May 2000

An Efficient State Space Generation for the Analysis of Real-Time Systems

Inhye Kang
Soongsil University

Insup Lee
University of Pennsylvania, lee@cis.upenn.edu

Young-Si Kim
Electronics and Telecommunications Research Institute

An Efficient State Space Generation for the Analysis of Real-Time Systems

Abstract
State explosion is a well-known problem that impedes analysis and testing based on state-space exploration. This problem is particularly serious in real-time systems because unbounded time values cause the state space to be infinite even for simple systems. In this paper, we present an algorithm that produces a compact representation of the reachable state space of a real-time system. The algorithm yields a small state space, but still retains enough information for analysis. To avoid the state explosion which can be caused by simply adding time values to states, our algorithm uses history equivalence and transition bisimulation to collapse states into equivalent classes. Through history equivalence, states are merged into an equivalence class with the same untimed executions up to the states. Using transition bisimulation, the states that have the same future behaviors are further collapsed. The resultant state space is finite and can be used to analyze real-time properties. To show the effectiveness of our algorithm, we have implemented the algorithm and have analyzed several example applications.

Keywords
Formal specification, reachability analysis, real-time analysis, state space minimization, timed automata

An Efficient State Space Generation for the Analysis of Real-Time Systems

Inhye Kang, *Member*, *IEEE*, Insup Lee, *Senior Member*, *IEEE*, and Young-Si Kim

Abstract—State explosion is a well-known problem that impedes analysis and testing based on state-space exploration. This problem is particularly serious in real-time systems because unbounded time values cause the state space to be infinite even for simple systems. In this paper, we present an algorithm that produces a compact representation of the reachable state space of a real-time system. The algorithm yields a small state space, but still retains enough information for analysis. To avoid the state explosion which can be caused by simply adding time values to states, our algorithm uses history equivalence and transition bisimulation to collapse states into equivalent classes. Through history equivalence, states are merged into an equivalence class with the same untimed executions up to the states. Using transition bisimulation, the states that have the same future behaviors are further collapsed. The resultant state space is finite and can be used to analyze real-time properties. To show the effectiveness of our algorithm, we have implemented the algorithm and have analyzed several example applications.

Index Terms—Formal specification, reachability analysis, real-time systems analysis, state space minimization, timed automata.

1 Introduction

As computers become ubiquitous, they are increasingly used in safety critical environments. Typical safety critical applications are control systems, monitoring systems and communication systems. Any failure of such computer systems may cause a great financial loss, environmental disaster, or even the loss of lives. The potential high cost associated with an incorrect operation of these systems has created a demand for a rigorous framework in which various design alternatives can be formally specified and rigorously analyzed and tested before implementation.

It is commonly believed that future safety critical systems will be more complex due to increased demands on their functionalities as well as the size of the problem domain. Thus, it will be difficult for one to analyze and test correctness without computer-aided tools. One common aspect of safety critical systems is that they must respond under stringent real-time constraints. That is, their correctness depends not only on how concurrent components interact, but also on the time at which these interactions occur. In addition, these systems are costly to prototype, requiring careful prediction of timing properties before implementation and evaluation of design alternatives.

Although the verification problem is in general undecidable, there exist several automatic verification and analysis techniques for finite state systems. Such techniques are usually based on *state space exploration*. That is, they identify a set of states that are reachable from the initial states and then analyze this set for verification. Such techniques exist for proving absence of deadlock or livelock, for proving properties expressed in propositional temporal logic or real-time logic, and for determining trace equivalence, testing preorder or bisimulation equivalence, etc.

The major weakness of the state space exploration based approach is that the size of the state space grows exponentially with the number of processes and thus creates the state space explosion problem. The problem is particularly serious in real-time systems because unbounded time values cause the state space to be infinite. Recently, there has been some work on constructing the finite representation of the reachable states, i.e., the reachability graph from a real-time system [27], [24], [17], [2].

Most of this work represents real-time using the discrete time model in which events can happen only at the integer time values [27], [24], [17]. The reachability analysis based on the discrete time model may not detect some reachable states in the real world where time is dense [1]. In the dense time model, events can happen at arbitrary points in time over real-line. For real-time systems with dense time, there exist little work on reachability analysis [2], [29]. This paper describes our approach to constructing a reachability graph for both discrete and dense time models. Our model for a real-time system is a timed automaton introduced in [2], [6]. The timed automaton is a finite automaton extended with

- *Inhye Kang is with the School of Computing, Soongsil University, Seoul, Korea.*
  E-mail: kang@computing.soongsil.ac.kr.
- *Insup Lee is with the Department of Computer and Information Science, University of Pennsylvania, Philadelphia, PA 19104-6389.*
  E-mail: lee@central.cis.upenn.edu.
- *Young Si Kim is with Electronics and Telecommunications Research Institute, 161 Gajong-Dong Yusong-Gu, Taejon, Korea.*
  E-mail: yskim@tdx.etri.re.kr.

Manuscript received 20 Nov. 1995; revised 13 Aug. 1998; accepted 1 Dec. 1998.
Recommended for acceptance by J. Gannon.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number 101166.

timing constraints. It has a finite set of nodes and transitions to represent control flow and a finite set of real-valued clocks to express timing constraints. A transition may depend on the values of the clocks and can reset some of the clocks. The values of the clocks increase at the same rate with the global time. The timed automaton can model a wide range of time-dependent behaviors such as time-out, delay, and lower/upper bounds between events using arbitrary number of clocks. Our goal is to develop a technique to efficiently represent the reachability graph of a timed automaton.

Timed automata have been extensively studied for verification of real-time systems [4], [3], [29], [23].

These are region-based approaches. A region is a set of states, where a state consists of a node and a clock valuation. In [2], a region includes states with the same node and a set of equivalent clock valuations in some sense, and a finite region graph is constructed using the partitioning algorithm given in [8]. The region graph has size exponential in the number of clocks and the size of the constants that appear in the enabling conditions of the transitions. Because the region graph is too fine-grained, minimization approaches [4], [29] have been proposed in order to equate more valuations. The minimal region graphs, however, still have the same worst case complexity as region graphs. Another region-based approach to generate finite reachable state space from a timed automaton is based on forward analysis [14], [5]. The forward analysis repeatedly computes the region that includes reachable states by initially starting from the initial region and then adding states reachable from the current region through time passage or transitions until the region contains all reachable states, i.e., the fixed point reaches. One drawback of the forward analysis is that it does not guarantee the termination of the procedure.

In this paper, we present an algorithm that produces a compact reachability graph from a timed automaton. The algorithm usually yields a small state space, but it retains reachability and event ordering information for analysis of real-time properties such as safety properties and bounded response time properties. Our algorithm uses the notions of history equivalence and transition bisimulation to cluster states into equivalence classes. In our approach, states are defined as histories (i.e., executions upto the states). In history equivalence, states that have the same untimed hisotries are equivalent and are merged into one. Since there exist infinitely many untimed histories, the state space minimized by history equivalence is still infinite. Using transition bisimulation, states that have the same future behaviors are further merged, so the resultant state space becomes finite. Comparing to the minimal region graph approaches, our approach is minimized based on traces, while the minimal region graph approaches are based on branching-time structures. Comparing to the forward analysis approaches, the number of equivalence classes in our approach is finite, whereas the forward analysis approaches may generate infinitely many regions. Our approach is implemented in a tool, called TREAT (Timed Reachability Analysis Tool).

The paper is organized as follows. Section 2 describes the syntax and semantics of timed automata and also reviews existing approaches for reachability analysis on timed automata. Section 3 defines equivalence relations, namely history equivalence and transition bisimulation. Section 4 presents the algorithm that generates a reachable state space according to the underlying equivalence relations. Section 5 reports on case studies that show the efficiency of our tool TREAT, and compares our results with other tools. Section 6 summarizes relevant research in state-space generation techniques for real-time systems. In Section 7, we conclude the paper with current and future research issues.

2 Timed Automata

Various kinds of timed automata have been used to describe real-time Systems [2], [23], [6]. In this paper, we adopt the timed automaton introduced by Nicollin et al. [23] which associates timing constraints with both nodes and transitions.

2.1 The Syntax

A timed automaton has a finite set of nodes and transitions to represent control flow and a finite set of variables called *clocks* to express timing constraints. The domain of clocks is the set of real numbers. The values of all clocks are initially zero and increase at the same rate, but any subset of them can be reset to zero on a transition. Timing constraints are associated with both nodes and transitions.

The syntax of timed automata is defined as follows. Let I be the set of non-negative integers, and let R be the set of nonnegative real numbers. Let C be the set of timing constraints expressed using the conjuctions over the atomic formulas of the form $x \sim i$ for clock x and integer i.[1]

The timing constraints do not include comparisons of two or more clock values such as $x_1 + i_1 \sim x_2 + i_2$.

Definition 2.1. A *timed automaton A* is a tuple $(N, n_{init}, X, \Sigma, Inv, T)$, where

1. N is a finite set of nodes;
2. n_{init} is the initial node;
3. X is a finite set of clocks;
4. Σ is a finite set of events;
5. $Inv : N \rightarrow C$ is a timing constraint on each node; and
6. $T \subseteq N \times C \times \Sigma \times 2^X \times N$ is a transition relation.

1. Disjunctions can be represented as separated edges with conjunctions and real numbers in constants can be modified to integers by multiplying all constants by 10^k for some k.

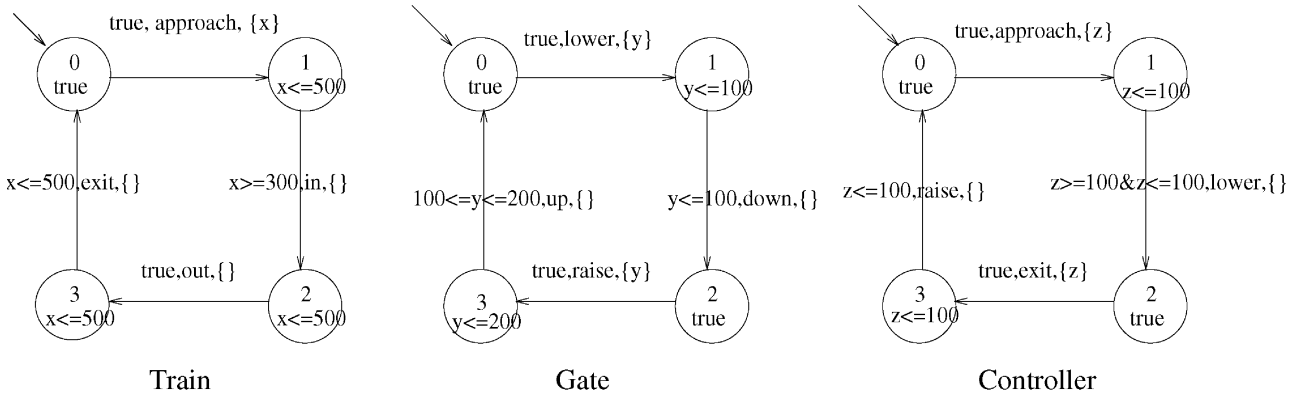


Fig. 1. Timed automata for Train, Gate, and Controller.

The function Inv associates with each node $n \in N$ a timing constraint called the *invariant* of n. The system's control can stay in a node n only while the current clock valuation satisfies $Inv(n)$. This constraint forces control to move to the next node before it becomes false to prevent control being stuck in a node. We restrict invariants to be conjunctions of atomic formulas of the form $x \leq i$. For a transition $a = (n_1, c, \sigma, Y, n_2) \in T$, if the current node n_1 satisfies the timing constraint c, then the system can take the transition. As the result of taking the transition, the system performs event e, resets all clocks in Y to zero, and instantaneously moves to the next node n_2.

We use the following notations on a transition $a = (n_1, c, \sigma, Y, n_2)$ for convenience: $source(a)$ is the source node n_1, $target(a)$ is the target node n_2, $condition(a)$ is the enabling condition c, $event(a)$ is the event σ, and $resetclocks(a)$ is the set of clocks Y.

Composition. In general, a system consists of several timed automata running in parallel and communicating with each other. These concurrent timed automata can be composed into a global timed automaton as follows: transitions of the timed automata that do not execute a shared event are interleaved, whereas transitions using a shared event are synchronized.

Definition 2.2. Let $A_1 = (N_1, n_{init1}, X_1, \Sigma_1, Inv_1, T_1)$ and $A_2 = (N_2, n_{init2}, X_2, \Sigma_2, Inv_2, T_2)$. The composition $A = A_1||A_2$ of A_1 and A_2 is a tuple $(N, n_{init}, X, \Sigma, Inv, T)$, where

1. $N = N_1 \times N_2$;
2. $n_{init} = (n_{init1}, n_{init2})$;
3. $X = X_1 \cup X_2$ (assume $X_1 \cap X_2 = \emptyset$);
4. $\Sigma = \Sigma_1 \cup \Sigma_2$;
5. $Inv(n_1, n_2) = Inv_1(n_1) \wedge Inv_2(n_2)$; and
6. T is given as follows:

- for all $(n_1, c_1, \sigma_1, Y_1, n_1') \in T_1$ and
$$(n_2, c_2, \sigma_2, Y_2, n_2') \in T_2,$$
if σ_1 is equal to σ_2, then T includes
$$((n_1, n_2), c_1 \wedge c_2, \sigma_1, Y_1 \cup Y_2, (n_1', n_2'));$$
- for all $(n_1, c_1, \sigma_1, Y_1, n_1') \in T_1$, if σ_1 is not in $\Sigma_1 \cap \Sigma_2$, then for all $n_2 \in N_2$, T includes $((n_1, n_2), c_1, \sigma_1, Y_1, (n_1', n_2))$; and
- for all $(n_2, c_2, \sigma_2, Y_2, n_2') \in T_2$, if σ_2 is not in $\Sigma_1 \cap \Sigma_2$, then for all $n_1 \in N_1$, T includes $((n_1, n_2), c_2, \sigma_2, Y_2, (n_1, n_2'))$.

Example: Railroad Crossing System. The standard railroad crossing problem has been used to compare different formal methods for real-time systems [13]. Fig. 1 shows an automatic controller that opens and closes a gate at a railroad crossing presented in [3]. The system is formed as the composition of three components, Train, Gate, and Controller, which execute in parallel and synchronize through the events: $approach$, $exit$, $lower$, and $down$. When a train approaches the crossing, Train sends an $approach$ signal to Controller and sends an in signal at least 300 seconds later to its environment to represent that a train enters the crossing. When a train leaves the crossing, Train generates an out signal to its environment for representing that a train leaves the crossing and then sends an $exit$ signal to Controller for synchronizing with it. The $exit$ signal is sent within 500 seconds after the $approach$ signal. Controller sends a signal $lower$ to Gate exactly 100 seconds after the $approach$ signal and sends a $raise$ signal within 100 seconds after $exit$. Gate responds to $lower$ by moving $down$ within 100 seconds and responds to $raise$ by moving up between 100 and 200 seconds. The composed timed automaton from Train, Gate, and Controller is shown in Fig. 2. For simplicity, it ignores nodes that have no path from the initial node because any such node is obviously unreachable. Node (i, j, k) represents that Train, Gate, and Controller are at nodes i, j, and k, respectively.

2.2 The Semantics

The semantics of a timed automaton is given by *executions* and *behaviors*. We first explain the executions using the railroad crossing system in Fig. 2. Initially, the system control resides at node $(0, 0, 0)$, and the values of clocks x, y, and z are all zero. At 20.5 seconds, the values of x, y, and z become 20.5 at node $(0, 0, 0)$. If transition $b1$ is taken at that time, the system executes event $approach$, and control

moves to node $(1,0,1)$. Since x and z are reset by transition $b1$, the values of x, y, and z are 0, 20.5, and 0, respectively. The invariant of the node $(1,0,1)$ is "$x \leq 500 \wedge z \leq 100$." Since the current values of x and z are zero, control can stay at node $(1,0,1)$ for at most 100 seconds. The enabling condition "$z \geq 100$" of transition $b2$ and the enabling condition "$x \geq 300$" of transition $b3$ remain false during this 100 second time period. At time 120.5, the values of x and z are both 100. Since the enabling condition of $b2$ becomes true at that time, the transition can be executed. On the other hand, the enabling condition of $b3$ is still false. Since control must leave from the node due to the invariant, the system executes $b2$, i.e., performs event *lower* and moves to the next node $(1,1,2)$ at time 120.5.

We now define executions of a timed automaton. Let $v, v_1, \ldots$ represent clock valuations. For a valuation v and a real number r, let $v+r$ be valuation v' such that $v'(x) = v(x) + r$ for $x \in X$.

