To Store or Not To Store

Gerd Behrmann[1], Kim G. Larsen[1], and Radek Pelánek[⋆2]

[1] Aalborg University, Frederik Bajers Vej 7E, 9220 Aalborg, Denmark
[2] Masaryk University Brno, Czech Republic

Abstract. To limit the explosion problem encountered during reachability analysis we suggest a variety of techniques for reducing the number of states to be stored during exploration, while maintaining the guarantee of termination and keeping the number of revisits small. The techniques include static analysis methods for component automata in order to determine small sets of covering transitions. We carry out extensive experimental investigation of the techniques within the real-time verification tool UPPAAL. Our experimental results are extremely encouraging: a best combination is identified which for a variety of industrial case-studies reduces the space-consumption to less than 10% with only a moderate overhead in time-performance.
Keywords. Timed automata model checking, Static analysis

1 Introduction

Reachability analysis has proved one of the most successful methods for automated analysis of concurrent systems. Several verification problems, e.g. trace-inclusion, invariant checking and model checking of temporal logic formula using test automata may be solved using reachability analysis. However the major problem in applying reachability analysis is the potential combinatorial explosion in the size of state spaces and the resulting excessive memory requirements. During the last decade numerous symbolic and reduction techniques have been put forward [BCM+90,GW91,Val90,BLP+99,LNAB+98,LL95] in order to avoid this explosion problem, playing a crucial role in the successful development of a number of verification tools for finite-state and timed systems (*e.g.* SMV, SPIN, visualSTATE, UPPAAL, KRONOS).

The explosion in memory consumption is closely linked to the need for storing states during exploration, primarily with the aim of guaranteeing termination but also to avoid repeated exploration (revisits) of states. However, one may maintain the guarantee of termination while keeping the number revisits small without necessarily storing *all* states. As an example consider the state space of the network in Fig. 1. Here, termination is guaranteed with storing of only two states (e.g. $AY0$ and $AY1$). The main question addressed in this paper is how to efficiently decide whether "to store" or "not to store" states during exploration.

In answering this question, we see that it suffices to store enough states so that all cycles in the global state space are covered. However, as finding

⋆ Supported by GA ČR grant no. 201/03/0509

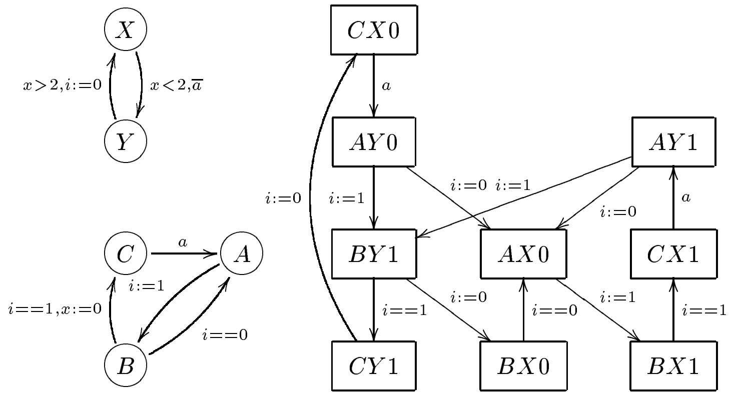

Fig. 1. A network of two timed automata and its discrete state space.

the smallest such set is NP-complete, we instead propose a number of efficient storing strategies yielding small covering sets. The techniques include storage only of every k-th state along a path, and storage only of states with multiple successors. Also, static analysis methods of cycles of the individual automata in a network are given in order to determine a (small) set of covering transitions based partially on a heuristic analysis of the quality of such sets using a (cheap) random walk analysis.

Within the real-time verification tool UPPAAL, we conduct a thorough experimental investigation of the various storing techniques identifying their best combinations. Our experimental results are extremely encouraging: for the majority of a range of industrial case-studies and examples studied in the literature it is possible to reduce the number of states stored to less than 10% compared with that of the currently distributed version of UPPAAL and with only a small overhead in time-performance.

The outline of the paper is as follows: section 2 introduces the timed automata network model used in UPPAAL together with the basic reachability algorithm. Section 3 provides a number of revised reachability algorithms based on different storing strategies. Section 4 revises the notion of covering set and provides a number of static analysis algorithms for identifying small such sets. Section 5 provides an extensive experimental analysis of the reduction techniques suggested, and, finally, section 6 contains the conclusion.

