A Verification Strategy for Timing Constrained Systems

Felice Balarin Alberto L. Sangiovanni-Vincentelli

Department of Electrical Engineering and Computer Science
University of California, Berkeley, CA 94720

Abstract. Verification of many properties can be done without regard to the speed of the components of a finite-state system. However, some of the properties can be verified only under certain timing constraints. We propose a new verification strategy for timing constrained finite-state systems. The strategy can avoid the state space explosion problem for a class of systems. A model of such systems, called *timed L-process*, compatible with the strategy, is also developed.

1 Introduction

Recently, Dill [Dil89] and Alur and Dill [AD90] proposed a method for incorporating timing restriction into a model of communicating finite-state systems by introducing the notion of a timed automaton, containing fictitious time-measuring elements called clocks [AD90] or timers [Dil89]. The verification problem is shown to be equivalent to the speed-independent verification problem on an appropriate automaton. The fundamental problem with both approaches is state space explosion, i.e. state space growing exponentially in the number of timers (clocks).

Kurshan [Kur91] suggested to carry out the verification process on timed systems with COSPAN [HK88], a verification system for untimed processes, by relaxing the time constraints, verifying the relaxed system and if the verification is unsuccessful check whether the run that violates the property to be verified is infeasible under the timing constraints. If this is so, Kurshan removes the run and repeats the process. This strategy is appealing but heuristic in nature. There was no proof that the process would eventually converge to provably the correct answer.

In this paper, we introduce the notion of *pauses*, and construct an equivalent (non-pausing) automaton. In contrast to previous approaches, we build an equivalent automaton as a composition of the speed-independent (or unrestricted) automaton and many small automata. This decomposition of timing constraints enable us to perform the verification on a smaller, abstracted automaton which includes only some aspects of timing constraints. This leads to an iterative verification strategy similar to the heuristic proposed by Kurshan, where a verification process is started with the unrestricted automaton, which is then composed with simple automata imposing timing constraints, but only after the verification has failed, and imposing only those constraints which are violated in the failure report.

The rest of this paper is organized as follows. In section 2, we introduce the notion of timed L-process, and then we construct the equivalent (not timed) L-process in section 3. In section 4 two main steps of the proposed verification strategy are described: extracting timing violations from the failure report, and imposing that subset of timing constraints to the model of the system. Final remarks are provided in section 5.

2 Timed L-Processes

An L-process [Kur90] is an automaton over infinite sequences, distinguished from others by its alphabet and its acceptance conditions.

An alphabet of L-processes is a set of atoms of Boolean algebra L. It is convenient to think of atoms of L as distinct assignments to several variables taking values in finite domains. A boolean algebra L can be than thought of as a power set of a set of atoms, which is obviously closed under intersection (or product) $*$, union (or sum) $+$ and complement $\sim$. A partial order $\leq$ can be thought of as a set inclusion, multiplicative identity 1, as a set of all atoms, and additive identity 0 as an empty set.

Although ideas presented here are applicable to other automata, we have chosen to develop them in the framework of L-processes, because algebraic structure on the alphabet enables us to describe easily manipulations we use, like adding additional variables, or changing the transition structure

Acceptance conditions of L-processes (called cycle sets and recur edges) are particular because of their negative nature, i.e. a run is accepted unless it is excepted by cycle sets or recur edges. Hence, if no acceptance conditions are given a language of the L-process contains all sequences that have a run from some of the initial states. A product $\otimes$ (or "composition") of L-processes satisfying: $\mathcal{L}(P_1 \otimes P_2) = \mathcal{L}(P_1) \cap \mathcal{L}(P_2)$, has been defined in [Kur90].

It can be verified automatically whether the language of an L-process is contained in the language describing some properties (e.g. [HK88]). If this is not the case, there exists at least one loop of states reachable from the initial states, that is an accepting run of some sequence not in the language of the task. Usually, one such a loop is included in the failure report produced by automatic tools.

We extend L-processes by allowing them to remain in designated "pause" states a limited amount of time. This extension is called a *simple timed L-process*. Intuitively, we describe one pause by a pair of states $\{v_i^{@}, v_i^{d}\}$, as shown in Figure 1. When a system enters a state $v_i^{@}$, a pause begins. A symbol p_i, uniquely associated with that state indicates that a pause is in progress. A pause finishes when a system exits a state v_i^{d}. The time spent in these two states must satisfy the lower bound l_i and the upper bound u_i. To be able to treat uniformly both constraints of type $x < c$ and of type $x \leq c$ we adopt the concept of *bounds* introduced in [AIKY92]. The set of bounds is an extension of the set of integers with expression of the form n^- which can be thought of as a number infinitesimally smaller than the integer n. Addition and comparison are then

naturally extended.