Definition 2.3 An *execution* of a timed automaton $A = (N, n_{init}, X, \Sigma, Inv, T)$ is defined as a finite sequence

$$(n_0, v_0) \xrightarrow{a_1,t_1} (n_1, v_1) \xrightarrow{a_2,t_2} (n_2, v_2) \cdots \xrightarrow{a_k,t_k} (n_k, v_k)$$

or an infinite sequence:

$$(n_0, v_0) \xrightarrow{a_1,t_1} (n_1, v_1) \xrightarrow{a_2,t_2} (n_2, v_2) \cdots \xrightarrow{a_k,t_k} (n_k, v_k) \cdots$$

satisfying the following constraints:

1. Initiality: $n_0 = n_{init}$, $v_0(x) = 0$ for all $x \in X$, and $t_0 = 0$;
2. Invariant Constraint: for each $i \geq 0$, $(v_i + r)$ satisfies $Inv(n_i)$ for $0 \leq r \leq (t_{i+1} - t_i)$ (if it is finite, $(v_k + r)$ satisfies $Inv(n_k)$ for $r \geq 0$);
3. Succession Constraint: for each $i \geq 0$ ($0 \leq i \leq k-1$ if it is finite), there exists transition a_{i+1} in T with source n_i and target n_{i+1} such that
   - $(v_i + t_{i+1} - t_i)$ satisfies $condition(a_{i+1})$ and
   - $$v_{i+1}(x) = \begin{cases} 0 & \text{if } x \in resetclocks(a_{i+1}) \\ v_i(x) + t_{i+1} - t_i & \text{otherwise; and} \end{cases}$$
   - Monotonicity: $t_i \leq t_{i+1}$ for $i \geq 0$.[2]

In an execution, t_i means the global time passed after the start of the execution. And "$\xrightarrow{a_i,t_i}$" means that transition a_i happens t_i time units after the system starts the execution, and "$\xrightarrow{a_i,t_i}(n_i, v_i)\xrightarrow{a_{i+1},t_{i+1}}$" indicates that control stays at node n_i from t_i to t_{i+1}.

In the railroad crossing example, we have an execution

$$((0,0,0),(0,0,0)) \xrightarrow{b1,20.5} ((1,0,1),(0,20.5,0)) \xrightarrow{b2,120.5} ((1,1,2),(100,0,100)) \cdots$$

where valuation (r_1, r_2, r_3) indicates that the values of x, y, and z are r_1, r_2, and r_3, respectively.

When we analyze a system we are usually interested in behaviors rather than the valuations of clocks, where a behavior is a sequence of events with their occurrence times. A behavior of a timed automaton can be obtained from an execution as described in the following definition.

Definition 2.4 For an execution

$$(n_0, v_0) \xrightarrow{a_1,t_1} (n_1, v_1) \xrightarrow{a_2,t_2} (n_2, v_2) \xrightarrow{a_3,t_3} (n_3, v_3) \cdots,$$

the corresponding *behavior* is

$$\langle\ (event(a_1), t_1),\ (event(a_2), t_2),\ (event(a_3), t_3),\ \cdots \rangle.$$

For example, the railroad crossing system has a behavior $\langle\ (approach,\ 20.5),\ (lower, 120.5),\ \cdots\rangle$ which comes from the execution

$$((0,0,0),(0,0,0)) \xrightarrow{b1,20.5} ((1,0,1),(0,20.5,0)) \xrightarrow{b2,120.5} \cdots.$$

The set of possible executions or behaviors can be combined into a labeled transition system. A labeled transition system is defined as follows.

Definition 2.5. A *labeled transition system* is a tuple $(S, S_0, L, \rightarrow)$, where

1. S is a set of states;
2. $S_0 \subseteq S$ is a set of initial states;
3. L is a set of labels; and
4. $\rightarrow \subseteq S \times L \times S$ is a transition relation.

The formal definition of the corresponding labeled transition system for a given timed automaton is described in [1].

3 Our Approach: Background Theory

Our approach is to add history equivalence to the definition of states instead of clock valuations because clock values cause state explosion. We also give a labeled transition system for a timed automaton according to newly defined states. We then define history equivalence and transition bisimulation for minimizing states and discuss properties which the minimized state space preserves.

3.1 States

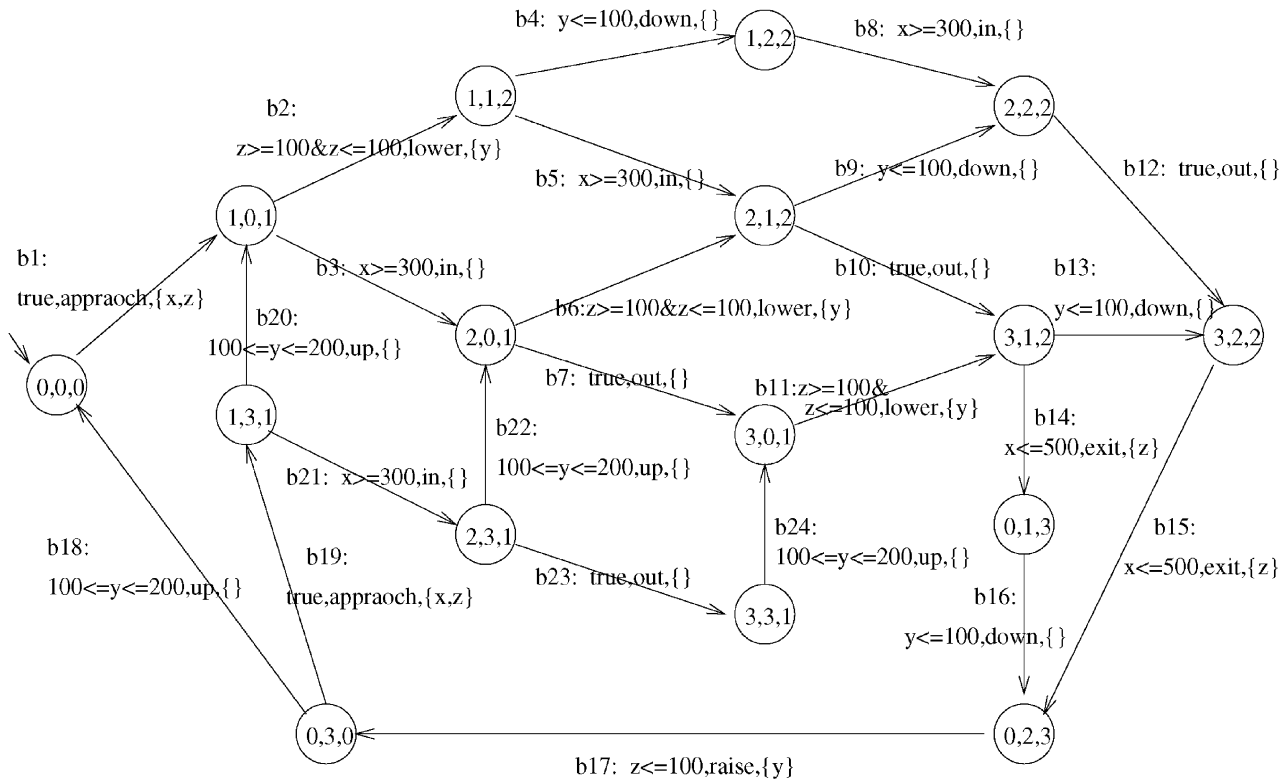
We first define states for a given timed automaton $A = (N, n_{init}, X, \Sigma, Inv, T)$. Let a_{init} be a dummy transition representing the start of the execution such that $target(a_{init})$ is the initial node n_{init} and $resetclocks(a_{init})$ is the set X of all clock variables. We define a *timed history* as a sequence

$$\langle (a_0, t_0), (a_1, t_1), (a_2, t_2), \ldots, (a_i, t_i), \ldots \rangle$$

where $a_0 = a_{init}$, $t_0 = 0$, $a_i \in T$, $t_i \in R$, $t_{i-1} \leq t_i$ for $i \geq 1$. We define a *state* as (node, timed history) instead of (node, clock valuation). State

$$(n, \langle (a_0, t_0), (a_1, t_1), (a_2, t_2), \ldots, (a_k, t_k) \rangle)$$

2. Monotonicity constraint ensures time that does not decrease but allows several actions at the same time.



Inv(0,0,0)=true
Inv(0,1,3)=(y<=100&z<=100)
Inv(0,2,3)=(z<=100)
Inv(0,3,0)=(y<=200)
Inv(1,0,1)=(x<=500&z<=100)
Inv(1,1,2)=(x<=500&y<=100)
Inv(1,2,2)=(x<=500)
Inv(1,3,1)=(x<=500&y<=200&z<=100)
Inv(2,0,1)=(x<=500&z<=100)
Inv(2,1,2)=(x<=500&y<=100)
Inv(2,2,2)=(x<=500)
Inv(2,3,1)=(x<=500&y<=200&z<=100)
Inv(3,0,1)=(x<=500&z<=100)
Inv(3,1,2)=(x<=500&y<=100)
Inv(3,2,2)=(x<=500)
Inv(3,3,1)=(x<=500&y<=200&z<=100)

Fig. 2. Timed automation for railroad crossing system.

represents that the system starts its execution at time t_0, executes transitions $a_1, a_2, \ldots$ at times $t_1, t_2, \ldots$, respectively, and control is currently in node n. We note that $n = target(a_k)$. With this definition of states, we give an *execution* of the system as follows.

Definition 3.1. (execution) For a timed automaton $A = (N, n_{init}, X, \Sigma, Inv, T)$, an execution is given by:

$$(let\ th_i = \langle (a_0, t_0), (a_1, t_1), \ldots, (a_i, t_i) \rangle)$$

$$(n_0, th_0) \overset{a_1,t_1}{\rightarrow} (n_1, th_1) \overset{a_2,t_2}{\rightarrow} (n_2, th_2) \cdots \overset{a_k,t_k}{\rightarrow} (n_k, th_k) \cdots$$

satisfying

- initiality: $n_0 = n_{init}$, $a_0 = a_{init}$, $t_0 = 0$;
- invariant constraint: $v_i + r$ satisfies $inv(n_i)$ for $0 \le r \le (t_{i+1} - t_i)$;
- succession constraint: $v_i + (t_{i+1} - t_i)$ satisfies $condition(a_{i+1})$; and
- time monotonicity: $t_i \le t_{i+1}$,

where $v_0(x) = 0$ for $x \in X$, $v_{i+1}(x) = 0$ for

$$x \in resetclocks(a_{i+1}),$$

and $v_{i+1}(x) = v_i(x) + (t_{i+1} - t_i)$ for $x \notin resetclocks(a_{i+1})$.

For a timed automaton A, all possible executions of A define a labeled transition system as follows. Let $execs(A)$ be the set of all possible executions of A.

Definition 3.2. Given a timed automaton $A = (N, n_{init}, X, \Sigma, Inv, T)$, the corresponding labeled transition system is $M_{lts}(A) = (S_{lts}, S_{lts0}, L_{lts}, \rightarrow_{lts})$, where

- $S_{lts} = \{(n, th) | \cdots (n, th) \cdots \in execs(A)\}$;
- $S_{lts0} = \{(n_{init}, \langle (a_{init}, 0) \rangle)\}$;
- $L_{lts} = T \times R$; and
- $$\rightarrow_{lts} (\subseteq S_{lts} \times L_{lts} \times S_{lts})$$
  $$= \{((n_1, th_1), (a, t), (n_2, th_2)) | th_2 = th_1 \cdot \langle (a, t) \rangle\}.$$

Here, we represent a transition $((n_1, th_1), (a, t), (n_2, th_2))$ by: $(n_1, th_1) \overset{a,t}{\rightarrow} (n_2, th_2)$.

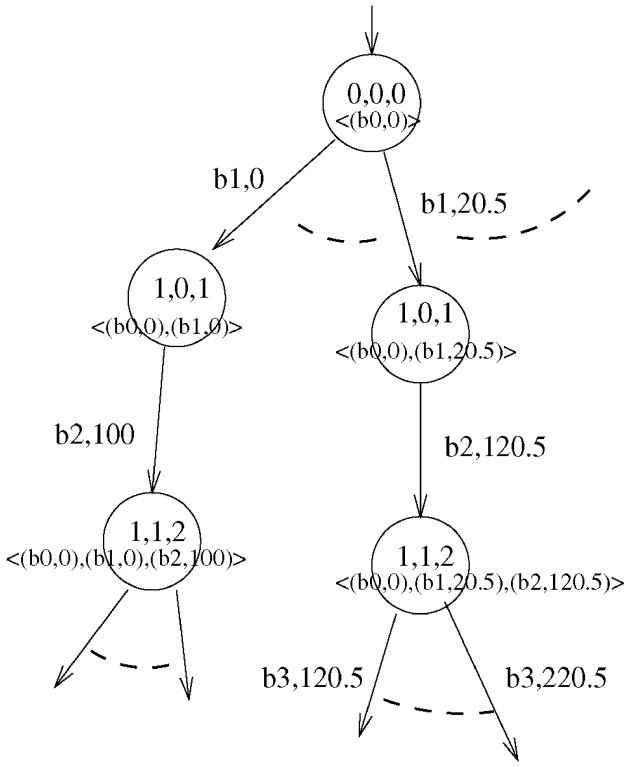


Fig. 3. Labeled transition system for railroad crossing system.

We relate the notion of execution (called old execution) in Definition 3.1 with the notion of execution in Definition 3.1 as follows. For an old execution

$$(n_0, v_0) \stackrel{a_1,t_1}{\rightarrow} (n_1, v_1) \stackrel{a_2,t_2}{\rightarrow} (n_2, v_2) \cdots,$$

the corresponding newly defined execution is

$$(n_0, th_0) \stackrel{a_1,t_1}{\rightarrow} (n_1, th_1) \stackrel{a_2,t_2}{\rightarrow} (n_2, th_2) \cdots,$$

where $th_i = \langle (a_{init}, 0), (a_1, t_1), \ldots, (a_i, t_i) \rangle$. And for an execution

$$(n_0, th_0) \stackrel{a_1,t_1}{\rightarrow} (n_1, th_1) \stackrel{a_2,t_2}{\rightarrow} (n_2, th_2) \cdots,$$

the corresponding old execution is

$$(n_0, v_0) \stackrel{a_1,t_1}{\rightarrow} (n_1, v_1) \stackrel{a_2,t_2}{\rightarrow} (n_2, v_2) \cdots,$$

where $v_0(x) = 0$ for $x \in X$, $v_{i+1}(x) = 0$ for

$$x \in resetclocks(a_{i+1}),$$

and $v_{i+1}(x) = v_i(x) + (t_{i+1} - t_i)$ for $x \notin resetclocks(a_{i+1})$.

Fig. 3 shows a part of the labeled transition system corresponding to the railroad crossing system in Fig. 2.

3.2 History Equivalence

For a timed history $th = \langle (a_0, t_0), (a_1, t_1), (a_2, t_2), \ldots \rangle$, we define *(untimed) history* by:

$$untimed(th) = \langle a_0, a_1, a_2, \ldots \rangle.$$

For a state (n, th), let $untimed(n, th) = (n, untimed(th))$.

Definition 3.3. (history equivalence) Two states s_1 and s_2 are history equivalent if $untimed(s_1) = untimed(s_2)$.

Given timed automaton A, if we minimize the labeled transition system $M_{lts}(A)$ with respect to history equivalence, then the minimal labeled transition system $M_{hist}(A)$ is defined as follows.

Definition 3.4. For a timed automaton

$$A = (N, n_{init}, X, \Sigma, Inv, T),$$

let $M_{lts}(A) = (S_{lts}, S_{lts0}, L_{lts}, \rightarrow_{lts})$. The minimal labeled transition system with respect to history equivalence is $M_{hist}(A) = (S_{hist}, S_{hist0}, L_{hist}, \rightarrow_{hist})$, where

- $S_{hist} = \{untimed(s)) | s \in S_{lts}\}$;
- $S_{hist0} = \{untimed(s_0) | s_0 \in S_{lts0}\}$ (i.e., $\{(n_{init}, \langle a_{init} \rangle)\}$);
- $L_{hist} = \{a | (a, t) \in L_{lts}\}$ (i.e., $L_{hist} = T$); and
- $$\rightarrow_{hist} = \{(untimed(s_1), a, untimed(s_2)) | ((s_1, (a, t), s_2) \in \rightarrow_{lts}\}.$$

Here, we represent a state as a (node, untimed history) pair after clustering history-equivalent states. For a state $s = (n, h)$ in S_{hist}, let $node(s) = n$ and $history(s) = h$. For a transition tt in $\rightarrow_{hist}$, let $label(tt) = a$.

Definition 3.5. For a timed automaton

$$A = (N, n_{init}, X, \Sigma, Inv, T),$$

let $M_{hist}(A) = (S_{hist}, S_{hist0}, L_{hist}, \rightarrow_{hist})$. For a state $s \in S_{hist}$,

- s is said to be reachable;
- $history(s)$ is said to be valid; and
- $node(s)$ is said to be reachable through $history(s)$.

For the railroad crossing system, the labeled transition system shown in Fig. 3 is minimized with respect to history equivalence as shown in Fig. 4.

