Timing Verification by Successive Approximation [1]

R. Alur A. Itai R.P. Kurshan M. Yannakakis

AT&T Bell Laboratories
Murray Hill, NJ 07974

Abstract. We present an algorithm for verifying that a model M with timing constraints satisfies a given temporal property T. The model M is given as a parallel composition of ω-automata P_i, where each automaton P_i is constrained by bounds on delays. The property T is given as an ω-automaton as well, and the verification problem is posed as a language inclusion question $\mathcal{L}(M) \subseteq \mathcal{L}(T)$. In constructing the composition M of the constrained automata P_i, one needs to rule out the behaviors that are inconsistent with the delay bounds, and this step is (provably) computationally expensive. We propose an iterative solution which involves generating successive approximations M_j to M, with containment $\mathcal{L}(M) \subseteq \mathcal{L}(M_j)$ and monotone convergence $\mathcal{L}(M_j) \rightarrow \mathcal{L}(M)$ within a bounded number of steps. As the succession progresses, the approximations M_j become more complex. At any step of the iteration one may get a proof or a counter-example to the original language inclusion question. The described algorithm is implemented into the verifier Cospan. We illustrate the benefits of our strategy through some examples.

1 Introduction

In recent years, there has been considerable interest in developing tools for automatic verification of concurrent systems. A verification algorithm determines whether a model of a finite-state system satisfies its specification given as a temporal logic formula or as an ω-automaton. For *delay insensitive* systems, the correctness can be proved by abstracting real-time as nondeterministic delay, retaining only the sequencing of state-transitions. For such abstraction, automatic verification has a history spanning over ten years, and has been shown to be useful in verifying protocols and circuits [AKS83, CES86, BCDM86, McM93]. However, for a large class of systems, correctness depends upon the actual magnitudes of various delays. Timing information can be incorporated in the models for delay-insensitive systems in a straightforward way assuming a *discrete* model of time [AK83, EMSS90, AH89]. A discrete model requires that all events happen simultaneously with the ticks of a global clock, and hence, limits the accuracy in modeling the timing constraints of a real-time system. Verification algorithms for real-time systems assuming a *continuous* representation of time have emerged only recently [Dil89, AD90, ACD93, Lew90, HNSY92, ACH+92]. These results indicate that the addition of timing constraints makes the verification problem harder: timing information can multiply the exponential complexity inherent in untimed verification by a factor proportional to the magnitudes of certain delay bounds. In this paper, we propose a new approach to cope with this increased computational complexity.

We use an automata-theoretic formulation of the verification problem. The theory of ω-automata has been well studied in the literature (see [Tho90] for an overview), and formal methods for temporal reasoning about concurrent systems have been developed based on this theory [Kur87, VW86]. A system is usually defined as a collection of components coordinating with each other, where each component is modeled as an ω-automaton. The ω-automaton M modeling the system is obtained using a parallel composition operator on the component automata P_i. The temporal property to be checked is modeled by an ω-automaton T. We want to verify that M has the property T, or more precisely, that the language inclusion $\mathcal{L}(M) \subseteq \mathcal{L}(T)$ holds. The language $\mathcal{L}(M)$ can be understood as the set of behaviors possible in the system M, while $\mathcal{L}(T)$ can be interpreted as the set of all behaviors consistent with the property T.

We impose timing constraints on an automata-theoretic specification through an association of certain events with the beginning and end of "delays". Each delay is constrained by a lower and an upper bound on its allowed duration. A *delay constraint* D consists of a set of such delays together with their lower and upper bounds. A sequence of events is said to be *timing consistent* with a delay constraint iff it is possible to associate increasing (real-valued) time values with all events such that the resulting duration of each

[1] A preliminary version of this paper appeared in the *Proceedings of the Fourth Conference on Computer Aided Verification* (CAV 1992), LNCS 663, and the full version appeared in *Information and Computation* 118(1):142-157, 1995

delay is within the allowed bounds. The possible behaviors of the model M are only the timing consistent sequences in the language of the parallel composition of the component automata P_i. One possible solution to the resulting verification problem is to employ the method of [AD90] by constructing an automaton P_D, the so-called *region automaton*, which rules out timing-inconsistent sequences. The language of the model M is simply the composition of the component automata P_i together with the region automaton P_D. Unfortunately, the size of P_D is exponential in the number of component automata and is proportional to the magnitudes of the bounds on the delays. Furthermore, it is known that the timing verification problem is PSPACE-complete [AD90]. We propose to construct successive approximations to the desired model as a heuristic for circumventing this computational complexity.

We first construct M_0, the composition of the automata P_i ignoring the delay constraints, and try to prove the language inclusion $\mathcal{L}(M_0) \subseteq \mathcal{L}(T)$. If this succeeds, then $\mathcal{L}(M) \subseteq \mathcal{L}(M_0) \subseteq \mathcal{L}(T)$. This means that the proof of the desired property does not require any timing assumptions, and we are done. If this fails, we can find $\mathbf{x} \in \mathcal{L}(M_0)\backslash\mathcal{L}(T)$ of the form $\mathbf{x} = \sigma'\sigma^\omega$, where σ and σ' are (finite) strings of events. We give an algorithm to check for timing consistency of $\mathbf{x}$ with respect to the delay bounds of M. The time complexity of this check is linear in the lengths of σ and σ' and cubic in the number of automata P_i. If $\mathbf{x}$ is consistent with the original delay constraint D of M, then $\mathbf{x}$ provides a counter-example to $\mathcal{L}(M) \subseteq \mathcal{L}(T)$. Otherwise, we identify a (locally) optimal delay constraint D' such that $\mathbf{x}$ is inconsistent with D', and every sequence consistent with D is also consistent with D'. Then we generate the region automaton $P_{D'}$ which accepts only those behaviors that are consistent with the delay constraint D'. Compared to the original delay constraint D, D' may have considerably fewer delay constraints, and may have bounds of smaller magnitudes, with the consequence that the size of $P_{D'}$ may be much smaller than that of P_D. We add $P_{D'}$ as a restriction to M_0, forming M_1, and iterate the algorithm. Thus, we construct successive approximations M_j to M, with containment $\mathcal{L}(M) \subseteq \mathcal{L}(M_j)$ and monotone convergence $\mathcal{L}(M_j) \to \mathcal{L}(M)$. At any step of the iteration, the computational effort required may be considerably less than the effort needed to solve the original language inclusion question, and give a proof or a counter-example to the original question. In the worst case, the number of iterations needed is exponential in the size of the original delay constraint.

We have implemented the proposed scheme in the verifier Cospan [HK90]. In Section 5, we present the results of our experiments on some examples including timing-based algorithms for mutual exclusion, and a toy controller for railroad crossing. In all these examples, the number of iterations needed before termination is much smaller than the worst case exponential bound, typically even less than the number of delays. Also the delay bounds get reduced to small numbers, and the computational effort required does not depend on the magnitudes of the bounds specified in the input. Thus, in these examples our heuristic gives significant improvements over the existing methods, and based upon these experiments, we believe it to be a promising approach. Apart from reducing the computational complexity, our iterative algorithm seems useful for another purpose, namely, for identifying the constraints that are relevant to prove the property. This identification is typical of approaches based on theorem proving, but was not possible in the previously proposed algorithmic methods.

The remaining paper is organized as follows. Section 2 gives an overview of the iterative algorithm. Section 3 gives the algorithms used to test whether a given sequence is timing consistent with a given delay constraint. Section 4 discusses the heuristics used for optimizing the delay constraint. The implementation and the examples are discussed in Section 5.

2 Overview

A system is modeled by an ω-automaton M. The underlying alphabet Σ corresponds to the possible events, and the language $\mathcal{L}(M)$ corresponds to the set of possible behaviors of the system. Given a temporal property modeled by another ω-automaton T, the verification problem is to test whether the the language inclusion $\mathcal{L}(M) \subseteq \mathcal{L}(T)$ holds.

While our development here is fairly general, we make one basic requirement on the semantic nature of the automata used to model the system. We require that the automata (over a common alphabet) are closed under a parallel composition operator, denoted by $\otimes$, supporting the language intersection property:

$$\mathcal{L}(M \otimes N) = \mathcal{L}(M) \cap \mathcal{L}(N)\ . \tag{1}$$

We will refer to any such class of automata as *processes*. Examples of process classes are *L-processes* [Kur87, Kur90] with composition defined by the tensor product, *deterministic Muller automata* [Cho74] and *deterministic Rabin-Scott acceptors* (conventional automata accepting strings) with composition defined by the Cartesian product (more precisely, the $\wedge$ operator defined in [Kur87]). Such processes support models of both synchronous and asynchronous coordination, including interleaving semantics [Kur90]. While we impose no requirements of associativity on $\otimes$, in the following development, for simplicity, we will write successive compositions without parentheses. Furthermore, we assume that if Σ is the underlying alphabet, then there exists a process $\mathbf{1}$ with $\mathcal{L}(\mathbf{1}) = \Sigma^*$ or $\mathcal{L}(\mathbf{1}) = \Sigma^\omega$. In this paper we refer both to strings: finite words over Σ, and sequences: infinite words over Σ.

A system M commonly is defined in terms of coordinating components, each component influencing the behavior of the other components with which it coordinates. Therefore, processes provide a natural class of automata for modeling such coordination. We begin with a system model P expressed as a composition of K processes over the alphabet Σ: $P = \otimes_{i=1}^{K} P_i$, with no reference yet to timing constraints. The actual model M will be obtained by introducing additional constraints. Suppose that certain events $x \in \Sigma$ correspond to the beginning or the end of a "delay event" relative to various P_i. Specifically, for each P_i we are given a finite set Δ_i of *delays* (as yet, uninterpreted); let Δ_P be $\cup_i \Delta_i$. The association between the events in Σ and the delays is given, for each P_i, by two partial functions b_i ("begin") and e_i ("end"), mapping some of the events in Σ to the delays in Δ_i. For an event $x \in \Sigma$, if $b_i(x) = \delta$, then this means that process P_i begins delay $\delta \in \Delta_i$ at event x, and if $e_i(x') = \delta$ then P_i ends this delay at x'. For example, x may correspond to the receipt of a signal by P_i, marking the onset of an internal delay $\delta \in \Delta_i$: $b_i(x) = \delta$, and x' may correspond to the subsequent emission of a signal from P_i marking the end of that delay: $e_i(x') = \delta$. If process P_i did not begin a delay at event x, then $b_i(x)$ is undefined; likewise for $e_i(x)$. If processes P_i and P_j both begin a delay at event x, then $b_i(x) \in \Delta_i$ and $b_j(x) \in \Delta_j$ are defined. For example, x may correspond to the simultaneous receipt of a signal by P_i and emission of a (perhaps unrelated) signal from P_j. For more examples of how this arises naturally in modeling systems of concurrent processes, see [Kur87, Kur90].