2 Preliminaries

Let X be a set of clocks, V a set of bounded integer variables and Σ a set of actions and co-actions s.t. $a = \overline{\overline{a}}$ and $a \in \Sigma \Leftrightarrow \overline{a} \in \Sigma$. An integer expression over V is an expression on the form v, c, $e_1 + e_2$ or $e_1 - e_2$, where $v \in V$, c is an integer constant, and e_1 and e_2 are integer expressions. Let $G(X, V)$ be the set of all guards, s.t. $x \bowtie c$, $e_1 \bowtie e_2$, and $g_1 \wedge g_2$ are guards, where x is a clock, c is an integer over V, e_1 and e_2 are integer expressions, g_1 and g_2 are guards, and $\bowtie \in \{<, \leq, =, \geq, >\}$. Let $U(X, V)$ be the set of updates, s.t $x := 0$, $v := e$, and $u_1;\ u_2$ are updates, where x is a clock, v is a bounded integer variable, e is an integer expression, and u_1 and u_2 are updates.

Definition 1 (Timed Automata). *A timed automaton $\mathcal{A}$ over X, V and Σ is a tuple $(L, l^0, E, \mathsf{guard}, \mathsf{sync}, \mathsf{up}, I)$ where L is a finite set of locations; $l^0 \in L$ is the initial location; $E \subseteq L \times L$ is the set of edges;* $\mathsf{guard} : E \to G(X, V)$, $\mathsf{sync} : E \to \Sigma \cup \{\tau\}$, $\mathsf{up} : E \to U(X, V)$ *assign guards, actions, and updates to edges; and $I : L \to G(X, V)$ assigns invariants to locations. A network of timed automata is a tuple of timed automata $A_1 \parallel \cdots \parallel A_n$ over X, V, and Σ.*

Notice that the terms *location* and *edge* refer to the syntactic elements of a timed automaton. A *concrete state* of a timed automaton is a semantic element and is defined as a triple (l, u, γ) of the current location l, and two functions u and γ assigning values to clocks and variables, respectively. More formally, let Γ be a function from V to intervals in $\mathbb{Z}$ s.t. $\forall v \in V : 0 \in \Gamma(v)$, thus defining the range of the variables. A clock valuation, u, is a function from X to the non-negative reals, $\mathbb{R}_{\geq 0}$. A variable valuation, γ, is a function from V to $\mathbb{Z}$ such that $\gamma(v) \in \Gamma(v)$. Let $\mathbb{R}^X$ be the set of all clock valuations and Z^V the set of all integer valuations. We write $(u, \gamma) \models g$ when the guard g is satisfied in the context of u and γ.

The semantics of (a Network of) Timed Automata is often stated in terms of a labelled transition system $\to$ with $(L_1 \times \cdots \times L_n) \times \mathbb{R}^X \times \mathbb{Z}^V$ being the set of states. There are two types of transitions: Whenever allowed by the invariant of the current locations, time can pass. The resulting *delay transitions* do not change the locations nor the variables, but all clocks are incremented by the exact same delay. Alternatively, if the guard of an edge (or two edges if they synchronise via complementing actions) is satisfied, we might trigger an *edge transition*. Edge transitions do not take time, *i.e.* the clock valuation is not changed except to reflect updates directly specified on the edge involved. The same is true for the variable valuation.

Given a state property φ, the reachability problem for timed automata is that of deciding whether there is a path from the initial state to a state satisfying φ. A standard forward breadth-first or depth-first search based directly on the semantics of a timed automaton is unlikely to terminate, since the state-space is uncountably infinite. It is well known that several exact and finite abstractions based on *zones* exist, see [AD94,BBFL03]. Zones are sets of clock valuations definable by conjunctions of constraints of the forms $x \bowtie c$ and $x - y \bowtie c$, where x and y are clocks and c is an integer. Using zones, one can define the *symbolic semantics* of a network of timed automata as a labelled transition relation $\Rightarrow$ over *symbolic states* on the form (l, Z, γ), where l is a location, Z is a zone, and γ is a variable valuation, s.t. $(l, Z, \gamma) \stackrel{t}{\Rightarrow} (l', Z', \gamma')$ iff $\forall \sigma' \in Z' : \exists \sigma \in Z : (l, \sigma, \gamma) \xrightarrow{t}\xrightarrow{\delta} (l', \sigma', \gamma')$ (t being a set of edges and δ being a non-negative real-valued delay). We will skip the formal definition and assume the existence

```
proc REACHABILITY
  W = {init_state}; P = ∅;
  while W ≠ ∅ do
        get s from W;
        if s ⊨ φ then return true; fi
        P = P ∪ {s};
        foreach s', t : s ⇒(t) s' do
          if s' ∉ P ∪ W then W = W ∪ {s'}; fi od
  od
  return false;
end
```

Fig. 2. Decision procedure for the reachability problem.

of such a symbolic semantics. In the following, we will refer to symbolic states simply as states.