Formally, a simple timed L-process T is a pair (P, d), where P is an L-process (called *unrestricted process of* T) and d is a set of pauses. A *pause* $i \in d$ is a 5-tuple $(v_i^@, v_i^d, p_i, l_i, u_i)$ satisfying the following:

- l_i and u_i are bounds satisfying $-\infty < l_i < 0$,[1] $0 < u_i \leq \infty$ and of course $-l_i \leq u_i$,
- $v_i^@$ and v_i^d are states of P and $p_i \in L$ is such that $M_P(v_i^@, v_i^@) = M_P(v_i^@, v_i^d) = p_i$, no other transitions depend on p_i, $v_i^@$ has no other fanouts and v_i^d has no other fanins,
- $V^@$ (a set of all $v_i^@$), V^d (a set of all v_i^d) and the set of initial states of P are mutually disjoint.

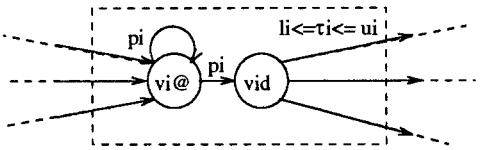


Fig. 1. A pair of states representing one pause

Figure 2a shows three examples of simple timed L-processes. Each process contains one pause, with associated bounds $-2^-, \infty$; $-1, 2^-$ and $-1, 3$, respectively.

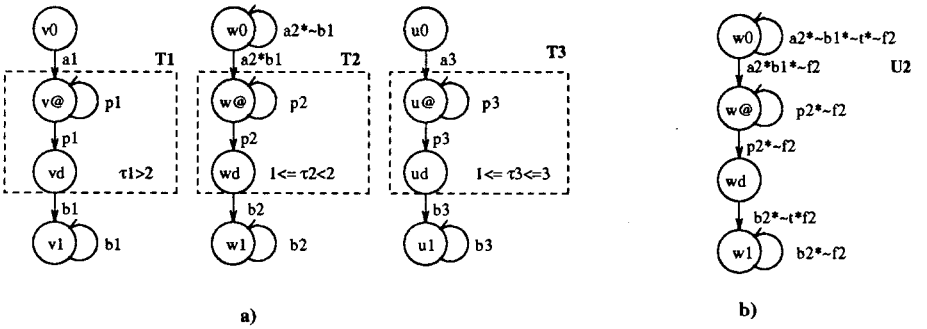


Fig. 2. Examples of simple timed L-processes (a) and corresponding unrestricted $\widehat{L}$-process (b)

A *timed sequence* (a, t) consists of a sequence a and a timing function t assigning a real positive time t_k to every a_k in a. A timing is naturally extended

[1] It is convenient to represent lower bound constraint of type $x \geq n$ as $-x \leq -n$, and of type $x > n$ as $-x \leq n^-$.

to some run v of a by: $t(v_k) = t(a_k) = t_k$. If pause i is active, the *elapsed time* of pause i at v_k, $(\tau_i)_k$ is defined to be the difference between $t(v_k)$ and the time the pause i has last started. It is convenient to extend this definition to the whole v by setting $(\tau_i)_k = 0$, if v_k is not an element of pause i.

We say that a t is a proper timing of v if all of the following consistency conditions hold:

1. $t_1 = 0$, $t_{k+1} \geq t_k$, for all $k \geq 1$ (time is non-decreasing),
2. if $v_k \notin V^d$ and $v_k \neq v_{k-1}$, then $t_k = t_{k-1}$ (all state changes outside a pause are instantaneous, the time can advance only inside a pause, or in a self-loop),
3. if $v_k = v_i^d \in V^d$ then $-(\tau_i)_k \leq l_i$ (pause-finishing times satisfy lower bounds),
4. $(\tau_i)_k \leq u_i$ (elapsed times satisfy upper bounds).

We do not make an usual requirement that time progresses without bounds, or equivalently we do allow that infinitely many events happen in a bounded amount of time. Hence, a failure to complete a pause can be acceptable. The time progress requirement can be easily added by making $\{v_i^@\}$ a cycle set. Since this has no implications to results presented here, we leave it out of the definition, as the users choice.

We say that a timed sequence (a, t) is in the *language* of a simple timed L-process $T = (P, d)$, and write $(a, t) \in \mathcal{L}(T)$, if and only if there exists v such that v is an accepting run of a in P, and t is a proper timing of v.

Pauses are tied with states, so in a simple timed L-process only one pause can be active at one time. To overcome this limitation we define a *timed L-process* as a N-tuple of simple timed processes $(T_1, \ldots, T_N)$. We say that T_n, $1 \leq n \leq N$ are the components of T. A language of the timed process T is defined to be an intersection of languages of T_n's. An *untimed language* of T is defined by: $Untime(\mathcal{L}(T)) = \{a | \exists t, (a, t) \in \mathcal{L}(T)\}$.