Now, recall that the verification problem is to prove $\mathcal{L}(P) \subseteq \mathcal{L}(T)$ for some process T. However, it may be the case that while $\mathcal{L}(P) \not\subseteq \mathcal{L}(T)$, it nonetheless holds that $\mathcal{L} \subseteq \mathcal{L}(T)$ for the subset $\mathcal{L} \subset \mathcal{L}(P)$ which is consistent with timing constraints we impose on the delays. Formally, we associate with each delay $\delta \in \Delta_P$ two nonnegative numbers $0 \leq \alpha(\delta) \leq \beta(\delta)$, where the lower bound $\alpha(\delta)$ is a nonnegative integer and the upper bound $\beta(\delta)$ is either a positive integer or is ∞. The intended interpretation is that the delay δ has duration at least $\alpha(\delta)$ and at most $\beta(\delta)$. This allows us to express timing constraints such as the delay between the transmission of a message at one end of a channel and its arrival at the other end is at least $2sec$ and at most $5sec$. A *delay constraint* D is a 3-tuple (Δ, α, β) where $\Delta \subseteq \Delta_P$. Now we define the notion of timing consistency:

> A sequence $\mathbf{x} = x_1 x_2 \cdots$ is said to be *timing-consistent* with respect to a delay constraint $D = (\Delta, \alpha, \beta)$ iff there exist real numbers $t_1 < t_2 < \cdots$ such that for all processes P_i and for all delays $\delta \in \Delta \cap \Delta_i$, if $b_i(x_j) = \delta$, and $e_i(x_k) = \delta$ for $k > j$, and both $b_i(x_l)$ and $e_i(x_l)$ are undefined for $j < l < k$, then $\alpha(\delta) \leq t_k - t_j \leq \beta(\delta)$.

In other words, $\mathbf{x}$ is timing-consistent if it is possible to assign an increasing succession of real times to the events of P modeled by $\mathbf{x}$, in such a way that the durations of delays of the respective component processes P_i as given by these time assignments are within the allowed bounds. Note that according to the above definition, a pair of "begin" and "end" events for process P_i with an intervening "begin" or "end" event for the same process P_i, does not contribute any additional timing constraint. Also even if $\alpha(\delta) = 0$, the time difference between a "begin" and the matching "end" event for δ has to be strictly greater than 0. We define a string $\sigma = x_1 x_2 \ldots x_n$ to be *timing-consistent* (with D) provided there are real numbers $t_1 < \cdots < t_n$ satisfying these same conditions. For any delay constraint D, let

$$\mathcal{L}_D(P) = \{\mathbf{x} \in \mathcal{L}(P) \mid \mathbf{x} \text{ is timing-consistent with } D\}.$$

The system model M consists of the untimed model P together with the delay constraint $D = (\Delta_P, \alpha, \beta)$, and $\mathcal{L}(M) = \mathcal{L}_D(P)$. Thus we seek an algorithm to answer the language inclusion question

$$\mathcal{L}_D(P) \subseteq \mathcal{L}(T). \tag{2}$$

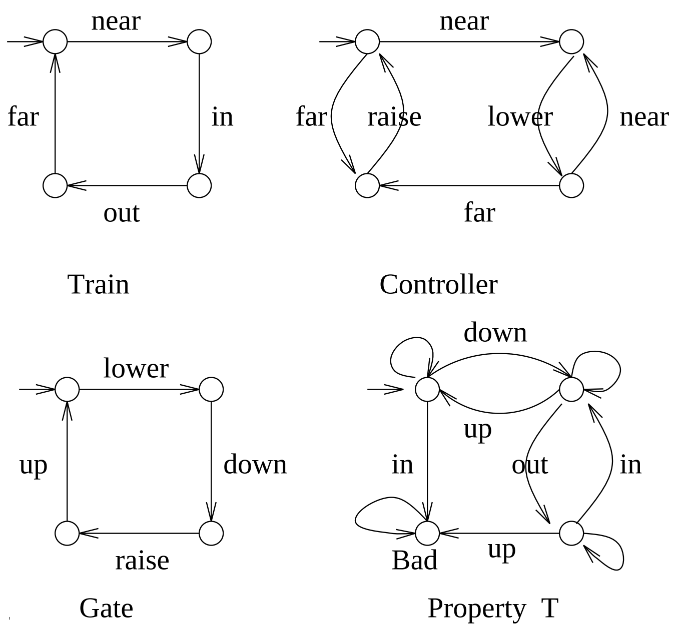

Figure 1: The railroad crossing

We note that there are many alternative ways of introducing timing in the automaton model. For instance, in a timed automaton [AD90], the automaton state is augmented with a finite set of (real-valued) clocks, and the edges are annotated with assignments and constraints on the clock values, whereas in a timed transition system [HMP91], each transition has a lower and an upper bound that constrains the interval for which it can be continuously enabled without being taken. In terms of expressiveness, all these formalisms are equivalent. Our choice of notation allows us to integrate our method into the verification system Cospan with greater ease. Let us consider a toy controller for railroad crossing to illustrate our definitions.

Example 1 We consider an example of a gate controller at a railroad crossing. The system is composed of three components: Train, Gate, and Controller. The train communicates with the controller with two events *near* and *far*. The events *in* and *out* mark the events of entry and exit of the train from the railroad crossing. The controller communicates with the gate using the events *raise* and *lower*, and the events *up* and *down* denote the opening and closing of the gate. The automata are shown in Figure 1. Assume that for each automaton and for each event x that it does not participate in, there is a self-loop on each state labeled with x. Thus, in the automaton Train, each state has a self-loop labeled with each of the events in $\{lower, raise, up, down\}$. The initial states are indicated by arrows. The acceptance conditions are trivial, and thus every possible sequence generated by an automaton is accepted. For instance, the language $\mathcal{L}(Train)$ is $(\Sigma'^{*}\ near\ \Sigma'^{*}\ in\ \Sigma'^{*}\ out\ \Sigma'^{*}\ far)^{\omega}$, where $\Sigma' = \{lower, raise, up, down\}$. The parallel composition of two automata corresponds to their (synchronous) product, and the language of the product is simply the intersection of the languages of the components. Let P be the composition of Train, Gate, and Controller.

We want to prove a safety property of the system that states that whenever the train is inside the gate, the gate should be closed. The property is specified by the deterministic automaton T of Figure 1. Assume that each state s has a self-loop labeled with an event x if there is no other edge starting from s and explicitly labeled with x. Thus the start state has a self-loop for every event other than *in* and *down*. The accepting sequences of T are those corresponding to the runs that do not visit the state labeled *Bad*.

To prove the property T, we need to impose timing constraints on the model P. Suppose we know that the train sends the signal *near* to the controller precisely $3sec$ before it enters the gate, and it sends the signal *far* within a time interval of $1sec$ to $2sec$ after its entry. We also assume that the separation between trains, that is, the delay between *far* and *near* is at least $5sec$. For the train, let the set of delays be $\Delta_{train} = \{\delta_1, \delta_2, \delta_3\}$. The event *near* marks the beginning of delay δ_1, and *in* marks its end. The event *in* also marks the beginning of delay δ_2, and *far* marks its end. The delay δ_3 begins with *far* and ends with *near*.

Thus $b_{train}(near) = \delta_1$, $b_{train}(in) = \delta_2$, $b_{train}(far) = \delta_3$, and b_{train} is undefined for the remaining events. The function e_{train} is defined similarly. The desired timing constraints are enforced by setting $\alpha(\delta_1) = \beta(\delta_1) = 3$, $\alpha(\delta_2) = 1$, $\beta(\delta_2) = 2$, and $\alpha(\delta_3) = 5$, $\beta(\delta_3) = \infty$.

In addition, suppose we know that the controller responds to the signal *near* by sending *lower* after a delay of $1sec$, and responds to the signal *far* by sending *raise* within $1sec$. Let $\Delta_{controller} = \{\delta_4, \delta_5\}$. The delay δ_4 begins with *near* and ends with *lower*, while δ_5 begins with *far* and ends with *raise*. Let $\alpha(\delta_4) = \beta(\delta_4) = 1$, and $\alpha(\delta_5) = 0$, $\beta(\delta_5) = 1$.

Similarly, assume that the response time of gate to *lower* is less than $1sec$, and its response time to *raise* is within $1sec$ to $2sec$. This assumption can be modeled by a delay δ_6 that begins with *lower* and ends with *down*, and a delay δ_7 that begins with *raise* and ends with *up*.

The language $\mathcal{L}_D(P)$ is a strict subset of $\mathcal{L}(P)$. For instance, the string $\sigma = near, lower, in, down$ is a prefix of a possible behavior in $\mathcal{L}(P)$. However, for σ to be timing consistent, we need to find $t_1 < t_2 < t_3 < t_4$ such that $t_3 - t_1 = 3$ and $t_2 - t_1 = 1$ and $t_4 - t_2 \leq 1$. This is not possible, and hence, σ is timing inconsistent, and is not a prefix of any behavior in $\mathcal{L}_D(P)$. In fact, $\mathcal{L}(P)$ is not included in $\mathcal{L}(T)$, but $\mathcal{L}_D(P)$ is. ∎

Now consider the problem of testing whether the language inclusion $\mathcal{L}_D(P) \subseteq \mathcal{L}(T)$ holds. Using the method of [AD90], we can construct an automaton P_D, the so-called *region automaton*, which rules out timing-inconsistent sequences; that is, $\mathcal{L}(P \otimes P_D) = \mathcal{L}_D(P)$. This strategy reduces the timing verification problem to an ordinary language inclusion test. Unfortunately, the size of P_D is exponential in the number of processes comprising P and is proportional to the magnitudes of the bounds given by α and β. These exponential dependencies are inherent to the problem, it is known that the timing verification problem is PSPACE-complete. The purpose of this paper is to develop a heuristic for circumventing this computational complexity. Our heuristic works like this. We first try to prove the language inclusion

$$\mathcal{L}(P) \subseteq \mathcal{L}(T) \ . \tag{3}$$

We assume that there is a verification tool that allows us to perform such language inclusion tests, and that reports counter-examples when the test fails. If the inclusion test (3) succeeds, then surely (2) holds, as $\mathcal{L}_D(P) \subseteq \mathcal{L}(P)$. This is the case when no timing assumptions are required to prove the property T.

If (3) fails, then for some $\mathbf{x} \in \mathcal{L}(P)$, $\mathbf{x} \notin \mathcal{L}(T)$. The next step of the algorithm is to test whether or not the counter-example $\mathbf{x}$ is timing consistent with respect to $(\Delta_P, \alpha, \beta)$. Let us first consider the case that $\mathbf{x}$ has a finite prefix σ such that every extension of σ does not belong to $\mathcal{L}(T)$. This is the case, for instance, when the temporal property defined by T is a "safety" property. The problem of testing timing consistency of σ can be reduced to the problem of finding negative cost cycles in a weighted directed graph as described in Section 3. In this case, the timing-consistency of $\mathbf{x}$ can be checked in time $O(|\sigma| \cdot K^2)$, where K is the number of processes. For the general case when T specifies an arbitrary ω-regular property, we can find a counter-example $\mathbf{x}$ of the form $\mathbf{x} = \sigma' \sigma^\omega$ for strings σ and σ'. We show that in this general case, checking timing consistency of $\mathbf{x}$ can be reduced to computing shortest distances in an infinite periodic weighted digraph, and develop an algorithm for this purpose. The time complexity of the algorithm is $O((|\sigma'| + |\sigma|) \cdot K^2 + K^3 \cdot \lceil \log K \rceil)$.