Given the symbolic semantics, defining a decision procedure for the reachability problem is easy and is shown in Fig. 2. Here, P is the set of already visited states (the passed list), and W is the set of states which are to be explored (the waiting list). The passed list is usually implemented as a hash table, and the waiting list is often a queue or a stack.

3 Storing Strategies

By storing states in a passed list, we avoid that states are explored more than once, thus ensuring termination and efficiency. For termination, not all states need to be stored. Theoretically, only enough states such that all cycles in the symbolic transition system are covered must be stored – this is called a *feedback vertex set*. However, finding the smallest such set is NP-complete[Kar72]. Our goal in this paper is to find efficient strategies to obtain small feedback vertex sets for the symbolic transition relation (thus ensuring termination), while keeping the number of revisited states as low as possible.

We start by presenting a revised version of the reachability algorithm, see Fig. 3, which adds an extra integer field, *flag*, to each state in the waiting list. The interpretation of this flag depends on the two functions $\mathit{to\_store}(s, \mathit{flag})$ and $\mathit{next\_flag}(s, t, \mathit{flag})$, which compute whether to store s and what the flag of a successor of s should be, respectively. The reductions presented in the following are all instances of this generic scheme and differ only in the actual definition of *to_store* and *next_flag*.

Distance In many real life examples, cycles are rather long and it is sufficient to store every k-th state along any path, *i.e.*

$$\mathit{to\_store}(s, \mathit{flag}) \equiv \mathit{flag} \bmod k = 0$$
$$\mathit{next\_flag}(s, t, \mathit{flag}) \equiv \mathit{flag} + 1$$

```
proc REACHABILITY-WITH-REDUCTION
  W = {(init_state, 0)};  P = ∅;
  while W ≠ ∅ do
        get (s, flag) from W;
        if s ⊨ φ then  return true; fi
        if to_store(s, flag) then P = P ∪ {s};  fi
        foreach s', t : s =t=> s' do
          if s' ∉ P ∪ W then W = W ∪ {(s', next_flag(s, t, flag))};  fi od
  od
  return false;
end
```

Fig. 3. Revised decision procedure for the reachability problem, with hooks for adding various reduction techniques.

For the example in Fig. 1 this strategy stores (provided that we use BF-Search which starts at CX0 with flag 0) for $k = 2$: {BY1, AX0, AY0, AY1}, and for $k = 3$: {CY1, BX0, BX1}.

Successors Observations of state spaces of industrial protocols revealed that there are usually chains of states with only one successor. It is practically useless to store several states from such a chain. Thus only states which have more than one successor need to be stored. To avoid that cycles of states with only one successor cause problems, we use the flag as a counter to store at least one state of such cycles, *i.e.*

$$to\_store(s, flag) \equiv |succ(s)| > 1 \vee flag = k$$

$$next\_flag(s, t, flag) \equiv \begin{cases} 0 & \text{if } to\_store(s, flag) \\ flag + 1 & \text{otherwise} \end{cases}$$

In practice these cycles are infrequent and it is safe to use a large value for k (in none of out numerous experiments we considered such cycles occurred). For the example in Fig. 1 this strategy stores: {AY0, BY1, AY1}.

Covering Set In [LLPY97] a strategy based on static analysis of the cycles of the individual automata in the system was proposed. Here we provide a generalisation of the idea, inspired by [KLM+98]. Suppose that we have a set of edges, *Cover*, with the property that each cycle in the global state space contains at least one edge from *Cover*. Then it is sufficient to store states that are targets of transitions derived from edges in *Cover*, *i.e.*

$$to\_store(s, flag) \equiv flag = 1$$

$$next\_flag(s, t, flag) \equiv \begin{cases} 1 & \text{if } t \in Cover \\ 0 & \text{otherwise} \end{cases}$$

We will consider the problem of finding *Cover* in the next section. Referring to the example of Fig. 1, there are three cycles: (CX0, AY0, BY1, CY1), (BY1, BX0, AX0, BX1, CX1, AY1), and (AX0,BX1, CX1, AY1). Possible candidates for *Cover* are { (Y,X) } and { (C,A) }, resulting in the states in {AX0, CX0, BX0} and {AY0, AY1} being stored, respectively.