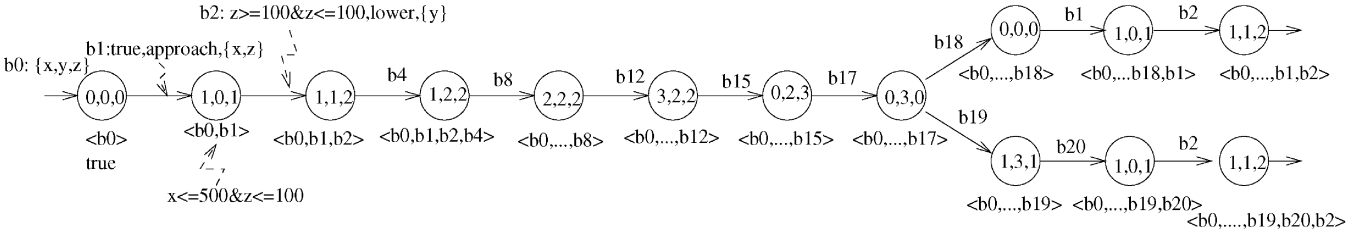


Fig. 4. Reachability tree for railroad crossing system.

$$\begin{array}{lccccccc} M_{lts}(A): & (n_0,th_0) & \stackrel{a_1,t_1}{\rightarrow} & (n_1,th_1) & \ldots & \stackrel{a_n,t_n}{\rightarrow} & (n_n,th_n) & \ldots \\ M_{hist}(A): & (n_0,h_0) & \stackrel{a_1}{\rightarrow}_{hist} & (n_1,h_1) & \ldots & \stackrel{a_n}{\rightarrow}_{hist} & (n_n,h_n) & \ldots \\ M_{hb}(A): & equiv((n_0,h_0)) & \stackrel{a_1}{\rightarrow}_{hb} & equiv((n_1,h_1)) & \ldots & \stackrel{a_n}{\rightarrow}_{hb} & equiv((n_n,h_n)) & \ldots \end{array}$$

Fig. 5. Relationship among labeled transition systems for timed automation A.

After collapsing history equivalence states, states lose absolute clock values and transitions lose absolute transition time values. However, history equivalence preserves reachability in the sense that all and only reachable nodes in the original system A appear in the minimal labeled transition system $M_{hist}(A)$. And also, history equivalence preserves untimed behaviors of A, that is, $\langle \sigma_1, \sigma_2, \ldots \rangle$ is a behavior of M if and only if it is a behavior of $M_{hist}(A)$, that is, there exists a sequence $a_1, a_2, \ldots$ in $M_{hist}(A)$ such that $event(a_i) = \sigma_i$).

3.3 Strong Transition Bisimulation

Definition 3.6. (transition bisimulation) For a timed automaton A, let $M_{hist}(A) = (S_{hist}, S_{hist0}, L_{hist}, \rightarrow_{hist})$. A relation $\rho \subseteq S_{hist} \times S_{hist}$ is a transition bisimulation if for all $(s_1, s_2) \in \rho$, for all $a \in L_{hist}$,

- whenever $s_1 \stackrel{a}{\rightarrow}_{hist} s_1'$, there exists s_2' such that $s_2 \stackrel{a}{\rightarrow}_{hist} s_2'$ and (s_1', s_2') is also in ρ;
- whenever $s_2 \stackrel{a}{\rightarrow}_{hist} s_2'$, there exists s_1' such that $s_1 \stackrel{a}{\rightarrow}_{hist} s_1'$ and (s_1', s_2') is also in ρ.

Two states s_1 and s_2 are said to be *transition bisimilar*. Transition bisimulation is the same as strong bisimulation [22]. We just call it transition bisimulation to emphasize that label a represents a transition in T instead of an event in Σ.

If we minimize the labeled transition system $M_{hist}(A)$ with respect to transition bisimulation, the minimal labeled transition system is given as follows. For a state s, let $equiv(s)$ be the set of states equivalent to s with respect to transition bisimulation.

Definition 3.7. For a timed automaton A, let

$$M_{hist}(A) = (S_{hist}, S_{hist0}, L_{hist}, \rightarrow_{hist}).$$

Then the minimal labeled transition system with respect to history equivalence and transition bisimulation is $M_{hb}(A) = (S_{hb}, S_{hb0}, L_{hb}, \rightarrow_{hb})$, where

- $S_{hb} = \{equiv(s) | s \in S_{hist}\}$;
- $S_{hb0} = \{equiv(s) | s \in S_{hist0}\}$;
- $L_{hb} = L_{hist}$;
- $\rightarrow_{hb} = \{(equiv(s), a, equiv(s')) | (s, a, s') \in \rightarrow_{hist}\}$.

We note that the minimal labeled transition system $M_{hb}(A)$ has a finite number of states and transitions. (See Theorem 4.2 later.) For each transition $tt = (s, a, s')$ in $\rightarrow_{hb}$, let $TAtransition(tt) = a$. And for each $s \in S_{hb}$, let $incoming(s)$ ($outgoing(s)$) be the set of transitions in $\rightarrow_{hb}$ whose target (source) states are s.

Fig. 5 describes the relationship among $M_{lts}(A)$, $M_{hist}(A)$ and $M_{hb}(A)$, where $h_i = untimed(th_i)$ for $i \geq 0$.

Theorem 3.1. *For a timed automaton A, $M_{hb}(A)$ preserves reachability and event ordering:*

1. *reachability: n is reachable through some execution of M iff $M_{hb}(A)$ includes some state s such that $node(s) = n$;*
2. *event ordering: $\langle \sigma_1 \sigma_2 \sigma_3 \ldots \rangle$ is a behavior of M iff M_{hb} has a sequence of transitions $tt_1 tt_2 tt_3 \ldots$ with labels $a_1 a_2 a_3 \ldots$ such that $event(a_i) = \sigma_i$.*

Proof. We omit the proof. However, it is easy to see from Fig. 5. □

4 Our Approach: Algorithm

Our approach is summarized as follows: given a timed automaton A,

1. construct the labeled transition system $M_{lts}(A)$;
2. minimize $M_{lts}(A)$ with respect to history equivalence (we have $M_{hist}(A)$); and
3. minimize $M_{hist}(A)$ with respect to transition bisimulation (we have $M_{hb}(A)$).

However, in practice it is impossible to construct the intermediate labeled transition systems $M_{lts}(A)$ and $M_{hist}(A)$ because they have an infinite number of states although $M_{hb}(A)$ is finite. We thus develop an algorithm that constructs the minimal labeled transition system $M_{hb}(A)$ with respect to history equivalence and transition bisimulation, directly from A without generating the intermediate labeled transition systems. In the algorithm, we assume that we have the following functions:

1. *transition-bisimilar*(s_1, s_2): returns true if two states s_1, s_2 are transition bisimilar; and
2. *valid-history*(h): returns true if history h is valid.

In the current implementation, the labeled transition system $M_{alg}(A)$ generated from the algorithm is bigger than $M_{hb}(A)$ because we have a sufficient (not a necessary) condition for checking transition bisimilarity among states, that is, *transition-bisimilar*(s_1, s_2) may return false although s_1, s_2 are transition bisimilar. In this section, we present the algorithm and then give how to implement the two functions.

4.1 Construction Algorithm

We now present an algorithm that constructs a reachability graph from a given timed automaton A. The resultant reachability graph is the labeled transition system in which

algorithm Reachability Graph Construction
input: a timed automaton $(N, n_{init}, X, \Sigma, Inv, T)$;
output: a minimal graph $M_{alg}(A) = (S_{alg}, S_{alg0}, L_{alg}, \rightarrow_{alg})$;

1: create the initial state $s_0 = (n_{init}, \langle(a_0, 0)\rangle)$;
$Explored := \emptyset$; $Edges := \emptyset$;
$Unexplored := \{s_0\}$;

2: **while** $Unexplored \neq \emptyset$ **do**
pick and remove a state s_1 from $Unexplored$;
2A: **if** there exists a state s_2 in $Explored$ such that
$transition\text{-}bisimilar(s_1, s_2)$ is true
2A-1: **then**
let tt be the incoming edge of s_1;
$Edges := (Edges - \{tt\})$
$\cup\{(source(tt), label(tt), s_2)\}$;
2A-2: **else**
add s_1 to $Explored$;
for every outgoing transition $a \in T$ of $node(s_1)$ **do**
2A-2a: create a state $s_2 = (target(a), history(s_1) \cdot \langle a\rangle)$;
2A-2b: **if** $history(s_2)$ is valid **then**
$Unexplored := Unexplored \cup \{s_2\}$;
$Edges := Edges \cup \{(s_1, a, s_2)\}$;
end for
end while
$S_{alg} := Explored$; $S_{alg0} := \{s_0\}$; $L_{alg} := T$; $\rightarrow_{alg} := Edges$;
end algorithm

Fig. 6. Construction algorithm.

the number of states is reduced using history equivalence and transition bisimulation.

The algorithm is given in Fig. 6. Step 1 is initialization. During the algorithm, $Explored$ keeps all explored states, $Unexplored$ keeps unexplored states which are immediately reachable from explored states, and $Transitions$ includes explored transitions. Here, a state is a (node, untimed history) pair. For a transition $tt = (s_1, a, s_2) \in Transitions$, let $source(tt) = s_1$, $target(tt) = s_2$ and $label(tt) = a$. Step 2 repeats as long as there is at least one unexplored state. In Step 2, it picks an unexplored state s_1. If there exists an explored state s_2 transition bisimilar to the selected state s_1 (at Step 2A), then it gets rid of s_1 and adjusts the target state of the incoming transition of s_1 as s_2 in Step 2A-1. On the other hand, if there is no such state, then the selected state is added to $Explored$ and states immediately reachable from the state are created and added to $Unexplored$ in Step 2A-2.

The algorithm generates the minimal labeled transition system $M_{hb}(A)$ with respect to history equivalence and transition bisimulation if two functions *transition-bisimilar* and *valid-history* are supported [18].

4.2 Implementation

We discuss how to compute *transition-bisimilar*(s_1, s_2) for states s_1 and s_2 and *valid-history*(h) for a history h which are used in the algorithm. We first define the minimum and maximum time distances between transitions in a history, and then give conditions, in terms of the distances, under which a history is valid and two states are transition bisimilar, respectively.

4.2.1 Minimum and Maximum Distances

Given a timed automaton A, let $M_{lts}(A)$ be

$$(S_{lts}, S_{lts0}, L_{lts}, \rightarrow_{lts}).$$

For a valid history $h = \langle a_0, a_1, \ldots, a_k\rangle$ and for

$$1 \leq i \leq j \leq k, min\_dist(a_i, a_j, h)$$

and $max\_dist(a_i, a_j, h)$ are defined as the minimum and maximum time distances, respectively, from a_i to a_j for all executions associated with the history:

$$\begin{aligned} min\_dist(a_i, a_j, h) = min\{t_j - t_i | (n, \langle(a_0, t_0), \ldots, (a_i, t_i), \ldots, \\ (a_j, t_j), \ldots, (a_k, t_k)\rangle) \in S_{lts}\} \\ max\_dist(a_i, a_j, h) = max\{t_j - t_i | (n, \langle(a_0, t_0), \ldots, \\ (a_i, t_i), \ldots, (a_j, t_j), \ldots, (a_k, t_k)\rangle) \in S_{lts}\}. \end{aligned}$$

Here, $min\_dist(a_i, a_j, h)$ determines the earliest time at which the transition a_j can take place after the transition a_i, whereas $max\_dist(a_i, a_j, h)$ determines the latest time by which the transition a_j must take place after the transition a_i. That is a_j happens at least $min\_dist(a_i, a_j, h)$ time units and at most $max\_dist(a_i, a_j, h)$ time units after a_i for all executions associated with h. For a clock x, $reset(x, h)$ is the last transition in h on which x is reset, and $last(h)$ is the last transition in the history, that is, $last(h) = a_k$. We note that there is not the case that x is not reset along h because x is reset on a_{init}. At the entering time to state (l, h), the value of x is in between $min\_dist(reset(x, h), last(h), h)$ and $max\_dist(reset(x, h), last(h), h)$.

We compute min_dist and max_dist using weighted graphs. This method is similar to the one used by Modechart [17]. For a history $h = \langle a_0, a_1, \ldots, a_k \rangle$, let $W(h)$ be the weighted graph (V, E, w), where V is a set of vertices, E is a set of directed edges, and $w : E \rightarrow I$ is a weight function of edges such that:

- $V = \{v_i | 0 \leq i \leq k\}$ (v_i is a vertex corresponding to a_i);
- $$\begin{aligned} E \ =& \{(v_{i-1}, v_i), (v_i, v_{i-1}) \mid 1 \leq i \leq k\} \cup \\ & \{(v_i, v_j) \mid x \geq l \in cond(a_j) \\ & \quad \text{and } a_i = reset(x, \langle a_0, a_1, \ldots, a_{j-1} \rangle)\} \cup \\ & \{(v_j, v_i) \mid (x \leq l \in cond(a_j) \\ & \quad \text{or } x \leq l \in inv(source(a_j)) \\ & \quad \text{and } a_i = reset(x, \langle a_0, a_1, \ldots, a_{j-1} \rangle)\}; \end{aligned}$$
- if $i < j$, then
$$w(v_i, v_j) := max(\{0\} \cup \{l | x \geq l \in cond(a_j) \text{ and } a_i = reset(x, \langle a_0, a_1, \ldots, a_{j-1} \rangle)\});$$
if $i > j$, then
$$w(v_i, v_j) := max(\{-\infty\} \cup \{-l | (x \leq l \in cond(a_i) \text{ or } x \leq l \in inv(source(a_i))) \text{ and } a_j = reset(x, \langle a_1, a_2, \ldots, a_{i-1} \rangle)\}.$$

We assign the earliest time that a_j can happen after a_i happens to $w(v_i, v_j)$, directly from conditions of M. For $cond(a_j)$, relations with "$\geq$" define $w(v_i, v_j)$ and relations with "$\leq$" defines $w(v_j, v_i)$ for $i < j$. Suppose that

$$cond(a_j) = x_1 \geq c_1 \wedge x_2 \leq c_2, \text{ and}$$

$$a_i = reset(x_l, \langle a_0, a_1, \ldots, a_{j-1} \rangle)$$

for $l = 1, 2$. Then, a_j can happen at least c_1 time units and at most c_2 time units after a_i happens. In other words, the earliest time from executing a_i to executing a_j is c_1 and the earliest time from executing a_j to executing a_i is $-c_2$. Thus, we assign c_1 to $w(v_i, v_j)$ and $-c_2$ as $w(v_j, v_i)$. If several relations with "$\geq$" exist, for example,

$$cond(a_j) = x_1 \geq 10 \wedge x_2 \geq 20$$

and $a_i = reset(x_l, \langle a_0, a_1, \ldots, a_{j-1} \rangle)$ for $l = 1, 2$, then a_j can happen both 10 and also 20 time units after a_i happens. Thus, a_j can happen 20 time units ($max\{10, 20\}$) after a_i happens, that is, $w(v_i, v_j) = 20$. On the other hand, if several relations with "$\leq$" exist (say,

$$cond(a_j) = x_1 \leq 10 \wedge x_2 \leq 20, \ inv(source(a_j)) = x_3 \leq 30,$$

and $a_i = reset(x_l, \langle a_0, a_1, \ldots, a_{j-1} \rangle)$ for $l = 1, 2, 3$), then a_j must happen within 10 time units, within 20 time units, and within 30 time units after a_i. Thus, a_j must happen within 10 time units ($min\{10, 20, 30\}$), that is, $w(v_j, v_i) = -10$.

For a valid history h, we compute min_dist and max_dist from the weighted graph $W(h)$ as follows. In $W(h)$, there can be several paths from v_i to v_j with different weights. Suppose that there exist two paths p_1 and p_2 from v_i to v_j for $i \leq j$ with weights c_1 and c_2. Then a_j happens at least c_1 time units after a_i, in order that conditions associated with p_1 are satisfied and happens at least c_2 time units after a_i in order that conditions associated with p_2 are satisfied. In order that all conditions in h are satisfied, a_j happens at least $max\{c_1, c_2\}$ time units after a_i. Similarly, if there exist two paths p_1 and p_2 from v_j to v_i with weights $-c_1$ and $-c_2$, then a_i happens at least $max\{-c_1, -c_2\}$ time units after a_j. In other words, a_j happens at most $-max\{-c_1, -c_2\}$ time units after a_i. We determine min_dist, max_dist as follows:

- $min\_dist(a_i, a_j, h)$ is equal to the maximum weight among all path weights from the node corresponding to a_i to the node corresponding to a_j; and
- $max\_dist(a_i, a_j, h)$ is equal to the absolute value of the maximum weight among all path weights from the node corresponding to a_j to the node corresponding to a_i.

4.2.2 Testing History Validity

Using the weighted graph, we can compute whether a given history is valid by the following theorem.

Definition 4.1. For a history h, h is valid if and only if $W(h)$ has no positive cycle.

