An Implementation of Three Algorithms for Timing Verification Based on Automata Emptiness

R. Alur * C. Courcoubetis † D. Dill ‡ N. Halbwachs § H. Wong-Toi ‡

1 Introduction

Designing correct algorithms for systems involving interaction among state machines, such as communication protocols and concurrent algorithms, is known to be a difficult task. Design errors are often overlooked because of the difficulty in reasoning about all possible executions of a concurrent system. Traditional debugging techniques, such as simulation and prototype testing, may fail because the potential state-space of the system is too large for all bugs to be revealed. Consequently, there has been increasing interest in the use of formal methods for the verification of concurrent systems.

Most work in this field has concentrated on the sequencing and coordination of system events, abstracting away the times at which events occur. However many systems are time-critical: their behaviors may be time-dependent, and their specifications may require events to occur within specific time bounds. For example, most communication protocols require a bounded response time, and an automatic flight controller must guarantee fast reaction times. Recently, much attention has been given to the problem of extending existing verification techniques to such real-time systems [1, 2, 3, 4, 5, 6].

We consider real-time systems described by timed automata [2, 3]. These automata have a finite-state control and a finite set of fictitious clocks which measure the passage of time. A state of the system consists of the control state of the automaton and the values of all the clocks. It is clear that a timed automaton has infinitely many states. Analysis of such systems involves constructing a finite graph called a *regions graph* which represents a finite quotient of the infinite state graph. States in the same region are in some sense equivalent, and the regions graph can be used for checking emptiness of an automaton [2], deciding bisimulation equivalence between automata [7], real-time model-checking [3, 8], deciding reachability questions [9], and controller synthesis [10]. The practical difficulty lies in the fact that the regions graph is exponentially large in the timing information of the original timed automaton. This state-space explosion due to the time bounds is in addition to the blow-up due to concurrent components. There is therefore great need for the development and testing of heuristics before formal verification becomes practical.

This papers describes modifications to and the implementation of algorithms previously described in [1, 11]. We first describe three generic (untimed) algorithms for constructing graphs of the reachable states of a system, and how these graphs can be used for verification. They all have as input an implicit description of a transition system. We then apply these algorithms to real-time systems. The first algorithm performs a straightforward reachability analysis on *sets* of states of the system, rather than on individual states. This corresponds to stepping symbolically through the system many states at a time. In the case of a real-time system this procedure constructs a graph where each node is the *union* of some regions of the regions graph. There is therefore no need for an *a priori* partitioning of the state space into individual regions; however, this approach potentially leads to exponentially worse complexity since its potential state space is the power set of regions [1]. The other two algorithms we consider are *minimization* algorithms [12, 13, 11]. These simultaneously perform reachability analysis and minimization from an implicit system description. These can lead to great savings when the minimized graph is much smaller than the explicit reachable graph.

Our paradigm for verification is to test for the emptiness of the set of all timed system executions that violate a requirements specification. One way to specify and verify non-terminating processes is to model them as languages of ω-sequences of events [14, 15, 16, 1, 17, 18]. Modular processes can be constructed via composition operations involving language intersection. Specifications are also given as languages: they contain all acceptable event sequences. Program correctness is then just language containment. If the process language P has the specification $Spec$, then to verify P is to establish that $P \subseteq Spec$, or equivalently that $P \cap \overline{Spec}$ is empty, where $\overline{Spec}$ represents the complement of the specification. Often both P and the complemented specification $\overline{Spec}$ can be expressed naturally by a *fair transition system* (or fts for short). In this case, program verification reduces to a test of emptiness of an fts since fts's are closed under intersection. This methodology is suitable for verifying real-time systems since timed automata are a form of fts.

We describe how these algorithms are implemented,

*AT&T Bell Laboratories, Murray Hill, New Jersey.

†Institute of Computer Science, FORTH & University of Crete, Heraklion, Greece. Partially supported by the Esprit Basic Research Action REACT

‡Stanford University, Stanford, California. Supported by the Department of the Navy, Office of the Chief of Naval Research under Grant N00014-91-J-1901.

§IMAG Institute, Grenoble, France.

and discuss strategies for analyzing only the "non-Zeno" runs of a timed automaton, *i.e.* runs where time progresses without bound. The performance of the implementations is compared using two examples.

2 Analysis of Fair Transition Systems

2.1 Fair Transition Systems

Suppose we have an implicit description of a labeled transition system $\mathcal{S} = (\Sigma, S, s_0, \rightarrow, \mathcal{F})$. Here Σ is a set of events and the set S is a (not necessarily finite) set of states. The state $s_0 \in S$ is the initial state, and $\rightarrow \subseteq S \times \Sigma \times S$ is the transition relation, also referred to as $\rightarrow_{\mathcal{S}}$. We denote the fact that a transition $t : (s, \sigma, s')$ is in $\rightarrow$ by the notation $s \xrightarrow{\sigma} s'$. For a state s and a subset $X \subseteq S$, we write $s \overset{\sigma}{\Rightarrow} X$ to mean that $s \xrightarrow{\sigma} s'$ for some $s' \in X$. We use $s \rightarrow s'$ to mean that there is some event σ such that $s \xrightarrow{\sigma} s'$. The notation $s \Rightarrow X$ for a set X is used similarly. The fairness constraint set $\mathcal{F} \in 2^{2^S}$ is a set of sets of states of S.

A *word* $\tilde{\sigma}$ over Σ is an infinite sequence of events σ_i. A *run* of a fts $\mathcal{S}$ over a word $\tilde{\sigma}$ is an infinite sequence of states $s_i \in S$ such that

$$s_0 \xrightarrow{\sigma_0} s_1 \xrightarrow{\sigma_1} s_2 \xrightarrow{\sigma_2} \cdots \xrightarrow{\sigma_{n-1}} s_n \xrightarrow{\sigma_n} \cdots$$

It is a *fair run* if for every set $F_j \in \mathcal{F}$, $s_i \in F_j$ for infinitely many i. The language accepted by the fts is the set of words that have fair runs.

2.2 Emptiness

Given a fts $\mathcal{S}$ the emptiness problem is to decide if the language of $\mathcal{S}$ is empty. When both S and Σ are finite, there are many well-known algorithms for constructing the reachable subgraph of $\mathcal{S}$. From such a subgraph, emptiness of $\mathcal{S}$ can be decided by looking for strongly connected components that satisfy the fairness constraints. However in many cases, and indeed for timed systems, this finiteness assumption is not true (the notion of state includes a real-valued time component).