3 Equivalent Non-Pausing Process

In this section we will sketch the construction of the equivalent untimed process $\widehat{P}$ for some timed L-process T. We will define such a process in some extension $\widehat{L}$ of Boolean algebra L. $\widehat{P}$ and T are equivalent in a sense that the projection on L of the language of $\widehat{P}$ is exactly equal to the untimed language of T. Therefore, to prove $Untime(\mathcal{L}(T)) \subseteq \mathcal{L}(A)$, where A is some L-automaton (hence also $\widehat{L}$-automaton), it is enough to prove: $\mathcal{L}(\widehat{P}) \subseteq \mathcal{L}(A)$. Full details of the construction, as well as the proof of equivalence are given in [BSV92].

Given a timed L-process $T = (T_1, \ldots, T_N)$ defined by its components' unrestricted L-processes $P_1, \ldots, P_N$, and its components' sets of pauses $d_1, \ldots, d_N$, we construct $\widehat{P}$ as a product of the *unrestricted $\widehat{L}$-process U* the *region $\widehat{L}$-process R*.

The unrestricted $\widehat{L}$-process U is a composition of $\widehat{L}$-processes $U_1, \ldots, U_N$, each U_n being basically the same as P_n, except that we add some additional information to its output. This information is needed to coordinate U with R.

The first piece of information we add is whether or not a transition in P_n can take some time, as required by consistency condition 2. We will extend the Boolean algebra L by a new variable t and use it to label all transition that must take some time. If we also label by $\sim t$ all transition that must not take any time, we will ensure that transition from these two classes will not happen simultaneously. Specifically, we multiply with $\sim t$ all entries in the transition matrix of P_n, except the diagonal (i.e. self-loops), and the entries corresponding to transitions inside a pause. Similarly, we will add a new "flag variable" f_i for each pause i in d_n. We use f_i as a signal that pause i is finishing. Consequently, we multiply with f_i all entries of the transition matrix corresponding to the fanouts of v_i^d, and multiply by $\sim f_i$ all other entries. The process U_2 for the example in Figure 2a is shown in Figure 2b.

It is easy to see that the projection of the language of U on algebra L contains exactly those sequences that are accepted by all P_n's, including those sequences that can not be properly timed, hence are not in $Untime(\mathcal{L}(T))$. It is the purpose of the process R to eliminate such sequences from the language. Basically, R keeps record of possible elapsed times in all pauses, and does not allow a finish of some pause if the elapsed can not satisfy given bounds. Alur and Dill [AD90] have shown that it is not necessary to remember exact values of the elapsed time, but only the integer part and the ordering of fractional parts. They have also shown that if no upper bound is given all values larger than the lower bound can be considered equivalent.

To keep track of the values of elapsed times we extend the Boolean algebra with one "multi valued variable" $\widehat{\tau}_i$ for each pause i. A variable $\widehat{\tau}_i$ is a finite abstraction of τ_i, more precisely if $u_i < \infty$ it takes a distinct value for each integer and open interval between integers in $[0, u_i]$. If $u_i = \infty$, all values of τ_i larger then the lower bound correspond to single value of $\widehat{\tau}_i$. In slight abuse of notation we use bounds to represent a domain of $\widehat{\tau}_i$. More precisely, we use integers to represent themselves and bounds of the form n^- to represent intervals $(n-1, n)$, or $(n-1, \infty)$ if $u_i = \infty$ and $l_i = (n-1)$ or $l_i = (n-1)^-$.

We construct the region process as a product of *difference tracking* automata $R_{\tau_i - \tau_j \leq c}$ and *zero tracking* automata $R_{\tau_i = 0}$. We build one automaton $R_{\tau_i = 0}$ for each pause i and one automaton $R_{\tau_i - \tau_j \leq c}$ for each pair of pauses i and j and every bound c necessary to uniquely determine the value of $\widehat{\tau}_i$ (or $\widehat{\tau}_j$) given that $\tau_i = 0$ ($\tau_j = 0$, respectively), and that truth values of $\tau_i - \tau_j \leq c$ for all c's are known. For example, variables $\widehat{\tau}_2$ and $\widehat{\tau}_3$ take values in $\{0, 1^-, 1, 2^-\}$ and $\{0, 1^-, 1, 2^-, 2, 3^-, 3\}$ respectively, so we need $R_{\tau_3 - \tau_2 \leq c}$ for each c in

$$\{-1^-, -1, 0^-0, 1^-, 1, 2^-, 2, 3^-\}$$

The purpose of the $R_{\tau_i - \tau_j \leq c}$ is to establish whether $\tau_i - \tau_j \leq c$ is satisfied or not. Its state space is an abstraction of a (possibly infinite) rectangle containing all possible pairs of values of τ_i and τ_j. All points satisfying $\tau_i - \tau_j \leq c$ are contained in a "good" state and all others make a "bad" state. A unique initial state is the one containing point $(0, 0)$. It has no cycle sets nor recur edges, and

its transition matrix reflects possible *trajectories* in the rectangle containing feasible values of τ_i and τ_j. The trajectory can be constructed for any sequence $a \in \mathcal{L}(U)$ and any timing t of a. The trajectory is constructed incrementally, adding a segment from $((\tau_i)_k, (\tau_j)_k)$ to $((\tau_i)_{k+1}, (\tau_j)_{k+1})$ according to a_k and t_k. The construction rules are as follows:

Rule 1 : we begin at point $(0,0)$ and stay there as long as neither pause i nor j are active, i.e. $a_k \leq\sim p_i* \sim p_j$,

Rule 2(3) : if pause i (j) is active and pause j (i) is not, i.e. if $a_k \leq p_i* \sim p_j$ ($a_k \leq\sim p_i * p_j$), we move forward, along the τ_i (τ_j) axis,

Rule 4 : if both pauses i and j are active, i.e. if $a_k \leq p_i * p_j$, we move forward, along a 45° line,

Rule 5(6) : if pause j (i) is finishing pausing and pause i (j) is not, i.e. if $a_k \leq f_j* \sim f_i$ ($a_k \leq\sim f_j * f_i$) we move to the point $((\tau_i)_k, 0)$ $((0, (\tau_j)_k)$ respectively),

Rule 7 : if both pauses i and j are finishing, i.e. if $a_k \leq f_j * f_i$, we move to the point $(0,0)$.

The length of all forward movements is determined by $t_{k+1} - t_k$. A transition between states of $R_{\tau_i - \tau_j \leq c}$ exists if a segment of some properly timed trajectory connects two points in those states. The transition is enabled if the conditions stated in the rule that generated the segment are met. More precisely, for any pair $v, w \in \{good, bad\}$ the corresponding transition matrix entry is of the form:

$$M(v,w) = \sum(enabling\_condition * \sum((\widehat{\tau}_i = \widehat{x}) * (\widehat{\tau}_j = \widehat{y})))$$

where the outer sum goes over all eight enabling conditions in the left column of Table 1, and the inner sum goes over all abstracted values $(\widehat{x}, \widehat{y})$ of some pair of positive real numbers[2] $(x,y) \in w$ satisfying the corresponding constraint in the right column of Table 1.

Table 1. Rules for building a transition matrix of difference tracking L-processes

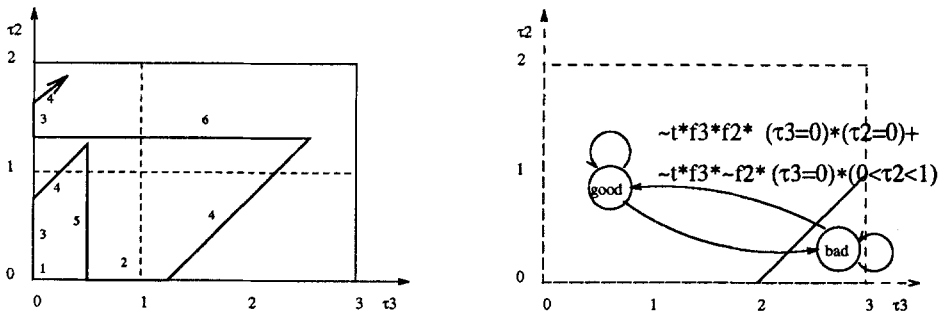
enabling condition	applied rule(s)	constraints on $(x,y) \in w$
$t* \sim p_i* \sim p_j$	1	$x = y = 0, (x,y) \in v$
$t * p_i* \sim p_j$	2	$y = 0, \exists \delta > 0 : (x - \delta, y) \in v$
$t* \sim p_i * p_j$	3	$x = 0, \exists \delta > 0 : (x, y - \delta) \in v$
$t * p_i * p_j$	4	$\exists \delta > 0 : (x - \delta, y - \delta) \in v$
$\sim t* \sim f_i * f_j$	5	$y = 0, \exists z \leq u_j : -z \leq l_j, (x,z) \in v$
$\sim t * f_i* \sim f_j$	6	$x = 0, \exists z \leq u_i : -z \leq l_i, (z,y) \in v$
$\sim t * f_i * f_j$	7	$\exists (q,z) \in v : q \leq u_i, -q \leq l_i, z \leq u_j, -z \leq l_j$
$\sim t* \sim f_i* \sim f_j$	1–7	$(x,y) \in v$

[2] If w is a good state $(x,y) \in w$ stands for $x - y \leq c$, and if w is a bad state it stands for $x - y > c$.

One possible trajectory for pauses 3 and 2 in Figure 2a is shown in Figure 3a. Each segment of the trajectory is labeled with the number of the rule that generated it. An $\widehat{L}$-process $R_{\tau_3-\tau_2\leq 2}$ is shown in 3b. The only transition which has a small enough expression to fit in the figure is *bad* $\rightarrow$ *good*. The rest of the transitions are given in Table 2. Each transition expression is formed as a sum over all rows of products of the expression in the first column and the expression in the column corresponding to that transition.

