Iterative Algorithms for Formal Verification of Embedded Real-Time Systems

Felice Balarin* Alberto L. Sangiovanni-Vincentelli
Department of Electrical Engineering and Computer Sciences
University of California, Berkeley, CA 94720

Abstract

Most embedded real-time systems consists of many concurrent components operating at significantly different speeds. Thus, an algorithm for formal verification of such systems must efficiently deal with a large number of states and large ratios of timing constants. We present such an algorithm based on timed automata*, a model where a finite state system is augmented with time measuring devices called* timers*. We also present a semi-decision procedure for an extended model where timers can be decremented. This extension allows describing behaviors that are not expressible by timed automata, for example interrupts in a real-time operating system.*

1 Introduction

Most of the recent developments in formal verification of real-time systems stem from the work of Alur and Dill [1]. They define *timed automata*, a model where a finite state system is augmented with real-valued, time measuring devices called *timers*. Even though the state space of timed automata is infinite, they provide a construction of an equivalent finite-state system called a *region automaton.*

Each state of a region automaton consists of an untimed part corresponding to the state of the original finite-state system, and a *region* corresponding to possible values of timers. Unfortunately, the number of reachable untimed parts is typically exponential in the number of components of the system, and the number of regions grows exponentially not only with the number of timers, but also in the sizes of constants used in timing constraints. The latter is particularly problematic for *stiff* real-time systems, i.e. systems consisting of components operating at widely different speeds. Most of the embedded systems can be classified as stiff real-time systems. For example, in an automotive system the engine revolution data may be collected several thousands times in one second, while the cabin temperature may be checked only once every ten seconds. Even when time constants in the environment do not vary so widely, most embedded systems contain both (typically slow) software and (typically fast) hardware components.

The primary contribution of this paper is a new iterative algorithm for stiff real-time systems with the following features:

1. it can be combined with implicit state enumeration techniques (e.g. [4]), therefore it can deal efficiently with large untimed state spaces, and
2. the complexity of every iteration does not depend on the sizes of timing constants, therefore iterations are efficient even for stiff systems.

One approach to managing a large number of regions is to represent sets of regions symbolically with matrices. This approach proved to be fairly insensitive to stiffness [8, 2], but untimed parts of all reachable states have to be explicitly enumerated. Thus, this approach can be used only for systems with a small untimed portion of the state space.

Another iterative approach was suggested by Alur et al. [3]. This approach is compatible both with explicit and implicit state enumeration techniques, but the problem is that in every iteration they construct what is essentially a complete region automaton for a subset of timers. Thus, this approach is not suitable for stiff real-time systems.

Finally, in our previous work [7] we have suggested an approach where in every iteration we construct an automaton the state space of which does not depend on the sizes of timing constants. However, in any iteration we may introduce some auxiliary I/O variables, with the number of values proportional to timing constants in the model. Thus, this approach may still suffer from stiffness, as shown by experiments presented

*Supported by SRC under grant # 94-DC-008.

in section 6.

Of course, the approach presented here is just a heuristic and not a general efficient solution for a problem that is known to be PSPACE-complete [1]. Even though every iteration is efficient, the number of iteration can be exponential, and in the worst case the full region automaton is constructed in the last iteration. Fortunately, the worst case is rare: we have yet to find a real example that requires the construction of the full region automaton.

Another contribution of this paper is a semi-decision procedure for an extended model that allow timers to be decremented. We show how this extension can be used to model features (such as interrupts) which cannot be modeled with standard timed automata.

2 Timed automata with decrement

To be able to treat uniformly both strict and non-strict inequalities we introduce the concept of bounds similar to [3]. The *domain of bounds* is the extension of the set of integers with expressions of the form n^- and n^+, which can be thought of as real numbers infinitesimally smaller (larger) than integer n. The addition, negation and comparison to reals is then naturally extended to bounds [5]. For example, $x \geq n^+$ is equivalent to $x > n$, and $-(n^-) = (-n)^+$.

Let $V = \{x_1, \ldots, x_n\}$ denote a set of *timer variables*. The set *timing constraints* Ψ is the set of formulas of the form $x \leq a$, $x \geq b$ or $x - y \leq c$, where x and y are timer variables, and a, b, and c are bounds. Let 2^Ψ denote the set of all *finite* subsets of Ψ. A *timer valuation* $\tau : V \rightarrow R$ assigns a real value to every timer variable. We say that a timer valuation τ satisfies timing constraints $x \leq a$, ($x \geq b$, $x - y \leq c$) if $\tau(x) \leq a$, ($\tau(x) \geq b$, $\tau(x) - \tau(y) \leq c$, respectively).