Proof. The proof is given in the appendix. □

4.2.3 Testing Transition Bisimulation

We show how to compute function *transition-bisimilar*(s_1, s_2) in Fig. 6. Transition bisimulation is a relation for future behaviors. We cannot enumerate all future behaviors due to their infiniteness. Thus, we develop a condition with which we decide transition bisimilarity among states without regarding future. The condition is given in terms of minimum and maximum time distances.

Let $a_1 \prec a_2$ represent that a_1 precedes a_2. Let Max_c be the largest among the constants appearing in the conditions of the timed automaton. We can partition clocks in X into three according to history h as follows:

Definition 4.1. For a history h,

1. $$P_1(h) = \{x \in X | min\_dist(reset(x,h), last(h), h) > Max_c\};$$

2. $$P_2(h) = \{x \in X | min\_dist(reset(x,h), last(h), h) \leq Max_c \text{ and } max\_dist(reset(x,h), last(h), h) > Max_c\};$$

3. $$P_3(h) = \{x \in X | max\_dist(reset(x,h), last(h), h) \leq Max_c\}.$$

Definition 4.2. For two states s_1 and s_2, let $h_1 = history(s_1)$ and $h_2 = history(s_2)$ and let

- $cond1(s_1, s_2) := node(s_1) = node(s_2)$;
- $$cond2(s_1, s_2) := P_1(h_1) = P_1(h_2) \wedge P_2(h_1) = P_2(h_2) \wedge P_3(h_1) = P_3(h_2)$$
- $$cond3(s_1, s_2) := cond3.1(s_1, s_2) \wedge cond3.2(s_1, s_2)$$
  $$cond3.1(s_1, s_2) := \forall x \in P_2(h_1). \; min\_dist(reset(x, h_1), last(h_1), h_1) = min\_dist(reset(x, h_2), last(h_2), h_2)$$
  $$cond3.2(s_1, s_2) := \forall x \in P_3(h_1). \; min\_dist(reset(x, h_1), last(h_1), h_1) = min\_dist(reset(x, h_2), last(h_2), h_2) \wedge max\_dist(reset(x, h_1), last(h_1), h_1) = max\_dist(reset(x, h_2), last(h_2), h_2)$$
- $$cond4(s_1, s_2) := \forall x_1, x_2 \in P_2(h_1) \cup P_3(h_1). \; reset(x_1, h_1) \prec reset(x_2, h_1) \Rightarrow cond4.1(s_1, s_2) \vee cond4.2(s_1, s_2)$$
  $$cond4.1(s_1, s_2) := max\_dist(reset(x_1, h_1), reset(x_2, h_1), h_1) > Max_c \wedge max\_dist(reset(x_1, h_2), reset(x_2, h_2), h_2) > Max_c \wedge min\_dist(reset(x_1, h_1), reset(x_2, h_1), h_1) = min\_dist(reset(x_1, h_2), reset(x_2, h_2), h_2)$$
  $$cond4.2(s_1, s_2) := min\_dist(reset(x_1, h_1), reset(x_2, h_1), h_1) = min\_dist(reset(x_1, h_2), reset(x_2, h_2), h_2) \wedge max\_dist(reset(x_1, h_1), reset(x_2, h_1), h_1) = max\_dist(reset(x_1, h_2), reset(x_2, h_2), h_2).$$

  Then,

  $$bisim\_cond(s_1, s_2) = cond1(s_1, s_2) \wedge cond2(s_1, s_2) \wedge cond3(s_1, s_2) \wedge cond4(s_1, s_2).$$

The first condition, $cond1(s_1, s_2)$, means that two states are associated with the same nodes. We note that states that come from the different nodes are not merged by history equivalence and transition bisimulation. (See the definition of history equivalence in Definition 3.3 and the definition of transition bisimulation in Definition 3.6.)

The conditions $cond2(s_1, s_2)$ and $cond3(s_1, s_2)$ show the relation between the time that each clock was reset and the current time. Obviously, if

$$min\_dist(reset(x, h_1), last(h_1), h_1)$$

is greater than Max_c and $min\_dist(reset(x, h_2), last(h_2), h_2)$ is greater than Max_c, then the enabling condition of s_1 and s_2 over x is evaluated to the same value in both s_1 and s_2. For example, suppose $Max_c = 10$ and the enabling condition is $x \leq 10$. Here the enabling condition evaluates to false in both s_1 and s_2 if their min_dist values are greater than 10. If their minimum distances are not greater than Max_c, then their minimum time distances should be the same and their maximum time distances should be either the same or any values greater than or equal to Max_c. Here, the evaluated value of the form $x \sim k$ for $\sim = \leq$ or $\geq$ is the same in both the states.

The conditions $cond2(s_1, s_2)$ and $cond4(s_1, s_2)$ give the relation between t he reset times of every two clocks. Suppose that

$$min\_dist(reset(x, h_1), last(h_1), h_1) = min\_dist(reset(x, h_2), last(h_2), h_2) = 5 \text{ and } min\_dist(reset(y, h_1), last(h_1), h_1) = min\_dist(reset(y, h_2), last(h_2), h_2) = 3.$$

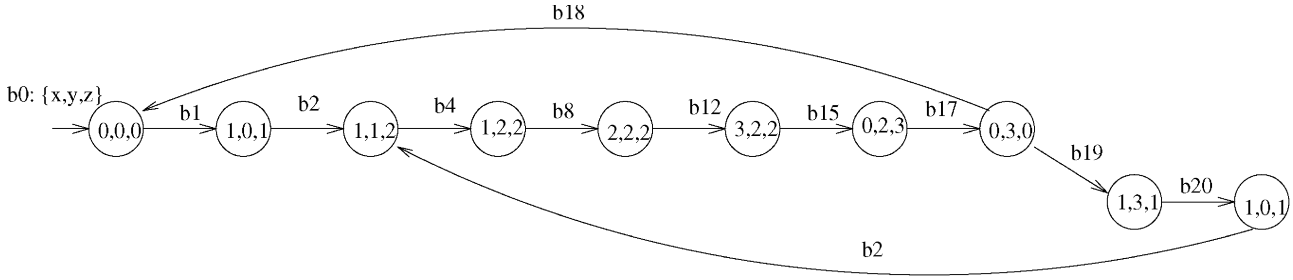


Fig. 7. Reachability graph for the railroad crossing system.

Here, $cond2(s_1, s_2)$ is true. However, if

$$min\_dist(reset(y, h_1), reset(x, h_1), h_1) = 1$$

and

$$min\_dist(reset(y, h_2), reset(x, h_2), h_2) = 2$$

(i.e., the value of $(x - y)$ is greater than or equal to 1 at state s_1 and 2 at state s_2), then an enabling condition $x = 5 \wedge y = 4$ is satisfiable at state s_1, but is always false at state s_2. This is why the third condition is necessary in addition to the second condition.

Lemma 4.1 states that $bisim\_cond$ is a condition for bisimulation. It is a sufficient condition, not a necessary condition. Lemma 4.2 states that the construction algorithm, shown in Fig. 6, always terminates.

Lemma 4.1. If $bisim\_cond(s_1, s_2)$ is true, then s_1 and s_2 are transition bisimilar.

Proof. The proof is given in the appendix. □

Theorem 4.2. *A relation* $\{(s_1, s_2) \mid bisim\_cond(s_1, s_2)\}$ *has finitely many equivalence classes.*

Proof. The proof is given in the appendix. □

The total number of states ever put into *Unexplored* is also finite. Thus, the algorithm always terminates. Theorem 4.3 shows that in the worst case, the size of the timed reachability graph generated using Definition 4.2 is doubly exponential to the number of clocks. This bound is the same as that of the minimal region graph [4].

Theorem 4.3. *For a given timed automaton, the number of equivalence classes using* $bisim\_cond$ *is bounded by* $O(|L| \times Max_c^{k \times k})$, *where* Max_c *is the largest among the constants appearing in the invariants and enabling conditions and* k *is the number of clocks.*

Proof. It follows directly from the proof of Theorem 4.2. □

Example. In the railroad crossing system in Fig. 4, let us consider two states

$$s_1 = ((1, 1, 2), \langle b_0, b_1, b_2 \rangle) \text{ and}$$
$$s_2 = ((1, 1, 2), \langle b_0, b_1, b_2, b_4, b_8, b_{12}, b_{15}, b_{17}, b_{19}, b_{20}, b_2 \rangle).$$

Let $h_1 = history(s_1)$ and $h_2 = history(s_2)$. Obviously, $cond1(s_1, s_2)$ is true. For clock x,

$$min\_dist(reset(x, h_1), last(h_1), h_1) = min\_dist(reset(x, h_2), last(h_2), h_2) =$$
$$max\_dist(reset(x, h_1), last(h_1), h_1) = max\_dist(reset(x, h_2), last(h_2), h_2) = 100.$$

The minimum and maximum distances for z are also 100. And the minimum and maximum distances for y are zero because $reset(y, h_i) = last(y, h_i)$ for $i = 1, 2$. Thus, $cond2(s_1, s_2)$ is true. Finally, $cond3(s_1, s_2)$ is also true because $min\_dist(reset(x, h_i), reset(y, h_i), h_i)$ equals to 100 for $i = 1, 2$ and so on. Thus, (s_1, s_2) is in transition bisimilar, that is, the two states s_1 and s_2 are transition bisimilar.

Fig. 7 shows the reachability graph for the railroad crossing system.

Implementation. We have implemented in TREAT the algorithm that generates the reachability graph with time relations using C++ and the algorithm that composes two timed automata into a global timed automaton. The program is about 2,000 lines of code. The reachability graph shown in Fig. 7 was drawn manually using the reachability information that was automatically generated.

4.3 Analysis of Properties

For real-time systems, the practical goal is to verify safety properties such as deadlock-freeness, mutual exclusion, and meeting timing constraints. Reachability analysis is used to prove that systems never enter *unsafe* states. We can prove verify safety properties using reachability graphs generated from the algorithm.

Absence of Deadlock. In the reachability graph, if we can find a state which has no outgoing transitions, we conclude that the system can deadlock or terminate.

General Properties as Timed Automata. In [3], properties given in timed automata are proved as follows:

1. Model the system with timed automata, $M_1, M_2 \ldots, M_n$;
2. Specify properties as a timed automaton M_s;
3. Construct the reachability graph from the composed timed automaton, $(M_1||M_2|| \ldots ||M_n||M_s)$; and
4. Decide whether the system is correct.

5 Applications

We now illustrate the application of our approach with three examples: the railroad crossing control system, the Fischer's mutual exclusion protocol, and the active structure control system, and compare the experimental results of TREAT with HyTech [15] and Kronos [10], [11].

The experiments were executed on Sun Microsystems 60 MHz SuperSPARC with 256 MB of physical memory. They were executed under the following three limitations:

1. Time limitation: every experiment is performed at most for 24 hours;
2. Memory limitation: memory usage of every experiment is limited to 256 MB; and
3. Tool limitation: each tool may have constraints such as the size of the system given as an input. If an experiment fails due to the time limitation, the memory limitation, or the tool limitation, then it is represented by $fail_{time}$, $fail_{mem}$, or $fail_{tool}$, respectively.

5.1 Railroad Crossing System

The railroad crossing control system is a benchmark example for real-time formal method. The railroad crossing lies in a region delimited by entry and exit sensors that detect the entry and exit of trains. There are n tracks in the crossing; that is, there can be up to n trains approaching, leaving or in the crossing. The correctness of the railroad crossing control system is given by: whenever a train is in the crossing, the gate is down.

Model. The railroad crossing control system $RCCS$ is given in Fig. 8. The system is modified from the system described in Section 2.1 to deal with multiple tracks. The Train process consists of processes $Train_i$ which models the behaviors of track i. If $RCCS$ has k tracks, then

$$RCCS = \mathrm{Train_1}||\ldots||\mathrm{Train_k}||\mathrm{Controller}||\mathrm{Gate}.$$

For $1 \leq i \leq k$, Train_i has four nodes. Controller has three nodes and Gate has four nodes. In $RCCS$, there exist $k+2$ clocks since for $1 \leq i \leq n$, the Train_i process has clock x_i, the Controller process has clock y and the Gate process has clock z. $RCCS$ has a data variable nt to represent the number of trains in the crossing.

Analysis. We now show whether the $RCCS$ process satisfies the safety property. To satisfy the property, Gate must stay in node $g2$ while Train_i is in node $t2$ for some i. If Train_i is in $t2$ for some i and Gate is in $g0$, $g1$, or $g3$, then the system is unsafe. Thus we can prove the safety property by showing whether there exists an unsafe state in $M_{alg}(RCCS)$ after TREAT generates $M_{alg}(RCCS)$.

Experimental Results. Table 1 shows experimental results of TREAT, HyTech, and Kronos. TREAT gives that the number of states for one track is 12, the number of states for two tracks is 83, and the number of states for three tracks is 10,892. For four tracks, TREAT fails due to the time limitation, i.e., TREAT fails to generate the minimal labeled transition system within 24 hours. For $k = 1, 2, 3$, $M_{alg}(RCCS)$ does not include a state that Train_i is in $t2$ for some i and Gate is in $g0$, $g1$, or $g3$. Thus, the railroad crossing control system satisfies the safety property.

For TREAT and HyTech, it gives the time taken and the number of states generated during analysis. Since Kronos dose not construct the state space but performs symbolic model checking, it gives the time taken for analysis. HyTech provides both forward and backward analysis techniques. The time taken by TREAT and HyTech includes the composition time as well as the analysis time. On the other hand, the time taken by Kronos includes only the analysis time because Kronos accepts the composed system as an input.

For HyTech's forward analysis approach, HyTech generates 12 regions in 2.69 seconds for one track. With two tracks, HyTech fails due to the time limitation. In fact, HyTech does not terminate because newly generated regions are not included in existing regions at each iteration. For example, there exists a behavior such that at least one train is always in the crossing, that is, before a train in track 1 leaves the crossing, a train in track 2 enters the crossing, and vice versa. For node $(t2, t0, c0, g2)$, the value of x_1 is always in [300, 500] and z increases at each iteration because z is never reset. At each iteration, a newly generated region for $(t2, t0, c0, g2)$ is not included in existing regions. As we see in this case, one advantage of our approach over the forward analysis is to generate a finite number of states for any given system.

For HyTech's backward analysis approach, it proves the property up to three tracks like TREAT. TREAT performs the analysis faster than HyTech up to two tracks while HyTech performs the analysis faster than TREAT for three tracks. The backward analysis does not generate the reachable state space of the system. The states generated by the backward analysis are not the states to which the initial state can reach but the states from which the error state is reached.

Compared to the results of Kronos, Kronos proves the safety property up to five tracks. With six tracks, Kronos fails to compose the system due to the tool limitation. Kronos allows the number of nodes less than 2^{16} because nodes are represents using the C type "short" (2 bytes). But, the number of nodes of the composed system with six tracks exceeds the limit. Kronos gives much better results than TREAT and HyTech in this example. Kronos does not construct the reachable state space.

5.2 Fischer's Mutual Exclusion Protocol

Mutual exclusion arises when it is necessary for a shared resource to be accessed by only one process at a time. With concurrent systems, more than one process may simultaneously try to access the same resource. Thus the systems

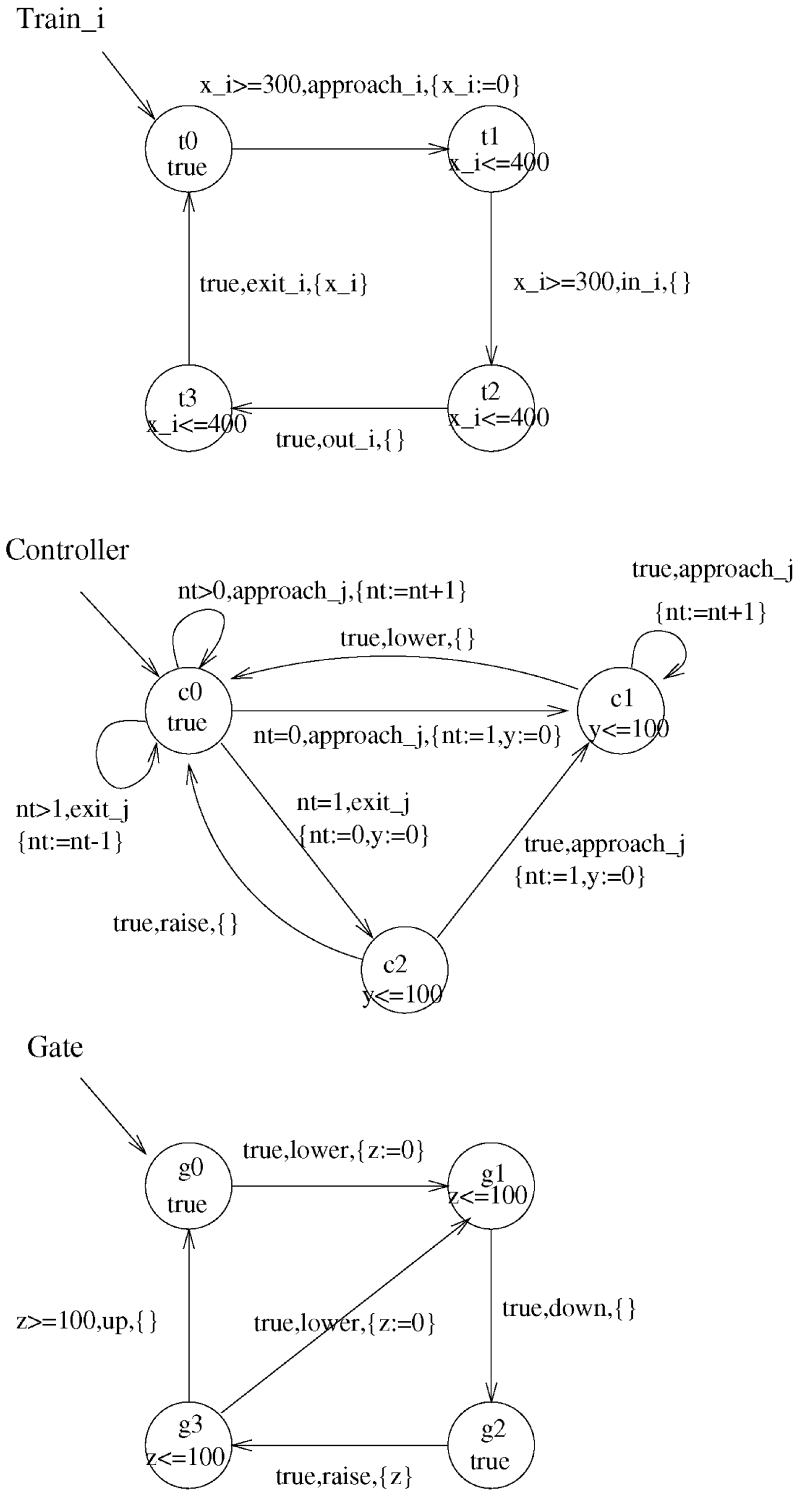


Fig. 8. Railroad crossing control system.

are required to provide a mechanism that makes accesses to a critical resource by concurrent processes mutually exclusive. One such technique is a simple timing-based mutual exclusion protocol due to Fischer [20].