Random The simplest way to decide which states to store is to use randomness. In this case the flag is actually not needed since we simply store states with some probability, p. A deterministic version of this strategy is to use a global counter, by counting the number of newly visited states and store each k-th state ($k \approx \frac{1}{p}$).

Combinations There are many possibilities for combining the previous strategies. One particular combination of *covering set*, *successors* and *distance* reductions is the following:

$$to\_store(s, flag) \equiv flag \geq k \wedge |succ(s)| > 1 \vee flag = k^2$$

$$next\_flag(s, t, flag) \equiv \begin{cases} 0 & \text{if } to\_store(s, flag) \\ flag + 1 & \text{if } \neg to\_store(s, flag) \wedge t \in Cover \\ flag & \text{otherwise} \end{cases}$$

We increase *flag* when the state is a target of covering edge. And we store states only if the flag is greater than k and the number of successors is greater than one or the flag is k^2 (the latter catches cycles where each state only has one successor). There are of course many other possible combinations. We have tried other combinations of covering set, successors, distance, and random strategies with non-uniform probabilities. However, the behaviour of these combinations was usually similar to the one presented above.

Lemma 1. *Procedure* REACHABILITY-WITH-REDUCTION *terminates with any of these storing strategies (resp. with probability one for random strategies).*

4 Covering Set

In this section we discuss the problem of finding a *covering set*, being a set of edges with the property that each cycle in the symbolic state space uses at least one edge from this set. The algorithms used are due to [KLM+98], where they were used in the context of partial order reduction. We contribute additional methods for identifying relations among cycles which lead to smaller covering sets and a heuristic analysis of the quality of a covering set based on random walk analysis.

Referring to the example in Fig. 1, possible candidates for *Cover* are {(Y,X)} and {(C,A)}. The challenge we face is to compute such a covering set for a network based solely on a static analysis of the control structure of each automaton, *i.e.*, without unfolding the symbolic transition relation. Elementary cycles in the control structure, henceforth called *local cycles*, are important elements of such an analysis, since cycles in the symbolic state space, henceforth called *global cycles*, are formed by unfolding local cycles. The straightforward way of finding a covering set is to compute all local cycles and pick an edge from each. In the example, there are three local cycles: $\{X, Y\}, \{A, B, C\}$, and $\{A, B\}$, leading to a covering set of $\{(Y, X), (A, B)\}$, for instance. But if we look closer, it becomes clear that the cycle $\{X, Y\}$ cannot be realised without the synchronisation provided by $\{A, B, C\}$, and that $\{A, B\}$ cannot be realised without the reset provided by $\{X, Y\}$. We say that $\{A, B\}$ is *covered by* $\{X, Y\}$. We will now formalise this.

For the rest of this section we assume the existence of a network of n timed automata, $A_i = (L_i, l_0, E_i, \mathsf{guard}_i, \mathsf{sync}_i, \mathsf{up}_i, I_i)$. A *local cycle* of A_i is a sequence $(e_1, \ldots, e_m)$ of edges in E_i iff there is a set of locations $l_1, \ldots, l_m$ in L_i s.t. $l_1 \rightarrow_i \cdots \rightarrow_i l_m \rightarrow_i l_1$, where $l \rightarrow_i l' \Leftrightarrow (l, l') \in E_i$. A *global cycle* is a sequence $(t_1, \ldots, t_m)$, where each t_i is a set of edges, iff there is a sequence of symbolic states, $s_1, \ldots, s_m$ s.t. $s_1 \xRightarrow{t_1} \cdots s_m \xRightarrow{t_m} s_1$, where $\Longrightarrow$ is the labelled transition relation of the network. A *projection* $P(o)$ of a global cycle $o = (t_1, \ldots, t_m)$ is the set of local cycles corresponding to subsequences of o, *i.e.*, $c \in P(o)$ iff $c = (e_1, \ldots, e_k)$ is a local cycle and $\exists 1 \leq i_1 < \cdots < i_k \leq m, \forall 1 \leq j \leq k : e_j \in t_{i_j}$. By an abuse of notation we sometimes treat sequences as sets.