Thus rather than constructing a graph with individual states as nodes, we build from $\mathcal{S}$ instead a *set-graph* $G = \langle \Sigma, \mathcal{N}, N_0, \rightarrow \rangle$ over the alphabet Σ. Each node of its *finite* set of nodes $\mathcal{N}$ is a set of states of $\mathcal{S}$, and $s_0 \in N_0$. A set-graph is *complete* if and only if every state $s \in S$ appears in some node $N \in \mathcal{N}$.

A set-graph G is *bisimulating* if and only if $\forall N_1, N_2 \in \mathcal{N}, \forall \sigma \in \Sigma$

$$\begin{array}{ll} & \exists s \in N_1 \; \exists s' \in N_2 : s \xrightarrow{\sigma} s' \\ \text{if and only if} & \forall s'' \in N_1 \; \exists s' \in N_2 : s'' \xrightarrow{\sigma} s' \\ \text{if and only if} & N_1 \xrightarrow{\sigma} N_2 \end{array}$$

A set-graph G is *surjective* if and only if $\forall N_1, N_2 \in \mathcal{N}, \forall \sigma \in \Sigma$

$$\begin{array}{ll} & \exists s' \in N_2 \; \exists s \in N_1 : s \xrightarrow{\sigma} s' \\ \text{if and only if} & \forall s'' \in N_2 \; \exists s \in N_1 : s \xrightarrow{\sigma} s'' \\ \text{if and only if} & N_1 \xrightarrow{\sigma} N_2 \end{array}$$

Proposition: Let G be a complete, bisimulating set-graph for $\mathcal{S}$. If there exists a run $\{s_i\}$ for $\{\sigma_i\}$ in $\mathcal{S}$, then there is a run $\{N_i\}$ for $\{\sigma_i\}$ in G such that $s_i \in N_i$. Conversely, if there exists a run $\{N_i\}$ for $\{\sigma_i\}$ in G, then there is a run $\{s_i\}$ for $\{\sigma_i\}$ in $\mathcal{S}$ such that $s_i \in N_i$. $\square$

A set of states N *respects* a partition ρ of S, if and only if for every class $X \in \rho$, $N \cap X$ is either empty or all of X. For a set $A \subseteq S$, we denote by $\overline{A}$ its complement with respect to S. Let $A, B \subseteq S$. We let $\otimes$ be the pairwise intersection operator on sets, *i.e.*

$$\{A_i\} \otimes \{B_j\} = \{A_i \cap B_j\}$$

The operator is generalized to multiple arguments. Consider the partition

$$\rho_0 = \bigotimes_{F_i \in \mathcal{F}} \{F_i, \overline{F_i}\}$$

A set-graph G for $\mathcal{S}$ is *fairness-respecting* if and only if every node N of G respects the partition ρ_0.

Theorem: Let G be a complete, bisimulating and fairness-respecting set-graph for $\mathcal{S}$. The fts $\mathcal{S}$ is non-empty if and only if G has a reachable strongly connected component which contains for every F_i a node N_i such that $N_i \cap F_i \neq \emptyset$. $\square$

We now outline three algorithms that build complete, fairness-respecting set-graphs. The first algorithm produces only graphs that are surjective, while the last two generate bisimulating graphs. While the above theorem does not hold in general for surjective set-graphs, we can show that in our special application the surjective setgraph built does maintain the property of emptiness.

3 Set-Reachability Algorithm

There are familiar algorithms for reachability which perform an exhaustive search over a discrete state-space. When examining one discrete state, all of its successors must also be examined in some order (commonly either depth-first or breadth-first order). A state need not be examined further if it has previously been visited.

This basic methodology can be applied to *sets* of states, rather than individual states. Given a partition of the state space, and an initial reachable set of states respecting that partition, we want to construct a graph where all nodes are sets of states, and there is an edge between two sets of states A and A' whenever every state in A' is directly accessible from some state in A, *i.e.* we want to build a surjective set-graph. Furthermore every node respects the partition.

```
procedure set_reachability {
   input A0;      /* an initial reachable set of
                        states respecting rho_0 */
   input S;       /* a transition system        */
   input rho_0;   /* a partition of the states  */
   output G;      /* a graph as described above */
   vertices(G) := {};
```

```
    edges(G) := {};
    stack := emptystack;
    push(A0, stack);
    while (not empty(stack)) do
       A := pop(stack);
       for (sigma in Events(S)) do
          As := successors(A,sigma);
          if (As <> {}) then
             A' := intersect(As, rho);
             while (A' <> {}) do
                B := select(A');
                A' := A' - B;
                if (not B in vertices(G)) then
                   vertices(G) := vertices(G) + B;
                   push(B, stack);
                endif
                edges(G) := edges(G) + <A,sigma,B>;
             endwhile
          endif
       endfor
    endwhile
}
```

Let $V(\hat{\sigma})$ be the set of states that are reachable from A_0 via the finite sequence of events $\hat{\sigma}$. The algorithm will terminate if and only if the cardinality of $\{V(\hat{\sigma})\}_{\hat{\sigma}\in\Sigma^*}$ is finite.

4 Minimization Algorithms

We now briefly describe two algorithms for *simultaneously* minimizing and generating the reachable subgraph of an *implicitly* defined transition system. First we introduce some common notation.

Let ρ be a partition of S. A class $X \in \rho$ is said to be *stable* with respect to another class $Y \in \rho$ if and only if $\forall \sigma \in \Sigma$

$$(\exists\, x \in X : x \overset{\sigma}{\Rightarrow} Y) \text{ implies } (\forall\, x \in X : x \overset{\sigma}{\Rightarrow} Y)$$

A class is stable with respect to ρ if it is stable with respect to every class in ρ, and the partition ρ is called a *bisimulation* if and only if every class of ρ is stable with respect to ρ. The *reduction* of $\mathcal{S}$ according to a partition ρ is the transition system $\mathcal{S}|\rho = (Acc(\rho), [s_0]_\rho, \longrightarrow_\rho)$, where

- $Acc(\rho)$ is the set of classes of ρ which contain at least one state accessible from s_0;
- $[s_0]_\rho$ denotes the class of ρ which contains s_0;
- $X \overset{\sigma}{\longrightarrow}_\rho Y$ iff $x \overset{\sigma}{\Rightarrow} Y$ for some $x \in X$.

4.1 Minimization Algorithm I

This minimization algorithm is due to Bouajjani et al [19, 12]. The algorithm starts from a transition system $\mathcal{S}$ and an initial partition ρ_0 and constructs a transition system $\mathcal{S}|\overline{\rho}$, where $\overline{\rho}$ is the coarsest bisimulation compatible with ρ_0 (every class of ρ_0 is a union of classes of $\overline{\rho}$). The algorithm will terminate whenever the bisimulation $\overline{\rho}$ has a finite number of classes. The generic algorithm is parameterized by the following functions which are specific to the transition system $\mathcal{S}$. For computational efficiency we use a variant of the *split* function found in [19, 12].