In Fischer's protocol, the system *MUTEX* has several processes, each P_i executes the algorithm shown in Fig. 9. Assume that the statement $s := i$ takes no more than a time units. Then we have two timing parameters a and b in the algorithm. The algorithm includes a shared variable s. In *MUTEX*, it violates the mutual exclusion if it includes a state in which any two processes P_i and P_j are in the critical sections CS_i and CS_j simultaneously. In other words, the correctness of the system is that the system never reaches such a state.

TABLE 1
Experimental Results: TREAT, HyTech, and Kronos for the Railroad Crossing Control System

	TREAT		HyTech (forward)		HyTech (backward)		Kronos
Track#	Time	State#	Time	State#	Time	State#	Time
1	0.1s	12	2.73s	12	5.2s	120	0.07s
2	1.2s	83	$fail_{time}$	(∞)	3m2s	2,472	1.17s
3	1h26m54s	10,892			38m37s	19,711	4.81s
4	$fail_{time}$				$fail_{time}$		62.75s
5							21m20s
6							$fail_{tool}$

Model. The shared variable is modeled as a process S since CTSM does not have shared variables. Then $MUTEX = P_1||P_2||\ldots||P_n||S$ for n concurrent processes. We describe *MUTEX*=$(P_1||\ldots||P_n||S)$ for $a = 10$ and $b = 20$, that is assignment $s := i$ takes at most 10 time units and the delay in Step 2 is 20 time units. The system *MUTEX* has channels $value(j,i)$, $set(0,i)$, $set(i,i)$ for $1 \leq i,j \leq n$, as shown in Fig. 10a. Through event $value(j,i)$, process S is synchronized with P_i to inform that the current value of s is j. Through $set(i,i)$, process P_i is synchronized with process S to inform that P_i changes the value of s to i. And, action $set(0,i)$ by process P_i is sent to process S to indicate that P_i changes the value of s to zero.

Process S consists of $(n+1)$ nodes representing the values $(0,1,\ldots,n)$ of s. The process is initially at node 0, where it outputs a signal to channel $value(0,i)$ if P_i wants to get the value of s for $1 \leq i \leq n$. If it receives a signal from channel $set(i,i)$, then it moves to node i. For $1 \leq j \leq n$, node j represents that the value of s is j. At node j, the process S outputs a signal to channel $value(j,i)$ if P_i wants to get the value of s for $1 \leq i \leq n$. If it receives a signal from channel $set(0,i)$, then it returns to the initial node 0. If it receives a signal from channel $set(i,i)$ for $1 \leq i \leq n$, then it moves to node k. Fig. 10b shows the process S for $n = 2$.

For $1 \leq i \leq n$, process P_i is shown in Fig. 10b. It has a clock x_i to represent the timing constraints. It is initially at node 0. It can move to the next node 1 at any time if $s = 0$ (i.e., it receives a signal from channel $value(0,i)$). At node 1, it sends a signal to process S through channel $set(i,i)$ within 10 time units because assignment $s := i$ takes at most 10 time units, and moves to node 2. The timing constraint is given by the invariant "$x_i \leq 10$" of node 1. At node 2, it gets the value of s from process S at least 20 time units after it enters to the node due to the statement. The timing constraint is given by the enabling conditions "$x_i \geq 20$" in outgoing transitions from node 2. If it receives a signal from channel $value(i,i)$ (i.e., $s = i$), then it enters the critical section CS_i. If it receives a signal from channel $value(j,i)$ for $i \neq j$ (i.e., $s \neq i$), then it fails to enter the critical section and returns to the initial node 0.

```
repeat
        repeat
0:              wait s = 0;
1:              s := i;
2:              delay b;
        until s = i;
CS_i:   Critical Section;
        s := 0;
forever
```

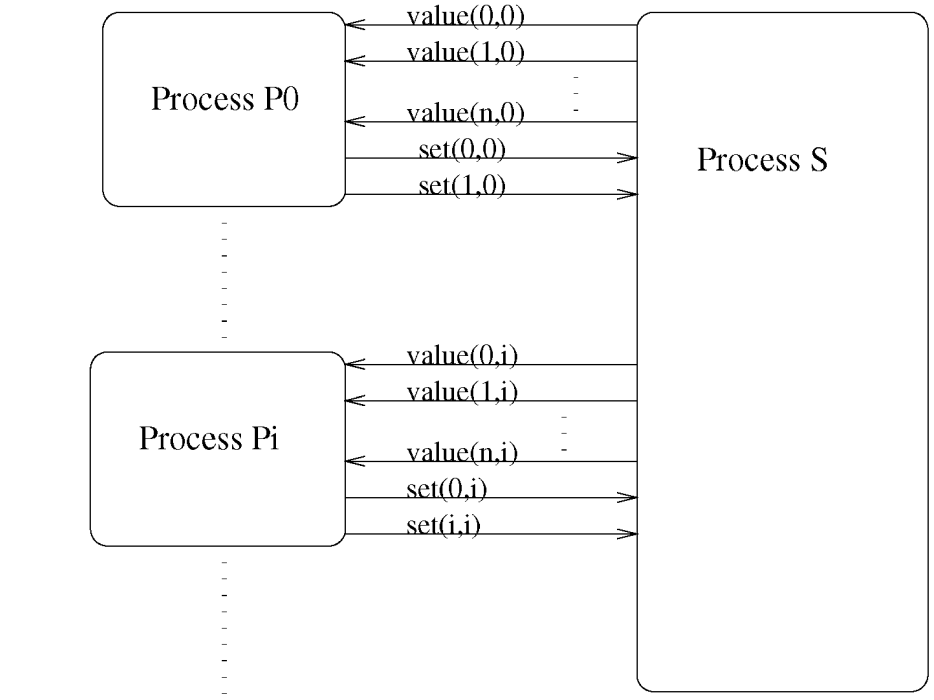
Fig. 9. Fischer's mutual exclusion protocol.

For $MUTEX$ with $a = 10, b = 10$, all processes are the same except for the enabling conditions "$x_i \geq 20$" in outgoing transitions from node 2 of process P_i is changed to "$x_i \geq 10$."

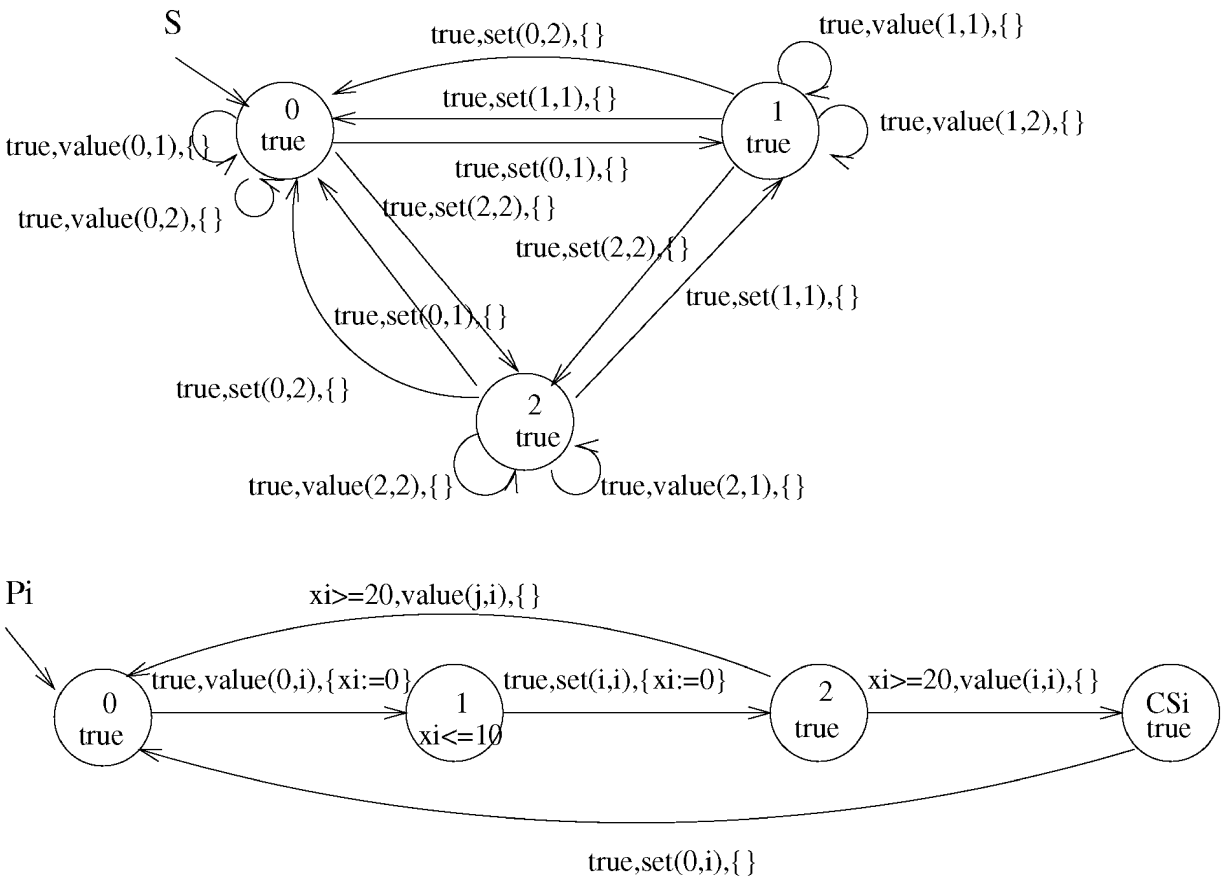
Analysis. We show that the system $MUTEX$ satisfies mutual exclusion using TREAT. The system violates the mutual exclusion property if and only if it includes a state in $M_{alg}(MUTEX)$ such that any two processes P_i and P_j are in the critical sections CS_i and CS_j simultaneously.

Experimental Results. Table 2 shows the experimental results. The results of TREAT for $MUTEX$ are as follows. The number of reachable states of the system with $a = 10, b = 20$ is 35, 825, 3,175 for $n = 2, 3, 4$, respectively, and the number of reachable states of the system with $a = 10, b = 10$ is 61, 1,091, 18,616 for $n = 2, 3, 4$, respectively, $MUTEX$ satisfies the mutual exclusion property for $a = 10, b = 20$, but $MUTEX$ violates the property for $a = 10, b = 10$. This property states that any two or more processes are in the critical sections are reachable in the system with $a = 10, b = 10$. From these experimental results, we recognize that the correctness of the mutual exclusion protocol depends on the values of timing parameters a and b. Table 2a shows the results for $a = 10$ and $b = 20$.

Compared to the results of HyTech's forward analysis, TREAT successfully performs the analysis up to four



(a)



(b)

Fig. 10. Timed automata for mutual exclusion protocol. (a) Process configuration. (b) Process description.

TABLE 2
Experimental Results: TREAT, HyTech, and Kronos for the Mutual Exclusion Protocol

	TREAT		HyTech (forward)		HyTech (backward)		Kronos
n	Time	State#	Time	State#	Time	State#	Time
2	0.1s	35	1.58s	27	1.9s	35	0.084s
3	28s	825	17.37s	307	14.72s	313	1.1s
4	10m1s	3,175	7m31s	4,769	3m27s	2,315	7.8s
5	$fail_{time}$		$fail_{time}$		3h19m41s	15,181	67s
6					$fail_{time}$		$fail_{tool}$

(a)

	TREAT		HyTech (forward)		HyTech (backward)		Kronos
n	Time	State#	Time	State#	Time	State#	Time
2	0.1s	61	2.35s	53	2.79s	74	0.12s
3	17.1s	1,091	1m40s	1,974	20.13s	464	0.88s
4	1h46m28s	18,616	$fail_{time}$		4m24s	2,849	10.9s
5	$fail_{time}$				2h57m33s	16,995	1m48s
6					$fail_{time}$		$fail_{tool}$

(b)

(a) Correct specification $a = 10$ *and* $b = 20$. *(b) Incorrect specification* $a = 10$ *and* $b = 10$.

processes while HyTech fails due to the time limitation for four processes when $a = 10$ and $b = 10$. For $a = 10$ and $b = 20$, HyTech generates less states than TREAT with two or three processes. On the other hand, HyTech generates 4,769 states and TREAT generates 3,174 states with four processes. For $a = 10$ and $b = 10$, HyTech generates less states than TREAT with two. However, HyTech explores 1,974 states while TREAT generates 1,091 with three processes.

For HyTech's backward analysis approach, it proves the property up to five processes. Let us consider the system with $n = 4$, $a = 10$ and $b = 10$. TREAT generates 18,616 states reachable from the initial state. And HyTech generates 2,849 states that can reach the error states in which the mutual exclusion violates.

Compared to the results of Kronos, Kronos gives the correctness result much faster than TREAT and HyTech and analyzes up to five processes because Kronos does not generate the state space but performs symbolic model checking.

5.3 Active Structure Control System

Elseaidy, Cleaveland, and Baugh [12] present an active structure control system ($ASCS$) which monitors the state of the system (e.g., accelerations and displacements), and provides the counter external excitation of the structure. The active structural control system contains three major components: a sensor, which monitors the state of the system; an actuator, which applies forces to the structure; and a control process, which feeds the data provided by the sensor to a control algorithm that calculates the appropriate forces which the actuator device should provide to counter external excitation of the structure. The result in [25] shows that the system provides satisfactory performance if the time between pulses is bounded by $T_0/8$ and $T_0/2$ for the natural period $T_0 = 29$msec of a structure. Thus the correctness of the system is given by: the time between successive pulse applications must be in $[37, 145]$.

Model. The system $ASCS$ consists of three components: Sensor, Actuator, and Controller, which are described in Fig. 11. Events tau_s, tau_a, tau_c represent internal actions of Sensor, Actuator, and Controller, respectively. There are two channels $sensor\_controller$ and $controller\_actuator$. Sensor sends a message to Controller via channel $sensor\_controller$, Controller sends a message to Actuator via $controller\_actuator$.

Sensor has a clock $x1$ to represent timing constraints. Sensor collects data for 50 to 55 time units at node 1. It sends the data to Controller. To send the data, Sensor first prepares communication for 10 time units at node 2, waits for synchronization with Controller using action $sensor\_controller$ at node 3, and sends the data to Controller for five time units at node 4.