A *timed automaton with decrement* (TAD) is a 8-tuple $(\Sigma, Q, I, TR, V, TO, TM, F)$ where Σ is some finite set of *I/O values*, Q is some finite set of *states*, $I \subseteq Q$ is a set of *initial states*, $TR \subseteq Q \times \Sigma \times Q$ is a *transition relation*, $V = \{x_1, \ldots, x_n\}$ is a set of timer variables, $TO : Q \times Q \rightarrow 2^\Psi$ is a *timing obligation*, $TM : Q \times Q \times V \rightarrow \{reset\} \cup \{0, 1, 2, \ldots\}$ is a *timer modifier*, and $F \subseteq Q$ is a set of *final states*.

We say that a sequence of states $q_0 q_1 \ldots q_n \in Q^*$ is a *run* of a sequence of I/O values $\sigma_1 \sigma_2 \ldots \sigma_n \in \Sigma^*$ if for all $i = 1, \ldots, n$: $(q_{i-1}, \sigma_i, q_i) \in TR$. A run is *initialized* if $q_0 \in I$. A run is *terminated* if $q_n \in F$. A sequence $0 = \delta_0 < \delta_1 < \delta_2 < \ldots < \delta_n$ of non-negative real numbers is a *consistent timing* of $q_0 q_1 \ldots q_n$ if there exists a sequence of timer valuation $\tau_1 \tau_2 \ldots \tau_n$ such that for all $i = 1, \ldots, n$:

the timing obligation is fulfilled: τ_i satisfies *all* timing constraints in $TO(q_{i-1}, q_i)$,

the timer modifier is obeyed: for all timer variables $x \in V$: $\tau_i(x) = \delta_i - \delta_{i-1}$ if $i = 1$ or $TM(q_{i-2}, q_{i-1}, x) = reset$, and otherwise:

$$\tau_i(x) = \tau_{i-1}(x) - TM(q_{i-2}, q_{i-1}, x) + \delta_i - \delta_{i-1} \ .$$

A run is *accepting* if it is initialized, terminated and admits a consistent timing. The formal verification problem typically reduces to deciding an existence of an accepting run [4]. This problem is undecidable for TAD's in general [5], but Alur and Dill have shown [1] that it is decidable (in fact PSPACE-complete) for *timed automata*, a subclass of TAD's satisfying $TM(q, s, x) \in \{reset, 0\}$ for all $q, s \in Q$ and all $x \in V$.

An *untimed automaton* is just a special case timed automaton with an empty set of timer variables. We denote it by (Σ, Q, I, TR, F). Checking language emptiness of untimed automata can be done in time proportional to the number of states [9].

3 Examples

In the railroad crossing in Figure 1 (adapted from [2]), the system has three components: the train, the gate, and the controller, and all states are final. The set of I/O values is a domain of the vector variable (t, c, g), where the the domain of component t is $\{o, a, i\}$ (indicating that the train can be out, approaching or in the crossing), the domain of c is $\{l, r\}$ (indicating that the controller can instruct the gate to lower or raise), and the domain of g is $\{u, d\}$ (indicating that the controller can be up or down).

The train approaches from outside of the crossing. After at least two time units of approaching, the train will enter the crossing, and then exit at most five time units from the beginning of the cycle. Exactly one time unit after the train approaches the controller commands the gate to lower, and with a delay of at most one time unit the gate will close. Similarly, at most one time unit after the train exits the crossing the controller commands the gate to raise, and with a delay of at least one and at most two time units the gate will open. For simplicity, we require the train to approach only if the gate is up.

Two properties to be verified are:

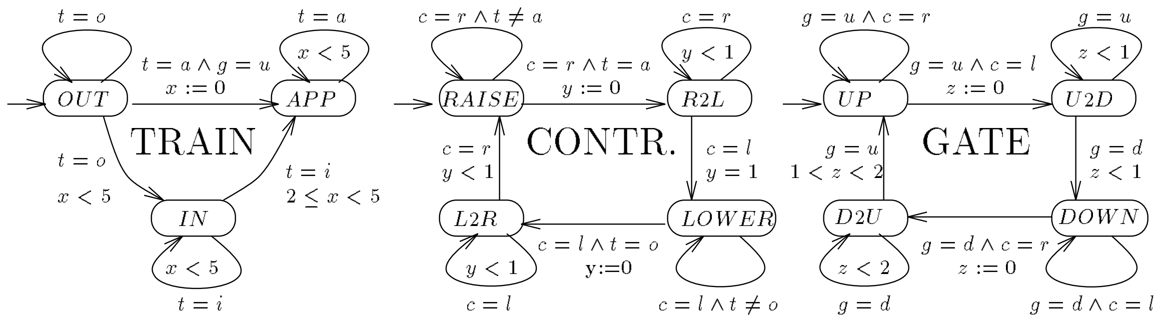


Figure 1: Railroad crossing example.

safety: the gate is down whenever the train is in the crossing, and

liveness: the gate is never down for more than seven time units.