- *split* is a non-deterministic function which "splits" a class X of a partition ρ into "more stable" subclasses. Let γ be the set of classes of ρ for which X is not stable. If γ is not empty, then *split* returns a minimal set of subclasses of X that are stable with respect to some $Y \in \gamma$. If γ is empty, then *split* returns $\{X\}$.
- For a class X, the set $post_\rho(X)$ is the set of classes in ρ which are directly accessible from some state of X: $post_\rho(X) = \{Y \mid \exists\, x \in X, x \Rightarrow Y\}$.
- The set $pre_\rho(X)$ is the set of classes in ρ which contain at least one state from which X is directly accessible: $pre_\rho(X) = \{Y \mid \exists\, y \in Y, y \Rightarrow X\}$.

In the following algorithm, ρ is the current partition, α is the set of classes of ρ which have been found accessible from the class of the initial state, and σ is the set classes of ρ which have been found stable with respect to ρ.

```
procedure min_algo_I;
begin
   rho := rho_0;
   alpha := { [s0] };
   sigma :=  {};
   while (alpha <> sigma) do
      X := select(alpha - sigma);
      alpha' :=  split(X,rho);
      if (alpha' = { X }) then
         sigma := sigma + {X};
         alpha := alpha + Post(rho)(X);
      else
         alpha := alpha - {X};
         if (some Y in alpha' contains s0) then
            alpha := alpha + {Y};
         endif
         sigma := sigma - Pre(rho)(X);
         rho := (rho - {X}) + alpha';
      endif
   endwhile
endprocedure
```

4.2 Minimization Algorithm II

Lee and Yannakakis' minimization algorithm [13] is similar in spirit to the one above. Space does not permit a full account of their algorithm, so we instead describe only the fundamental differences between their algorithm and Bouajjani et al's, and refer the reader to [13] for details.

Lee and Yannakakis specify an explicit strategy for choosing which class of the partition to split next. Their selection strategy guarantees an upper bound on the running time that is quadratic in the size of the minimal graph, provided there is a finite minimal graph and a means of detecting termination. The idea is to search forward to find classes that need to be split, and to give every class a fair chance of being split. It also ensures that no effort is wasted unnecessarily splitting classes that are not reachable (this idea in fact led to the variant of the *split* function we use in Minimization Algorithm I).

Whenever a class is known to be reachable, a reachable point is chosen from within the class and used to *mark* the class. Consequently future splitting may be done "around" this reachable point, thereby ensuring that all splitting is done on reachable classes. Once a class is reached, it is placed in a marked graph. An edge from one marked class to another indicates that there is some state in the second class that is directly accessible from the marked state of the first.

5 Timed Automata

Timed automata [1, 2, 3] are a form of finite-state automata augmented with a finite set of real-valued clocks. Clocks may only be reset when transitions occur, and each transition has an associated enabling condition. Thus to express a bound on the delay between two transitions, we reset a clock on the first transition, and associate an enabling condition with the other transition.

For each transition, the enabling condition is given as a set of points in $\mathbb{R}^n$ ($\mathbb{R}$ denotes the set of non-negative reals, and n is the number of clocks in the automaton). The condition is enabled provided the n-vector of clock values lies in the enabling region. We require enabling conditions to be a form of convex polyhedron of $\mathbb{R}^n$, consisting of all the solutions of a system of linear inequalities where each inequality is of one of the following forms:

- $x \leq k$, $x < k$, $x \geq k$, $x > k$, where x is a clock and k is an integer constant, or
- $x - y \leq k$, $x - y < k$, where x and y are clocks and k is an integer constant.

We call such a polyhedron a (time) *zone*. Let $\mathcal{Z}(n)$ be the set of zones of $\mathbb{R}^n$. We consider also a set of *reset actions* $\mathcal{A}(n)$, which are functions from $\mathbb{R}^n$ to $\mathbb{R}^n$. For each $a \in \mathcal{A}$, there is a set of indexes $I_a \subseteq \{1 \ldots n\}$ such that $\forall \vec{x} \in \mathbb{R}^n, \forall i = 1, \ldots, n$, $a(\vec{x})_i$ is equal to 0 if $i \in I_a$, and $(\vec{x})_i$ otherwise.

Definition: A *timed automaton* G is a tuple $\langle \Sigma, Q, C, q_{init}, T, F \rangle$ where

1. Σ is a finite set of events,
2. Q is a finite set of locations,
3. $C = \{x_1, \ldots, x_n\}$ is a (finite) set of clocks,
4. $q_{init} \in Q$ is an initial location,
5. $T \subseteq Q \times \Sigma \times \mathcal{Z}(n) \times \mathcal{A}(n) \times Q$ is a transition relation, and
6. $F \in 2^{2^Q}$ is a set of fairness constraints.

Control of the automaton starts in location q_{init}. Initially all clocks have value 0. A transition (q, σ, z, a, q') in T, denoted by $q \xrightarrow{\sigma,z,a} q'$, means that control may pass from location $q \in Q$ in the automaton to location $q' \in Q$ via the event σ provided the n-vector of clock values lies in the enabling region z. At the same time the reset action a is applied to the current clock values. At any instant, the state of the system can be fully described by the current location and the values of all its clocks.

A timed automaton is an implicit description of an underlying fts. The states of this fts are pairs $\langle q, \vec{x} \rangle$, where $q \in Q$ and $\vec{x} \in \mathbb{R}^n$. Its initial state is $\langle q_{init}, \vec{0} \rangle$. The fairness constraints are $\mathcal{F} = F \times \mathbb{R}^n$. We must consider two forms of events in the evolution of the fts: either transitions from T occur, or time passes. All events occur instantaneously: the event τ represents the event that non-zero time has passed since the last event. Thus the fts has alphabet $\hat{\Sigma} = T \cup \{\tau\}$. Now we define a timed consecution relation on the states of a timed automaton.

Definition: For $e \in \hat{\Sigma}$, a state $\langle q', \vec{x'} \rangle$ is said to be an e-successor of another state $\langle q, \vec{x} \rangle$, written $\langle q, \vec{x} \rangle \xrightarrow{e} \langle q', \vec{x'} \rangle$, if and only if either

- $e = t : q \xrightarrow{\sigma,z,a} q' \in T$ such that $\vec{x} \in z$ and $\vec{x'}$ equals $a(\vec{x})$, or
- $e = \tau$, and $q' = q$ and for some δ, $\vec{x'} = \vec{x} + \vec{\delta}$ (where $\vec{\delta}$ denotes the n-vector $[\delta, \delta, \ldots] \in \mathbb{R}^n$).