If $\mathbf{x}$ is timing-consistent, then $\mathbf{x}$ provides a counter-example to (2). Otherwise, we identify first a minimal set of processes, and for those, a minimal set of delays $\Delta_{\mathbf{x}}$, giving a delay constraint $D' = (\Delta_{\mathbf{x}}, \alpha, \beta)$ with respect to which $\mathbf{x}$ is not timing-consistent. Next, we relax the delay bound maps α and β by decreasing respective $\alpha(\delta)$'s and increasing $\beta(\delta)$'s, in a fashion which preserves the non-timing-consistency of $\mathbf{x}$ but which, after dividing each bound by their collective greatest common divisor, results in bounds which are as small as possible. The result of these operations gives an optimized delay constraint $D(\mathbf{x})$ such that $\mathbf{x}$ is timing inconsistent with it and every sequence timing inconsistent with it is also inconsistent with the original constraint D. Using an algorithm from [AD90], we generate a process $E_1 = P_{D(\mathbf{x})}$ such that $\mathcal{L}(E_1) = \mathcal{L}_{D(\mathbf{x})}(\mathbf{1})$, where $\mathbf{1}$ is a process satisfying $\mathcal{L}(\mathbf{1}) = \Sigma^\omega$. If $\Delta_{\mathbf{x}}$ contains less number of delays than the original set Δ_P, or if the bounds of the delays in $D(\mathbf{x})$ are smaller than the corresponding bounds in the original delay constraint D, then the size of E_1 will be smaller than the size of P_D. Our strategy for selecting $D(\mathbf{x})$ is only a heuristic to keep the size of E_1 small. It is not guaranteed to give the smallest process that excludes $\mathbf{x}$, nor is it clear that the smallest process would be a better choice in the context of the successive iterations. Experience suggests that in many cases only a few delay bounds are material to the verification of

Input: Processes $P_1, \ldots P_k, T$, and a delay constraint $D = (\Delta_P, \alpha, \beta)$.
Output: Decides whether the inclusion $\mathcal{L}_D(\otimes_i P_i) \subseteq \mathcal{L}(T)$ holds.
Algorithm:
Initially: $P = \otimes_i P_i$ and $E = \mathbf{1}$ and $X = \emptyset$.
Loop

1. If $\mathcal{L}(P \otimes E) \subseteq \mathcal{L}(T)$ then stop (the desired inclusion holds).
2. Choose $\mathbf{x} = \sigma' \sigma^\omega$ in $\mathcal{L}(P \otimes E) - \mathcal{L}(T)$.
3. If $\mathbf{x}$ is timing consistent with the delay constraint D then stop (the desired inclusion does not hold).
4. Add $\mathbf{x}$ to X.
5. Find a minimal set of delays $\Delta_X \subseteq \Delta_P$ such that each $\mathbf{y} \in X$ is timing inconsistent with $(\Delta_X, \alpha, \beta)$.
6. Using relaxation, find the "smallest" bounds α_X and β_X such that each $\mathbf{y} \in X$ is timing inconsistent with $D(X) = (\Delta_X, \alpha_X, \beta_X)$.
7. Let E be the region automaton $P_{D(X)}$.

Figure 2: Algorithm for timing verification

any specific property T, especially in case T is a "local" property (derived, perhaps, through decomposition [Kur90] of a global property). In such cases $D(\mathbf{x})$ involves only a few processes, and hence, the size of E_1 is small.

We test the language inclusion

$$\mathcal{L}(P \otimes E_1) \subseteq \mathcal{L}(T) \tag{4}$$

and proceed as before, either verifying (4) and thus (2), or finding a counter-example $\mathbf{y}$ to (4). In the latter case, either $\mathbf{y}$ is a counter-example to (2), or it is timing inconsistent. In this last case, we find an optimal delay constraint $D(\mathbf{x}, \mathbf{y})$ as before, with respect to which both $\mathbf{x}$ and $\mathbf{y}$ are timing inconsistent, and choose the next approximation E_2 to be the region automaton $P_{D(\mathbf{x},\mathbf{y})}$.

The outline of the algorithm is shown in Figure 2. In each iteration, E is the current approximation to P_D, and X is the set of counter-examples obtained in the previous iterations. Note that as $\Delta_P = \cup \Delta_i$ is finite, there are only finitely many choices for Δ_X at step 5. The optimization heuristic used at step 6 can construct only finitely many choices of the bounding functions α_X and β_X for a given set Δ of delays. The algorithm terminates in at most $n = 2^{|D|}$ steps, in the worst case generating $E_1, \ldots, E_n$ with

$$\mathcal{L}(P) \supset \mathcal{L}(P \otimes E_1) \supset \mathcal{L}(P \otimes E_2) \supset \cdots \supset \mathcal{L}(P \otimes E_n) = \mathcal{L}_D(P). \tag{5}$$

Here, $|D|$ is the length of the delay constraint $(\Delta_P, \alpha, \beta)$ assuming binary encoding for the constants. In the examples of Section 5, the termination is obtained much faster, in less than $|\Delta_P|$ number of iterations.

3 Checking Timing Consistency

In this section, we address the problem of checking timing consistency of $\mathbf{x}$ with respect to a delay constraint $D = (\Delta, \alpha, \beta)$.

3.1 A graph-theoretic formulation of the problem

Consider a sequence $\sigma = x_1 x_2 \ldots$. Recall that σ is timing consistent with respect to D if and only if we can assign "time" values t_i to the respective events x_i such that

1. $t_1 < t_2 < \ldots$, and
2. if process P_i begins a delay $\delta \in \Delta$ at x_j (i.e., $b_i(x_j) = \delta$), and the matching end is at x_k (i.e., $e_i(x_k) = \delta$ and both $b_i(x_l)$ and $e_i(x_l)$ are undefined for $j < l < k$), then

$$\alpha(\delta) \leq \ t_k - t_j \ \leq \beta(\delta).$$

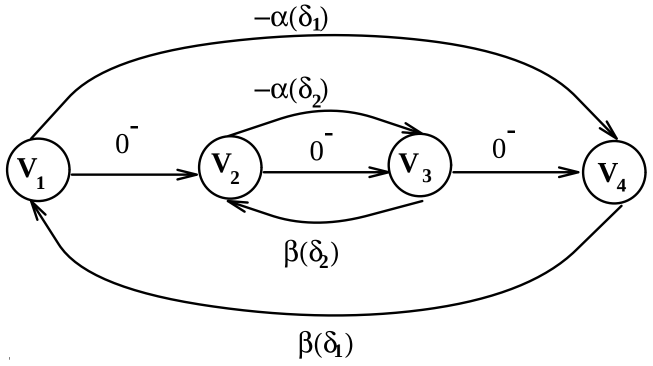

Figure 3: The weighted graph for $\sigma = b_1 b_2 e_2 e_1$

Checking feasibility of this system of inequalities can be reduced to detecting negative cost cycles in a weighted directed graph. Let us first consider an example.

Example 2 Consider two processes P_1 and P_2. The process P_1 has a delay δ_1, and the process P_2 has a delay δ_2. Let b_i and e_i denote the beginning and ending of the respective delays of P_i. Consider the string $\sigma = (b_1, b_2, e_2, e_1)$. Checking consistency of σ corresponds to testing consistency of the following set of inequalities:

$$t_1 < t_2 < t_3 < t_4, \quad \alpha(\delta_1) \leq (t_4 - t_1) \leq \beta(\delta_1), \quad \alpha(\delta_2) \leq (t_3 - t_2) \leq \beta(\delta_2).$$

The graph associated with these inequalities is shown in Figure 3. The graph has a node v_i for each t_i, and each inequality gives rise to an edge. Thus, the constraint $\alpha(\delta_1) \leq (t_4 - t_1) \leq \beta(\delta_1)$ gives rise to two edges: a forward edge from v_1 to v_4 with weight $-\alpha(\delta_1)$ and a backward edge from v_4 to v_1 with weight $+\beta(\delta_1)$. Similarly the constraint $\alpha(\delta_2) \leq (t_3 - t_2) \leq \beta(\delta_2)$ gives rise to two edges. For every constraint $t_i < t_{i+1}$, i.e. $t_i - t_{i+1} < 0$, there is an edge from v_i to v_{i+1} with weight 0^-. The superscript "$-$" indicates that, on account of the strict inequality, nonzero time must have elapsed.

The cost of a cycle is the sum of the costs of its edges. While adding the costs we need to account for the superscripts also: the sum of two costs has the superscript "$-$" iff one of the costs has the superscript "$-$". It is easy to see that the string σ is timing inconsistent iff the graph has a negative cost cycle (the cost 0^- is considered to be negative). Assuming $\alpha(\delta_1) \leq \beta(\delta_1)$ and $\alpha(\delta_2) \leq \beta(\delta_2)$, the graph has a negative cost cycle iff $\alpha(\delta_2) \geq \beta(\delta_1)$. Note that if $\alpha(\delta_2) = \beta(\delta_1)$, then there is a negative cost cycle with cost 0^-. ■

The domain of bounds

The above example illustrates that the cost of an edge reflects whether the corresponding constraint is a strict or a nonstrict inequality. In order to deal with different types of bounds uniformly, we define the domain of bounds, similar to [Dil89], to be the set

$$\mathcal{B} \;=\; \{\ldots -2, -1, 0, 1, 2, \ldots\} \;\cup\; \{\ldots -2^-, -1^-, 0^-, 1^-, 2^-, \ldots\} \;\cup\; \{-\infty, \infty\}.$$

The costs of the edges of the graph will be from the domain $\mathcal{B}$. To compute shortest paths, we need to add costs and compare costs. The ordering $<$ over the integers is extended to $\mathcal{B}$ by the following law: for any integer a, $-\infty < a^- < a < (a+1)^- < \infty$. The addition operation $+$ over integers is extended to $\mathcal{B}$ by: (i) for all $b \in \mathcal{B}$, $b + \infty = \infty$, (ii) for all $b \in \mathcal{B}$ with $b \neq \infty$, $b + (-\infty) = -\infty$, and (iii) for integers a and b, $a + b^- = a^- + b = a^- + b^- = (a+b)^-$.

Now the constraints corresponding to a sequence σ can be rewritten as follows. A constraint of the form $t_i < t_{i+1}$ is written as $t_i - t_{i+1} \leq 0^-$. A constraint of the form $a_1 \leq t_k - t_j \leq b_1$ gives rise to two constraints: $t_k - t_j \leq b_1$ and $t_j - t_k \leq -a_1$.