A set of transitions T *covers* a set of cycles C iff for each cycle $c \in C$: $T \cap c \neq \emptyset$. A set of transitions T is a *covering set* of a network N iff T covers all global cycles of N. The straightforward way to find some covering set is to find all local cycles and choose one transition from each cycle. However, it is often the case that a cycle cannot occur without one of the other cycles occurring as well. Formally, a set of local cycles D covers a local cycle c iff for each global cycle o holds $c \in P(o) \Rightarrow P(o) \cap D \neq \emptyset$. A set of local cycles C is covered by D iff D covers each cycle in C. If some set of cycles D covers all local cycles, then it is sufficient to choose one transition from each cycle of D. In our studies of existing models, we have identified four common situations which can be used to identify when a cycle is covered by a set of other cycles:

- Let c be local cycle that at some point updates the value of a variable v to some value k' and later requires it to have another value k without updating v in the meantime. In order to be realisable, such cycles must be combined with a cycle which updates v to k. Let *need_change_to*(v, k) be the set of cycles which need v to be changed to k in order to be realisable. Let *changes_to*(v, k) be the set of cycles which change v to k.
- A variation of this idea is based on cycles which either always increase or decrease a variable v (similar to a for-loop). These must be combined with cycles updating v in a non-increasing or non-decreasing manner, respectively. Let *increasing*(v) be the set of cycles which always increase v (and similarly for *decreasing*) and let *changes*(v) be the set of cycles which update v.
- Cycles synchronising via an action must obviously be paired. Let *sync*(a) be the set of cycles which synchronise on a.
- Cycles which at some point require a clock x to be bigger than k and later to be smaller than k without resetting x in the meantime must be combined with cycles resetting x (the opposite does not hold since clocks can be incremented by delaying). Let *needs_reset*(x) be the set of cycles which need x to be reset and let *resets*(x) be the set of cycles which reset x.

Lemma 2. *Let $A_1 \parallel \cdots \parallel A_n$, $A_i = (L_i, l_i^0, E_i, \mathsf{guard}_i, \mathsf{sync}_i, \mathsf{up}_i, I_i)$ be a network of timed automata. Let C_i be the set of local cycles of A_i, and $C = \cup_i C_i$. For some i, let $c \in C_i$. Then D covers c if at least one of the following holds:*

$$\exists v, k : c \in \mathit{needs\_change\_to}(v,k) \wedge \mathit{changes\_to}(v,k) \subseteq D$$
$$\exists v : c \in \mathit{increasing}(v) \wedge \mathit{changes}(v) \setminus \mathit{increasing}(v) \subseteq D$$
$$\exists v : c \in \mathit{decreasing}(v) \wedge \mathit{changes}(v) \setminus \mathit{decreasing}(v) \subseteq D$$
$$\exists a : c \in \mathit{sync}(a) \wedge \mathit{sync}(\overline{a}) \setminus C_i \subseteq D$$
$$\exists x : c \in \mathit{need\_reset}(x) \wedge \mathit{resets}(x) \subseteq D$$

Returning to our running example, we now see that $\{A, B\}$ is covered by $\{X, Y\}$ because the former is in $\mathit{need\_change\_to}(i, 0)$ and the latter is the only cycle in $\mathit{changes\_to}(i, 0)$. Similarly, $\{X, Y\}$ is covered by $\{A, B, C\}$ because the former is in $\mathit{sync}(a)$ and the latter is the only cycle in $\mathit{sync}(\overline{a})$.

All functions used in Lemma 2 may be obtained by static analysis, perhaps with the need of over-approximation *e.g.* in case an effect of a transition on a variable is found too complicated, the static analysis may simply suppose that the transition can change the variable to any value of its domain. The usefulness of the different heuristics depends on the type of models we want to analyse. For communication protocols using *need_change_to* seems sensible, whereas *increasing* and *decreasing* are much more useful in models containing **<u>for</u>** like constructs.

The choice of covering set determines the number of stored states. However, the number of stored states is not proportional to the size of the covering set, but rather to the frequency of the edges of the covering set in the symbolic state space (because we store targets of these edges). For example, in Fig. 1 the frequency of edge (Y, X) is $4/12$, whereas the frequency of edge (C, A) is $2/12$. The number of states stored for the covering set $\{(Y, X)\}$ is indeed larger than for $\{(C, A)\}$. Unfortunately, it is very difficult to estimate the frequency of an edge from a static analysis. Therefore we propose to perform a *Random Walk Analysis*: Before the construction of the covering set we perform several random walks through the system and count the occurrences of edges. In this way we identify *random walk coefficients*. These are fairly good estimates of the frequencies. Thus we would like to find covering set with the smallest sum of random walk coefficients. However, since even finding the smallest covering set is as hard as reachability, we have to use a heuristic approach.