Actuator has a clock $x4$ to represent timing constraints. Actuator receives from Controller a message via channel

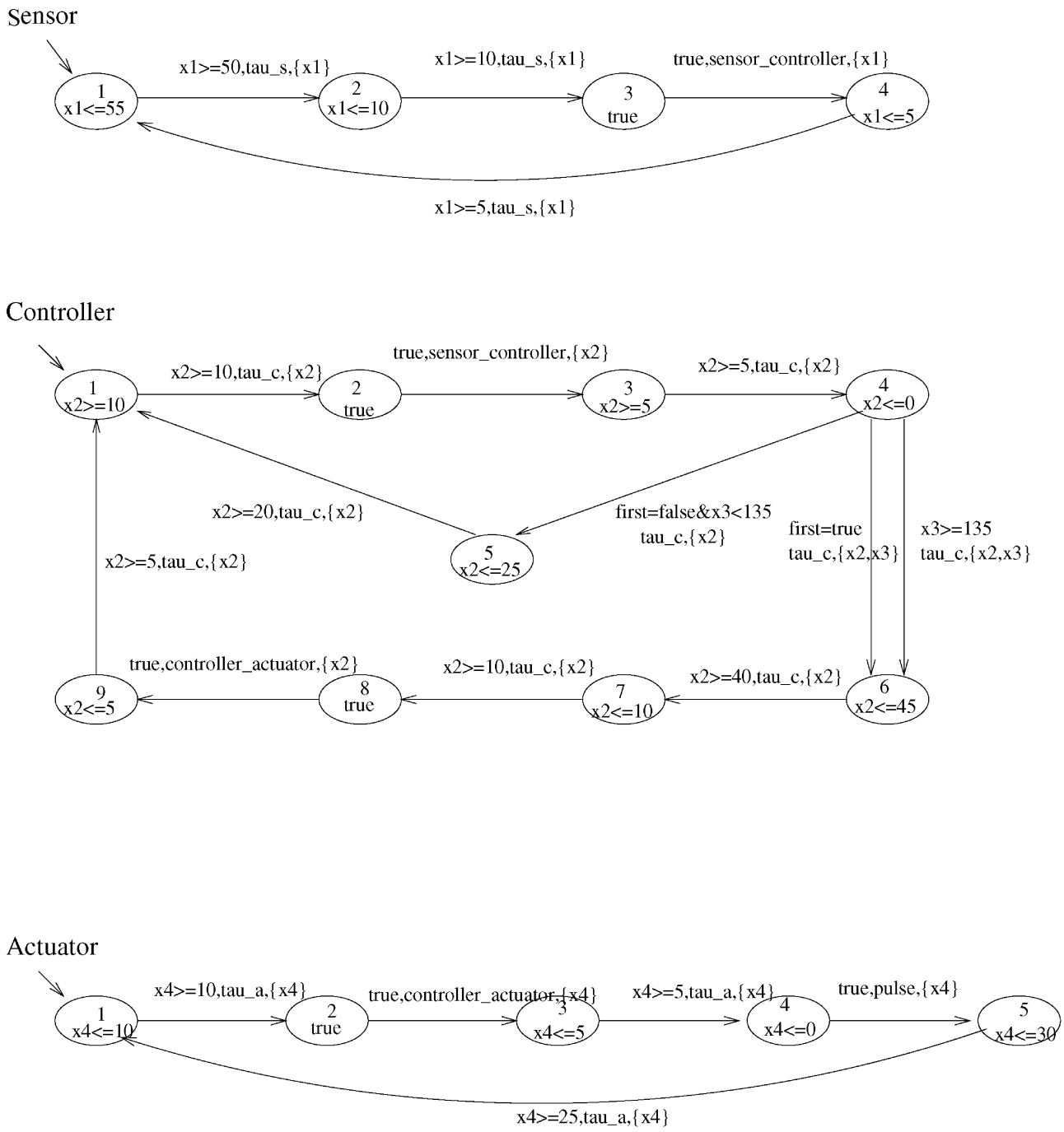


Fig. 11. Timed automata for active structure control system.

controller_actuator. To receive the message, it sets up communication for 10 time units at node 1, waits for synchronization with Controller through action *controller_actuator* at node 2, and receives the message for five time units at node 3. After getting a message from Controller, Actuator generates a *pulse* event for observation of the start of pulse application at node 4. It then applies forces to the structure for 25 to 30 time units at node 5.

Controller has two clocks $x2$ and $x3$. Clock $x2$ is used to represent the total time elapsed in each node, and clock $x3$ is used to hold the total time elapsed since the previous pulse application. Controller also has a Boolean variable *first* which is true at the first iteration and then becomes false. Controller repeatly gets data from Sensor, and if enough time has elapsed since the previous pulse application, it calculates the appropriate pulse magnitude and sends it to Actuator. In detail, to get the data from Sensor, Controller first prepares communication for 10 time units at node 1, waits for synchronization with Sensor using action *sensor_controller* at node 2, and receives the data for five time units at node 3. After the communication, it moves to node 4. At node 4, we have three cases: 1) if there is no

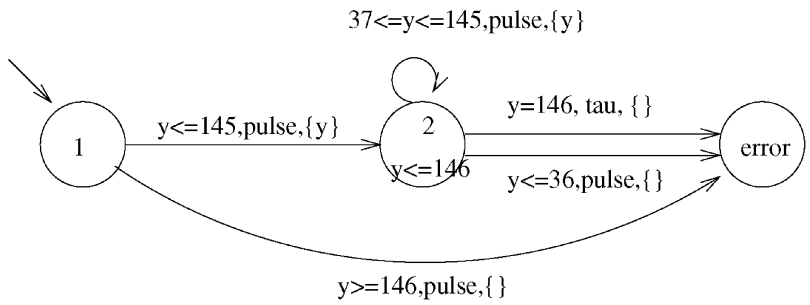


Fig. 12. Timed automata for correctness property (Active Structure Control System).

previous pulse application (i.e. $first$ is true), then Controller moves to node 6; 2) if enough time has elapsed (i.e. $x3 \geq 135$), then Controller also moves to node 6; and 3) otherwise it moves to node 5. At node 5, Controller waits for 20 to 25 time units and then returns to the initial node 1. At node 6, Controller calculates the pulse magnitude for 40 to 45 time units. To send it to Actuator, Controller prepares communication for 10 time units at node 7, waits for synchronization with Sensor using action $controller\_actuator$ at node 8, sends it for five time units to Sensor at node 9, and returns to the initial node 1.

Analysis. One of the correctness properties for the system is a bounded response time property that the time between successive pulse applications must be in the range $[37, 145]$. To show the timing requirement, we have a monitoring process Mon which goes to the *error* state whenever the time between successive pulses is less than 37 or greater than 145, as shown in Fig. 12.

Experimental Results. TREAT constructs the composed CTSM process $ASCS||Mon$. The composed process has 360 nodes and 767 transitions. The composed process has five clocks, $x1$, $x2$, $x3$, $x4$, and y.

Table 3 shows the experimental results for the system. The first row shows the result for the correctness that the time between successive pulse applications is in the range $[37, 145]$. TREAT outputs the labeled transition system with 133 states, which takes 3.1 seconds. In the system, there is no *error* state. Therefore, we conclude that the system satisfies the timing requirement. As a comparison, Elseaidy, Cleaveland, and Baugh report that the reachability graph has 3,174 states [12]. It takes 171 seconds on Sparc 2. Thus, TREAT reduces the state space by 1/14 because TREAT minimizes the time state space.

Compared to HyTech's forward analysis, it explores smaller number of states than TREAT but takes more time than TREAT. HyTech's backward analysis fails due to memory overflow. For Kronos, it takes 4.8 hours. This shows that symbolic model checking is not always faster than the state space exploration approach.

The second row gives the result for the correctness with the time range $[37, 125]$. TREAT outputs the labeled transition system with 142 states, which takes 3.3 seconds. The labeled transition system includes *error* states, and thus, the system violates the timing requirement for range $[37, 125]$.

5.4 Summary

We summarize the experimental results as follows.

TREAT gave better performance than HyTech's forward analysis except for the active structure control system. TREAT generated smaller state space than HyTech and performed faster than HyTech for the railroad crossing control system, the Fischer's mutual exclusion protocol. Moreover, TREAT successfully analyzed the railroad crossing system for two and three tracks and the Fischer's mutual exclusion protocol for $n = 4, a = 10, b = 10$ while HyTech failed.

For HyTech's backward analysis, it gave better performance than TREAT for the railroad crossing control system and the Fischer's mutual exclusion protocol. However, it failed to analyze the active structure control system even though TREAT analyzed it within several seconds.

Kronos does not generate the state space but performs symbolic model checking. Thus, it showed better performance than TREAT and HyTech for most of the experiments. To our surprise, the results of the active

TABLE 3
Comparison: TREAT, HyTech, and Kronos for Active Structural Control System

	TREAT		HyTech (forward)		HyTech (backward)		Kronos
	Time	State#	Time	State#	Time	State#	Time
Correct	3.1s	133	18.30s	87	$fail_{mem}$		4h45m46s
Incorrect	3.3s	142	15.44s	109	$fail_{mem}$		4h7m58s

structure control system disproves the popular belief that symbolic model checking always gives better performance than state space exploration approach. Kronos takes 4.8 hours for the analysis while TREAT takes just 3.1 seconds. Through these experiments we observed that for Kronos it is hard to debug when the descriptions of given systems include errors because Kronos does not generate the explored states or erroneous traces. For the railroad crossing control system with six tracks, Kronos fails to compose the system due to the tool limitation. Kronos has strong constraints on the input descriptions that the number of nodes, the number of transitions and the largest constant are required to be less than 2^8 because they are represents using the C type "short" (2 bytes).

One of the lessons from these experiments is that there is no analysis approach that always performs better than others and thus it would be advantageous to incorporate various analysis approaches into TREAT.

6 Related Work

We briefly overview other work on reachability analysis for real-time systems. In real-time systems, a state can be unreachable due to timing constraints. Although timing constraints have different expressions in different models, the property that time increases uniformly and unboundedly is the same. The domain of time is either discrete or dense. Many real-time models [27], [24], [17] follow the discrete time semantics since it is easier to handle and analyze. For real-time systems with dense time, little work has been done on timed reachability analysis. The most successful method for dense time is proposed by Alur et al. [2].

In Communicating Real-time State Machines (CRSMs) [27], a system consists of a set of CRSMs connected with one-to-one communication channels. CRSMs use the set of integers to represent time. Each CRSM has a finite set of data variables, control locations and transitions. Transitions consist of an enabling condition, an action, a transformation function, and lower and upper time bounds. For a transition with time bound $[l, u]$, the system can execute the action of the transition at least l time units and at most u time units after the transition is enabled. The behaviors of the global system are time-stamped traces of actions. Raju [26] gives a method to generate a reachability graph representing the behaviors. In the reachability graph, a node consists of the current location of each CRSM, the variable valuation, and the time spent by each CRSM in its current location. An edge is labeled with a set of actions executed and the time gap between nodes. The domain of each variable is restricted to be finite, and thus the number of possible variable valuations are finite. The time spent by each CRSM labeling a node can be distinguished using $(c+1)$ different values, where c is the largest value of upper bounds of transitions. Since time is given by the set of integers, the reachability graph is always finite. This approach is based on discrete time, so each state can be represented as time spent by each CRSM in its current location. But, in dense time semantics, if time spent in its current location is given in a state, infinitely many states exist. On the other hand, in our approach, each minimized state is represented as relative time intervals between clocks, so finitely many states exist in dense time semantics.

Timed Transition Models (TTMs) [24] also uses discrete time. Time is modeled using an external and conceptual global clock which ticks infinitely often. A system has a set of TTMs, each of which consists of locations and transitions. In transitions, there are enabling conditions, transformation functions and lower and upper time bounds, similarly to CRSM. In the reachability graph, a node consists of the possible transitions as well as the current variable valuations. The possible transitions are decorated with the current time bound. Each edge represents either a TTM's transition or a tick transition. The tick transition represents a unit of time passage. With a tick transition, the current time bounds decorated in transitions are decreased by one down to zero. As long as a TTM has a finite number of valuations, the reachability graph is finite. We note that edges represent at most one time unit passage in TTM, whereas edges represent several time units in CRSM.

Modechart [16] is a graphical specification language for real-time systems which allows a user to describe a system in a hierarchical and modular way. A Modechart specification consists of modes that can be running in parallel or sequentially. There are three kinds of modes: primitive modes, serial modes, and parallel modes. A primitive mode contains an action with lower and upper time bounds. In Modechart, events is an instantaneous change such as the start of an action, the end of an action, the start of a mode, the end of a mode, etc. A serial mode has several modes connected by transitions. Transitions are labeled with either events or lower and upper time bounds. A parallel mode includes a set of modes running simultaneously. The reachability analysis for a Modechart is described in [17]. In the reachability graph, each node represents an event occurrence, and each edge represents the causality that the target node can happen as the result of the event of the source node and time passage. The timing relation between nodes is computed using the time bounds in actions and transitions, and is used to compute the reachability of nodes. The resulting graph is finite because there exist finitely many distinguishable nodes and timing relations. The condition *distance equivalence relation(deq)* used to distinguish nodes is computed using weighted graph like the condition $bisim\_cond$ in Definition 4.2. The difference is that while *deq* compares the distances between a node and its parent, $bisim\_cond$ compares the distances between a

nodes and its all predecessors that reset clocks because clocks can be reset anytime and can be compared in any enabling conditions.

Timed automata introduced in this paper has dense time semantics unlike CRSM, TTM, and Modechart. Because there can be an arbitrary number of clock variables and transitions can reset any subset of clock variables, time dependent behaviors of a real-time system are more expressive. Alur et al. [2] propose region graphs as reachability graphs. A region consists of a location and a set of clock valuations that are equivalent. Two valuations are equivalent if the integral parts of each clock are the same and the orderings of fractional parts of all the clocks are the same. If the valuations are equivalent, then they have the same reachability. The region graph has size exponential in the number of clocks and the size of the constants that appear in the enabling conditions of the transitions [2]. Because the region graph is too fine-grained, more valuations that have the same reachability are equated in [4], [29]. Their algorithms, called minimization, partitions the whole state space until all regions in the partition include bisimilar states. Comparing to our approach, a node includes states that have the same enabled immediate transitions in the minimal region graph approaches. On the other hand, our approach is minimized based on traces, so a node is a collection of states that have the same transitions enabled immediately or in the future.

One approach to generate the reachable state space for timed automata is forward and backward analysis [14], [5]. If the reachability problem is given as: "Can the system reach from a state in region R_i to a state in region R_f?," then the problem is solved using fixed-point methods: forward and backward fixed-point computations [21]. The forward (backward) fixed-point procedure starts with the set $Q = R_i$ ($Q = R_f$) and repeatly adds states to (from) which any state in Q can reach (be reached). The procedure terminates if at some stage $Q \cap R_f$ ($Q \cap R_i$) is not empty or no new states can be added. The procedure may not terminate at all. On the other hand, our approach always terminates. This verification method is implemented in Kronos [10], [11] and Hytech [15]. In this approach, regions are constructed independently from properties that are verified.

To reduce the state space for timed automata, Yi et al. [30], [21] present a symbolic technique that partitions the set of clock valuations according to the particular property to be verified. More clock valuations can be equated because the approach partitions the set of valuations only if they affect the satisfiability of the given property differently. This approach is implemented in Uppaal [7]. In [28], Sokolsky and Smolka also present a symbolic technique that explores the portion necessary to determine the truthhood the given property. If a real-time system is represented as the composition of a collection of timed automata, then the global automaton is constructed on-the-fly.

7 Conclusion

We have presented an algorithm to cope with the state explosion problem in generating the state space of a timed automaton. Our algorithm clusters a set of states that are equivalent under the notions of history equivalence and transition bisimulation. To show the usefulness of the reachability graph, we have presented our experimental results for the railroad crossing example and the mutual exclusion protocol.

Although the reachability graph presented in this paper is similar to the computation graph of Modechart [17], there are several differences: 1) The underlying time domain of the computation graph is discrete in Modechart; 2) Timing constraints in a Modechart specification are much simpler than those in a timed automaton.

We have developed a data space minimization algorithm with respect to bisimulation for states with arbitrary data variables [19]. We plan to integrate the data space minimization algorithm and the reachability graph construction algorithm presented in this paper. At the same time, we will optimize our implementation to give better results.

The work is part of our research in developing effective tools based on state space exploration [9]. We are also currently investigating other properties such as time bounds between events that can be checked directly from the reachability graph generated by our algorithm.

Appendix

Proofs of Theorems and Lemmas

Lemma 4.1. L1. *If h is valid, then for an edge (v_1, v_2) in $W(h)$, $t_1 + w(v_1, v_2) \leq t_2$ where t_1 and t_2 are execution times of transitions a_1 and a_2 corresponding to v_1 and v_2, respectively.*

Proof. If a_1 precedes a_2 in h, then $w(v_1, v_2) \geq 0$, and for some clock x, x is reset on transition a_1 and $x \geq w(v_1, v_2)$ is in $cond(a_2)$ by the definition of $W(h)$. The value of x is $(t_2 - t_1)$ at t_2. At t_2, $cond(a_2)$ should be true to execute a_2. Thus, $t_2 - t_1 \geq w(v_1, v_2)$. That is, $t_1 + w(v_1, v_2) \leq t_2$.