The weighted graph $G(\mathbf{x})$

For a sequence $\mathbf{x}$, now we define an infinite weighted digraph $G(\mathbf{x})$, where the costs of the edges are from the set $\mathcal{B}$, as follows. The graph $G(\mathbf{x})$ has a node v_i for every variable t_i; for each constraint $t_j - t_k \leq b$, $b \in \mathcal{B}$, there is an edge from v_j to v_k with cost b. Thus if a process begins its delay at x_i with a matching end at x_j, then the graph has a forward edge from v_i to v_j with negative cost showing the lower bound and a backward edge from v_j to v_i with positive cost showing the upper bound. For a string σ of length N, a weighted graph $G(\sigma)$ with N vertices is defined similarly. In both cases, the problem of checking consistency reduces to finding negative cost cycles in the associated digraph.

Proposition 1 *The sequence $\mathbf{x}$ is timing inconsistent iff the graph $G(\mathbf{x})$ has a negative cost cycle.*

Proof. First observe that $\mathbf{x}$ is timing inconsistent iff there is some finite prefix σ of $\mathbf{x}$ such that σ is timing inconsistent. Also, $G(\mathbf{x})$ has a negative cost cycle iff $G(\sigma)$ has for some finite prefix σ of $\mathbf{x}$. Hence, it suffices to show that a string σ is timing inconsistent iff the graph $G(\sigma)$ has a negative cost cycle. This follows from the theory linear programming: the problem of computing shortest paths in $G(\sigma)$ is the dual of finding consistent assignment of time values to the events in σ (see, for instance, [PS82]). ■

3.2 Testing consistency of strings

For a string σ of length N, the graph $G(\sigma)$ is finite with N vertices and $O(N{\cdot}K)$ edges, where K is the number of processes. There exist standard polynomial-time algorithms to detect negative cost cycles. However, the best known time complexity is $O(N^2 \cdot K)$. Since N is usually much larger than K, we prefer an alternative solution with time complexity $O(K^2 \cdot N)$.

The reduced graph $G^*(\sigma)$

Consider a string $\sigma = x_1 x_2 \ldots x_N$. The graph $G(\sigma)$ has N vertices $v_1, v_2, \ldots v_N$. Let $V_b^*(\sigma)$ consist of vertices v_j such that some process P_i finishes a delay δ at x_j with no prior matching beginning of the delay (i.e., $e_i(x_j) = \delta$ and for all $1 \leq k < j$, both $b_i(x_k)$ and $e_i(x_k)$ are undefined). Similarly, let $V_e^*(\sigma)$ consist of vertices v_j such that some process P_i begins a delay δ at x_j and there is no matching end of the delay (i.e., $b_i(x_j) = \delta$ and for all $j < k \leq N$, both $b_i(x_k)$ and $e_i(x_k)$ are undefined). Now consider the string $\sigma\sigma'$ and the graph $G(\sigma\sigma')$. The graphs $G(\sigma)$ and $G(\sigma')$ are subgraphs of $G(\sigma\sigma')$. An edge in $G(\sigma\sigma')$ is either in $G(\sigma)$, or in $G(\sigma')$, or it connects the last vertex of $G(\sigma)$ to the first vertex of $G(\sigma')$, or it connects some vertex in $V_e^*(\sigma)$ to some vertex in $V_b^*(\sigma')$.

For a string σ, let $V^*(\sigma)$ be the union $V_b^*(\sigma) \cup V_e^*(\sigma) \cup \{v_1, v_N\}$. Observe that the size of $V^*(\sigma)$ is at most $(2K+2)$. Now consider a superstring σ' (i.e., $\sigma' = \sigma_1\sigma\sigma_2$ for some strings σ_1 and σ_2). If a vertex v in the subgraph $G(\sigma)$ and a vertex u not in $G(\sigma)$ are the end-points of an edge in $G(\sigma')$, then v must be in $V^*(\sigma)$. Thus the vertices not in $V^*(\sigma)$ are "internal" to the subgraph $G(\sigma)$.

From the graph $G(\sigma)$ let us define another weighted graph $G^*(\sigma)$, called the *reduced graph* of σ, as follows: the vertex set is $V^*(\sigma)$, and for every pair of vertices v_j and v_k in $V^*(\sigma)$ there is an edge from v_j to v_k with cost equal to the cost of the shortest path from v_j to v_k in the graph $G(\sigma)$ (note that this cost can be ∞ if there is no path from v_j to v_k, and can be $-\infty$ if there is no "shortest" path because of a negative cost cycle). Thus, the graph $G^*(\sigma)$ is obtained from $G(\sigma)$ by first computing the shortest paths and then discarding the internal vertices. Thus, if we replace the subgraph $G(\sigma)$ by $G^*(\sigma)$ in $G(\sigma')$, the resulting graph has a negative cost cycle iff $G(\sigma')$ does.

Constructing the reduced graph

Using these ideas, the consistency of strings can be checked efficiently using a dynamic programming approach. The outline of the algorithm is shown in Figure 4. Given a string σ, it checks if the graph $G(\sigma)$ has a negative cost cycle, and if not, computes the reduced graph $G^*(\sigma)$. While implementing the algorithm, a graph will be represented by a matrix that gives, for every pair of vertices, the cost of the edge connecting them (the entries in the matrix are from the domain $\mathcal{B}$). If there are multiple edges connecting two vertices then we need to consider only the one with the least cost.

Input: A string $\sigma = x_1 \dots x_N$.
Output: Decides if σ is timing consistent, and outputs the reduced graph $G^*(\sigma)$.
Algorithm:

Initially: Let G be the graph with a single vertex v_1.
For $j := 2$ **to** N **do**
{ **Comment**: G equals $G^*(x_1 \dots x_{j-1})$.}
1. To G, add the vertex v_j and all edges of $G(\sigma)$ that connect v_j with the vertices of G.
2. Compute new shortest distances within G; if a negative cost cycle is detected, stop (σ is inconsistent).
3. Remove all the vertices not in $V^*(x_1 \dots x_j)$.

Figure 4: Algorithm for testing consistency of finite strings

Consider a matrix A representing the reduced graph $G^*(x_1 \dots x_{j-1})$. Step 1 corresponds to adding an extra row and column to A. At step 2, we need to check if the updated matrix has a negative cost cycle, and if not, compute the new shortest distances. Observe that, for any pair of vertices v and v', the new shortest distance between v and v' is different from the old one, only if the new shortest path visits the new vertex v_j. This fact can be used to compute the new shortest distances efficiently: in time $O(m^2)$, where m is the number of vertices in the current graph. Step 3 ensures that the updated matrix A stores only the vertices that are external to $x_1, \dots x_j$, and hence at most $(2K+2)$ vertices. Thus, the overall time complexity of the algorithm is $O(N \cdot K^2)$.

Theorem 2 *The problem of deciding whether a string σ is timing consistent can be solved in time $O(|\sigma| \cdot K^2)$.*

3.3 Testing consistency of sequences

If the language inclusion $\mathcal{L}(P) \subseteq \mathcal{L}(T)$ fails, since P and T have finite state spaces, we can find a sequence $\mathbf{x} \in \mathcal{L}(P) \backslash \mathcal{L}(T)$ of the form $\mathbf{x} = \sigma' \sigma^\omega$ (for strings $\sigma', \sigma \in \Sigma^+$).

First observe that it is possible that $\sigma'\sigma$ is consistent, while for some i, $\sigma'\sigma^i$ is not.

Example 3 Consider two processes P_1 and P_2. The process P_1 has a delay δ_1, and the process P_2 has a delay δ_2. We use b_i and e_i to denote the beginning and ending of the delays of the processes P_i, as in Example 1. Let

$$\begin{aligned} \sigma' &= (b_1) \\ \sigma &= (b_2, \{e_1, b_1\}, e_2). \end{aligned}$$

If $\alpha(\delta_1) = \beta(\delta_1) = n$ and $\alpha(\delta_2) = \beta(\delta_2) = n+1$, it is easy to see that $\sigma'\sigma^i$ is consistent iff $i \leq n$. ■

Now we proceed to develop an algorithm for checking consistency of $\mathbf{x}$. We start by showing how to reduce the problem of checking consistency of a sequence to checking consistency of its subsequences.

Combining two reduced graphs

Consider a string σ_1 that is the concatenation of two strings σ_2 and σ_3. If either σ_2 or σ_3 is timing inconsistent then so is σ_1. Consider the reduced graphs $G^*(\sigma_2)$ and $G^*(\sigma_3)$. We can put these two graphs together by connecting the vertices in $V_e^*(\sigma_2)$ to the vertices in $V_b^*(\sigma_3)$ by the appropriate edges (i.e., by connecting the begin-delay events in σ_2 with their matching end-delay events in σ_3), and also connecting the last vertex in $G^*(\sigma_2)$ to the first vertex of $G^*(\sigma_3)$. If the resulting graph has a negative cycle, then σ_1 is timing inconsistent. Otherwise, we can compute the shortest distances in this new graph, and then delete the vertices not in $V^*(\sigma_1)$ to obtain $G^*(\sigma_1)$. Recall that the reduced graphs contain only $O(K)$ vertices irrespective of the length of the string, and computing shortest distances in a graph can be done in time cubic in the number of its vertices. Hence, given $G^*(\sigma_2)$ and $G^*(\sigma_3)$, we can compute $G^*(\sigma_1)$ in time $O(K^3)$.

In the sequel, we will need to compute graphs of the form $G^*(\sigma^m)$. Observe that $G^*(\sigma)$ can be computed in time $O(|\sigma| \cdot K^2)$, and the reduced graph $G^*(\sigma^m)$ can be computed from $G^*(\sigma^{m/2})$ in time $O(K^3)$. This leads to the following bound on computing $G^*(\sigma^m)$.

Lemma 3 *For a string σ, the reduced graph $G^*(\sigma^m)$ can be computed in time $O(|\sigma| \cdot K^2 + K^3 \cdot \lceil \log m \rceil)$.*

Now consider a sequence $\mathbf{x}$ that is the concatenation of a string σ' and a sequence $\mathbf{y}$. In the previous section, we defined reduced graphs for strings. We can define reduced graphs for sequences in an analogous fashion: for a sequence $\mathbf{y}$, $V^*(\mathbf{y})$ is $V_b^*(\mathbf{y})$ plus the first vertex of $G(\mathbf{y})$. Now we can test consistency of $\mathbf{x}$ by putting together the reduced graphs $G^*(\sigma')$ and $G^*(\mathbf{y})$. From this, it follows that

> If there is an algorithm which tests whether σ^ω is timing consistent, and if so, computes the reduced graph $G^*(\sigma^\omega)$, then it is possible to check timing consistency of $\sigma'\sigma^\omega$ with additional time complexity $O(|\sigma'| \cdot K^2 + K^3)$.

