symposium on Programming.* LNCS 137, Springer Verlag, April 1982.
23. R. Segala. *Modeling and Verification of Randomized Distributed Real-Time Systems.* PhD thesis, Massachusetts Institute of Technology, 1995.
24. M. Stoelinga and F. Vaandrager. Root contention in IEEE 1394. In *Proceedings of ARTS'99*, *LNCS* 1601, 1999.

Appendix: Details on the Binary Exponential Backoff Method

In the following, we report the RAPTURE model of the binary exponential backoff algorithm. This algorithm assumes the time is divided in slots. Process `Clock` controls such division. In order to proceed in an orderly fashion, each slot is divided in three sections marked by actions Tick, Tack, and Tock. In the first section (previous to Tick), hosts attempt to seize the line. Each host (modelled by process Host_i) indicates its will to access the line by incrementing the global variable chan_req. The second section (between Tick and Tack) is the middle of the slot and is used for each attempting host to check if there was a collision (i.e. if chan_req>1) and proceed according to the algorithm explained in Section 5. The last section (between Tack and Tock) is only used to reset variable chan_req in order to restart the process.

```
channel
   Tick, Tack, Tock ;

var
   chan_req         : uint ( DIM_HOST ) ;
   line_seized_flag : bool ;            // These two flags are used to
   gave_up_flag     : bool ;            // analyse some properties
                                        // not reported in this article.

process Clock {

  sync
      Tick, Tack, Tock ;

  init ( #start  and  ( chan_req = 0 )
         and  not line_seized_flag  and  not gave_up_flag ) ;

  loc start:
    when true goto begin_slot ;
  loc begin_slot :
    when true sync Tick goto mid_slot ;
  loc mid_slot :
    when true sync Tack goto reset ;
  loc reset :
    when true goto end_slot assign { chan_req := 0 } ;
  loc end_slot :
    when true sync Tock goto begin_slot ;
}
```

Process Host_i uses two constants: MAX_NR_ATTEMPTS, that represents the maximum number of attempts to access the line (N in Section 5), and MAX_EXP that is the maximum exponential value allowed (MAX_EXP $= 2^K$ where K is as in Section 5). Variables nr_attempts and exp_val are used to save the current number of attempts and the current exponential value (from which the random value will be chosen) respectively. Variable slots_to_wait contains the slots to wait before attempting to seize the line. Originally, slots_to_wait takes a random value uniformly distributed between 0 and exp_val $- 1$. This random choice is not straightforward since the RAPTURE modelling language does not allow probabilities to depend on variables. Therefore it is coded in a loop with the help of an auxiliary variable aux_exp. The random setting of slots_to_wait takes place in location process_collision, where the host will iterate ($\log_2$ exp_val) times. In each iteration i, variable slots_to_wait will be incremented by 2^i with probability 0.5. Variable aux_exp carries the appropriate 2^i value.

```
process Host_i {

  sync
       Tick, Tack, Tock ;

  var
       // Be aware that dimensions depend on constants MAX_NR_ATTEMPTS
       // and MAX_EXP
       nr_attempts   : uint ( DIM_TRI ) ;
       exp_val       : uint ( DIM_EXP ) ;
       aux_exp       : uint ( DIM_EXP ) ;
       slots_to_wait : uint ( DIM_EXP ) ;

  init ( #wait_tick  and  (nr_attempts = 0)  and  (exp_val = 1)
          and  (aux_exp = 1)  and  (slots_to_wait = 0) ) ;

  // Begining of the slot: try to seize the line or just wait.
  loc wait_tick :
    when ( slots_to_wait > 0 ) sync Tick
         // Wait these slot
         goto wait_tack
         assign { slots_to_wait := slots_to_wait - 1 } ;
    when ( slots_to_wait = 0 )
         // Do not wait any longer and try to seize the line
         goto init_cycle assign { chan_req := chan_req + 1 };
  loc init_cycle :
    when true sync Tick goto check_collision ;
  loc check_collision :
    when ( chan_req = 1 )
         // The attempt was succesful. Line seized
         goto line_seized
         assign { line_seized_flag := true } ;
    when ( (chan_req > 1)  and  (nr_attempts >= MAX_NR_ATTEMPTS) )
         // There was collision and maximum number of attepmts exceeded.
         // Give up.
         goto gave_up
         assign { gave_up_flag := true } ;
    when ( (chan_req > 1)  and  (nr_attempts < MAX_NR_ATTEMPTS) )
         // The attempt was unsuccesful. A collision occurred.
         // Set values for next attempt.
         goto process_collision
         assign { nr_attempts := nr_attempts + 1 ;
                  aux_exp := 1 ;
                  slots_to_wait := 0 } ;
```

```
  // Choose a random value to wait until next attempt.
  loc process_collision :
    when ( (aux_exp > exp_val)  and  (exp_val >= MAX_EXP) )
         goto wait_tack ;
    when ( (aux_exp > exp_val)  and  (exp_val < MAX_EXP) )
         goto wait_tack
         assign { exp_val := 2 * exp_val } ;
    when ( aux_exp <= exp_val )
         goto {
           process_collision .5
             assign { aux_exp := 2 * aux_exp } ;
           process_collision .5
             assign { slots_to_wait := slots_to_wait + aux_exp ;
                      aux_exp := 2 * aux_exp }
         } ;
  // Wait until the middle section of the slot is finished.
  loc wait_tack :
    when true sync Tack goto wait_tock ;
  // Wait until the last section of the slot is finished.
  loc wait_tock :
    when true sync Tock goto wait_tick ;
  loc line_seized :
    when true goto line_seized ;
  loc gave_up :
    when true goto gave_up ;
}
```