Similarly, if a_2 precedes a_1 in h, then $w(v_1, v_2) \leq 0$, and for some clock x, x is reset on transition a_2 and $x \leq -w(v_1, v_2)$ is in $cond(a_1)$. The value of x is $(t_1 - t_2)$ at t_1. At t_2, $cond(a_1)$ should be true to execute a_1. Thus, $t_1 - t_2 \leq -w(v_1, v_2)$. That is, $t_1 + w(v_1, v_2) \leq t_2$. □

Theorem 4.1 *For a history h, h is valid if and only if $W(h)$ has no positive cycle.*

Proof. We first show that if h is valid, then $W(h)$ has no positive cycle, and then show that if $W(h)$ has no positive cycle, then h is valid.

1. If h is valid, then $W(h)$ has no positive cycle. Suppose that h is valid and $W(h)$ has a positive cycle.

Let $v_1v_2v_3 \dots v_k (= v_1)$ be a positive cycle in the weighted graph $W(h)$. Then, the total weight of the cycle is greater than zero, i.e.,

$$\Sigma_{i=1}^{k-1} w(v_i, v_{i+1}) \; 0. \tag{R1}$$

Since h is valid, there exists an execution $exec$ corresponding to h. For $exec$, let t_i be the execution time of the transition corresponding to v_i for $0 \le i \le k$. Then, by **L1**,

$$\begin{aligned} t_1 + w(v_1, v_2) &\le t_2, t_2 + w(v_2, v_3) \\ &\le t_3, \cdots, t_{k-1} + w(v_{k-1}, v_k) \le t_k. \end{aligned}$$

That is,

$$t_1 + \Sigma_{i=1}^{k-1} w(v_i, v_{i+1}) \le t_k.$$

Since $v_1 = v_k$, i.e., $t_1 = t_k$,

$$\begin{aligned} t_1 + \Sigma_{i=1}^{k-1} w(v_i, v_{i+1}) &\le t_1 \\ \Sigma_{i=1}^{k-1} w(v_i, v_{i+1}) &\le (t_1 - t_1). \\ \Sigma_{i=1}^{k-1} w(v_i, v_{i+1}) &\le 0. \end{aligned} \tag{R2}$$

Then, **R2** contradicts to **R1**. Therefore, $W(h)$ has no positive cycle.

2. If $W(h)$ has no positive cycle, then h is valid.

Let $h = \langle a_0, a_1, \dots, a_k \rangle$, and let v_i be the corresponding vertex of a_i. We define $m(v_i, v_j)$ as the maximum weight among all path weights from v_i to v_j for $0 \le i, j \le k$. Let $t_0 = 0$ and $t_i = m(v_0, v_i)$ for $1 \le i \le k$.

If $x \ge c$ is in $cond(a_j)$ and

$$reset(x, \langle a_0, \dots, a_{j-1} \rangle) = a_i,$$

then $w(v_i, v_j) \ge c$ by the definition of w, and thus $m(v_i, v_j) \ge c$ by the definition of m. And also,

$$\begin{aligned} m(v_0, v_j) &\ge m(v_0, v_i) + m(v_i, v_j) \\ m(v_0, v_j) - m(v_0, v_i) &\ge m(v_i, v_j) \\ t_j - t_i &\ge m(v_i, v_j) \\ t_j - t_i &\ge c. \end{aligned}$$

If $x \le c$ is in $cond(a_j)$ and

$$reset(x, \langle a_0, \dots, a_{j-1} \rangle) = a_i,$$

then $w(v_j, v_i) \ge -c$, and thus $m(v_j, v_i) \ge -c$. If

$$m(v_0, v_j) - m(v_0, v_i) > -m(v_j, v_i),$$

i.e., $m(v_0, v_j) + m(v_j, v_i) > m(v_0, v_i)$, then the weight of $v_0 \dots v_j v_i$ is the greater than $m(v_0, v_i)$. It contradicts the definition of m. Thus,

$$m(v_0, v_j) - m(v_0, v_i) \le -m(v_j, v_i).$$

Since $m(v_j, v_i) \ge c$, $m(v_0, v_j) - m(v_0, v_i) \le c$ and, thus, $t_j - t_i \le c$. Therefore, there exists an execution,

$$s_0 \xrightarrow{a_1, t_1} s_1 \xrightarrow{a_2, t_2} s_2 \cdots \xrightarrow{a_n, t_n} s_n,$$

for some $s_0, s_1, \dots, s_n$. That is, h is valid. □

Lemma 4.1. L2. *Let $s_1 = (n, h_1)$ and $s_2 = (n, h_2)$. Let f be an enabling condition. If $bisim\_cond(s_1, s_2)$ is true, then f is satisfiable at s_1 iff f is satisfiable at s_2.*

Proof. It is proved by the induction on the length of f.

Base case. $f = true|false|x \le c|x \ge c$

Case $f = true$ **or** $false$: trivial

Case $f = x \le c$:

For $i = 1, 2$, f is satisfiable at s_i only if

$$min\_dist(reset(x, h_i), last(h_i), h_i) \le c.$$

If $min\_dist(reset(x, h_1), last(h_1), h_1) \le Max$, then by Definition 4.2,

$$min\_dist(reset(x, h_1), last(h_1), h_1) = min\_dist(reset(x, h_2), last(h_2), h_2),$$

i.e., $min\_dist(reset(x, h_1), last(h_1), h_1) \le c$ iff

$$min\_dist(reset(x, h_2), last(h_2), h_2) \le c.$$

Thus, f is satisfiable at s_1 iff it is satisfiable at s_2.

Case $f = x \ge c$:

For $i = 1, 2$, f is satisfiable at s_i only if

$$max\_dist(reset(x, h_i), last(h_i), h_i) \ge c.$$

If $max\_dist(reset(x, h_1), last(h_1), h_1) \le Max$, then by Definition 4.2,

$$max\_dist(reset(x, h_1), last(h_1), h_1) = max\_dist(reset(x, h_2), last(h_2), h_2),$$

i.e., $max\_dist(reset(x, h_1), last(h_1), h_1) \ge c$ iff

$$max\_dist(reset(x, h_2), last(h_2), h_2) \ge c.$$

Thus, f is satisfiable at s_1 iff it is satisfiable at s_2.

Induction Step. Suppose that f is satisfiable at s_1 and s_2. Then for $f' = f \wedge x \le c | f \wedge x \ge c$, we show that f' is satisfiable at s_1 iff f' is satisfiable at s_2.

We prove it using weighted graphs. For $i = 1, 2$, let W_i be the weighted graph $W(h_i \hat{} \langle a \rangle)$ for $cond(a) = f$. Since f is satisfiable at s_1 and s_2, there is no positive cycle in W_1 and W_2 by Theorem 4.1. Let vl_i be the vertex for a, vx_i be the vertex for $reset(x, h_i)$ and vy_i be the vertex for $reset(y, h_i)$ in W_i. Let $w_i(v, u)$ be the maximum weight among all path weights from v to u for vertices v and u in W_i.

Case $f' = f \wedge x \le c$:

Let W_i' be the weighted graph which is the same as W_i except for adding an edge from vl_i to vx_i with weight "$-c$." We show that W_1' has no positive cycle iff W_2' has no positive. Then f' is satisfiable at s_1 iff it is satisfiable at s_2.

Case $w_1(vx_1, vl_1) > Max$:

Then, $w_2(vx_2, vl_2) > Max$ by Definition 4.2. In W_1', there is a positive cycle vx_1, vl_1, vx_1 with weight $w_1(vx_1, vl_1) + (-c)$ because $w_1(vx_1, vl_1) > Max \ge c$.

Similarly, in W_2', there is a positive cycle vx_2, vl_2, vx_2 with weight $w_2(vx_2, vl_2) + (-c)$.

Case $w_1(vx_1, vl_1) \leq Max$:

Then, $w_1(vx_1, vl_1) = w_2(vx_2, vl_2)$ by Definition 4.2.

Case $w_1(vx_1, vl_1) > c$:

There exist positive cycles vx_i, vl_i, vx_i with weight $w_i(vx_i, vl_i) + (-c)$ in W_i for $i = 1, 2$.

Case $|w_1(vl_1, vx_1)| \leq c$:

Since $w_1(vl_1, vx_1) \geq -c$, the maximum weight from vl_1 to vx_1 in W_1' is the same as $w_1(vl_1, vx_1)$.

Thus, there is no positive cycle in W_1'. Similarly, there is no positive cycle in W_2'.

Case $w_1(vx_1, vl_1) \leq c < |w_1(vl_1, vx_1)|$:

Then the maximum weight from vl_1 to vx_1 in W_1' is $-c$, not $w_1(vl_1, vx_1)$. Suppose that there exists a positive cycle $vx_1, u_1, u_2, \ldots, u_n, vl_1, vx_1$ in W_1'.

That is,

$$w_1(vx_1, u_1) + w_1(u_1, u_2) + \ldots + w_1(u_n, vl_1) + (-c) > 0.$$

By the definition of w_1,

$$w_1(vx_1, vl_l) \geq w_1(vx_1, u_1) + w_1(u_1, u_2) + \ldots + w_1(u_n, vl_1).$$

Thus, $w_1(vx_1, vl_l) + (-c) > 0$, which contradicts the assumption $w_1(vx_1, vl_1) \leq c$.

Therefore, there is no positive cycle in W_1'. Similarly, in W_2'.

Case $f' = f \wedge x \geq c$:

Let W_i' be the weighted graph which is the same as W_i except for adding an edge from vx_i to vl_i with weight "c." We show that W_1' has no positive cycle iff W_2' has no positive. Then $'$ is satisfiable at s_1 iff it is satisfiable at s_2.

case $w_1(vx_1, vl_1) > Max$:

Then the maximum weight from vx_1 to vl_1 in W_1' is the same as $w_1(vx_1, vl_1)$.

Thus, there is no positive cycle in W_1'. Similarly, there is no positive cycle in W_2'.

case $w_1(vx_1, vl_1) \leq Max$:

By Definition 4.2,

$$w_1(vx_1, vl_1) = w_2(vx_2, vl_2).$$

Case $w_1(vx_1, vl_1) \geq c$:

Then the maximum weight from vx_1 to vl_1 in W_1' is the same as $w_1(vx_1, vl_1)$.

Thus, there is no positive cycle in W_1'. Similarly, there is no positive cycle in W_2'.

Case $|w_1(vl_1, vx_1)| < c$:

There exist a positive cycle vx_1, vl_1, vx_1 with weight $c + w_1(vl_1, vx_1)$ in W_1'. Similarly, in W_2'.

Case $w_1(vx_1, vl_1) < c \leq |w_1(vl_1, vx_1)|$:

Then the maximum weight from vx_1 to vl_1 in W_1' is c, not $w_1(vx_1, vl_1)$. Suppose that there exists a positive cycle $vx_1, vl_1, u_1, u_2, \ldots, u_n, vx_1$ in W_1'.

That is,

$$c + w_1(vl_1, u_1) + w_1(u_1, u_2) + \ldots + w_1(u_n, vx_1) > 0.$$

By the definition of w_1,

$$w_1(vl_1, vx_1) \geq w_1(vl_1, u_1) + w_1(u_1, u_2) + \ldots + w_1(u_n, vx_1).$$

Thus, $c + w_1(vl_1, vx_1) > 0$, which contradicts the assumption $c \leq |w_1(vl_1, vx_1)|$.

Therefore, there is no positive cycle in W_1'.

Similarly, in W_2'.

Therefore, if $bisim\_cond(s_1, s_2)$ is true, then f is satisfiable at s_1 iff f is satisfiable at s_2. □

Lemma 4.1. L3. *Let $s_1 = (n, h_1)$ and $s_2 = (n, h_2)$. Let f be an enabling condition. Suppose node n has a transition a to n' with f. Let*

$$s_1' = (n', h_1 \hat{} a), s_2' = (n', h_2 \hat{} a).$$

If $bisim\_cond(s_1, s_2)$ is true and f is satisfiable at s_1 and s_2 then $bisim\_cond(s_1', s_2')$ is true.

Proof. We prove it using the proof of **L2**. We show that whenever weighted graphs are changed in **L2**, we show that the maximum weights are preserved. Let $w_i'(v, u)$ be the maximum weight among all path weights from v to u for vertices v and u in W_i'.

Case $f' = f \wedge x \leq c$:

As shown in the proof of **L2**, if $w_1(vx_1, vl_1) > c$ then f' is false, and if $|w_1(vl_1, vx_1)| \leq c$, then maximum weights in W_i' is the same as W_i. Thus we just consider the case of T/P

$$w_1(vx_1, vl_1) \leq c < |w_1(vl_1, vx_1)|.$$

Then for any clock y,

$$w_1'(vy_1, vx_1) = max\{w_1(vy_1, vx_1), w_1(vy_1, vl_1) - c\},$$
$$w_2'(vy_2, vx_2) = max\{w_2(vy_2, vx_2), w_2(vy_2, vl_2) - c\} \text{ and}$$
$$w_1'(vl_1, vy_1) = max\{w_1(vl_1, vy_1), -c + w_1(vx_1, vy_1)\},$$
$$w_2'(vl_2, vy_2) = max\{w_2(vl_2, vy_2), -c + w_2(vx_2, vy_2)\}.$$

Case $w_1(vy_1, vl_1) \leq w_1(vx_1, vl_1)$: (i.e., $x \prec y$)

If $|w_1(vy_1, vx_1)| > Max$ then $|w_2(vy_2, vx_2)| > Max$. Since $|w_1(vy_1, vl_1) - c| \leq Max$.

$$\begin{aligned} w_1'(vy_1, vx_1) &= w_1(vy_1, vl_1) - c \\ &= w_2(vy_2, vl_2) - c \\ &= w_2'(vy_2, vx_2). \end{aligned}$$

Otherwise, $w_1'(vy_1, vx_1) = w_2'(vy_2, vx_2)$ because

$$w_1(vy_1, vx_1) = w_2(vy_2, vx_2).$$

If $|w_1(vl_1, vy_1)| > Max$ then $|w_2(vl_2, vy_2)| > Max$. Since $|-c + w_1(vx_1, vy_1)| \leq Max$,

$$\begin{aligned} w_1'(vl_1, vy_1) &= -c + w_1(vx_1, vy_1) \\ &= -c + w_2(vx_2, vy_2) \\ &= w_2'(vl_2, vy_2). \end{aligned}$$

Otherwise, $w_1'(vl_1, vy_1) = w_2'(vl_2, vy_2)$ because

$$w_1(vl_1, vy_1) = w_2(vl_2, vy_2).$$

Case $w_1(vy_1, vl_1) > w_1(vx_1, vl_1)$: (i.e., $y \prec x$)

If $|w_1(vl_1, vy_1)| \leq Max$, then

$$w_1'(vl_1, vy_1) = w_2'(vl_2, vy_2)$$

since $w_1(vl_1, vy_1) = w_2(vl_2, vy_2)$ and

$$w_1(vx_1, vy_1) = w_2(vx_2, vy_2).$$

If $|w_1(vx_1, vy_1)| > Max$, then

$$|w'_1(vl_1, vy_1)|, |w'_2(vl_2, vy_2)| > Max.$$

If $|w_1(vx_1, vy_1) - c| > Max$ then

$$w'_1(vl_1, vy_1) = w'_2(vl_2, vy_2)$$

since $|w'_1(vl_1, vy_1)|$, $|w'_2(vl_2, vy_2)| > Max$.

If $|w_1(vl_1, vy_1)| > Max$ and $|w_1(vx_1, vy_1) - c| \leq Max$ then $w'_1(vl_1, vy_1) = w'_2(vl_2, vy_2) = w_1(vx_1, vy_1) - c$ since $w_1(vx_1, vy_1) = w_2(vx_2, vy_2)$.

If $w_1(vy_1, vl_1) \leq Max$, then $w'_1(vy_1, vx_1) = w'_2(vy_2, vx_2)$ since $w_1(vy_1, vx_1) = w_2(vy_2, vx_2)$ and

$$w_1(vy_1, vl_1) = w_2(vy_2, vl_2).$$

If $w_1(vy_1, vx_1) > Max$, then

$$w'_1(vy_1, vx_1), w'_2(vy_2, vx_2) > Max.$$

If $w_1(vy_1, vx_1) \leq Max$, then $w_1(vy_1, vl_1) = w_2(vy_2, vl_2)$ since

$$w_1(vy_1, vl_1) = w_1(vx_1, vl_1) + w_1(vy_1, vx_1),$$

$$w_1(vx_1, vl_1) = w_2(vx_2, vl_2),$$

$$w_1(vy_1, vx_1) = w_2(vy_2, vx_2).$$

So, $w'_1(vy_1, vx_1) = w'_2(vy_2, vx_2)$.