Computing the shortest distances within the periodic graph $G(\sigma^\omega)$

We compute the periodic graph $G(\sigma^\omega)$ by considering the concatenation of infinitely many copies of the graphs $G^*(\sigma)$. Suppose $G^*(\sigma)$ has m vertices, $m \leq (2K+2)$, and let us denote the vertices in the j-th copy of $G^*(\sigma)$ by $v_1^j, \ldots v_m^j$. For $l, k > 0$, we will use G_l^k to denote the subgraph of $G(\sigma^\omega)$ consisting of l copies of $G^*(\sigma)$ starting from k-th copy, and G^k to denote the subgraph consisting of infinite number of copies starting from k-th copy. It should be clear that for every k, k', l, the graphs G_l^k and $G_l^{k'}$ are isomorphic, and the graphs G^k and $G^{k'}$ are also isomorphic.

Note that all the vertices of $G^*(\sigma^\omega)$ lie in the first copy. Hence, to compute $G^*(\sigma^\omega)$ we need to compute the shortest distance between every pair of vertices v_i^1 and v_j^1. Let $d^k(i,j)$ denote the shortest distance between v_i^1 and v_j^1 in the subgraph G_k^1 (i.e., the graph with only the first k copies), and let $d(i,j)$ be the shortest distance between v_i^1 and v_j^1 in the entire graph $G(\sigma^\omega)$. Equivalently, $d^k(i,j)$ is the shortest distance between v_i^l and v_j^l in the subgraph G_k^l, and $d(i,j)$ the shortest distance between v_i^l and v_j^l in the subgraph G^l, for any $l > 0$.

The question is whether $d(i,j)$ can be determined by computing the values of $d^k(i,j)$ up to a small k. The following lemma gives the answer in the case where there is a shortest path between v_i^1 and v_j^1 (i.e., the case where $d(i,j) \neq -\infty$). It shows that if the shortest distance between v_i^1 and v_j^1 can be reduced by considering additional copies of $G(\sigma)$ beyond m^2 copies, then, in fact, it can be reduced repeatedly by "pumping" appropriate paths, and hence, is unbounded.

Lemma 4 *For every pair of vertices v_i^1 and v_j^1, either $d(i,j)$ is $-\infty$ or equals $d^{m^2}(i,j)$.*

Proof. Consider a pair of vertices v_i^1 and v_j^1 such that $d(i,j) < d^{m^2}(i,j)$. We will show that, in such a case, $d(i,j)$ is $-\infty$. Let ρ be a smallest, in terms of the number of edges it contains, path between v_i^1 and v_j^1 such that the cost of ρ is less than $d^{m^2}(i,j)$. Let us denote this cost by $c(\rho)$.

We use ρ to define a parent-child relationship over pairs of indices. A pair (i'', j'') is said to be a child of (i', j') iff for some k, ρ can be written as

$$v_i^1 \xrightarrow{\rho_1} v_{i'}^k \xrightarrow{\rho_1'} v_{i''}^{k+1} \xrightarrow{\rho_2'} v_{j''}^{k+1} \xrightarrow{\rho_3'} v_{j'}^k \xrightarrow{\rho_3} v_j^1$$

such that the path ρ_2' lies entirely in the subgraph G^{k+1}, and the path $\rho_1'\rho_2'\rho_3'$ lies entirely in the subgraph G^k. Now consider a tree over the pairs of indices: the root node is (i,j), and the edges are defined by the above child relationship. Since the path ρ visits at least m^2+1 copies, this tree has a path containing m^2+1 nodes. Since there are only m^2 vertex pairs, it follows that there exists a pair (i', j') that is a descendant of itself. That is, the path ρ can be split as:

$$v_i^1 \xrightarrow{\rho_1} v_{i'}^k \xrightarrow{\rho_1'} v_{i'}^{k'} \xrightarrow{\rho_2'} v_{j'}^{k'} \xrightarrow{\rho_3'} v_{j'}^k \xrightarrow{\rho_3} v_j^1$$

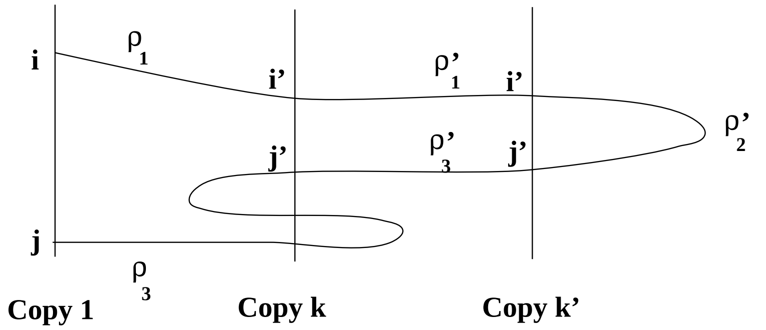

Figure 5: The splitting of the path ρ

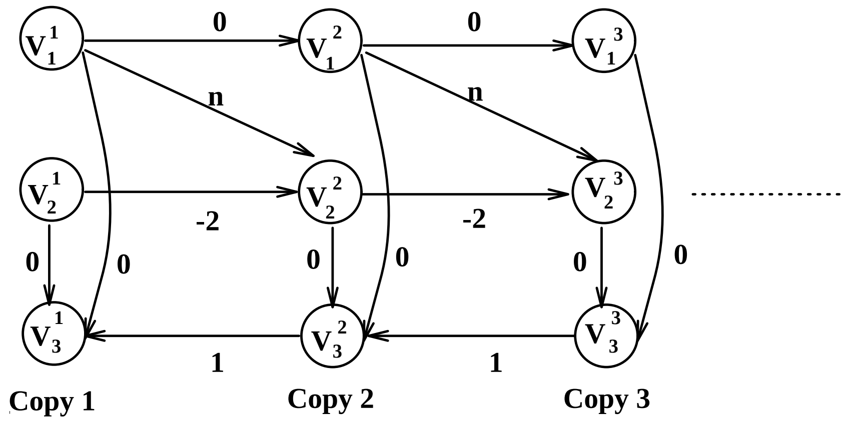

Figure 6: The periodic graph of Example 4

such that the path ρ_2' lies entirely within the subgraph $G^{k'}$ and the path $\rho_2 = \rho_1'\rho_2'\rho_3'$ lies entirely within G^k (see Figure 5).

Recall that the subgraphs G^k and $G^{k'}$ are isomorphic, and hence ρ_2' can be considered to be a path between $v_{i'}^k$ and $v_{j'}^k$ (to be precise, the superscripts of the vertices appearing on ρ_2' need to be shifted by $(k'-k)$, but we will slightly abuse the notation). In fact, for every $n \geq 0$, we have the path $(\rho_1')^n\rho_2'(\rho_3')^n$ between $v_{i'}^k$ and $v_{j'}^k$.

Now consider the path $\rho_1\rho_2'\rho_3$ between v_i^1 and v_j^1. If $c(\rho_1\rho_2'\rho_3) \leq c(\rho)$ then we have a path from v_i^1 and v_j^1 with cost less than $d^{m^2}(i,j)$ and with less edges than ρ. Hence, by the choice of ρ, $c(\rho_1\rho_2'\rho_3) > c(\rho)$. This implies that $c(\rho_1') + c(\rho_3') < 0$. This means that the segments ρ_1' and ρ_3' can be "pumped" to reduce the cost further and further: $c((\rho_1')^n\rho_2'(\rho_3')^n)$ forms a strictly decreasing sequence with increasing values of n. This implies that there is no "shortest" path between the vertices $v_{i'}^k$ and $v_{j'}^k$, and hence, $d(i,j) = -\infty$. ■

The next question is to determine the pairs of vertices v_i^1 and v_j^1 for which there is no "shortest" path and $d(i,j)$ is $-\infty$. Clearly, if $d^{2m^2}(i,j) < d^{m^2}(i,j)$ then $d(i,j)$ is $-\infty$ by the above lemma. But the converse of this statement does not hold.

Example 4 Consider the graph shown in Figure 6. A single copy consists of 3 vertices. Note that the distance between the vertices v_2^1 and v_3^1 decreases at every step: $d^k(2,3) = -k+1$. Thus $d^{18}(2,3) < d^9(2,3)$, and this allows us to correctly conclude that $d(2,3) = -\infty$. The situation is quite different for the distance between v_1^1 and v_3^1: $d^k(1,3) = 0$ for $k \leq n+3$ and $d^k(1,3) = n+3-k$ for $k > n+3$. Thus the shortest distance does not change in the beginning. If the cost n exceeds 15, then $d^{18}(1,3) = d^9(1,3)$, and yet, $d(1,3) = -\infty$. Thus a different criterion is required to conclude $d(1,3) = -\infty$; it should follow from the facts that $d(2,3) = -\infty$ and there exists a finite cost edge from v_1^1 to v_2^2 and from v_3^2 to v_3^1. ■

The next lemma characterizes pairs (i,j) such that $d(i,j) = -\infty$.

Lemma 5 *Let v_i^1 and v_j^1 be a pair of vertices such that $d(i,j) = -\infty$ and $d^{m^2}(i,j) \neq -\infty$. Then there exists some pair (i',j') such that $d^{2m^2}(i',j') < d^{m^2}(i',j')$ and in the graph G^1 there exist paths from v_i^1 to $v_{i'}^k$ and from $v_{j'}^k$ to v_j^1, for some $k > 0$.*

Proof. Consider a pair of vertices v_i^1 and v_j^1 such that $d(i,j) < d^{m^2}(i,j)$. The proof is very similar to the proof of Lemma 4. Let ρ be a smallest, in terms of the number of edges, path between v_i^1 and v_j^1 such that the cost of ρ is less than $d^{m^2}(i,j)$. Let (i',j') be a pair such that the path ρ can be split as

$$v_i^1 \xrightarrow{\rho_1} v_{i'}^k \xrightarrow{\rho_1'} v_{i'}^{k'} \xrightarrow{\rho_2'} v_{j'}^{k'} \xrightarrow{\rho_3'} v_{j'}^k \xrightarrow{\rho_3} v_j^1$$

such that the path $\rho_2 = \rho_1'\rho_2'\rho_3'$ visits the least number of different copies. This added restriction on the choice of (i',j') implies that the path ρ_2 lies within the subgraph $G_{m^2}^k$. Let ρ_2'' be the shortest path between $v_{i'}$ and $v_{j'}$ that lies within m^2 copies, that is, $c(\rho_2'') = d^{m^2}(i',j')$. Now the path $\rho_1'\rho_2''\rho_3'$ lies within $2m^2$ copies, and, since $c(\rho_1') + c(\rho_3') < 0$ (as in the proof of Lemma 4), has cost less than $c(\rho_2'')$. This means that $d^{2m^2}(i',j') < d^{m^2}(i',j')$, and this proves the lemma. ■

To find all pairs (i,j) with $d(i,j) = -\infty$ using the above lemma, we construct an edge-labeled graph G_{inf} over the vertex set $\{v_1, \ldots v_m\}$ with the labels $\{a,b,c,d\}$ as follows:

1. If $G^*(\sigma)$ has an edge from v_i to v_j then G_{inf} has a-labeled edge from v_i to v_j.
2. If $d^{2m^2}(i,j) < d^{m^2}(i,j)$ then G_{inf} has b-labeled edge from v_i to v_j.
3. If G_2^1 has an edge from v_i^1 to v_j^2 then G_{inf} has c-labeled edge from v_i to v_j.
4. If G_2^1 has an edge from v_i^2 to v_j^1 then G_{inf} has d-labeled edge from v_i to v_j.