The property under study checks the probability of reaching a state in which `Host_0` gives up. The probability of such final condition must be calculated from the beginning of the process, namely when the `Clock` is about to start and no host attempted yet to seize the line (i.e., `chan_req = 0`). The property is stated in RAPTURE by the following specification lines:

```
initial ( #Clock.start  and  ( chan_req = 0 ) ) ;
final   #Host_0.gave_up ;
```

The initial partition init_1 is specified in RAPTURE by the sentence

```
control  #Host_0.aux_val, #Host_0.exp_val ;
```

which states that states with different values for variables `aux_val` and `exp_val` must be distinguished in the initial partition. The initial partition init_2 is given by

```
control  #Host_0.exp_val ;
```

In this case, only states with different values for variable `exp_val` must be distinguished.

YAHODA: verification tools database*

J. Crhová, P. Krčál, J. Strejček, D. Šafránek, P. Šimeček

Faculty of Informatics, Masaryk University Brno,
Czech Republic
{xcrhova,xkrcal,strejcek,xsafran1,xsimece1}@fi.muni.cz

Abstract. We present a web server YAHODA [9], which is designed to provide unified information about currently available verification tools. The server software allows the tools developers to insert and maintain the information about their tools by their own. In the paper we describe the organization of the database, its main features, and the maintenance of the repository.

1 Introduction

The need of a comprehensive information resource on verification tools has been already reflected by several public sources [4, 5, 3]. There are specialized databases concerning Petri Nets [7, 6] or *HOL*-related tools [8]. See also the CONCUR project [10] that includes comparison of verification tools. However, most of those sources aimed at verification tools in general, are not well-structured and difficult to maintain. These databases often do not support the user in choosing the best tool according to the user's needs (the only exception is [6]). One must browse through all the tools and carefully read the info about each of them. Moreover, some of the databases, like [1, 2], are rather behind the times. It goes with the fact that the development in this area is very fast and therefore it is difficult to keep the information up-to-date.

YAHODA [9] is a project that aims at overcoming at least some of the problems mentioned above. YAHODA is a web application based on a relational database. It is the tool developer, not the YAHODA administrator, who maintains the information about the tool via an authorized web access. We hope this is the most effective way of fighting the inaccuracy. The information in the database is structured and thus allows the YAHODA user more flexible search for desired information. At the same time, YAHODA provides some basic taxonomy.

* This work has been partially supported by the Grant Agency of Czech Republic, grant No. 201/00/1023.

2 Information in YAHODA

In this section, we describe the structure of database records.

The amount of information to be stored is set up to be general enough on one hand (e.g., for searching and comparing tools), on the other hand we tried to include all the necessary details.

Every tool record includes the following information:

Purpose of the tool – the purpose of the tool is basically characterized as (combination of) *model checking (linear or branching time), equivalence checking, or theorem proving.*

Specific features – the distinguished specific features are *real-time verification, probabilistic verification or verification of the hybrid systems.*

Graphical interface – recognized graphical features are *graphical user interface, graphical specification of the system, graphical simulation.*

Description – a word description of the tool.

Modelling languages – languages for specification of systems.

Logics – logics used by the tool (only for model checkers).

Equivalences – equivalences checked by the tool (only for equivalence checkers).