A run of the automaton started in a state $\langle q, \vec{x} \rangle$ is obtained by iterating the relation $\longrightarrow$. Not all runs represent the unbounded passing of time; in fact there is no guarantee that any non-zero time passes at all. However if we are to model the execution of non-terminating runs of a timed system we wish to rule out such runs. A run r is *progressive* (or non-Zeno) if and only if $\Sigma_{i=0}\ \delta_i$ is unbounded, where

$$\delta_i = \begin{cases} 0 & e_i \in T \\ \delta & e_i = \tau \text{ and } (\vec{x})_{i+1} = (\vec{x})_i + \vec{\delta} \end{cases}$$

This property ensures that only a finite number of real events can occur in any finite interval of time.

6 Checking Emptiness of Timed Automata

We now describe how each of the three algorithms above can be adapted to check for emptiness of timed automata. The graphs we construct as set-graphs for a timed transition system have nodes of the form $\langle q, Z \rangle$, where $q \in Q$ and $Z \in \mathcal{Z}(n)$. We demonstrate how the operations required by each algorithm can be expressed as operations on time zones, and show how termination is guaranteed.

6.1 Zones and regions

6.1.1 Time operations

The following partial order will be considered on $\mathbb{R}^n$:

$$\vec{x} \preceq \vec{y} \text{ iff } \exists \delta \in \mathbb{R} \text{ such that } \vec{y} = \vec{x} + \vec{\delta}.$$

For a zone Z, the *set of time predecessors* $Z_{\swarrow}$, and the *set of time successors* $Z_{\nearrow}$ is given by

$$Z_{\swarrow} = \{ \vec{y} \mid \exists \vec{x} \in Z, \vec{y} \preceq \vec{x} \}$$

$$Z_{\nearrow} = \{\, \vec{y} \mid \exists \vec{x} \in Z, \vec{x} \preceq \vec{y} \,\}$$

If Z is a zone and Z' is a zone, then $Z \setminus Z'$ is some set of disjoint zones Z''_i, satisfying $Z - Z' = \bigcup_i Z''_i$. Notice that in general the set $Z - Z'$ is not a zone. If Z and Z' are two zones, then define

$$Z \sqcup Z' = \{Z \cap Z'\} \cup (Z \setminus Z') \cup (Z' \setminus Z).$$

In the following, we will consider *regions* of the form $\langle q, Z\rangle$, for some zone Z. By convention, $\langle q, Z\rangle \sqcup \langle q, Z'\rangle = \{\langle q, Z''\rangle \mid Z'' \in Z \sqcup Z'\}$.

Let $Z \Uparrow Z'$ denote the largest subset Z'' of Z such that it is possible to pass continuously from any point of Z'' through Z'' to a point of Z' by letting time elapse:

$$\forall \vec{x}_1 \in Z'', \exists \vec{x}_2 \in Z', \ \vec{x}_1 \preceq \vec{x}_2 \text{ and } \forall \vec{x}_1 \preceq \vec{x}_3 \preceq \vec{x}_2, \ \vec{x}_3 \in Z \cup Z'$$

It is easy to show that $Z \Uparrow Z'$ is a zone. A zone Z' is *directly time-accessible* from another zone Z if and only if $Z \Uparrow Z'$ is all of Z.

6.1.2 Transitions

Recall that the transitions of a timed automaton are considered the events of its fts. Thus to find the images and preimages of events in this fts, we need to be able to compute the images and inverse images of the transitions of a timed automaton.

The *image* of a class $\langle q, Z\rangle$ under the transition $t : q_1 \xrightarrow{\sigma,z,a} q_2$, denoted $t(\langle q, Z\rangle)$, is empty if $q \neq q_1$, or $Z \cap z = \emptyset$. Otherwise it is simply $\langle q_2, Z'\rangle$ where $Z' = a(Z \cap z)$. The *pre-image* of a class $\langle q, Z\rangle$ under the transition $t : q_1 \xrightarrow{\sigma,z,a} q_2$, denoted $t^{-1}(\langle q, Z\rangle)$, is empty if $q \neq q_2$, or $a(z) \cap Z = \emptyset$. Otherwise it is $\langle q_1, Z'\rangle$ where $Z' = a^{-1}(Z) \cap z$. To find Z', we compute

$$a^{-1}(Z) = \bigcup\{Z'' \mid a(Z'') \subseteq Z\}$$

We first find Z_1, the part of Z that is in the image of a. This zone is simply $Z \cap Z_a$ where Z_a is the zone $a(\mathbb{R})$. We can then express $a^{-1}(Z)$ as $a^{-1}(Z) = \Pi_a^{-1}(Z_1)$ where Π_a^{-1} is the inverse projection of all clocks in I_a.

A class $\langle q', Z'\rangle$ is *directly accessible* from another class $\langle q, Z\rangle$ if and only if

1. either $q = q'$ and Z' is directly time-accessible from Z, or
2. there exists a transition $q \xrightarrow{\sigma,z,a} q'$ such that $a(Z \cap z) \cap Z' \neq \emptyset$.

6.2 Set Reachability

We represent the nodes of the set-graph as regions, and show each step of the algorithm yields reachable sets that are also regions. This essentially repeats results in [1].

The set-reachability algorithm requires us to specify how to find the successors of a given class, and how to compute the pairwise intersection operator. The successor sets of a region $\langle q, Z\rangle$ are themselves regions: they are the time successors $\langle q, Z_{\nearrow}\rangle$, and the images of $\langle q, Z\rangle$ under some transition t, namely $t(\langle q, Z\rangle)$. Both of these operations were given in the previous subsection. Pairwise intersection is easily handled since the intersection of any two time zones is a time zone.

The initial partition taken here, and also for the other algorithms, is

$$\rho_o = \{q \times \mathbb{R}^n \mid q \in Q\}$$

Clearly this partition is fairness-respecting, and so the constructed set-graph is too.

6.3 Minimization Algorithm I

We briefly describe how to define the functions *split*, *pre*, and *post* in terms of operations on time zones. For a more detailed account see [11].