Next we define a language L_{inf} over the alphabet $\{a,b,c,d\}$ to consist of all the words w such that

1. the number of c's is equal to the number of d's;
2. in every prefix of w, the number of c's is at least as large as the number of d's;
3. there is at least one b.

From the graph G_{inf} we build the desired set S_{inf}:

$(i,j) \in S_{inf}$ iff there exists a path ρ from v_i to v_j in G_{inf} such that ρ spells a word in L_{inf}.

The next proposition follows from the previous lemmas and the definitions:

Proposition 6 *For every pair of vertices v_i^1 and v_j^1, if $(i,j) \in S_{inf}$ then $d(i,j) = -\infty$, otherwise $d(i,j) = d^{m^2}(i,j)$.*

Computing the set S_{inf}

Given an edge-labeled graph and a language L, the L-transitive closure problem is to compute all pairs of nodes (v_i, v_j) for which there exists a path from v_i to v_j that spells a word in L. Thus computing the set S_{inf} corresponds to computing the L_{inf}-transitive closure of the graph G_{inf}.

It is easy to see that the language L_{inf} can be recognized by a deterministic 1-counter machine A (a special case of a pushdown automaton). Let A be a machine which reads the symbols of the input word one by one and does the following: on symbol a it does nothing, on b it moves to a state that signifies that A has seen at least one b, on symbol c it increments the counter (staying in the same state), and on d it decrements the counter if it was positive and rejects the whole input if the counter was 0. The machine accepts if after processing the input word it has seen some b and the counter is 0. Thus, the language L_{inf} is context-free. Thus we can adopt the algorithm of [Yan90] for computing the L-transitive closure of context-free languages.

The algorithm is derived from a context-free grammar for L_{inf} in Chomsky normal form. We will describe now in more concrete terms the algorithm for our case. It uses the following grammar:

$$\begin{aligned} A &\rightarrow a \mid b \mid AA \mid A'd \\ A' &\rightarrow c \mid cA \\ B &\rightarrow b \mid BA \mid AB \mid B'd \\ B' &\rightarrow cB \end{aligned}$$

We will compute iteratively four sets A, A', B, B' of pairs of nodes. A pair (v_i, v_j) will be in A at the end of the algorithm iff there is a path from v_i to v_j which spells a word that satisfies conditions 1 and 2 in the definition of L_{inf} (but possibly not 3); it will be in A' iff the word satisfies the following modified conditions: $1'$. the number of c's is one more than the number of d's; and $2'$. in every prefix the number of c's is strictly larger than the number of d's. The sets B and B' are defined analogously except that, in addition, the word must also contain one b (i.e., satisfy condition 3 in the definition of L_{inf}). Thus, the desired set S_{inf} is given by the final value of B. These sets are represented by Boolean matrices whose rows and columns are indexed by the nodes of the graph. In addition, for each of the four sets we have a list of "unprocessed" pairs, $S_A, S_{A'}$ etc.

The data structures are initialized as follows: For every a-labeled edge, insert the corresponding pair of nodes to A (i.e., set the corresponding entry of matrix A to 1) and to the list S_A. For every b-labeled edge, insert the corresponding pair to A, B, S_A, S_B. For every c-labeled edge, insert the pair to A' and $S_{A'}$; we do nothing for the d-labeled edges.

In the iterative step, we remove a pair from one of the lists and "process" it; the algorithm terminates when the lists are empty. A pair (v_i, v_j) is processed as follows depending on the list it is taken from.

- Case 1: List S_A. For every member (v_j, v_k) of A (respectively B), if (v_i, v_k) is not in A (resp. B), then add it to A and to S_A (resp. B and S_B). For every member (v_k, v_i) of A (respectively B), if (v_k, v_j) is not in A (resp. B), then add it to A and to S_A (resp. B and S_B). For every edge (v_k, v_i) labeled c, if (v_k, v_j) is not in A', then add it to A' and to $S_{A'}$.

- Case 2: List S_B. For every member (v_j, v_k) of A, if (v_i, v_k) is not in B, then add it to B and to S_B. For every member (v_k, v_i) of A, if (v_k, v_j) is not in B, then add it to B and to S_B. For every edge (v_k, v_i) labeled c, if (v_k, v_j) is not in B', then add it to B' and to $S_{B'}$.

- Case 3: List $S_{A'}$. For every edge (v_j, v_k) labeled d, if (v_i, v_k) is not in A, then add it to A and to S_A.

- Case 4: List $S_{B'}$. For every edge (v_j, v_k) labeled d, if (v_i, v_k) is not in B, then add it to B and to S_B.

Removing a pair from a list and processing it takes time $O(m)$. Since every pair is inserted (and therefore also removed) at most once in each list, it follows that the time complexity is $O(m^3)$.

Lemma 7 *The set of pairs S_{inf} can be computed in time $O(m^3)$.*

Algorithm for testing consistency of x

Now we can put together all the pieces to obtain the algorithm of Figure 7. Algorithm of Figure 4 is used to test the consistency of σ' and σ, and to compute the reduced graphs $G^*(\sigma')$ and $G^*(\sigma)$. Step 3 takes time $O(m^3 \cdot \lceil \log m \rceil)$. Computing the set S_{inf} at step 4 can be performed in time $O(m^3)$ as outlined earlier. Combining the two graphs $G^*(\sigma')$ and $G^*(\sigma^\omega)$, and testing for negative cost cycles is easy. This gives the following theorem:

Theorem 8 *The problem of deciding whether a sequence $\sigma'\sigma^\omega$ is timing consistent is solvable in time $O((|\sigma'| + |\sigma|) \cdot K^2 + K^3 \cdot \lceil \log K \rceil)$.*

Input: Two strings σ' and σ.
Output: Decides if $\sigma'\sigma^\omega$ is timing consistent.
Algorithm:

1. If σ' is timing inconsistent then stop
   else compute $G^*(\sigma')$.
2. If σ is timing inconsistent then stop
   else compute $G^*(\sigma)$ (with vertex set $\{v_1, \cdots v_m\}$).
3. Compute $G^*(\sigma^{m^2})$ and $G^*(\sigma^{2m^2})$.
4. Compute the set of pairs S_{inf}.
5. Construct $G^*(\sigma^\omega)$ using the rule:
   If $(i,j) \in S_{inf}$ then $d(i,j) = -\infty$ else $d(i,j) = d^{m^2}(i,j)$.
6. Check if $G^*(\sigma')$ connected with $G^*(\sigma^\omega)$ has a negative cost cycle.

Figure 7: Algorithm for testing consistency of infinite sequences

4 Finding the optimal delay constraint

Given a delay constraint $D = (\Delta, \alpha, \beta)$ it is possible to construct an automaton P_D that accepts precisely those sequences that are timing-consistent with respect to D. For a set $\Delta \subseteq \Delta_P$, let $I(\Delta)$ be the set $\{i \mid \Delta \cap \Delta_i \neq \emptyset\}$, and for a delay δ, let $\gamma(\delta)$ be $\beta(\delta)$ when it is not ∞, and $\alpha(\delta)$ otherwise.

Consider a delay constraint $D = (\Delta, \alpha, \beta)$. First we define an automaton R_D with infinitely many states such that R_D accepts only those sequences that are timing consistent with D. A state of R_D records, for each process in $I(\Delta)$, which delay is "active", that is, which delay has begun, but not ended. If no such delay is active for a process, we use a special marker $\perp$. In addition, with each process, we associate a timer that records the time elapsed since the beginning of the delay; whenever a delay begins, the corresponding timer is reset to 0, and at each step, all the timer values are incremented by some amount. A delay can end only if the value of its timer is within the bounds specified by the maps α and β. Formally, a state s of R_D is a $|I(\Delta)|$-tuple; for $i \in I(\Delta)$, s has a component s_i which is a pair (d_i, ν_i) where $d_i \in \{\perp\} \cup (\Delta \cap \Delta_i)$ and ν_i is a nonnegative real number. In the initial state each component is $(\perp, 0)$. There is an edge from state s to state s' labeled with event $x \in \Sigma$ iff there exists a positive real number t such that for each $i \in I(\Delta)$,

- if $b_i(x) = \delta$ and $\delta \in \Delta$ then $d'_i = \delta$ and $\nu'_i = 0$.
- if $d_i = \delta$ and $e_i(x) = \delta$ then $\alpha(\delta) \leq d_i + t \leq \beta(\delta)$.
- if both $b_i(x)$ and $e_i(x)$ are undefined then $d'_i = d_i$ and $\nu'_i = \nu_i + t$.
- if $e_i(x)$ is defined and $b_i(x)$ is undefined, then $d'_i = \perp$.

The above rules encode the semantics of timing consistency, and it follows that a sequence $\mathbf{x}$ is timing consistent iff R_D can read it starting from its initial state. Of course, we cannot use R_D as a generator for the language $\mathcal{L}_D(\mathbf{1})$, since it has infinitely many states. The region automaton construction defines an equivalence relation $\sim$ over the states of R_D. Define $s \sim s'$ iff

- the delay components of s and s' are identical: $d_i = d'_i$ for each $i \in I(\Delta)$,
- the two states agree on the ordering of the fractional parts of the timer values: for $i, j \in I(\Delta)$, the fractional part of ν_i is less than that of ν_j iff the fractional part of ν'_i is less than that of ν'_j, and the fractional part of ν_i is zero iff that of ν'_i is zero.
- the two states agree on the integral parts of the timer values: for each $i \in I(\Delta)$, either $d_i = \perp$ or the truncated values of ν_i and ν'_i are either identical or both exceed $\gamma(d_i)$.

The equivalence relation $\sim$ has the following two important properties:

- The number of equivalence classes of $\sim$ is only finite, and bounded by

$$|I(\Delta)|! \cdot 2^{|I(\Delta)|} \cdot \prod_{i \in I(\Delta)} (|\Delta_i \cap \Delta| + 1) \cdot max_{\delta \in \Delta_i \cap \Delta}\{\gamma(\delta) + 1\}.$$

- The relation $\sim$ is a bisimulation relation on the states of R_D; that is, if $s \sim s'$ and if R_D has an edge from s to u labeled with an event x then for some state u' with $u \sim u'$, R_D also has an edge from s' to u' labeled with x.

The region automaton P_D is the quotient of R_D with respect to the equivalence relation $\sim$: the states of P_D are the equivalence classes of $\sim$, and P_D has an edge from a class v to a class v' labeled with an event x iff R_D has an edge from some state in v to some state in v' with label x. Since $\sim$ is a bisimulation relation, it follows that $\mathcal{L}(R_D) = \mathcal{L}(P_D)$. We refer the reader to [AD90, ACD93] for proofs and details.