Table 2. An example of the transition matrix of a difference tracking L-process

enabling condition	transition expression		
	good $\rightarrow$ *good*	*good* $\rightarrow$ *bad*	*bad* $\rightarrow$ *bad*
$t* \sim p_3 * \sim p_2$	$\widehat{\tau}_3 = 0 * \widehat{\tau}_2 = 0$	0	0
$t * p_3 * \sim p_2$	$0 < \widehat{\tau}_3 \leq 2 * \widehat{\tau}_2 = 0$	$\widehat{\tau}_3 > 2 * \widehat{\tau}_2 = 0$	$\widehat{\tau}_3 > 2 * \widehat{\tau}_2 = 0$
$t* \sim p_3 * p_2$	$\widehat{\tau}_3 = 0 * \widehat{\tau}_2 > 0$	0	0
$t * p_3 * p_2$	$\widehat{\tau}_3 > 0 * \widehat{\tau}_2 > 0*$ $(\widehat{\tau}_3 < 3 + \widehat{\tau}_2 \geq 1)$	0	$\widehat{\tau}_3 > 2 * 0 < \widehat{\tau}_2 < 1$
$\sim t* \sim f_3 * f_2$	$0 < \widehat{\tau}_3 \leq 2 * \widehat{\tau}_2 = 0$	$\widehat{\tau}_3 > 2 * \widehat{\tau}_2 = 0$	0
$\sim t * f_3* \sim f_2$	$\widehat{\tau}_3 = 0$	0	0
$\sim t * f_3 * f_2$	$\widehat{\tau}_3 = 0 * \widehat{\tau}_2 = 0$	0	0
$\sim t* \sim f_3* \sim f_2$	$\widehat{\tau}_3 \leq 2 + \widehat{\tau}_2 \geq 1+$ $\widehat{\tau}_3 < 3 * \widehat{\tau}_2 > 0$	0	$\widehat{\tau}_3 > 2 * \widehat{\tau}_2 < 1$



a) b)

Fig. 3. A trajectory (a) and one difference tracking L-process (b)

Next, we define processes $R_{\tau_i=0}$ which track whether the elapsed time in pause i can be zero or not. For each pause i we define an $\widehat{L}$-process $R_{\tau_i=0}$ with two states v_i^0 and v_i^1, the first one being a unique initial state, no recur edges

nor cycle sets and transition matrix:

$$M_{Q_i}(v_i^0, v_i^0) = (t* \sim p_i + \sim t* \sim f_i) * (\widehat{\tau}_i = 0) \quad M_{Q_i}(v_i^0, v_i^1) = t * p_i * (\widehat{\tau}_i > 0)$$
$$M_{Q_i}(v_i^1, v_i^1) = \sim f_i * (\widehat{\tau}_i > 0) \quad M_{Q_i}(v_i^1, v_i^0) = f_i * (\widehat{\tau}_i = 0)$$

Intuitively, $R_{\tau_i=0}$ is in v_i^0 if $\tau_i = 0$ and in v_i^1 if $\tau_i > 0$. A transition from v_i^0 to v_i^1 must absolutely take some time. A transition from v_i^1 to v_i^0 occurs if the pause i has finished. Note that f_i is not accepted in v_i^0.

4 Verification Strategy

Verifying a task on $\widehat{P}$ can run into difficulties, due to the large size of the state space that has to be searched. We propose a verification strategy to avoid this problem. We start a verification process with the unrestricted $\widehat{L}$-process U. If the verification succeeds, we have verified the task. If the verification fails, there is at least one sequence which is in the language of the current abstraction of $\widehat{P}$, but not in the language of the task. We analyze one run of such a sequence. If that run violates no timing constraints, we have proved that the task is not satisfied. However, if the run does violate some timing constraints, we compose the current abstraction of $\widehat{P}$ with some simple abstraction of the process R, which is guaranteed to eliminate that 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. The verification strategy is outlined in Algorithm 1.

Algorithm 1: verification strategy
procedure verify_task()
  initialize $P_c = U$
  while not stop
    try to verify a task on P_c
    if success **then** stop, the task is verified
    find a timing violating loop G
    if such a loop does not exist **then** stop, the task is not verified
    P_c =eliminate_loop(G, P_c)
  end while
end procedure

4.1 Identifying Timing Violation

Assume that the error report from the verifier contains a loop and a path to that loop from the initial state. We can unfold the loop, thus forming an infinite

sequence of states. We form a graph with nodes being states in the sequence and the edges representing constraints on elapsed time between states. There are four kinds of edges corresponding to four consistency constraints:

backward non-pause edges: (induced by consistency condition 1) for all $k > 1$ we add an edge $(k, k-1)$ and label it with "≥ 0",

forward non-pause edges: (induced by consistency condition 2) if $t(v_k) = t(v_{k+1})$ must be satisfied by consistency condition 2, we add an edge from $(k, k+1)$ and label it with "≤ 0",

backward pause edges: (induced by consistency condition 3) if some pause i starts at node k and is finishing at node k', we add an edge (k', k) and label it with "$-\tau_i \leq l_i$",

forward pause edges: (induced by consistency condition 4) if some pause i starts at node k and is still active at node k' and $u_i < \infty$, we add an edge (k, k') and label it with "$\tau_i \leq u_i$".