A class $\langle q, Z\rangle$ is *stable* with respect to a transition $t \in T$ and another class $\langle q', Z'\rangle$ if and only if

$$t^{-1}(\langle q', Z'\rangle) = \langle q'', Z''\rangle$$

with

$$q'' = q \text{ implies } Z'' \cap Z \text{ equals } \emptyset \text{ or } Z$$

It is *time-stable* with respect to $\langle q', Z'\rangle$ if and only if

$$q = q' \text{ implies } Z \Uparrow Z' \text{ equals } \emptyset \text{ or } Z$$

A class is stable with respect to a partition ρ if and only if it is stable with respect to every transition - class pair, and time-stable with respect to every class in ρ. For any class $\langle q, Z\rangle$, the nondeterministic function *split* is defined such that *split*$(\langle q, Z\rangle)$ is $\{\langle q, Z\rangle\}$ if $\langle q, Z\rangle$ is stable with respect to ρ. Otherwise it may return $\langle q, Z\rangle \sqcup \langle q, Z \Uparrow Z'\rangle$ if it is not time-stable with respect to $\langle q, Z'\rangle \in \rho$, or $\langle q, Z\rangle \sqcup t^{-1}(\langle q', Z'\rangle)$ if it is not stable with respect to t and $\langle q', Z'\rangle$ for some $t \in T, \langle q', Z'\rangle \in \rho$. Thus *split* always yields a set of classes that are either stable with respect to ρ or stable with respect to a greater number of transitions - class pairs.

$$pre_\rho(\langle q, Z\rangle) = \{\langle q, Z'\rangle \in \rho \mid Z' \Uparrow Z \neq \emptyset\} \cup \bigcup_{q' \xrightarrow{z,a} q} \{\langle q', Z'\rangle \in \rho \mid a(Z' \cap z) \cap Z \neq \emptyset\},$$

$$post_\rho(\langle q, Z\rangle) = \{\langle q, Z'\rangle \in \rho \mid Z \Uparrow Z' \neq \emptyset\} \cup \bigcup_{q \xrightarrow{z,a} q'} \{\langle q', Z'\rangle \in \rho \mid a(Z \cap z) \cap Z' \neq \emptyset\}$$

6.4 Minimization Algorithm II

This algorithm uses similar procedures to the first two algorithms: images, inverse images, and time successors and predecessors are computed as above. The only functions that differ are for marking blocks with single points and finding the successors of a single state. It turns out that not every time zone contains a time zone that represents a single point in $\mathbb{R}^n$: this is because time zones are defined using only integer constants. Consider for example, the time zone defined by $0 < x < 1$ for $n = 1$. Thus the algorithm is modified slightly to mark classes not with a single reachable state, but instead with a set of states. Thus marking can also be done using time zones. Consequently, the marked subset must be updated whenever a class is split, so it always lies within the class it is marking.

6.5 Termination

The termination of the algorithms relies on the fact that there are only finitely many time zones that need be considered in the analysis of a timed automaton. We merely state the result needed here and refer the reader to [3, 2, 20] for details.

Theorem: For any timed automaton G, there is a finite set-graph (called its *detailed regions graph*) whose nodes, all of the form $\langle q, Z\rangle$ for some time zone in $\mathcal{Z}$, are a stable partition of the state-space of G. □

Notice that the initial partition of the algorithms is coarser than the stable partition represented by the detailed regions (the nodes of detailed regions graph), and the minimization algorithms only ever split unstable classes. Thus at any point during the algorithm, every class of the generated set-graph is the union of detailed regions.

Theorem: The minimization algorithms described above terminate. □

The set-reachability algorithm as outlined above can generate regions that are proper subsets of an automaton's detailed regions. Thus the algorithm in fact does not terminate. We briefly describe how it is modified to guarantee termination. The idea is that we may add to each reachable set of states N in the set-graph any states s' for which there exists an $s \in N$ such that for every N' in the regions graph $s \overset{\sigma}{\Rightarrow} N'$ if and only if $s' \overset{\sigma}{\Rightarrow} N'$. Thus as each node N is generated it is replaced by the set of nodes in the regions graph with which it has non-empty intersection. The property of emptiness is preserved and only finitely many nodes are generated.

Theorem: The set-reachability algorithm terminates. □

6.6 Progressiveness

From the results in [3] it follows that the progressiveness assumption can be modeled as *fairness constraints*. In particular, these fairness conditions assert that every clock either increases without bound or is infinitely often reset, but not continuously so. Let c_i be the largest constant to which clock i is ever compared in the enabling conditions of G.

One possibility for handling this fairness condition in the analysis of timed automata is to distinguish, in the initial partition, the cases where $(\vec{x})_i = 0$ and those where $(\vec{x})_i > c_i$, for every clock i. Now an infinite path in the set-graph is called progressive if and only if for every $i = 1 \ldots n$:

- it contains an infinite number of regions $\langle q, F\rangle$ such that for all $\vec{x} \in F$ either $(\vec{x})_i = 0$ or $(\vec{x})_i > c_i$.
- it contains an infinite number of regions $\langle q, F\rangle$ such that for all $\vec{x} \in F$, $(\vec{x})_i \neq 0$.

However this procedure is computationally expensive since it causes the initial partition to be fragmented into exponentially many pieces. We prefer to add this fairness information *iteratively*. We first let the initial partition be $\{q \times \mathbb{R} \mid q \in Q\}$, and generate a reachable subgraph, thus ignoring the progressiveness requirement for now. If it is empty, then there are no runs of the timed graph, and thus certainly no progressive runs. If not, we may prune from the graph all nodes that do not lie on any fair path. They need no longer be considered since they cannot be reached by any fair run. Now we split each remaining node set $\langle q, Z\rangle$ into $\langle q, Z_i\rangle$ such where

$$\{Z_i\} = Z \otimes \{\{\vec{x} \mid (\vec{x})_1 = 0\}, \{\vec{x} \mid 0 < (\vec{x})_1 \leq c_1\}, \{\vec{x} \mid (\vec{x})_1 > c_1\}\}$$

Thus all classes now respect the fairness constraints for clock 1. We now recompute a reachable graph starting from these classes as a partition of the state-space. This iterative step is continued until either the automaton is declared empty, or the fairness information for all clocks has been added.

7 Implementing Operations on Time Zones

We now discuss how to perform these operations on a particular representation for time zones due to Dill [1].

7.1 Difference Bounds Matrices

Recall that a time zone $Z \in \mathcal{Z}(n)$ is any (possibly unbounded) polyhedron defined by integer constraints on clocks and clock differences. If we identify a new fictitious clock variable x_0 with the constant value 0, these constraints can be represented as bounds on the difference between two clock values. For instance, $x < 5$ can be expressed as $x - x_0 < 5$. Furthermore, if we introduce $-\infty$ as a bounding value, we can restrict ourselves to upper bounds without loss of generality. More precisely, each inequality can be re-expressed in one of the following forms:

- $x_i - x_j < k$ or $x_i - x_j \leq k$, for some integer k,
- $x_i - x_j < -\infty$,
- $x_i - x_j < \infty$

Thus to describe these inequalities in a uniform fashion we introduce the domain of *bounds*. A bound is an order pair in $Z \times \{<, \leq\} \cup \{(\infty, <), (-\infty, <)\}$. Each bound is intended to represent an upper bound on a real value. We define an ordering on bounds as

$$(x, r) \prec (x', r') \text{ iff } \begin{cases} x < x' \text{ or} \\ x = x', r \text{ is } <, \text{ and } r' \text{ is } \leq \end{cases}$$