The verification problem is to decide an existence of an accepting run in the composition of the automata in Figure 1 with the automata specifying the behavior violating the properties.

An example of a TAD which is not a timed automaton is shown in Figure 2. It represents a model of the i-th task in a real-time operating system running on a single processor machine [6]. The whole system consists of n such tasks, for $i = 0, \ldots, n-1$. The task i is activated by an interrupt of the i-th type. Between two occurrence of such interrupts at least p_i time units must pass, as measured by the timer x_i. When an interrupt occurs, the corresponding task starts running if no other task is running. Otherwise, it goes to the *pend* state. The task i requires r_i time units to execute, as measured by the timer y_i. Once running, the task cannot be preempted by another task, but if an interrupt occurs (i.e. $t_j = \mathit{interrupt}$ for some j), the service routine that takes d time units is executed. To model this we decrement y_i by d whenever an interrupt occurs. Since y_i continues to increase, this is equivalent to "freezing" y_i for d time units.[1] The tasks are prioritized, so when the processor is available (i.e. $t_j = \mathit{done}$ for some j), the pending task with the lowest index will start running.

The property to be verified is that every task completes before the next interrupt of its type occurs. This property can be violated if the time a task needs to complete $(r_i - y_i)$ becomes larger than the time until the next possible interrupt of the same type $(p_i - x_i)$. Therefore, whenever such a condition occurs, the automaton moves to the *dead* state, which is the unique final state. The system is verified if it has an empty language, i.e. no paths to the *dead* state.

[1]The idea that decrementing a timer is in some cases equivalent to freezing it is due to McManis [10].

4 Verification algorithm

We start a verification by relaxing all the timing constraints. If the verification succeeds, we have verified the task. If the verification fails, there is at least one accepting run in the current abstraction. If that run violates no timing constraints, the property is not satisfied and the verification fails. However, if the run does violate some timing constraints, we compose the current abstraction of the system with some simple automata that:

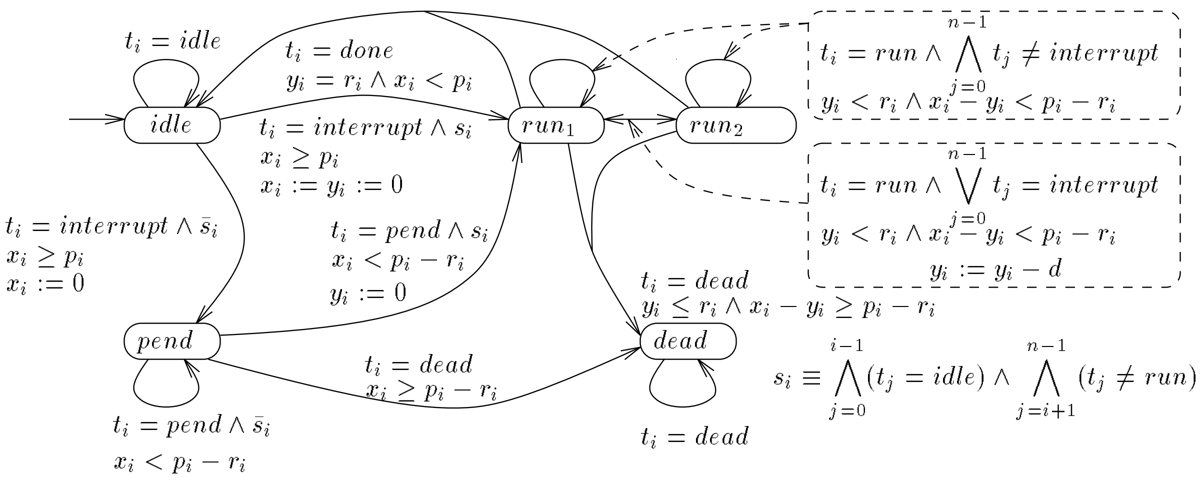
- preserve all the accepting runs in the original TAD,
- eliminate the reported run.