In the algorithm of Figure 2, after finding that $\mathbf{x}$ is timing inconsistent with the delay constraint $D = (\Delta_P, \alpha, \beta)$, the next step is to find an optimal delay constraint, namely a delay constraint $D(X)$ with the size of the region automaton $P_{D(X)}$ as small as possible, subject to:

1. each $\mathbf{y} \in X$ is timing inconsistent with $D(X)$, and
2. $\mathcal{L}(P_D) \subseteq \mathcal{L}(P_{D(X)})$.

Notice that the condition 2 ensures that to prove that the model augmented with P_D satisfies the specification, it suffices to prove that the model augmented with $P_{D(X)}$ satisfies the specification.

We find the desired delay constraint in two steps: in the first step we find a small set Δ_X of delays such that each $\mathbf{y} \in X$ is timing inconsistent with the delay constraint $(\Delta_X, \alpha, \beta)$; and in the second step we try to modify the bounds α and β to obtain $D(X) = (\Delta_X, \alpha_X, \beta_X)$. Our approach does not guarantee the minimality of the size of $P_{D(X)}$; it is only a heuristic to reduce the size.

4.1 Reducing the set of delays

First observe that, if $D = (\Delta, \alpha, \beta)$ and $D' = (\Delta', \alpha, \beta)$ with $\Delta' \subseteq \Delta$, then $\mathcal{L}_D(\mathbf{1}) \subseteq \mathcal{L}_{D'}(\mathbf{1})$. Thus we can discard delays that do not contribute to the inconsistency of the counter-examples in X. Consequently, we try to find a minimal set of delays that is necessary for the timing inconsistency of all $\mathbf{y} \in X$. This is done in two steps.

First we find a set Δ of delays such that each $\mathbf{y} \in X$ is timing inconsistent with (Δ, α, β), and Δ involves the least number of processes, that is, $|I(\Delta)|$ is minimum. We, therefore, look for a minimum size subset $I \subseteq \{1, ..., K\}$, such that each $\mathbf{y} \in X$ is inconsistent also with the delay constraint $D_I = (\cup_{i \in I} \Delta_i, \alpha, \beta)$. That is to say, if we run the algorithm of Section 3 ignoring the delay events of the processes not in I, we should still end up with timing inconsistency for each counter-example in X. The problem of finding such a subset I of minimum size can be easily shown to be NP-hard. Therefore, we exhaustively consider subsets of $\{1, ..., K\}$ in order of increasing size, starting with subsets of size 2. If the smallest I has size n, then the time complexity increases by a factor of $\min\{2^K, K^n\}$.

Having identified the minimal set I of processes, the second step is to find a minimal subset Δ_X of $\cup_{i \in I} \Delta_i$ preserving the timing-inconsistency of all $\mathbf{y} \in X$. This is again done by an exhaustive search over all the subsets of $\cup_{i \in I} \Delta_i$. Clearly, the set Δ_X consists of only the delays corresponding to the edges involved in the negative cost cycles.

4.2 Relaxing the bounds

Having identified the optimal set Δ_X of delays, we want to adjust the bounding functions α and β so as to reduce the sizes of the constants.

We start with a simple observation that dividing all bounds by a common factor does not affect timing consistency. Let D be a delay constraint (Δ, α, β), and k be the greatest common divisor of all the constants bounding the delays in Δ. Define new lower and upper bounds for the delays by: $\alpha'(\delta) = \alpha(\delta)/k$ and $\beta'(\delta) = \beta(\delta)/k$ (let ∞/k be ∞). It is easy to prove that $\mathcal{L}_D(\mathbf{1}) = \mathcal{L}_{D'}(\mathbf{1})$. This property can be used to reduce the size of the region automaton. Instead of using the delay constraint $D = (\Delta_X, \alpha, \beta)$ we use scaled down versions of α and β, and construct the region automaton $P_{D'}$. If the greatest common divisor k is large, this leads to a big saving: the size of $P_{D'}$ is smaller than that of P_D by a factor of $k^{|I(\Delta_X)|}$.

It is unlikely that we can apply the optimization of dividing by the greatest common divisor, by itself. However, the situation may improve dramatically if we "preprocess" the bounding functions by relaxing the lower and upper bounds. Again let $D = (\Delta, \alpha, \beta)$ be a delay constraint. Consider a delay $\delta \in \Delta$ with lower bound a and upper bound b. Suppose we replace these bounds by $a' \leq a$ and $b' \geq b$, respectively; that is, we relax the delay bounds by decreasing the lower bound and increasing the upper bound. Let D' be the new delay constraint. It is obvious that any sequence that is timing-consistent with D is also consistent with D' (but not vice versa). Hence, $\mathcal{L}_D(\mathbf{1}) \subseteq \mathcal{L}_{D'}(\mathbf{1})$. However, if we use D' obtained by this transformation as it is, there is no computational benefit; in fact, since we are increasing the upper bounds the size of the region automaton increases. But note that the scaling transformation may be applicable to the new delay bounds in a more effective way than it was to the original bounds. Thus the objective of changing the bounds is to make them all integral multiples of some large common factor. However we should not relax the bounds too much: in particular, we require that the counter-examples in X are timing inconsistent with respect to D' also. This can be easily understood by an example:

Example 5 Consider two delays: delay δ_1 of P_1 with lower bound 0 and upper bound 2, and delay δ_2 of P_2 with lower bound 5 and upper bound ∞. Suppose in the counter-example $\mathbf{x}$ both P_1 and P_2 begin their delays at the first step, and end their delays at the second step: $b_1(x_1) = e_1(x_2) = \delta_1$, and $b_2(x_1) = e_2(x_2) = \delta_2$. Clearly this scenario is timing inconsistent. If we construct a region automaton, the number of states is $2{\cdot}5{\cdot}l$ (for some l). To reduce the size, we first replace the lower bound $\alpha(\delta_2)$ by 4 which imposes a weaker bound. Now we can divide all bounds by their common factor, 2. Then δ_1 has lower bound 0 and upper bound 1, whereas δ_2 had lower bound 2 and upper bound ∞. The number of states in the new region automaton is $1{\cdot}2{\cdot}l$, a saving by a factor of 5. Note that had we replaced the original lower bound for δ_2 by 2, we could have eventually reduced all bounds to 1 after scaling. But this would not have been helpful because $\mathbf{x}$ would have been timing-consistent with the new constraints. ■

For a delay constraint $D = (\Delta, \alpha, \beta)$, define a set S_D to be the union of $\{\alpha(\delta) \mid \delta \in \Delta\}$ and $\{\beta(\delta) \mid \delta \in \Delta \text{ and } \beta(\delta) \neq \infty\}$. For a set S of nonnegative integers let $gcd(S)$ denote the greatest common divisor of the numbers in S. Now the optimization problem can be stated precisely as follows.

> Given a delay constraint $D = (\Delta, \alpha, \beta)$ and a set X of sequences such that each $\mathbf{y} \in X$ is timing inconsistent with D, find another delay constraint $D' = (\Delta, \alpha', \beta')$ such that
>
> 1. for each $\delta \in \Delta$, $0 \leq \alpha'(\delta) \leq \alpha(\delta)$,
> 2. for each $\delta \in \Delta$, $\beta'(\delta) \geq \beta(\delta)$ ($\beta'(\delta)$ may be ∞),
> 3. each $\mathbf{y} \in X$ is timing inconsistent with D',
>
> so as to minimize the maximum of the set $\{n/gcd(S_{D'}) \mid n \in S_{D'}\}$.

To solve the problem the following observations are useful. First notice that if $\alpha'(\delta) = 0$ for each $\delta \in \Delta$, then every sequence $\mathbf{y}$ is timing consistent with D'. Hence, $\alpha'(\delta) \geq 1$ for at least one delay δ. Now the requirement that $\alpha'(\delta) \leq \alpha(\delta)$ for all delays δ ensures that $gcd(\{\alpha'(\delta) \mid \delta \in \Delta\})$ cannot exceed the maximum of $\{\alpha(\delta) \mid \delta \in \Delta\}$. This gives us

> The greatest common divisor of the set $S_{D'}$, where D' is the optimal delay constraint, cannot exceed the maximum of $\{\alpha(\delta) \mid \delta \in \Delta\}$.

Now suppose that the greatest common divisor of $S_{D'}$ for the optimal constraint D' is k. The number of choices for the optimal α' and β' are finite. For any delay $\delta \in \Delta$, either

- $\beta'(\delta)$ is ∞, and in this case the possible choices for $\alpha'(\delta)$ are $0, k, 2k, \ldots, k \cdot \lfloor \alpha(\delta)/k \rfloor$, or
- $\beta'(\delta)$ is $k \cdot \lceil \beta(\delta)/k \rceil$ and $\alpha'(\delta)$ is $k \cdot \lfloor \alpha(\delta)/k \rfloor$.

Thus, we need to consider only finitely many choices for α' and β'; the number of choices is

$$\Sigma_{k=1,\ldots max(\alpha(\delta)|\delta\in\Delta)}\Pi_{\delta\in\Delta}(\lfloor\alpha(\delta)/k\rfloor+2).$$

For each choice of α' and β', we need to check whether each $\mathbf{y} \in X$ is timing consistent with D', and choose the D' that maintains inconsistency of all counter-examples and minimizes $max\{n/gcd(S_{D'}) \mid n \in S_{D'}\}$. From this optimal D', we obtain α_X and β_X: for $\delta \in \Delta_X$, set $\alpha_X(\delta) = \alpha'(\delta)/gcd(S_{D'})$, and $\beta_X(\delta) = \beta'(\delta)/gcd(S_{D'})$. This gives the optimal delay constraint $D(X) = (\Delta_X, \alpha_X, \beta_X)$, and then we construct the corresponding region automaton.

Now we can conclude that there is an upper bound on the possible choices of $D(X)$, and hence, a bound on the number of iterations before termination. Since Δ_X is a subset of Δ_P, the set of all delays, there are only $2^{|\Delta_P|}$ choices for it, and for a fixed choice of Δ_X, the optimization of relaxing the bounds generates only finitely many choices of α' and β'. This gives the following theorem:

Theorem 9 *The algorithm of Figure 2 terminates in at most*

$$\Sigma_{\Delta\subseteq\Delta_P}\Sigma_{k=1,\ldots max(\alpha(\delta)|\delta\in\Delta)}\Pi_{\delta\in\Delta}(\lfloor\alpha(\delta)/k\rfloor+2)$$

number of iterations.

Thus the worst case number of iterations is exponential in the number of delays, and also proportional to the magnitudes of constants in the lower bounds. It may be worthwhile to consider a slightly different strategy for choosing the sets Δ_X of delays, namely, never removing elements of Δ_X from one iteration to the next. In this case at most $|\Delta_P|$ different delay sets will ever be considered instead of $2^{|\Delta_P|}$, and hence, potentially reducing the number of iterations by an exponential factor. This strategy may, however, result in constructing bigger region automata. We have not yet done sufficient experiments to determine which strategy is better in practice.