Bounds can be added, where $(x, r) \oplus (x', r') =$

$$\begin{cases} (x + x', <) & \text{if } x \;\&\; x' \text{ finite, and } r \text{ or } r' \text{ is } < \\ (x + x', \leq) & \text{if } x \;\&\; x' \text{ finite, and } r \;\&\; r' \text{ are } \leq \\ (-\infty, <) & \text{if } x \text{ or } x' \text{ is } -\infty \\ (\infty, <) & \text{otherwise} \end{cases}$$

A difference bounds matrix (DBM) for $\mathbb{R}^n$ is an $(n+1)\times(n+1)$ matrix of bounds, with rows and column indexed from 0 to n. The DBM A represents the polyhedron consisting of all points that satisfy the inequality

$$\begin{cases} x_i - x_j \le r & \text{if } a_{ij} = (r, \le) \\ x_i - x_j < r & \text{if } a_{ij} = (r, <) \end{cases}$$

Clearly every time zone can be described by a DBM. However there are many DBMs defining the same zone, because some of the upper bounds need not be tight. For example, the system

$$x_1 < 2 \quad (1)$$
$$x_1 \ge 1 \quad (2)$$
$$x_2 \le 5 \quad (3)$$

can be represented by any matrix

$$\begin{matrix} (0,\le) & (-1,\le) & (0,\le) \\ (2,<) & (0,\le) & b_1 \\ (5,<) & b_2 & (0,\le) \end{matrix}$$

where $b_1 \not< (2,<)$ and $b_2 \not< (-4,\le)$.

7.2 Canonical form for DBM's

The key idea in performing operations on zones is to represent them as canonical DBM's, where all upper bounds are as "tight" as possible. We denote this canonical matrix cf(Z). Dill [1] showed that this matrix can be computed from an arbitrary matrix for Z by applying an all-pairs shortest path algorithm. This representation therefore leads to easy tests for equality and emptiness of time zones.

7.3 Intersection

The intersection of two time zones Z and Z' can easily be computed from their DBM's. Intuitively we take the union of all the inequalities for each zone, by taking the lower of the two bounds for each pair of clock differences. Let A and A' be DBM's for Z and Z'. The zone $Z \cap Z'$ is represented by the matrix B where for all i and j, $b_{ij} = min\{a_{ij}, a'_{ij}\}$.

7.4 Time successors

If SI is a system of inequalities defining Z, then $Z_{\nearrow}$ is defined by the system SI' obtained from SI by removing all inequalities that place upper bounds on the absolute values of the clocks, *i.e.* inequalities of the form $x \le k$ or $x < k$. The following pseudo-code describes how this operation can be performed on a DBM.

```
procedure time_successors(A,B) {
   input  DBM A;    /* DBM for Z */
   output DBM B;    /* DBM for time successors */
   B := A;
   for i := 1 to n do
      B[i][0] := (infty,<);
   endfor
}
```

Notice that the input need not be canonical. However if it is, then the output will also be.

7.5 Time Predecessors

Similar to computation of time successors, we may replace all lower bounds on clocks with $(0,\le)$. However in this case, canonical input does not in general imply the output will be canonical.

7.6 Inverse Images of Reset Actions

Let a be a reset action of the variables in I_a. Its inverse $a^{-1}(Z)$ is computed by first finding the possible image a within Z (this is done by setting to 0 the bounds on the absolute value of each clock in I_a and canonicalizing), and then taking the inverse projection of the reset variables (by making all bounds relating to clocks in I_a trivial).

7.7 Zone Difference

We generate a partition of $Z - Z'$ by successively slicing off parts of Z that do not lie in Z'. We consider in turn each inequality $x_i - x_j < z'_{ij}$ from Z' as a potential face along which to slice Z. A slice is necessary along this face if it "touches" any points in Z. Let $cl(Z')$ denote the closure of Z', *i.e.* all points in $\mathbb{R}^n$ for which there are arbitrarily close points in Z'. If the restriction of $cl(Z')$ to points satisfying $x_i - x_j = z'_{ij}$ intersects Z, then slice off from Z all points for which $x_i - x_j < z'_{ij}$ is not satisfied. These points form a zone which is then added to the partition of $Z - Z'$.

```
procedure difference
begin
  input DBM Z;
  input DBM Z';
  output DBMset Z''; /* set of DBMs for Z-Z' */
  done := FALSE;
  Z'' := {};
  C := closure(Z');
  for (i,j distinct clocks) do
    if (NOT done) then
      A := intersect(Z,Z');
      if (A = {}) then
        /* what remains of Z can be put in Z'' */
        Z'' := Z'' + {Z};
        done := TRUE;
      else
        if (A = Z) then
          /* no more of Z does not lie in Z' */
          done := TRUE;
        else
          /* check whether to slice this face */
          /* val(<k,r>) = k */
          B := res(C,x_i-x_j=val(z'_ij));
          if (intersect(B,Z) <> {}) then
            /* slice along this face */
            Z'' := Z''+res(Z, x_i-x_j >= z'_ij);
            Z := res(Z, x_i-x_j < z'_ij);
          endif
        endif
      endif
    endif
  endfor
endprocedure
```

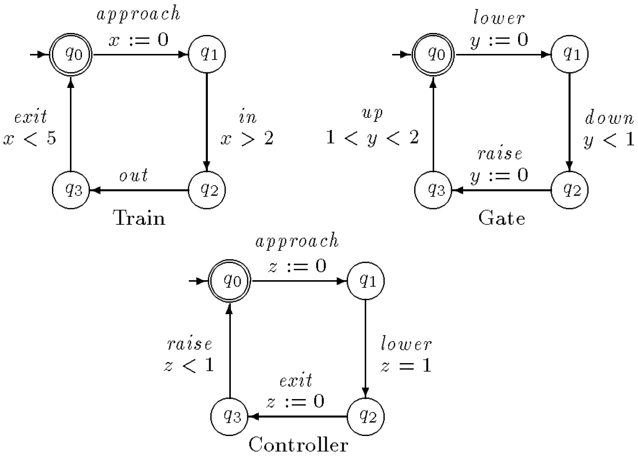

Figure 1: Automata for Train-Gate Controller Example

The restriction operator `res` is implemented by adding new constraints to a DBM and recomputing a canonical form. The closure operation is performed by replacing all strict inequalities on finite bounds with nonstrict inequalities.

8 Examples

In the examples below, a global automaton for the system is formed as the composition of automata for each component. Components synchronize their actions through shared events. Associated with each component is an alphabet of event symbols, and an event can occur provided it is enabled in every component automaton whose alphabet includes the event.