Figure 4 presents two possible algorithms for construction of a covering set. Different heuristics can be used for *choose_cycle* and *choose_transition* functions. It is advantageous to take these random walk coefficients into account in these heuristics. We refer to section 5.1 for a discussion of these heuristics and their comparison. Moreover, the One-Phase-Construction algorithm can be used for verification of user-proposed sets. Note that this algorithm does not need to actually construct all local cycles – in the worst case this can be exponential in the size of the automaton, although in practice the worst case is seldomly reached (for our industrial test cases it was maximally 500 cycles). If an automaton has

```
proc TWO-PHASE-CONSTRUCTION
  H = covered = ∅;
  while covered ≠ C do
        c = choose_cycle(C \ covered);
        H = H ∪ {c};
        covered = CLOSURE(covered ∪ {c});
  od
  T = ∅;
  while H ≠ ∅ do
        t = choose_transition(H);
        T = T ∪ {t};
        foreach c ∈ H, t ∈ c do
          H = H \ {c}; od
  od
  return T;
end
```

```
proc ONE-PHASE-CONSTRUCTION
  H = T = ∅;
  while H ≠ C do
        t = choose_transition(C \ H);
        T = T ∪ {t};
        foreach c ∈ C, t ∈ c do
          H = H ∪ {c}; od
        H = CLOSURE(H);
  od
  return T;
end
```

Fig. 4. Algorithms for computation of Covering set; CLOSURE(H) computes set of cycles already covered by H (using Lemma 2).

a very large number of local cycles, it might be advantageous to find a set of covering transitions for this automaton using, for instance, depth first search, and then use the ONE-PHASE-CONSTRUCTION algorithm for the rest with this set as a starting set for T.

5 Experiments

A prototype of the techniques presented in this paper has been implemented in the real-time model-checker UPPAAL. We have performed exhaustive experiments with different heuristics for covering set construction, storing strategies (including different parameters), and combinations of strategies. Experiments were done on 12 different examples, including industrial case-studies and examples previously studied in the literature. Due to space considerations we only summarise the results and give the conclusions we have drawn from the experiments. The results reported in this section are based on the following examples (*Aut.* is the number of automata in the network; *Edges* is the number of edges):

Name	Aut.	Edges	Description
Fischer	4	20	Mutual exclusion protocol for 4 processes
Train-Gate	6	42	Train crossing with 4 trains
CSMA	7	58	CSMA/CD protocol for 6 processes
BRP	6	46	Bounded Retransmition Protocol
Dacapo	6	206	Start-up phase of a TDMA protocol
Token Ring	8	70	FDDI token ring for 7 stations
BOCDP	9	130	Bang & Olufsen Collision Detection Protocol
BOPDP	9	142	Bang & Olufsen Power Down Protocol
Buscoupler	16	198	Data link layer of ABB field bus protocol

Table 1. Different heuristics for covering set construction; for each heuristic (01 - T4) and model we report number of Edges (E) in the covering set, the sum of the random walk coefficients (RWC), and peak memory consumption of the reachability algorithm with this covering set.

	BRP			Train-gate			BOPDP			BOCDP		
	E	RWC	Memory	E	RWC	Memory	E	RWC	Memory	E	RWC	Memory
O1	2	0.15	31.5%	**2**	0.13	27.4%	**3**	0.2	18.3%	**5**	0.13	17.1%
O2	9	0.21	39.9%	**2**	0.12	30.8%	8	0.16	17.9%	18	0.15	22.5%
O3	11	0.21	39.9%	4	0.29	30.3%	12	0.16	17.9%	20	**0.08**	**13.3%**
O4	8	0.2	37.1%	5	0.23	28.8%	6	**0.14**	17.4%	14	0.25	29.9%
O5	**1**	**0.08**	**16.5%**	3	0.24	36.3%	**3**	0.26	33.8%	7	0.21	24.1%
T1	7	0.14	23.3%	**2**	0.12	30.8%	5	0.16	**9.3%**	10	0.18	24.9%
T2	10	0.14	27.1%	3	**0.11**	**25.1%**	10	0.15	11.3%	28	0.17	25.8%
T3	**1**	**0.08**	**16.5%**	17	0.53	51.5%	57	0.41	37.5%	53	0.37	48.8%
T4	**1**	**0.08**	**16.5%**	12	0.37	70.2%	22	0.4	46.8%	25	0.36	45.9%

In the following, the quality of different methods for the covering set construction is evaluated and then the proposed storing strategies are compared. Here *memory consumption* refers to the peak size of $P \cup W$ during the computation, given as a percentage of all reachable states; *overhead* is the ratio between the number of visited states and the number of reachable states.