The sequence can not be consistently timed only if there exists a loop in the graph such that every sum of numbers satisfying upper-bound constraints in forward edges is smaller than any sum of number satisfying lower-bound constraints in backward edges. We call such a loop an overconstrained loop. If we set a weight of an edge to be the right hand side of its label, then overconstrained loops are exactly those with weights smaller than zero. Finding a negative weighted loop is well studied problem running in a low polynomial time in the size of the graph (e.g. [Tar83]). However, in our case the graph is infinite. Therefore we have have modified the existing algorithm to process nodes in natural order (determined by the sequence) and to stop as soon as a solution to constraints which can be repeated infinitely often is found. It can be shown that if such a solution exists it will be found in finite number of steps.

Without loss of generality, we assume that the loop is minimal, in the sense that removing any edge enables proper timing of nodes. Once a loop has been identified, we collapse all non-pause edges, by merging their incident nodes. However, we mark nodes obtained by collapsing forward non-pause edges. Such a loop is an input to the algorithm which eliminates a timing constraint, described in the next subsection.

For example, for the timed L-process in Figure 2a, a sequence of states:

$$v_1 = \begin{pmatrix} v_0 \\ w_0 \\ u_0 \end{pmatrix}, v_2 = \begin{pmatrix} v^{@} \\ w_0 \\ u^{@} \end{pmatrix}, v_3 = \begin{pmatrix} v^{d} \\ w_0 \\ u^{@} \end{pmatrix}, v_4 = \begin{pmatrix} v_1 \\ w^{@} \\ u^{@} \end{pmatrix}, v_5 = \begin{pmatrix} v_1 \\ w^{d} \\ u^{@} \end{pmatrix}, \ldots \quad (1)$$

is not possible under the timing restrictions, because no timing can satisfy conflicting constraints in the following table:

constraint	edge	edge label
$t(v_5) - t(v_2) \leq 3$	$v_2 \rightarrow v_5$	$\tau_3 \leq 3$
$t(v_5) - t(v_4) \geq 1$	$v_5 \rightarrow v_4$	$-\tau_2 \leq -1$
$t(v_4) - t(v_3) \leq 0$	$v_4 \rightarrow v_3$	≤ 0
$t(v_3) - t(v_2) > 2$	$v_3 \rightarrow v_2$	$-\tau_1 \leq -2^-$

The overconstrained loop corresponding to the edges in the table is shown in Figure 4a with non-pause edges collapsed.

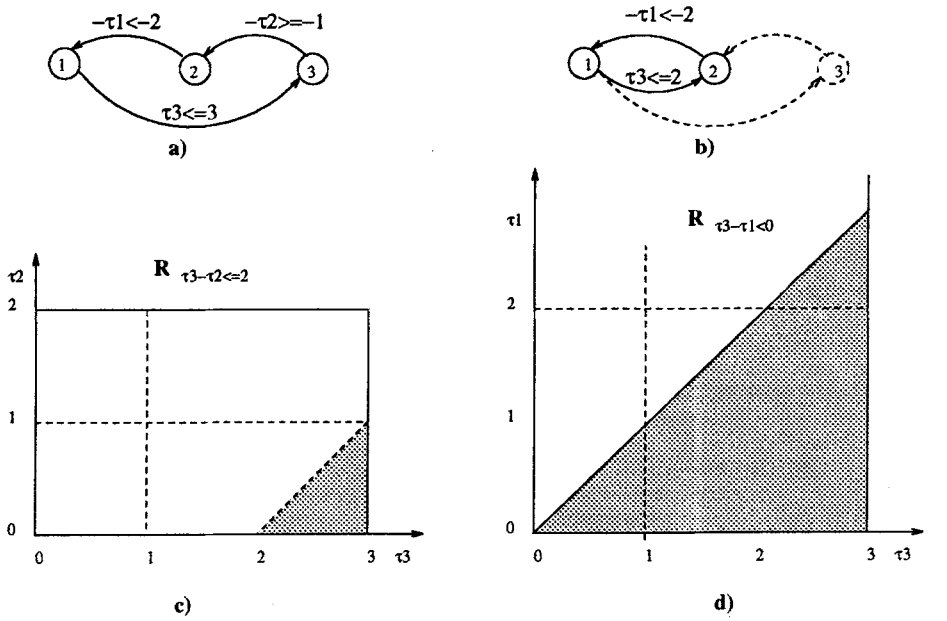


Fig. 4. An overconstrained loop and processes to eliminate it

4.2 Eliminating Timing Violations

Given an overconstrained loop G, we want to build some abstraction of R which eliminates that run. The procedure is outlined in Algorithm 2. We will follow the execution of the algorithm for the overconstrained loop in Figure 4a.