8.1 Train-Gate Controller

Our first example is an automatic controller that opens and closes a gate at a railway track intersection [21, 20]. The automata modeling the system's components are shown in Figure 1. Whenever a train enters the intersection, it sends an *approach* signal at least 2 minutes in advance to the controller. The controller also detects the train leaving the intersection, and this event occurs within 5 minutes after it started its approach. The gate responds to *lower* and *raise* commands by moving *down* and *up* respectively within certain time bounds. The controller sends a *lower* command to the gate exactly 1 minute after receiving an approach signal from the train. It commands the gate to raise within 1 minute of the train's exit from the intersection.

We verify a simple real-time safety property, namely that whenever the gate goes down, it is moved back up within a certain upper time bound K. In other words, the gate is never down for as long as K minutes. See the automaton in Figure 2. Its complement is expressed by the same automaton with state q_2

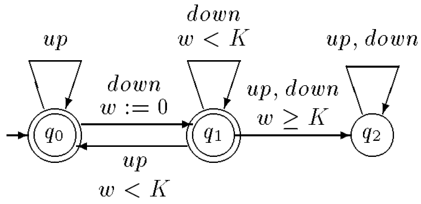

Figure 2: Real-time safety specification

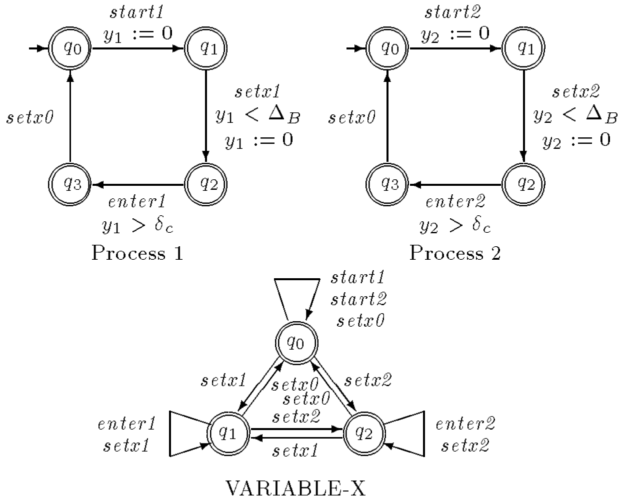

Figure 3: Automata for Mutual Exclusion Protocol

final instead of q_0 and q_1. Whenever the specification constant is greater than or equal to 7 the specification is satisfied.

8.2 Timed Mutual Exclusion

We also examine a simplified version of a mutual exclusion protocol due to Fischer [22]. A similar example appears in [23, 24]. There are n processes, each labeled with a unique identifier $i \in 1..n$. Initially each process is idle, but at any time may begin executing the protocol provided the value of a global variable x is 0. It then delays for up to Δ_B seconds before assigning the value i to the variable x. It may enter its critical section within δ_c seconds provided the value of x is still i. Upon leaving its critical section, it reinitializes x to 0. The automata for the case of two processes is given in Figure 3.

The conditions on the value of the global variable x are maintained by the special process called VARIABLE-X. If this process is at control location q_i then x equals i. Constraints on each process's behavior are expressed by disallowing certain process events when the value of x would prohibit it. For example the lack of a *start1* action from states q_1 and q_2 indicates Process 1 cannot start the protocol if x equals 1 or 2.

We verify the safety property (shown in Figure 4)

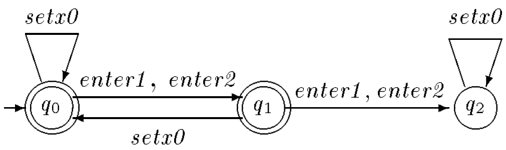

Figure 4: Mutual exclusion specification

that no two processes are ever in their critical sections at the same time.

9 Results and Comparison

9.1 Implementation

All three algorithms above have been implemented in C. The input to the program consists of a description of a timed automaton. The output indicates whether the automaton is empty or not. Composite timed automata may be specified as the product of individual timed automata. The input is passed through lex-yacc preprocessors to generate C source code, which is then compiled and executed. This allows constants to be hard-wired into the executable code: thus the exact amount of space required can be allocated to each DBM.

As classes are generated, they are stored in a global name table with pointers used to maintain the necessary stacking and queuing data-structures. Although not specified explicitly by the algorithms, information is stored to allow faster checks for stability. For each class $\langle q, Z\rangle$, there are pointers to classes which may be directly accessible from some state in $\langle q, Z\rangle$. These are maintained bidirectionally so that whenever a class is split it is easy to determine which classes may now be unstable. The check for strongly connected components is done using Tarjan's algorithm [25].

9.2 Results

The train-gate controller was tested for two cases. When the specification constant is $K = 10$, the controller meets its specification, and there are no violating runs. For $K = 5$, the specification is not satisfied.

In the timed mutual exclusion example, setting $\Delta_B > \delta_c$ allows the possibility of two processes entering their critical sections at the same time. The three implementations were tested for the values $\Delta_B = 5$ and $\delta_c = 12$, for which the specification is met, and $\Delta_B = 5$ and $\delta_c = 4$, where the specification is violated. Results were produced for up to 4 processes.

The data in Figure 5 was obtained on a DEC 5100 with 40 MB of main memory.

9.3 Comparison

Correctness vs Incorrectness : It is easier to prove a system behaves correctly than to prove it violates its specification. This is because proving non-emptiness of a timed system necessitates the iterative generation of a set-graph that respects the fairness constraints for the progressiveness requirement on *every* clock. On the other hand, emptiness may be detected much earlier.

Example	SR		MAI		MAII	
	Time	Mem	Time	Mem	Time	Mem
MUX-2	1	1	1	1	2	2
MUX-2-e	1	2	3	3	6	3
MUX-3	2	2	8	3	146	5
MUX-3-e	108	13	893	7	—	—
MUX-4	45	16	496	9	—	—
MUX-4-e	—	*	—	—	—	—
GTC	1	1	6	3	12	3
GTC-e	—	—	57	4	155	10

MUX-i indicates i-process mutual exclusion protocol
GTC Gate, train, controller example
-e contains error, specification not satisfied
— could not allocate more memory
$*$ managed to complete first iteration
Time is measured in seconds, and memory in Mbytes

Figure 5: Results Table

Removing Regions Not on Fair Runs : Between iterations of adding fairness constraints for each clock, the implementation removes regions known at the time not to lie on any fair runs. Although the comparative results are not given above, this optimization leads to reductions in computation time of around 50% for those examples where there were fair runs in the timed graph.