5.1 Covering Set Construction

At the moment, our prototype supports the heuristics based on *need_change_to* and *sync* for the identification of covering sets. Table 1 reports some representative results from using these heuristics. We have tried both one phase methods (O1-O5) and two phase methods (T1-T4) with different heuristics for the selection of a next cycle/transition, e.g. selection according to the number of newly covered cycles (O2, O3), the length of the cycle (T1, T2); random selection (O4); approach which chooses a suitable variable (channel) and then selects all cycles covering this variable (channel) (T3, T4). The main observations from these experiments are the following:

- The number of edges that are needed to cover all cycles is quite small. From 12 tested models, some of them consisting of more then 10 automata and 150 edges, half of them can be covered by one or two edges and only one of them needs more than 5 edges.
- The sum of the random walk coefficients is a better static measure of the 'quality' of a covering set than the size of the set.
- None of the nine methods is 'dominant'. The most 'stable' method seems to be one phase construction which selects next transition according to the ratio of newly covered cycles and random walk coefficient (O2).

Thus we conclude that the best approach is to have several different heuristics, construct several covering sets, and then choose the one with the smallest sum of random walk coefficients. This is the method that we have used in the following experiments.

5.2 Storing Strategies

The results from using different storing strategies are reported in Table 2. The 'entry point' strategy is a slightly improved[1] version of the strategy in [LLPY97], which is a special case of the covering set construction presented in this paper and which is already implemented in UPPAAL. The results for distance, random, and combination strategies depend on the choice of parameters. It is usually possible to achieve better results than those reported here by choosing parameters optimised for the given model. The parameters used in the table are those which give the best 'overall' results. We have made experiments with some other (combinations of these) strategies, but the results were usually similar to those presented.

For many of the examples, the combined strategy significantly reduces the amount of memory used (*e.g* for BRP, token ring, Train gate, Dacapo, BOCDP, BOPD and Buscoupler less than 20% of the state space needs to be represented at any time). In those cases the number of revisits is often relatively small (less than a factor of 2) and the runtime overhead is even smaller (the runtime overhead is bounded by the relative increase in the number of visited states). In fact, in some cases the reachability algorithm with reduction is even faster than the standard algorithm because a lot of time consuming operations are eliminated (since the passed list is smaller) and hence the exploration rate is higher.

Table 2. Comparison of strategies – the memory and overhead are reported.

	entry points	covering set	successors	random $p = 0.1$	distance $k = 10$	combination $k = 3$
Fischer	27.1%	42.1%	47.9%	53.7%	67.6%	56.9%
3,077	1.00	1.66	1.00	4.51	2.76	6.57
BRP	70.5%	16.5%	19.8%	18.3%	15.8%	7.6%
6,060	1.01	1.20	1.03	1.78	1.34	1.68
Token Ring	33.0%	10.3%	20.7%	17.2%	17.5%	16.8%
15,103	1.16	1.46	1.03	1.63	1.43	7.40
Train-gate	71.1%	27.4%	24.2%	31.8%	24.2%	19.8%
16,666	1.22	1.55	1.68	2.90	2.11	5.08
Dacapo	29.4%	24.3%	24.9%	12.2%	12.7%	7.0%
30,502	1.07	1.08	1.07	1.21	1.16	1.26
CSMA	94.0%	75.9%	81.2%	105.9%	114.9%	120.3%
47,857	1.06	2.62	1.40	7.66	2.83	6.82
BOCDP	25.2%	22.5%	6.5%	10.2%	9.3%	4.5%
203,557	1.00	1.01	1.08	1.02	1.01	1.09
BOPDP	14.7%	13.2%	42.1%	15.2%	11%	4.3%
1,013,072	2.40	1.33	1.02	1.52	1.14	1.74
Buscoupler	53.2%	13.6%	40.5%	31.7%	24.6%	14.3%
3,595,108	1.29	2.48	1.18	3.17	2.13	8.73

[1] The improved version exploits information about communication channels to a certain extent.

The surprising fact is the small number of revisits. It is difficult to provide any theoretical explanation of this phenomenon (the worst-case behaviour is exponential). We just point out that the order in which vertices are visited becomes very important with reductions. Let us suppose that there are two paths p_1 and p_2 from state s_1 to state s_2 and that state s_2 is not going to be stored. Thus when the state is reached by the second path, it will be revisited. But if the paths have the same length (and this is very often the case) and we use breadth-first search order (*i.e.* W is implemented as queue), then s_2 will still be in W when it is visited via the second path and thus it will not be revisited (at least not for now). With depth-first search order, it would be revisited anyway. We have verified this hypotheses on our examples. The overhead was significantly larger for depth-first search order than for breadth-first.