Since no nodes are marked we skip the first two steps of the algorithm. We start with any "peak" node, i.e. a node with one in-coming forward edge and one outgoing backward edge. In our example, node 3 is the only peak node. Labels $\tau_3 \leq 3$ and $-\tau_2 \leq -1$ indicate that pause 2 finishes while pause 3 is still active or just finishing. This is possible only if those two conditions can be simultaneously satisfied, or in other words, if $R_{\tau_3-\tau_2\leq 2}$ is in the good state. If $R_{\tau_3-\tau_2\leq 2}$ is in the bad state at that time, a finish of pause 2 will not be accepted and the sequence will be eliminated from the language. Therefore, in step 3 of the Algorithm 2 we compose a current abstraction of $\widehat{P}$ with the process $R_{\tau_3-\tau_2\leq 2}$. We do not need to consider edges $(1,3)$ and $(3,2)$ any more, so in step 4 we remove them from the graph. However, we do need to consider under which conditions will the process $R_{\tau_3-\tau_2\leq 2}$ be in a good or bad state. It is clear from Figure 4c that it will be in the good state at node 3, only if $\tau_3 \leq 2$ when pause 2 starts at node 2. Therefore, in step 5 we add an edge $(1,2)$ and label it with $\tau_3 \leq 2$, as

shown in Figure 4b. A dual case, when the start of the pause associated with the backward edge precedes the start of the pause associated with the forward edge, is considered in step 6.

We repeat steps 3–6 while there are peak nodes. In our example only one additional iteration is necessary, generating an abstracted pair region process $R_{\tau_3-\tau_1\leq 0^-}$, shown in Figure 4d. These two processes are enough to eliminate the sequence (1), because the process $R_{\tau_3-\tau_1\leq 0^-}$ will initially be in the bad state and remain there until pause 1 finishes, so it will accept the finish of pause 1 only if $\widehat{\tau}_2 > 2$, which in turn will force $R_{\tau_3-\tau_2\leq 2}$ to move to the bad state, where it will not accept the finish of pause 2.

This new abstraction is also enough to verify the task: "b_3 *always appear before* b_2", which is not satisfied if timing constraints are ignored. Using our strategy, we have verified the property using the abstraction of R that has only 4 states, in contrast to the full process R that has 960 states.

Algorithm 2: eliminating a timing violation

procedure eliminate_loop(G, P_c)
/* G - an overconstrained loop, P_c - a current abstraction of $\widehat{P}$ */
step 1: **for each** (k,m), labeled $-\tau_i \leq b$, m marked **do** $P_c = P_c \otimes R_{\tau_i=0}$
step 2: **for each** pair (k,m), (m,n) labeled $\tau_i \leq b$, $-\tau_j \leq c$, m marked **do**
$P_c = P_c \otimes R_{\tau_i=0} \otimes R_{\tau_i-\tau_j\leq c}$
while there exist a pair of edges (k,m), (m,n) labeled $\tau_i \leq b$, $-\tau_j \leq c$
step 3: $P_c = P_c \otimes R_{\tau_i-\tau_j\leq b+c}$
step 4: remove from G edges (k,m) and (m,n)
step 5: **if** $k < n$ **then** add to G edge (k,n) and label it $\tau_i \leq b+c$
step 6: **if** $k > n$ **then** add to G edge (k,n) and label it $-\tau_j \leq b+c$
end while
return P_c
end procedure

Step 1 is executed only if there is a backward edge coming into a marked node. For example, had the node 2 in Figure 4a been marked, the processes in Figure 4c and d would not eliminate the sequence (1). The marking of the node 2 would indicate that pause 2 starts before pause 1 finishes, but no time can elapse between these two events. The process in Figure 4d would still force $\widehat{\tau}_3 > 2$ when pause 1 finishes, but this would not be enough to force the process in Figure 4c to the bad state, because pause 2 would be active, making for example $\widehat{\tau}_3 = 3^-, \widehat{\tau}_2 = 1$ a possible choice to remain in the good state. This could be easily fixed by composing P_c with $R_{\tau_2=0}$ which would ensure $\widehat{\tau}_2 = 0$ until some transition that can take time occurs.