We repeat this process until the verification is terminated, either successfully or unsuccessfully. This strategy can lead to significant savings in time and space, provided that the behavior of the system is not heavily dependent on the timing constraints.

```
function verify_timed($\Sigma, Q, I, TR, V, TO, TM, F$)
  $T := (\Sigma, Q, I, TR, F)$;
  repeat
    if ($r$ :=verify($T$)) = $NULL$ then return $NULL$;
    if ($G$ :=analyze($T, r$)) = $NULL$ then return $r$;
    $T$ :=modify($G, T$);
  end repeat
end function
```

Figure 3: Iterative verification procedure.

The verification procedure is shown in Figure 3. The function verify_timed returns some accepting run



$t_i = idle$
$idle$
$t_i = done$
$y_i = r_i \wedge x_i < p_i$
$t_i = interrupt \wedge s_i$
$x_i \geq p_i$
$x_i := y_i := 0$
run_1
run_2
$t_i = run \wedge \bigwedge_{j=0}^{n-1} t_j \neq interrupt$
$y_i < r_i \wedge x_i - y_i < p_i - r_i$
$t_i = run \wedge \bigvee_{j=0}^{n-1} t_j = interrupt$
$y_i < r_i \wedge x_i - y_i < p_i - r_i$
$y_i := y_i - d$
$t_i = interrupt \wedge \bar{s}_i$
$x_i \geq p_i$
$x_i := 0$
$t_i = pend \wedge s_i$
$x_i < p_i - r_i$
$y_i := 0$
$t_i = dead$
$y_i \leq r_i \wedge x_i - y_i \geq p_i - r_i$
$pend$
$dead$
$t_i = dead$
$x_i \geq p_i - r_i$
$s_i \equiv \bigwedge_{j=0}^{i-1}(t_j = idle) \wedge \bigwedge_{j=i+1}^{n-1}(t_j \neq run)$
$t_i = pend \wedge \bar{s}_i$
$x_i < p_i - r_i$
$t_i = dead$

Figure 2: A model of a real-time operating system.

in $(\Sigma, Q, I, TR, V, TO, TM, F)$ if such a run exist, and otherwise it returns $NULL$. It calls three auxiliary functions: verify, analyze, and modify. The function verify is an untimed analog of verify_timed. It returns some accepting run in T, if such a run exist, and otherwise it returns $NULL$. Our procedure does not depend on any particular implementation of verify as long as it generates a failure report in the form of a run.

In the rest of this section, we first describe functions analyze (section 4.1) and modify (section 4.3), and sketch the proof of correctness (section 4.4) for the restricted case of timed automata with no constraints of the form $x - y \leq c$. Then we provide an extension to include these constraints (section 4.5). Finally, in section 4.6 we give a semi-decision procedure for arbitrary TAD's.

4.1 Failure analysis

Assume that the function verify in Figure 3 returns a run $q_0 q_1 \ldots q_n$. We want to check whether there exists a consistent timing $\delta_1, \ldots, \delta_n$ of that run.

The consistent timing must satisfy two classes of constraints: those induced by the timing obligation and those induced by the requirement that the time is strictly increasing. To treat constraints of both type uniformly, we assume (without loss of generality) that there exists a timer $x \in V$, such that it is reset on every transition, and $x > 0$ is an enabling condition of every transition (i.e. $\forall q, r.(TM(q, r, x) = reset$ and $(x > 0) \in TO(q, r))$).

To check whether a run $q_0 q_1 \ldots q_n$ admits a consistent timing we form a graph. To every transition (q_{i-1}, q_i) we associate a node i, and we add to the graph a distinguished node 0. Edges in the graph correspond to timing constraints. Given some constraint $x \leq c$ or $x \geq c$ in $TO(q_{i-1}, q_i)$ let k be the last node before i on which the timer x was reset, i.e.:

$$k = \max\{j < i | j = 0 \text{ or } TM(q_{j-1}, q_j, x) = reset\} \ ,$$

and let B be some set of transitions conditioned by that constraint that includes at least the transition (q_{i-1}, q_i). For every constraint of the form $x \leq c$, we add an edge from i to k weighted c (corresponding to the constraint $\tau_i(x) = \delta_i - \delta_k \leq c$). For every constraint of the form $x \geq c$, we add an edge from k to i weighted $-c$ (corresponding to the constraint $-\tau_i(x) = \delta_k - \delta_i \leq -c$). In both cases, we label the edge with x and B.