Case $f' = f \wedge x \geq c$:

As shown in the proof of **L2**, if $w_1(vx_1, vl_1) \geq c$, then maximum weights in W'_1 is the same as maximum weights in W_1, and if $|w_1(vl_1, vx_1)| \leq c$, then f' is false. Thus we just consider the case of

$$w_1(vx_1, vl_1) \leq c < |w_1(vl_1, vx_1)|.$$

Then for any clock y,

$$w'_1(vx_1, vy_1) = max\{w_1(vx_1, vy_1), c + w_1(vl_1, vy_1)\},$$
$$w'_2(vx_2, vy_2) = max\{w_2(vx_2, vy_2), c + w_2(vl_2, vy_2)\} \text{ and}$$
$$w'_1(vy_1, vl_1) = max\{w_1(vy_1, vl_1), w_1(vy_1, vx_1) + c\},$$
$$w'_2(vy_2, vl_2) = max\{w_2(vy_2, vl_2), w_2(vy_2, vx_2) + c\}.$$

Case $w_1(vy_1, vl_1) \leq w_1(vx_1, vl_1)$: (i.e., $x \prec y$)

Since $w_1(vx_1, vl_1) \leq c$, $w_1(vy_1, vl_1) = w_2(vy_2, vl_2)$ and $w_1(vx_1, vy_1) = w_2(vx_2, vy_2)$.

If $|w_1(vl_1, vy_1)| \leq c$ then $w_1(vl_1, vy_1) = w_2(vl_2, vy_2)$. So,

$$w'_1(vx_1, vy_1) = max\{w_1(vx_1, vy_1), c + w_1(vl_1, vy_1)\} = max\{w_2(vx_2, vy_2), c + w_2(vl_2, vy_2)\} = w'_2(vx_2, vy_2).$$

Otherwise, $w_1(vl_1, vy_1) < -c$, and thus,

$$w_1(vl_1, vy_1) + c < 0.$$

Since $w_1(vx_1, vy_1) \geq 0$, $w'_1(vx_1, vy_1) = w_1(vx_1, vy_1)$.
Similarly, $w'_2(vx_2, vy_2) = w_2(vx_2, vy_2)$.
Thus, $w'_1(vx_1, vy_1) = w'_2(vx_2, vy_2)$.
If $|w_1(vy_1, vx_1)| \leq c$ then $w_1(vy_1, vx_1) = w_2(vy_2, vx_2)$.
So,

$$w'_1(vy_1, vl_1) = max\{w_1(vy_1, vl_1), w_1(vy_1, vx_1) + c\} = max\{w_2(vy_2, vl_2), w_2(vy_2, vx_2) + c\} = w'_2(vy_2, vl_2).$$

Otherwise, $w_1(vy_1, vx_1) < -c$, and thus,

$$w_1(vy_1, vx_1) + c < 0.$$

Since $w_1(vy_1, vl_1) \geq 0$, $w'_1(vy_1, vl_1) = w_1(vy_1, vl_1)$.
Similarly, $w'_2(vy_2, vl_2) = w_2(vy_2, vl_2)$.
Case $w_1(vy_1, vl_1) > w_1(vx_1, vl_1)$: (i.e., $y \prec x$)
If $|w_1(vl_1, vy_1)| \leq Max$ then $w_1(vl_1, vy_1) = w_2(vl_2, vy_2)$ and $w_1(vx_1, vy_1) = w_2(vx_2, vy_2)$.
So,

$$w'_1(vx_1, vy_1) = max\{w_1(vx_1, vy_1), c + w_1(vl_1, vy_1)\} = max\{w_2(vx_2, vy_2), c + w_2(vl_2, vy_2)\} = w'_2(vx_2, vy_2).$$

Otherwise, $w_1(vl_1, vy_1) = w_1(vl_1, vx_1) + w_1(vx_1, vy_1)$.
Then, $w_1(vl_1, vy_1) + c \geq w_1(vx_1, vy_1)$.
Thus, if $|w_1(vx_1, vy_1)| \leq Max$,

$$w'_1(vx_1, vy_1) = w_1(vx_1, vy_1) = w_2(vx_2, vy_2) = w'_2(vx_2, vy_2).$$

If $|w_1(vx_1, vy_1)| > Max$, $|w'_1(vx_1, vy_1)| > Max$.
Similarly, $|w'_2(vx_2, vy_2)| > Max$.
If $w_1(vy_1, vl_1) > Max$ then $w'_1(vy_1, vl_1) > Max$.
Similarly, $w'_2(vy_2, vl_2) > Max$.
Otherwise, $w_1(vy_1, vl_1) \leq Max$ and, thus,

$$w_1(vy_1, vl_1) = w_2(vy_2, vl_2),$$
$$w_1(vy_1, vx_1) = w_2(vy_2, vx_2).$$

Thus,

$$w'_1(vy_1, vl_1) = max\{w_1(vy_1, vl_1), w_1(vy_1, vx_1) + c\} = max\{w_2(vy_2, vl_2), w_2(vy_2, vx_2) + c\} = w'_2(vy_2, vl_2).$$

Therefore, if $bisim\_cond(s_1, s_2)$ is true and f is satisfiable at s_1 and s_2 then $bisim\_cond(s'_1, s'_2)$ is true. □

Lemma 4.1. *If $bisim\_cond(s_1, s_2)$ is true, then s_1 and s_2 are transition-bisimilar.*

Proof. Let $s_1 = (n, h_1)$ and $s_2 = (n, h_2)$. Let f be an enabling condition. Suppose $bisim\_cond(s_1, s_2)$ is true. Then f is satisfiable at s_1 iff f is satisfiable at s_2 by **L2**. Suppose node n has a transition a to n' with f. Since f is satisfiable at states s_1 and s_2, the states have outgoing transition labeled with a. Let $s'_1 = (n', h_1 \hat{} a), s'_2 = (n', h_2 \hat{} a)$. Then $bisim\_cond(s'_1, s'_2)$ is true by **L3**. By the definition of transition bisimulation, if $bisim\_cond(s_1, s_2)$ is true, then s_1 and s_2 are transition bisimilar. □

Theorem 4.2. *For a timed automaton A, a relation $\{(s_1, s_2) \mid bisim\_cond(s_1, s_2)\}$ has finitely many equivalence classes.*

Proof. Let X and L be the set of clocks and the set of locations in M, resectively. And, let Max_c be the largest among the constants appearing in the conditions of M.

Let X be $\{x_1, \ldots, x_{|X|}\}$. Let $k = |X| + 1$. Let x_k be an extra clock that is reset on the last transition in a history. That is, $reset(x_k, h) = last(h)$.

Based on Lemma 4.1, for each state (l, h), the corresponding equivalence class can be represented by a pair (l, α):

$$\alpha = \{\alpha_{ij} | 1 \leq i, j \leq k\},$$

where if $reset(x_i, h)$ precedes $reset(x_j, h)$, then

- $\alpha_{ij} = min\_dist(reset(x_i, h), reset(x_j, h), h)$; and
- $\alpha_{ji} = max\_dist(reset(x_i, h), reset(x_j, h), h)$;

otherwise,

- $\alpha_{ij} = max\_dist(reset(x_j, h), reset(x_i, h), h)$; and
- $\alpha_{ji} = min\_dist(reset(x_j, h), reset(x_i, h), h)$.

Here, $\alpha_{ii} = 0$.

Let Max_c^+ be an value larger than Max_c.

For $1 \leq i < j \leq k$, if $reset(x_i, h)$ precedes $reset(x_j, h)$, then α_{ij} has at most $(Max_c + 2)$ distinguishable values:

$$0, 1, \ldots, Max_c, Max_c^+$$

since any values greater than Max_c cannot be distinguishable by cond3 and cond4 in Lemma 4.1.

Suppose that α_{ij} equals to n for $0 \leq n \leq Max_c$. Then α_{ji} can have at most $(Max_c + 2 - n)$ different values:

$$n, n+1, \ldots, Max_c, Max_c^+$$

since $\alpha_{ij} \leq \alpha_{ji}$. If α_{ij} equals to Max_c^+, then α_{ji} is also Max_c^+.

Therefore, the total number of different values of $(\alpha_{ij}, \alpha_{ji})$ is at most

$$\sum_{n=0}^{Max_c} (Max_c + 2 - n) + 1 = (Max_c + 2)(Max_c + 3)/2.$$

Similarly, if $reset(x_j, h)$ precedes $reset(x_i, h)$, then the total number of different values of $(\alpha_{ij}, \alpha_{ji})$ is also at most $(Max_c + 2)(Max_c + 3)/2$.

For $1 \leq i < j \leq k$, the total number of different values of $(\alpha_{ij}, \alpha_{ji})$ is at most

$$(Max_c + 2)(Max_c + 3)/2 + (Max_c + 2)(Max_c + 3)/2 = (Max_c + 2)(Max_c + 3).$$

There exists $k(k-1)/2$ pairs of $(\alpha_{ij}, \alpha_{ji})$ in α such as $(\alpha_{12}, \alpha_{21})$, $(\alpha_{13}, \alpha_{31})$, $\ldots$, $(\alpha_{1k}, \alpha_{k1})$, $(\alpha_{23}, \alpha_{32})$, $\ldots$, $(\alpha_{2k}, \alpha_{k2})$, and so on.

Thus, for each location l, the number of distinguishable values of α is at most

$$((Max_c + 2)(Max_c + 3))^{k(k-1)/2} < (Max_c + 3)^{k(k-1)/2}.$$

Therefore, the number of distinguishable equivalence classes is less than $|L| \times (Max_c + 3)^{k(k-1)/2}$ since an equivalence class is a pair of (l, α), where L is the set of locations. □

ACKNOWLEDGMENTS

This research was supported in part by the U.S. Air Force Office of Scientific Research grant F49620-95-1-0508, Northwestern University grants ARODAAG55-98-1-0393, ARODAAG55-98-1-0466, U.S. National Science Foundation grant CCR-9619910, and Office of Naval Research grant N00014-97-1-0505.

REFERENCES

[1] R. Alur, "Techniques for Automatic Verification of Real-Time Systems," PhD dissertation, Department of Computer Science, Stanford Univ., Aug. 1991.

[2] R. Alur, C. Courcoubetis, and D. Dill, "Model Checking for Real-time Systems," *Proc. of IEEE Symp. Logic in Computer Science*, 1990.

[3] R. Alur, C. Courcoubetis, D. Dill, N. Halbwachs, and H. Wong-Toi, "An Implementation of Three Algorithms for Timing Verification Based on Automata Emptiness," *Proc. IEEE Real-Time Systems Symp.*, 1992.

[4] R. Alur, C. Courcoubetis, N. Halbwachs, D. Dill, and H. Wong-Toi, "Minimization of Timed Transition Systems," *Proc. Int'l Conf. Concurrency Theory*, vol. 630, Aug. 1992.

[5] R. Alur, C. Courcoubetis, N. Halbwachs, T. Henzinger, ,P. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine., "The Algorithm Analysis of Hybrid Systems," *Theoretical Computer Science*, vol. 138, pp. 3–34, 1995.

[6] R. Alur and D. Dill, "A Theory of Timed Automata," *Theoretical Computer Science*, pp. 183–235, 1994.

[7] J. Bengtsson, K. Larsen, F. Larsson, P. Petterson, and W. Yi., "UPPAAL — A Tool Suite for Automatic Verification of Real-Time Systems," *Proc. DIMACS Workshop on Verification and Control of Hybrid Systems*, 1995.

[8] A. Bouajjani, J.-C. Fernandez, N. Halbwachs, and P. Raymond, "Minimal State Graph Generation," *Science of Computer Programming*, vol. 18, pp. 247–269 1992.

[9] D. Clarke, I. Lee, and H.-L. Xie, "VERSA: A Tool for the Specification and Analysis of Resource-Bound Real-Time Systems," *J. Computer and Software Eng.*, vol. 3, no. 2, Apr. 1995.

[10] C. Draws, A. Olivero, and S. Yovine, "Verifying ET-LOTOS Programs with KRONOS," *Proc. Seventh Int'l Conf. Formal Description Techniques*, 1994.

[11] C. Draws and S. Yovine, "Two Examples of Verification of Multirate Timed Automata with KRONOS," *Proc. IEEE Real-Time Systems Symp.*, Dec. 1995.

[12] W. M. Elseaidy, R. Cleaveland, and J.W. Baugh, Jr., "Verifying an Intelligent Structural Control System: A Case Study," *Proc. IEEE Real-Time Systems Symposium*, 1994.

[13] C. Heitmeyer, R. Jeffords, and B. Labaw, "Comparing Different Approaches for Specifying and Verifying Real-Time Systems," *Proc. Tenth IEEE Workshop Real-Time Operating Systems and Software*, May 1993.

[14] T. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine, "Symbolic Model Checking for Real-Time Systems," *Information and Computation*, vol. 111, no. 2, pp. 193–244, 1994.

[15] T.A. Henzinger, P. Ho, and H. Wong-Toi, "A User Guide to HYTECH," Technical Report CSD-TR-95-1532, Cornell Univ., 1995.

[16] F. Jahanian and A.K. Mok, "A Graph-Theoretic Approach for Timing Analysis and its Implementation," *IEEE Trans. Computers*, vol. 36, no. 8, pp. 961–975, Aug. 1987.

[17] F. Jahanian and D.A. Stuart, "A Method for Verifying Properties of Modechart Specifications," *Proc. IEEE Real-Time Systems Symposium*, 1988.

[18] I. Kang, "Real-Time System Analysis based on State-Space Exploration," PhD dissertation, May 1997.

[19] I. Kang and I. Lee, "State Minimization for Concurrent System Analysis Based on State Space Exploration," *Proc. Conf. Computer Assurance*, June 1994.

[20] L. Lamport, "A Fast Mutual Exclusion Algorithm," *Proc. ACM Trans. Computer Systems*, vol. 1, no. 5, pp. 1–11, 1987.

[21] K.G. Larsen, P. Petterson, and W. Yi, "Model-Checking for Real-time Systems," *Proc. Conf. Fundamentals of Computation Theory*, 1995.

[22] R. Milner, *Communication and Concurrency.* Prentice Hall, 1989.

[23] X. Nicollin, J. Sifakis, and S. Yovine, "Compiling Real-Time Specifications into Extended Automata," *IEEE Trans. Software Eng.,* vol. 18, no. 10, Sep. 1992.
[24] J. S. Ostroff, "Deciding Properties of Timed Transition Models," *IEEE Trans. Parallel and Distributed Systems,* vol. 1, no. 2, Apr. 1990.
[25] Z. Prucz, T.T. Soong, and A. Reinhorn, "An Analysis of Pulse Control for Simple Mechanical Systems," *Dynamic Systems, Measurement, and Control,* vol. 107, pp. 123–131, Jun. 1985.
[26] S.C.V. Raju, "An automatic Verification Technique for Communicating Real-Time State Machines," Technical Report 93-04-08, Univ. of Washington, April 1993.
[27] A.C. Shaw, "Communicating Real-Time State Machines," *IEEE Trans. Software Eng.,* vol. 18, no. 9, Sep. 1992.
[28] O. Sokolsky and S. A. Smolka, "Local Model Checking for Real-Time Systems," *Proc. Conf. Computer-Aided Verification,* 1995.
[29] M. Yannakakis and D. Lee, "An Efficient Algorithm for Minimizing Real-time Transition Systems," *Proc. Workshop Computer-Aided Verification,* 1993.
[30] W. Yi, P. Petterson, and M. Daniels, "Automatic Verification of Real-Time Systems by Constraint Solving," *Proc. Seventh Int'l Conf. Formal Description Techniques,* 1994.

Inhye Kang received her BS and MS degrees in computer engineering from Seoul National University, Korea in 1987 and 1989, respectively. She received her PhD degree in computer science from University of Pennsylvania in 1997. She is a visiting professor at Soongsil University, Korea. She is a member of the IEEE. Her research interests include formal methods, software engineering, and real-time systems.

Insup Lee received the BS degree in mathematics from the University of North Carolina, Chapel Hill, in 1977, and the PhD degree in computer science from the University of Wisconsin, Madison, in 1983.

He is currently professor in the Department of Computer and Information Science at the University of Pennsylvania, where he has been since 1983. He was CSE Undergraduate Curriculum Chair from September 1994 to August 1997. His research interests include real-time systems, formal methods, mobile computing, operating systems, and software engineering.

He was the cochair of the program committee for the 1992 IEEE Real-Time Systems Symposium, and the general cochair for the 1993 IEEE Real-Time Systems Symposium. He was the cochair of the program committee for First International Workshop on Real-Time Computing Systems and Applications held in December 1994 at Seoul, Korea. He was the cochair of CONCUR '95: International Conference on Concurrency Theory held in August 1995 at the University of Pennsylvania. He was the cochair of the Program Committee for Third International Workshop on Real-Time Computing Systems and Applications held in October 1996 at Seoul, Korea. He was the cochair of the program committee for the First IEEE International Symposium on Object-Oriented Real-time Distributed Computing in April 1998 at Kyoto, Japan. He is on the editorial boards of *IEEE Transactions on Computers and Formal Methods in System Design*.

Young-Si Kim received the BS degree in computer science from Chung Ang University in 1977, the MS degree from Yon Sei University in 1980, and a the PhD degree from Chung Ang University in 1991. He joined Electronics and Telecommunications Research Institute (ETRI) in 1977 and has been developing electronic switching systems, called TDX. His research interests are switching system software, distributed real-time system software, system verification and software testing. He is a principal member of engineering staff at ETRI.