Step 2 is executed only if there is a marked node with in-coming forward edge and out-going backward, indicating that the pause associated with the forward edge finishes before the pause associated with the backward edge, but no time

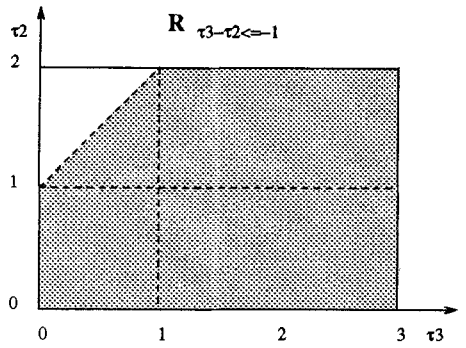


Fig. 5. Additional process needed when the peak node is marked

can elapse between these two events. Assume, for example, that the node 3 in Figure 4a is marked. Even if the process shown in Figure 4c is in a bad state, when pause 3 finishes it will move to the good state, where the finish of pause 2 is acceptable. We fix this in step 2 by composing P_c with $R_{\tau_3=0}$ and $R_{\tau_3-\tau_2\leq -1}$ (Figure 5). Now, when pause 3 finishes and the process Figure 4c is in the bad state it must be that $\widehat{\tau}_3 = 0$ and $\widehat{\tau}_2 = 1^-$, so the process in Figure 5 must be in the bad state. Since no time can elapse, it will remain there until pause 2 finishes. But $R_{\tau_3=0}$ will force $\widehat{\tau}_3 = 0$ (as long as no time elapses) and the process in Figure 5 accepts the finish of pause 2 only if $\widehat{\tau}_3 > 0$. Therefore, the sequence (1) is eliminated.

In general case, Algorithm 2 ensures, that the original sequence no longer has a run in the updated abstraction of $\widehat{P}$, because at least one of the difference tracking processes will be in the bad state at the corresponding peak node, hence it will not accept the finish of the "x-axis pause".

By Algorithm 2, it is possible to eliminate any timing inconsistent sequence, by composing the current abstraction of the system with some difference tracking and zero tracking processes. Since there are only finitely many of those, the iteration will converge in a finite number of steps.

5 Conclusions

To model timing behavior of finite-state systems, we have proposed timed L-processes. We believe that timed L-processes offer two major advantages over previous approaches. First, an equivalent L-process is defined as a composition of an unrestricted L-process and many smaller processes. We provide a transition matrix for each of these processes. In this way, the automatic generation of the equivalent process is simpler than in [Dil89] where there is one big region automaton and the computation of the next state function includes non-trivial matrix manipulation, and in [AD90] where the equivalent automaton is defined as a single automaton with a very large state space.

More importantly, we propose a verification strategy to deal with the state

space explosion problem. Basically, we propose a "trial and error" strategy, starting with the unrestricted process, and using at each step only the minimum subset of timing constraints necessary to eliminate the reported error. Although in the worst case the construction of the full region process is necessary, in our experience that is rarely the case. In fact none of the examples we tried required it. However, even if the region process is only partially constructed, the verification of timing constrained systems remains a complex and time-consuming task, requiring further research and development of more efficient techniques.

Besides time and space saving, the proposed strategy could also have a positive impact on the design. Indeed, to perform the required task, a design does not have to meet all timing constraints, but only those used in the verification. Relaxing of constraints could be used to optimize the design.

Acknowledgment

The authors would like to thank Prof. R. Brayton, R. Murgai and T. Villa for many useful discussions. We also acknowledge R. Kurshan for his presentations at UCBerkeley that sparked our interest in the subject. This work has been supported by DARPA under contract JFBI90-073.

References

[AD90] Rajeev Alur and David L. Dill. Automata for modelling real-time systems. In M.S. Paterson, editor, *ICALP 90 Automata, languages, and programming: 17th international colloquium*. Springer-Verlag, 1990. LNCS vol. 443.

[AIKY92] Rajeev Alur, Alon Itai, R. P. Kurshan, and M. Yannakakis. Timing verification by successive approximation. In *Proceeding of the Forth Workshop on Computer-Aided Verification (CAV '92)*, June 1992.

[BSV92] Felice Balarin and Alberto L. Sangiovanni-Vincentelli. Formal verification of timing constrained finite-state systems. Technical report, University of California Berkeley, 1992. UCB ERL M92/8.

[Dil89] David L. Dill. Timing assumptions and verifications of finite-state concurrent systems. In Joseph Sifakis, editor, *Automatic Verification Methods for Finite-State Systems*. Springer-Verlag, 1989. LNCS vol. 407.

[HK88] Z. Har'El and R. P. Kurshan. Software for analysis of coordination. In *Proceedings of the International Conference on System Science*, pages 382–385, 1988.

[Kur90] R. P. Kurshan. Analysis of discrete event coordination. In J.W. de Bakker, W.P. de Roever, and G. Rozenberg, editors, *Stepwise Refinement of Distributed Systems : Models, Formalisms, Correctness*, pages 414–453. Springer-Verlag, 1990. LNCS vol. 430.

[Kur91] R. P. Kurshan, 1991. private communications.

[Tar83] Robert Endre Tarjan. *Data Structures and Network Algorithms*. Society for Industrial and Applied Mathematics, Philadelphia, PA, 1983.