The sequence $q_0 q_1 \ldots q_n$ does not admit a consistent timing if and only if the graph generated by the rules above has a negative weighted loop (called an *over-constrained loop*). Finding a negative weighted loop in a graph is a well understood problem for which a polynomial algorithm exists [11].

There is a significant freedom in the choice of the set of transition B. Some of those transitions will eventually be eliminated from the current abstraction of the system. The larger the set, more behaviors will be eliminated in every iteration, thus the algorithm will converge faster. On the other hand, to keep the abstraction small, we want the representation of B to be as small as possible. This freedom can be used to fine tune the algorithm.

4.2 Notation

Given some untimed automaton:

$$A = (\Sigma, Q, I, TR, F) \ ,$$

and some set of transitions $B \subseteq Q \times \Sigma \times Q$, let $A - B$ be an automaton the same as A except that all transitions in B are removed from TR, i.e.:

$$A - B = (\Sigma, Q, I, TR - B, F) \ .$$

Given a TAD $(\Sigma, Q, I, TR, V, TO, TM, F)$ and some timer $x \in V$, let $R(x)$ be the set of all transitions on which x is reset, i.e.:

$$R(x) = \{(q, \sigma, r) | TM(q, r, x) = reset\} \ .$$

Let $\overline{B}$ denote the complement of some set of transitions $B \subseteq Q \times \Sigma \times Q$, i.e. let $\overline{B} = (Q \times \Sigma \times Q) - B$.

If $x, y \in V$ are timer variables and c is a non-negative bound, let $\langle x - y \leq c \rangle$ denote an untimed automaton as shown in Figure 4a. The automaton has two states, the "*good*" one corresponding to all valuations of x and y satisfying $x - y \leq c$, and the "*bad*" corresponding to valuations satisfying $x - y > c$. Because $c \geq 0$ and both x and y must be initially zero, the unique initial state is $x - y \leq c$. Both states are final. When c is a negative upper bound, the definition of $\langle x - y \leq c \rangle$ is slightly changed, as shown in Figure 4b.

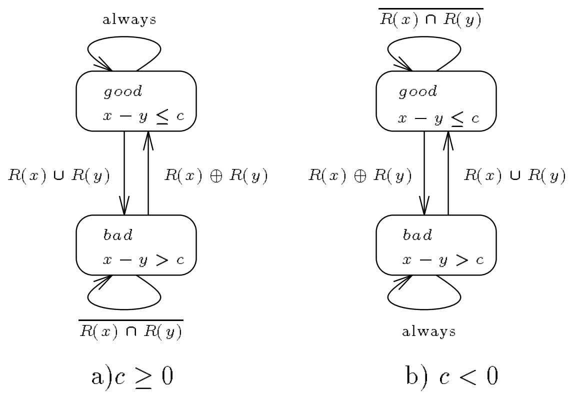


Figure 4: Automata $\langle x - y \leq c \rangle$. $S \oplus R$ is an abbreviation for $(S \cup R) \cap \overline{S \cap R}$.

We use $A(bad, *)$ to denote the set of all transitions in which the present state component of A is *bad*. Similarly, we use $A(*, good)$ to denote the set of all transitions in which the next state component of A is *good*.

```
function modify($G, T$)
  /* $G$ - an over-constrained loop */
  /* $T$ - an abstraction of the system to be verified */
  forall edges $(i, k, x, w, B)$, $(k, j, y, v, C)$ s.t. $i, j < k$
1:    $A_k := \langle y - x \leq w + v \rangle$;
2:    $T := T \otimes A_k - A_k(bad, *) \cap B \cap C$;
3:    remove edges $(i, k, x, w, B)$ and $(k, j, y, v, C)$;
4:    if $i < j$ then add to $G$ an edge:
        $(i, j, x, w + v, A_k(*, good) \cap R(y) \cap \overline{R(x)})$;
5:    else if $j < i$ then add to $G$ an edge:
        $(i, j, y, w + v, A_k(*, good) \cap R(x) \cap \overline{R(y)})$;
6:    else return $T$; /* $i = j$ */
      end if
  end while
end function
```

Figure 5: Failure elimination procedure for timed automata.

4.3 Failure elimination

In this section we describe the modify function used in the verification procedure in Figure 3. The inputs to modify are the current abstraction of the system T and an over-constrained loop G, and it returns a modified abstraction of the system, in which the failure report that has induced G is no longer a run. The modify function is shown in Figure 5, where we use (i, j, x, w, B) to represent an edge from node i to node j weighted w, and labeled with timer x, and behavior B.

