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'.

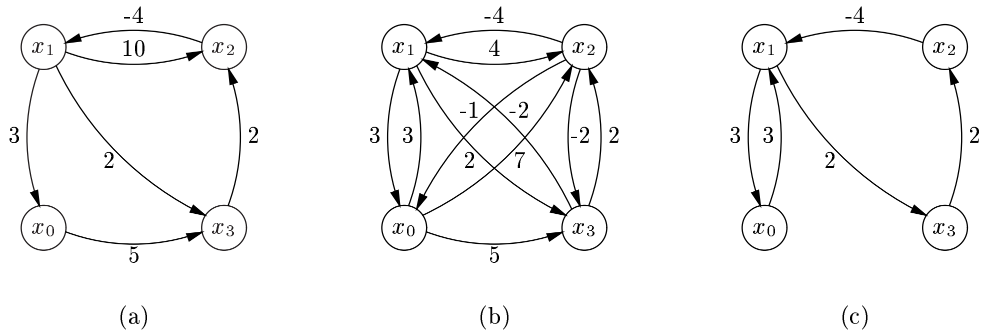

Figure2. Graph for E (a), its shortest–path closure (b), and shortest–path reduction (c).

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

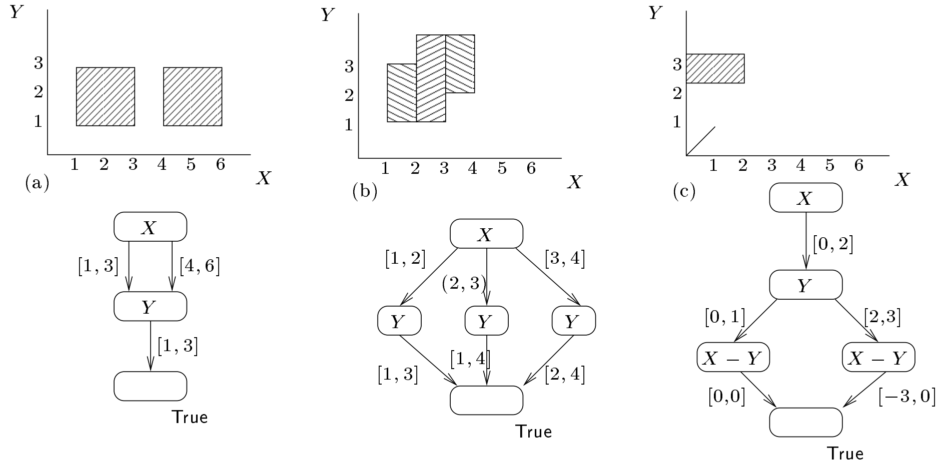

Figure3. Three example CDD's. Intervals not shown lead implicitly to False.

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*

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

– $get(P, W) = (P, W \setminus \{(l, Z)\}, (l, Z))$ *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 = {(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
```

→

```
Q = PW = {(l_0, Z_0 ∧ I(l_0))}
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

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

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.

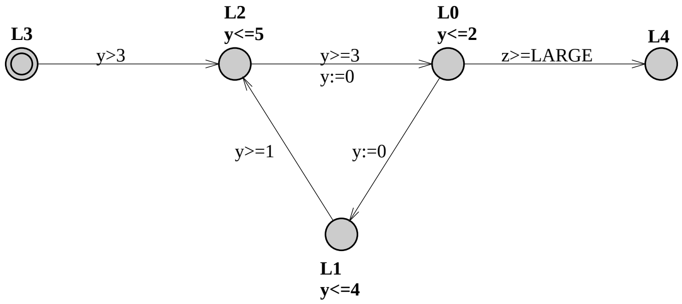

Figure6. Timed automaton P.

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

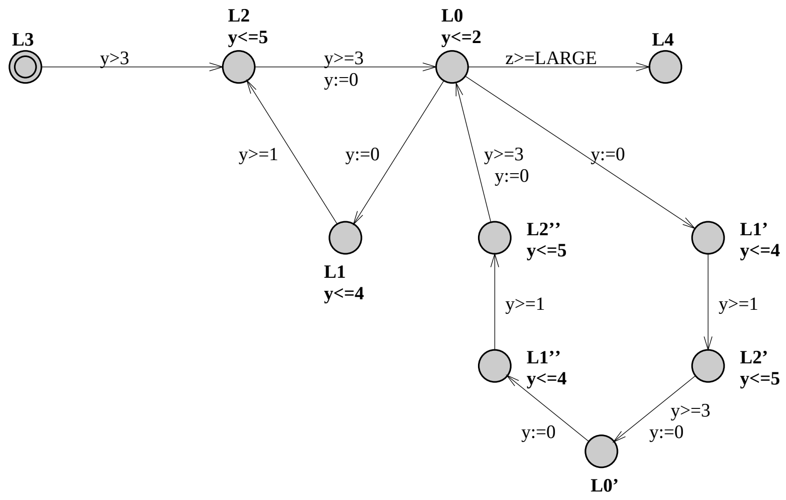

Figure7. The accelerated version of P.

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 4*th *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 16*th *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 12*th *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 18*th *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.