Minimization vs Set-Reachability : The results are slightly surprising in that the naive set-reachability algorithm often outperforms the minimization algorithms. We outline two possible reasons why it executes faster. First, the minimization algorithms require time to check classes for stability. Second, splitting classes into stable subclasses is an expensive operation: its counterpart in the set-reachability algorithm is simply to find the images of a class under all transitions.

However the graphs constructed from each of the minimization algorithms are far smaller than those generated by the SR algorithm, and we believe this fact will prove crucial for larger examples.

Minimization Algorithms I and II : In our particular implementations, MAI outperforms MAII. We believe this is mainly due to several implementational factors, rather than the algorithms themselves. Our implementation of MAII requires extra overhead in marking blocks with *regions* rather than states. A more efficient alternative would be to use new routines for manipulating single states with non-integer valued clocks. MAII has extra overhead in deciding which class to split next. Finally, MAII's performance could be significantly improved through more efficient data-structures.

10 Conclusions and further work

We have implemented three algorithms for checking the emptiness of a timed transition system. Preliminary results over two simple examples indicate that memory usage is a more limiting factor than time. The algorithms need to be tested on larger examples to demonstrate any practical benefits. However we believe that since the minimization algorithms produce far smaller graphs, they are more likely than explicit methods to be successful on bigger examples.

Much work needs to be done to make these algorithms more efficient in practice. Possible heuristics include regular checks that regions lie on fair runs (if they do not they may be discarded), and developing efficient yet simple strategies for choosing which region to split next. However, the emptiness problem is known to be PSPACE-complete, so there is no guarantee that any of the algorithms described here will be effective on any given example.

Acknowledgements
We thank David Lee and Mihalis Yannakakis for helpful discussions and suggestions.

References

[1] David L. Dill, Timing assumptions and verification of finite-state concurrent systems, In *Automatic Verification Methods for Finite State Systems, International Workshop, Grenoble, Lecture Notes in Computer Science 407*, pages 197–212. Springer-Verlag, 1989.

[2] R. Alur and D.L. Dill, Automata for modeling real-time systems, In *Proceedings of 17th International Colloquium on Automata, Languages and Programming*, pages 322–335. Springer Verlag, LNCS 443, 1990.

[3] Rajeev Alur, Costas Courcoubetis, and David Dill, Model-checking for real-time systems, In *Proceedings of the Fifth IEEE Symposium on Logic in Computer Science*, pages 414–425, 1990.

[4] Harry Lewis, A logic of concrete time intervals, In *Proceedings of the Fifth IEEE Symposium on Logic in Computer Science*, pages 380–389, 1990.

[5] Rajeev Alur and Thomas A. Henzinger, A really temporal logic, In *Proceedings of 30th Symposium on Foundations of Computer Science*, pages 164–169, 1989.

[6] N.A. Lynch and H. Attiya, Using mappings to prove timing properties, In *Proceedings of the Ninth ACM Symposium on Principles of Distributed Computing*, pages 265–280, 1990.

[7] K. Cerans, Decidability of bisimulation equivalence for parallel timer processes, In *Proceedings of Fourth Workshop on Computer-Aided Verification*, Montreal, Canada, 1992, to appear.

[8] Thomas A. Henzinger, Xavier Nicollin, Joseph Sifakis, and Sergio Yovine, Symbolic model checking for real-time systems, In *Proceedings of the Seventh IEEE Symposium on Logic in Computer Science*, pages 394–406, 1992.

[9] C. Courcoubetis and M. Yannakakis, Minimum and maximum delay problems in real-time systems, In *Proceedings of the Third Workshop on Computer-Aided Verification*, 1990.

[10] H. Wong-Toi and G. Hoffmann, The control of dense real-time discrete event systems (extended abstract), In *Proceedings of 30th IEEE Conference on Decision and Control*, pages 1527–1528, Brighton, England, December 1991.

[11] R. Alur, C. Courcoubetis, N. Halbwachs, D. Dill, and H. Wong-Toi, Minimization of timed transition systems (extended abstract), In *Proceedings of CONCUR '92*, pages 340–354, Stony Brook, NY, August 1992. Springer-Verlag, Lecture Notes in Computer Science, Volume 630.

[12] A. Bouajjani, J. Fernandez, and N. Halbwachs, Minimal model generation, In *Proceedings of Second Workshop on Computer-Aided Verification, Rutgers University*, 1990.

[13] David Lee and Mihalis Yannakakis, Online minimization of transition systems (Extended Abstract), In *Proceedings of ACM STOC 1992*, Vancouver, B.C., 1992.

[14] D.M.R. Park, Concurrency and automata on infinite sequences, In *Proceedings of 5th GI conference (P. Deussen. ed.)*, pages 167–183, 1981, LNCS 104.

[15] A.P. Sistla, M.Y. Vardi, and P. Wolper, The complementation problem for büchi automata with applications to temporal logic, *Theoretical Computer Science*, **49**:217–237, 1987.

[16] Z. Manna and A. Pnueli, Specification and verification of concurrent programs by ∀-automata, In *Proceedings of 14th Annual ACM Symposium on Principles of Programming Languages*, 1987.

[17] R.P. Kurshan, Analysis of discrete event coordination, In J.W. deBakker and G. Rozenberg W.-P. de Roever, editors, *Stepwise Refinement of Distributed Systems: models, formalisms, correctness, REX 89*. Springer-Verlag, 1990, LNCS 430.

[18] N.A. Lynch and M.R. Tuttle, Hierarchical correctness proofs for distributed algorithms, In *Proceedings of the Seventh ACM Symposium on Principles of Distributed Computing*, pages 137–151, 1987.

[19] A. Bouajjani, J. Fernandez, N. Halbwachs, P. Raymond, and C. Ratel, Minimal state graph generation, *Science of Computer Programming*, 1992, to appear.

[20] Rajeev Alur, *Techniques for Automatic Verification of Real-Time Systems*, PhD thesis, Department of Computer Science, Stanford University, August 1991.

[21] N. Leveson and J. Stolzy, Analyzing safety and fault tolerance using timed petri nets, In *Proceedings of International Joint Conference on Theory and Practice of Software Development*, pages 339–355, 1985, Lecture Notes in Computer Science 186.

[22] L. Lamport, A fast mutual exclusion algorithm, *ACM Transactions on Computer Systems*, **5**(1), February 1987.

[23] M. Abadi and L. Lamport, An old-fashioned recipe for real time, In *Proceedings of REX workshop "Real Time: Theory in Practice"*, 1991.

[24] Fred B. Schneider, Bard Bloom, and Keith Marzullo, Putting time into proof outlines, In *REX Workshop "Real-time: Theory in Practice"*, 1991.

[25] Robert Tarjan, Depth-first search and linear graph algorithms, *SIAM Journal of Computing*, **1**(2):146–160, June 1972.