The failure trace $q_0 q_1 \ldots q_n$ that has induced the over-constrained loop G is no longer a run in the modified automaton returned by modify because:

1. If A_k is the automaton generated in the last pass through the **while** loop, then A_k must be in the *bad* state after the rest of the system has gone through the sequence of states $q_0 \ldots q_{k-1}$.

2. If A_k is in the *bad* state the rest of the system has gone through the sequence of states $q_0 \ldots q_{k-1}$, then either:

   (a) the transition (q_{k-1}, q_k) is disabled in the modified automaton, or

   (b) there exists $m > k$ such that A_m is in the *bad* state after the rest of the system has gone through the sequence of states $q_0 \ldots q_{k-1} \ldots q_{m-1}$.

For example, consider the safety property of the railroad crossing. When all the timing constraints are relaxed, the property fails, and a possible failure trace is:

$$\begin{array}{l} TRAIN: \\ GATE: \\ CONT.: \end{array} \underbrace{\begin{bmatrix} OUT \\ RAISE \\ UP \end{bmatrix}}_{q_0} \underbrace{\begin{bmatrix} APP \\ R2L \\ UP \end{bmatrix}}_{q_1} \underbrace{\begin{bmatrix} IN \\ R2L \\ UP \end{bmatrix}}_{q_2} .$$

This sequence does not admit a consistent timing, as shown by the following over-constrained loop:

$$\underbrace{(q_0, q_1)}_{\text{node } 1} \xrightarrow{x \geq 2} \underbrace{(q_1, q_2)}_{\text{node } 2} \xrightarrow{y < 1} \underbrace{(q_0, q_1)}_{\text{node } 1} ,$$

In this case there are only two edges in the graph: $(1, 2, x, -2, B)$ and $(2, 1, y, 1^-, C)$, where B and C are some sets of transitions that contain (q_1, q_2). In step 1 of modify T is composed with $A_2 = \langle y - x < -1 \rangle$, and then all transition where (q_1, q_2) occurs while A_2 is in the $x - y \leq 1$ state are disabled. This is enough to eliminate the sequence $q_0 q_1 q_2$, because A_2 must move to $x - y \leq 1$ when x and y are both reset on (q_0, q_1).

4.4 Correctness

The correctness of the verified_timed algorithm can be proven by the following steps:

1. The initial abstraction of the system contains the language of the original.
2. The sequences eliminated by the modify function do not admit a consistent timing.
3. Every run which cannot be consistently timed induces an over-constrained loop.
4. Every run that induces an over-constrained loop is eliminated by the modify function.
5. Only finitely many different A_k's can be generated in step 1 of the modify function.

Together, parts 1 and 2 show that at all times the language of the current abstraction contains the language of the original system. Parts 3 and 4 show that in every iterations of the verified_timed algorithm we get closer to the language of the original system. Part 5 is used to show that the algorithm terminates in finitely many iterations. The detailed proof appears in [5].

4.5 Extension to $x - y \leq c$ constraints

To extend our approach to constraints of the form $x - y < c$, only the analyze function in the verify_timed algorithm needs to be changed. The rules for building a graph in the failure analysis are augmented such that for every timing constraint $x - y \leq c$ in $TO(q_{i-1}, q_i)$ we add an edge from m to k weighted c where k (respectively m) is the last node before i on which the timer x (y) was reset, i.e.:

$$\begin{aligned} k &= \max\{j < i | j = 0 \text{ or } TM(q_{j-1}, q_j, x) = reset\} \ , \\ m &= \max\{j < i | j = 0 \text{ or } TM(q_{j-1}, q_j, y) = reset\} \ . \end{aligned}$$

Such an edge corresponds to the following constraint that every consistent timing must satisfy:

$$\tau_i(x) - \tau_i(y) = \delta_m - \delta_k \leq c \ .$$

If $k < m$ we label such an edge with x, and otherwise we label it with y. In either case we also label the edge with some set B of transitions conditioned by $x - y \leq c$ that includes at least the transition (q_{i-1}, q_i). If that edge appears in the over-constrained loop, then before calling the modify function we compose the current abstraction of the system with $A = \langle x - y \leq c \rangle$, and eliminate transitions where transitions in B occur while A is in the *bad* state.

It can shown that all five parts of the correctness proof still hold, thus even with constraints of the form $x - y < c$, the verified_timed function will terminate with the correct result in finitely many steps.

4.6 Extension to decrements

Due to space limitations, we only list the extensions necessary to handle arbitrary TAD's. Full discussion can be found in [5].