In two cases (Fischer's and CSMA), the combined strategy turns out to be less efficient than the existing entry point strategy used in UPPAAL. It turns out when using reductions not only the number of revisits increases, but also the peak size of the waiting list. In some cases, the size of the waiting list dominates the memory consumption. For example, the combination strategy in the CSMA example stores only 16% of the states in the passed list, but due to the large waiting list the peak memory consumption is much larger. We have observed this behaviour mainly for parametrised systems which consists of several copies of one process – these systems have a high degree of nondeterminism and interleaving and thus the waiting list grows faster. This problem can partially be solved by implementing the waiting list as a priority queue and giving priority to states which are not going to be stored (intuitively, we want to get rid of them as soon as possible). Since the order is not breadth-first the overhead increases considerably. We have performed experiments confirming this reasoning.

There is clearly a trade-off between space and time and also between passed list size and waiting list size. The combined strategy usually stores the least amount of states in the passed list (but due to the waiting list the actual memory consumption can be larger than for other strategies). The successor strategy is not the most memory efficient, but causes nearly no overhead in terms of revisited states.

6 Conclusion

The contributions of this paper include a number of complementary storing strategies as well as static techniques for determining small sets of covering transitions with the aid of a random walk analysis. Extensive experiments have been carried out with different heuristics for covering set construction, storing strategies and combinations of strategies. The experiments are very encouraging: on a variety of industrial case-studies the space-saving is more than 90% with only a moderate increase in runtime. Though the experiments have been conducted within the real time tool UPPAAL the strategies proposed are equally applicable to other model checkers (including finite-state ones). However, the strategies are particular useful for timed systems as partial order reduction has yet to be successfully developed in this context (*e.g.* see [BJLY98]).

References

[AD94] R. Alur and D. Dill. Automata for Modelling Real-Time Systems. *Theoretical Computer Science*, 126(2):183–236, April 1994.

[BBFL03] Gerd Behrmann, Patricia Bouyer, Emmanuel Fleury, and Kim G. Larsen. Static guard analysis in timed automata verification. In *Proc. of TACAS'03*, LNCS. Springer–Verlag, 2003.

[BCM+90] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic Model Checking: 10^{20} states and beyond. In *Proc. of IEEE Symp. on Logic in Computer Science*, 1990.

[BJLY98] Johan Bengtsson, Bengt Jonsson, Johan Lilius, and Wang Yi. Partial Order Reductions for Timed Systems. In *Proc. of CONCUR '98: Concurrency Theory*, 1998.

[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 CAV'99*, LNCS. Springer–Verlag, 1999.

[GW91] P. Godefroid and P. Wolper. A Partial Approach to Model Checking. In *Proc. of IEEE Symp. on Logic in Computer Science*, pages 406–415, 1991.

[Kar72] R. M. Karp. Redcibility among combinatorial problems. In R. E. Miller and J. W. Thatcher, editors, *Complexity of Computer Computations*, pages 85–103, New York, 1972. Plenum Press.

[KLM+98] R. Kurshan, V. Levin, M. Minea, D. Peled, and H. Yenigün. Statical partial order reduction. In Bernhard Steffen, editor, *Proc. of TACAS'98*, volume 1384 of *LNCS*, pages 345–357, Lisbon, Portugal, Mar.–Apr. 1998. Springer–Verlag.

[LL95] Francois Laroussinie and Kim G. Larsen. Compositional Model Checking of Real Time Systems. In *Proc. of CONCUR '95*, LNCS. Springer–Verlag, 1995.

[LLPY97] Fredrik Larsson, Kim G. Larsen, 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.

[LNAB+98] Jørn Lind-Nielsen, Henrik Reif Andersen, Gerd Behrmann, Henrik Hulgaard, Kåre J. Kristoffersen, and Kim G. Larsen. Verification of Large State/Event Systems Using Compositionality and Dependency Analysis. In Bernard Steffen, editor, *Proc. of TACAS'98*, number 1384 in LNCS, pages 201–216. Springer–Verlag, 1998.

[Val90] A. Valmari. A Stubborn Attack on State Explosion. *Theoretical Computer Science*, 3, 1990.