Counterexample – distinguished features are generation and possible visualization of the counterexample.

Availability – the basic categories are *free, commercial, or free under specific conditions.* In the last case, the conditions can be explicitly added. *Availability of the source code* can be checked and *developing languages* can be specified.

Platforms – platforms for which the tool is available. The basic categories are *Unix* and *Windows.*

Last release – number and date of release of last available version are specified.

References – homepage and email of the tool are specified. Other relevant links can be added.

Moreover, supplementary information about logics, equivalences, and modelling languages used by the tools can be added. Whenever, it consists of a brief description and references to the literature.

The Fig. 1 shows the presentation of the tool record in YAHODA.

3 Searching in YAHODA

The most important feature for the users of YAHODA is the support for quick and transparent search in the database.

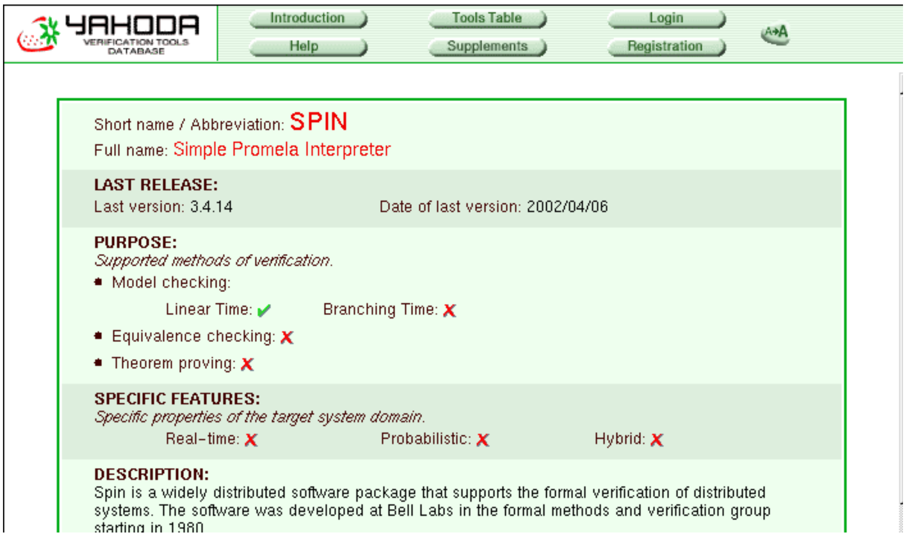


Fig. 1. Detailed information about a tool

All tools and their main features are displayed in graphical table, so called *tools table* (Fig. 2). The user can set some requirements on the tools concerning purpose of the tool, specific features (real-time verification, probabilistic verification or verification of the hybrid systems), information about the graphical interface, availability of the tool, and the platforms for which the tools are available. Requirement (called filtering query) consists of the column names of the tools table connected by the Boolean operators (Fig. 2). The user can choose them by clicking on the corresponding hypertextual column names and symbols of the Boolean operators. Tools table consists of tools satisfying the requirement only.

Users can get the detailed information about tool by clicking on the tool name in the tools table.

4 Maintaining records in YAHODA

To cope with the inaccuracy and the obsoleteness of the information we decided to open the database to the authors of the tools. Authors can add their tools into the database and maintain their records on their own (via the authorized access). Thus, after any modification of the tool, the author does not need to wait for the database administrator to update the tool's record.