The definition of the automata $\langle x - y \leq c \rangle$ needs to be extended to allow the following behavior when the rest of the system is making the transition (q, r), and neither $TM(q, r, x)$ nor $TM(q, r, y)$ are *reset*:

- if $TM(q, r, x) > TM(q, r, y)$, then $\langle x - y \leq c \rangle$ can move (from any state) to the *good* (i.e. $x - y \leq c$) state,
- if $TM(q, r, y) > TM(q, r, x)$, then $\langle x - y \leq c \rangle$ can move (from any state) to the *bad* (i.e. $x - y > c$) state.

The rules for building a graph in failure analysis phase must be modified as follows (k, m and B are defined as before).

For every timing constraint $x \leq c$ in $TO(q_{i-1}, q_i)$ we add add an edge:

$$(i, k, x, c + \sum_{p=k+1}^{i-1} TM(q_{p-1}, q_p, x), B) \ . \qquad (1)$$

For every timing constraint $x \geq c$ in $TO(q_{i-1}, q_i)$ we add an edge:

$$(k, i, x, -c - \sum_{p=k+1}^{i-1} TM(q_{p-1}, q_p, x), B) \ . \qquad (2)$$

For every timing constraint $x - y \leq c$ in $TO(q_{i-1}, q_i)$ we add an edge:

$$(m, k, x, c + \sum_{p=k+1}^{i-1} TM(q_{p-1}, q_p, x) - \sum_{p=m+1}^{i-1} TM(q_{p-1}, q_p, y), B) \ . \qquad (3)$$

After an over-constrained loop is found, the weights of edges generated by (1) and (2) are set to c and $-c$ respectively, and every edge generated by (3) is replaced by edges (i', k, x, c, B) and $(m, i', y, 0, B)$, where i' is a distinct copy of node i.

The modify function must be extended as shown in Figure 6. If in steps 6 and 7 node m already appears in the over-constrained loop G, new edges must be connected to a distinct copy of m.

It is possible to show that first four parts in the proof of correctness of the algorithm in Figure 5, also hold for the algorithm in Figure 6. However, the fifth part does not hold, and in fact the algorithm in Figure 6 may not terminate.

5 Hints

In our implementation of the algorithm in Figure 5, we allow the user to give "hints", i.e. to specify which timing constraints are not to be ignored initially. Actually, every hint forces our implementation to modify the initial abstraction of the system as in steps 1 and 2 of the algorithm. Therefore, to specify a hint, one needs to specify some constraint $x \geq -w$, some set of transitions B that are enabled only if that constrained is satisfied, another constraint of the form $y \leq v$, and some set of transitions C that are enabled only if the second constrained is satisfied. Given a hint, our tool will compose the current abstraction of the system with $A = \langle y - x \leq w + v \rangle$, and then eliminate all transitions in $A(bad, *) \cap B \cap C$.

```
function modify($G, T$)
  /* $G$ - an over-constrained loop */
  /* $T$ - an abstraction of the system to be verified */
  forall edges $(i, k, x, w, B)$, $(k, j, y, v, C)$ s.t. $i, j < k$
1:   $A_k := \langle y - x \leq w + v \rangle$;
2:   $T := T \otimes A_k - A_k(bad, *) \cap B \cap C$;
3:   remove edges $(i, k, x, w, B)$ and $(k, j, y, v, C)$;
4:   $m := \max\{p < k | p = 0$ or $TM(q_{p-1}, q_p, x) \neq 0$ or
                                   $TM(q_{p-1}, q_p, y) \neq 0\}$;
5:   $D := A_k(*, good) \cap$
          $\{(q, \sigma, r) | TM(q, r, x) = TM(q_{m-1}, q_m, x)$
          and $TM(q, r, y) = TM(q_{m-1}, q_m, y)\}$;
     if $m > i$ and $m > j$ then
6:     add an edge $(i, m, x, w + TM(q_{m-1}, q_m, x), D)$;
7:     add an edge $(m, j, y, v - TM(q_{m-1}, q_m, y), D)$;
     else if $i < j = m$ then
8:     add an edge $(i, m, x, w + v + TM(q_{m-1}, q_m, x), D)$;
     else if $j < i = m$ then
9:     add an edge $(m, j, y, w + v - TM(q_{m-1}, q_m, y), D)$;
     else   /* $i = j = m$ */
10:    return $T$;
     end if
  end forall
end function
```

Figure 6: Failure elimination for timed automata with decrement.

With hints, we can specify all the modifications done in step 1 and 2, and therefore we can exactly capture all the implications of timing constraints. However, specifying the right hints requires deep understanding of the system being verified.

6 Experimental results

Overall results for the timed automata verification algorithm are summarized in Table 1. All experiments were performed on a DEC workstation with 440Mb of physical memory. The examples can be divided into two basic groups, communication protocols: CSMA/CD, FDDI, and Fischer's (denoted in Table 1 with prefix fis), and control examples: railroad crossing (cross), seat-belt alarm (belt), and automated factory (fact). Results that are not available (either because none are published, or because the verification without hints is efficient enough, so we haven't developed any hints) are marked NA, and memory overflow is labeled MO. In the last column we give the best available result we have found in the literature.

The value of hints is obvious from Table 1. Without them, only the smallest examples can be verified.

Table 1: Results for the timed automata algorithm

	no hints	with hints		others
example	time	hints	time	time
csma	$1.5s$	NA		$45s$
fddi-l	$0.5s$	NA		NA
fddi-s	$39.0s$	NA		NA
fis3	26.4	6	$0.8s$	$0.4s$
fis4	$1,141s$	12	6.7	$7s$
fis5	MO	20	$168.4s$	159.4
belt	MO	5	$0.8s$	NA
fact	MO	58	$84.7s$	NA
cross-s	$0.8s$	3	$0.3s$	$0.6s$
cross-l	$3.6s$	11	$0.4s$	$1.6s$

Table 2: Sensitivity to stiffness

cycle	5	100	200	500	$1,000$	$2,000$
old	$1.8s$	$2.9s$	$4.5s$	$14s$	$45s$	$162s$
new	$0.8s$	$0.8s$	$0.8s$	$0.8s$	$0.8s$	$0.8s$

This suggests that better failure analysis techniques are needed. With hints, all but the factory example terminated in one iteration. The automated factory examples illustrates how automatic verification can complement hints. After several tries, we were able to develop a set of hints that enforces most of the timing constraints necessary to verify the property. Automatic verification then filled-in the remaining gaps in three iterations.

Compared to other available results our approach is comparable without hints and almost always better with hints. That is not surprising because the hints rely on the knowledge (and the effort) of the user, while other approaches are completely automatic.

To test the sensitivity to stiffness we have compared our old approach [7] to the algorithm in Figure 5 on the safety property of the example in Figure 1 with different values of the train cycle time (instead of the original value 5). The results are shown in Table 2. The time in the old approach grows more than linearly with the cycle time, while with the new approach the time stays constant. All experiments were performed without hints.

Finally, we mention some results for timed automata with decrements. It takes $16s$ of CPU time to verify that the interrupt of highest priority task will never be missed in the system of three tasks of Figure 2. For the task with middle priority it takes $54s$, and for the lowest priority task the program did not terminate after more than twelve hours of CPU time. This indicates that even for simple systems, the verification problem is hard and further advancements are necessary to make the verification practical.

References

[1] R. Alur and D. L. Dill. Automata for modelling real-time systems. In *Proceeding of ICALP'90.* Springer-Verlag, 1990. LNCS vol. 443.

[2] R. Alur et al. An implementation of three algorithms for timing verification based on automata emptiness. In *Proceedings of IEEE Real-time Systems Symposium*, 1992.

[3] R. Alur et al. Timing verification by successive approximation. In *Proceedings of CAV'92.* Springer-Verlag, 1993. LNCS vol. 663.

[4] A. Aziz et al. HSIS: A BDD-based environment for formal verification. In *Proceedings of the 31th ACM/IEEE Design Automation Conference*, 1994.

[5] F. Balarin. *Iterative Methods for Formal Verification of Discrete Event Systems.* PhD thesis, University of California Berkeley, 1994. in preparation.

[6] F. Balarin et al. Formal verification of the PATHO real-time operating system. In *Proceedings of 33th Conference on Decision and Control*, 1994.

[7] F. Balarin and A. L. Sangiovanni-Vincentelli. An iterative approach to verification of real-time systems. *Formal Methods in System Design: An International Journal*, 1994. to be published.

[8] T. A. Henzinger et al. Symbolic model-checking for real-time systems. In *Proceedings of 7th LICS.* IEEE Computer Society Press, 1992.

[9] J. Hopcroft and J. Ullman. *Introduction to Automata Theory, languages and Computation.* Addison Wesley, 1979.

[10] J. McManis and P. Varaiya. Suspension automata: A decidable class of hybrid automata. In *Proceedings of CAV'94.* Springer-Verlag, 1994. LNCS vol. 818.

[11] R. E. Tarjan. *Data Structures and Network Algorithms.* Society for Industrial and Applied Mathematics, Philadelphia, PA, 1983.