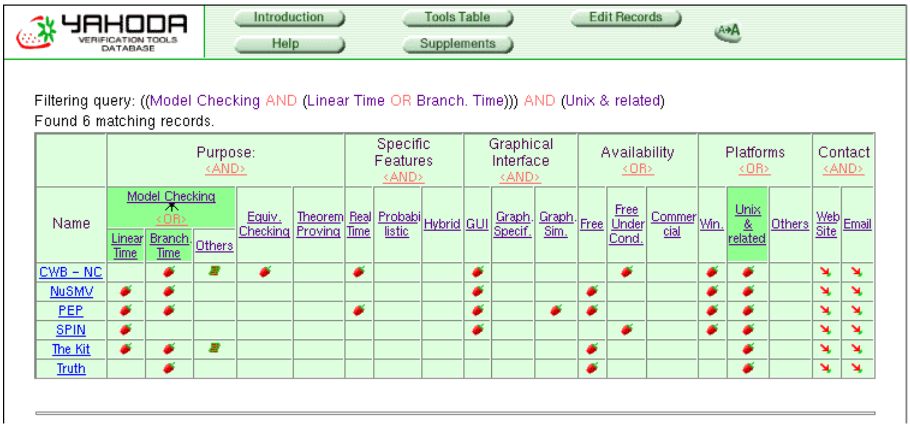


Fig. 2. Tools table

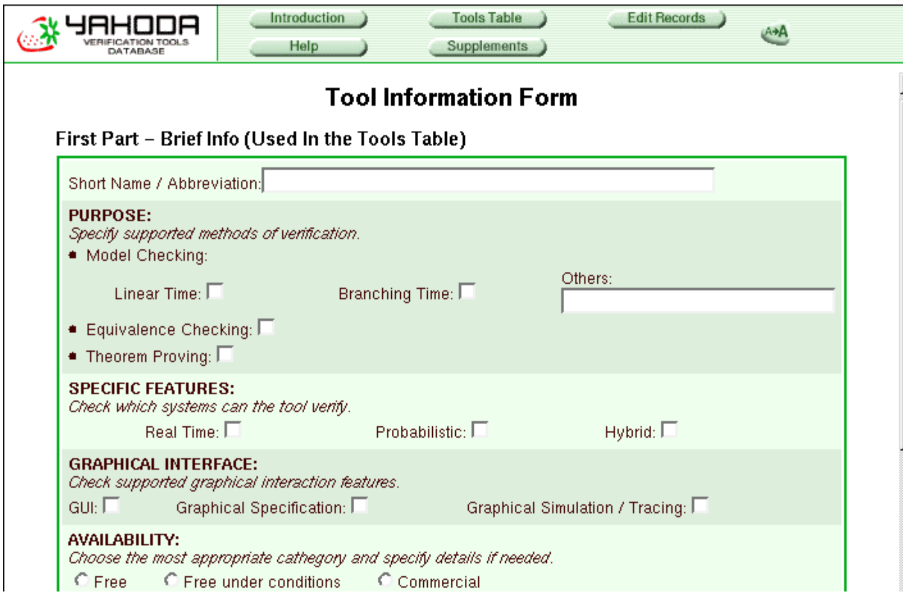


Fig. 3. Adding new tool

To obtain an authorized access, authors of tools should register to YAHODA. This can be done by filling authors' name and email address into the registration form. The server automatically creates authors' accounts and will send them a login and password immediately. Now the authors can insert new records of the tools by filling the form (Fig. 3). Moreover they can update this information any time in the future.

5 Conclusion and future plans

We described the database of verification tools YAHODA and showed how it is organized. Further, we described the main features of this database.

We intend to start YAHODA with the information about the tools presented at the Tools Day. After this, we will challenge all other tools' authors to add their tools to YAHODA.

Moreover, we are prepared to modify the database design, according either to external requests or to the development in the verification tools area. In general, we are asking all the users to send us their comments and suggestions how to improve the server.

Our objective is to make YAHODA a comprehensive source containing up-to-date information about verification tools.

Acknowledgments: We want to thank our colleagues who participated in the work on YAHODA, especially Luboš Brim, Ivana Černá, Jan Obdržálek, and Radek Pelánek.

References

1. BRICS tools home page.
   http://www.brics.dk//users/btools/index.html.
2. Database of existing mechanized reasoning systems.
   http://www.calculemus.org/MathUniversalis/3/listsoft.html.
3. Formal methods.
   http://www.afm.sbu.ac.uk/.
4. Formal methods education resources, tool pages.
   http://www.cs.indiana.edu/formal-methods-education/Tools/.
5. Formal methods europe.
   http://www.fmeurope.org/.
6. The petri net tools survey.
   http://home.arcor-online.de/wolf.garbe.petrisoft.hml.
7. Petri Nets tool database.
   http://www.daimi.au.dk/PetriNets/tools/db.html.
8. Systems related to HOL.
   http://www.cl.cam.ac.uk/Research/HVG/HOL/HOL.html#related.
9. YAHODA.
   http://www.fi.muni.cz/yahoda/.
10. Eric Madelaine. Verification tools from the CONCUR project. *EATCS Bulletin*, 47:110–120, 1992.

Publications in the FI MU Report Series are in general accessible
via WWW and anonymous FTP:

`http://www.fi.muni.cz/informatics/reports/`
`ftp  ftp.fi.muni.cz (cd pub/reports)`

Copies may be also obtained by contacting:

Faculty of Informatics
Masaryk University
Botanická 68a
602 00 Brno
Czech Republic