5 Implementation and examples

We have implemented the proposed scheme in the verifier Cospan [HK90]. Cospan is a tool for verifying logical properties of a system modeled as a collection of coordinating processes. The system is specified in a language called S/R [AKS83]. The syntax of S/R allows one to specify the association between the state-transitions of individual processes and the beginning and ending of various delay events. It was straightforward to write the code to extract from an S/R program the sets Δ_i of delays, and the maps b_i, e_i, α, and β. At each iteration step, Cospan performs the language inclusion test, and reports a counter-example if the inclusion does not hold. Our method gives encouraging results, provided the search is performed in a breadth-first fashion, thereby producing relatively short counter-examples.

Once the counter-example is obtained, it is tested for timing consistency. The next step is to obtain the optimal delay constraint $D(X)$. Having identified $D(X)$, we generate the region automaton $P_{D(X)}$ as a collection of coordinating S/R processes. These processes are included in the original S/R program when Cospan performs the search in the next iteration. All these steps are automated, and require no intervention from the user. If the desired language inclusion does not hold, then the output is a counter-example. In the other case, the tool reports that the property holds along with the delay constraint used in the final iteration.

In the remainder of the section we will consider some examples including the controller of Example 1 and timing-based algorithms for mutual exclusion. Our experiment consists of verifying the correctness in each case with a variety of values for the bounds on delays. We summarize our observations from these experiments below:

- The number of iterations needed before termination is much smaller than the worst case exponential bound, often even less than the number of delays.
- The iterative approach performs better, both in terms of space and time, than the approach of constructing the full region automaton P_D.

	Case 1	Case 2	Case 3	Case 4	Case 5
Estimated size of P_D	62206	228096	62206	10^8	$5 \cdot 10^{10}$
States explored of $P \otimes P_D$	30	185	29	533	2646
Time to explore $P \otimes P_D$ (in min)	0.12	0.15	0.18	0.31	11.54
Total iterative time (in min)	0.45	0.54	0.19	0.48	0.56
# of iterations	5	5	2	5	5
Estimated size of P_{D^*}	34992	228096	192	427680	165888
States explored in the last iteration	30	185	8	117	73

Figure 8: Experimental data for verifying train-gate controller

- The delay bounds get reduced to small numbers (or ∞), and thus, the computational effort required for timing verification does not grow proportionately with the actual specified bounds.
- The timing verification algorithm can help in understanding the behavior of the system being verified by identifying the delays that are relevant in proving the task being verified.

Based on our experiments, we believe that our iterative solution should be used instead of starting with the full region automaton P_D. We intend to continue our experiments with the tool on more practical and more complex examples.

Train-gate controller

Let us revisit the controller of Example 1. We will consider the performance for different properties and for different values of the bounds on delays. The table of Figure 8 shows the number of iterations before termination, the total time taken by the iterative algorithm, the number of delays in the constraint D^* used in the last iteration, and the number of states of $P \otimes P_{D^*}$ visited in the last iteration. This data should be compared to the time and space requirements if we require Cospan to use the original delay constraint D in the very first iteration. The table shows the number of states of $P \otimes P_D$ visited, and time it takes. It also lists the theoretical estimates of the number of states of the region automata P_D and P_{D^*}.

The original delay constraint D consists of 3 timed processes with 7 delays $\delta_1, \ldots \delta_7$. The lower and upper bounds for the delays are given by $\delta_1 : [3,3], \delta_2 : [1,2], \delta_3 : [5,\infty], \delta_4 : [1,1], \delta_5, \delta_6 : [0,1], \delta_7 : [1,2]$. Case 1 of Figure 8 shows the experimental data for proving the safety property T. The delay constraint D^* used in the final iteration has all the delays of the original constraint D except δ_2. Thus, the fact that the train sends the signal *far* within $1sec$ to $2sec$ after it enters the gate, is not required to prove the safety property. The remaining bounds stay unchanged. Note that, in this example, though the estimated size of P_D is much larger than the estimated size of P_{D^*}, the actual number of states visited while proving T are almost the same for both $P \otimes P_D$ and $P \otimes P_{D^*}$.

We also consider a real-time liveness property requiring that the gate never stays closed for more than $10sec$. That is, we want to prove that the maximum delay between *down* and the following *up* is at most $10sec$. To verify such properties that involve timing constraints, we need to introduce additional timer processes in the model. In this case, we use an additional automaton, called Timer, which has two events *start* and *stop*. Timer has one delay that begins with *start* and ends with *stop*. The lower and upper bound on this delay is 10. The system model now consists of Train, Gate, Controller, and Timer. The task T' rules out all sequences in which a *down* event is followed by a *start*, and then followed by a *stop*, with no intervening *up*. This property is proved after 5 iterations as shown in Case 2 of Figure 8. In this case, the final constraint D^* is the same as the original constraint D, and thus, there is no advantage in using the iterative scheme.

If we change the lower bound on the delay δ_1 between *near* and *in* to $2sec$, then the safety property T does not hold. Our tool discovers the counter-example in the second iteration, see Case 3 of Figure 8.

Now let us change the bounds to the following: $\delta_1 : [51,100], \delta_2 : [15,55], \delta_3 : [50,\infty], \delta_4, \delta_5 : [0,25]$, and $\delta_6, \delta_7 : [11,20]$. For this constraint, the experimental data is shown in Case 4 of Figure 8. Like in case 1, the task is proved in the 5th iteration, and the final delay constraint does not involve the delay δ_2. The bounds for the other delays are updated to the following: $\delta_1, \delta_3 : [10,\infty]$, δ_4, δ_5:[0,5], $\delta_6, \delta_7 : [2,4]$. Thus there is a significant reduction in the magnitudes of the bounds. Finally, let the bounds be $\delta_1 = [501, 700]$, $\delta_2 : [0, 199]$, $\delta_3 : [450, \infty]$, $\delta_4, \delta_5, \delta_6, \delta_7 : [100, 201]$. In this case also, we obtain termination after 5 iterations (see Case 5 of Figure 8). Comparing cases 1, 4, and 5, observe that, in the iterative approach both the time required for termination and the number of reachable states of $P \otimes P_{D^*}$, show no considerable change. On the other hand, the number of reachable states of $P \otimes P_D$ and time required to explore it, increase with increasing magnitudes.

Mutual exclusion protocol I

Now we consider Fischer's protocol for mutual exclusion [Lam87]. The mutual exclusion problem is to design a protocol that guarantees mutually exclusive access to a critical section among competing processes P_i. Fischer proposed a very simple protocol that exploits the knowledge about the timing delays of the system. In this protocol, a shared variable *lock* is used for communication; initially *lock* has the value 0. Each process P_i executes the following algorithm in an infinite loop:

(0) *remainder region*;
  repeat (1) **await** (*lock* = 0);
    (2) *lock* := *i*;
    (3) *delay*
  until (4) (*lock* = *i*);
(5) *critical section*;
(6) *lock* := 0

The correctness of the protocol depends on the assumptions about the time taken to execute each read and write operation. There are two read operations 1 and 4, and two write operations 2 and 6. The statement 3 corresponds to an explicit delay statement. For each process P_i, with each labeled statement j, we associate a delay δ_i^j. Each process P_i is modeled as an S/R process with 7 states, one per each possible value of the location of control, and the shared variable *lock* is modeled as another S/R process with 4 states, one per each possible value. We present results for the case of 3 competing processes. When Cospan is run ignoring the timing constraints, there are 6936 reachable states.

The property to be verified is the mutual exclusion property, namely, no two processes are ever in their respective critical sections at the same time. Note that mutual exclusion is a safety property. Without any assumptions on the bounds of the various delays, mutual exclusion does not hold. We assume that the bounds are specified by four numbers: each read or write operation of each process has lower bound a and upper bound b, and the *delay* statement has lower bound c and upper bound d. The statements 0 and 5 have lower bound 0 and upper bound ∞.

First consider the case when the delay constraint D is given by $a = 0$, $b = 2$, $c = 5$, and $d = 8$. The mutual exclusion property holds for these bounds. The experimental data appears in Case 1 of Figure 9. Termination is obtained after 5 iterations. The final delay constraint D^* consists of delays $\{\delta_i^2, \delta_i^3 \mid i = 1, 2, 3\}$, and the bounds are given by $a' = 0$, $b' = 1$, $c' = 1$, and $d' = \infty$. This means that only the upper bound on the write statement (2), and the lower bound on the *delay* statement (3) are sufficient to prove the mutual exclusion property. Notice that the constants are very small, in fact, the estimated sizes of the region automata P_D and P_{D^*} differ by a factor of 1000. In the final iteration, the product $P \otimes P_{D^*}$ has 3285 reachable states, smaller by a factor of 10 than the number of reachable states of $P \otimes P_D$. This also explains why the total time required for all the iterations combined is less than the time for a single iteration that explores $P \otimes P_D$. Also notice that the number of reachable states of $P \otimes P_{D^*}$ is smaller than that of P itself. In fact, this is typical in timing verification: if the magnitudes of the delay bounds are small, then taking the product with the region automaton cuts down on the number of reachable states.

Now let us consider the (liveness) property of deadlock freedom, which says that, if some process is at step 1, then eventually some process is in the critical section. The property is verified in the very first iteration. This is because, for Fischer's protocol, deadlock freedom does not depend upon the timing assumptions. Case 2 (Figure 9) shows that the iterative scheme gives a big improvement over exploring $P \otimes P_D$.

When we run the algorithm with $a = 1$, $b = 20$, $c = 50$, and $d = 51$, the results are same as in Case 1, and are shown in Case 3 of Figure 9. Termination is obtained in 5 iterations with the same values $a' = 0$,

Protocol	Fischer				AT92		LS92	
Reachable states of P	6936				71593		37504	
# of delays in D	15				33		27	
	Case 1	Case 2	Case 3	Case 4	Case 5	Case 6	Case 7	Case 8
Estimated size of P_D	$7 \cdot 10^6$	$7 \cdot 10^6$	$6 \cdot 10^9$	$7 \cdot 10^6$	$6 \cdot 10^7$	$5 \cdot 10^6$	$4 \cdot 10^7$	$4 \cdot 10^7$
States explored of $P \otimes P_D$	34677	34677	1587588	71777	477960	346145	49409	49409
Time to explore $P \otimes P_D$	11.29	11.47	$\sim$ 1 day	10.28	92.22	24.14	7.12	7.31
Total iterative time	3.17	0.22	3.07	0.11	62.25	32.26	0.15	1.34
# of iterations	5	1	5	1	15	11	1	6
# of delays in D^*	6	0	6	0	15	12	0	6
Estimated size of P_{D^*}	10368	1	10368	1	279936	48000	1	34992
States in last iteration	3285	6936	3285	3742	55167	57863	37504	5433

Figure 9: Experimental data for verifying mutual exclusion protocols

$b' = 1$, $c' = 1$, and $d' = \infty$. This illustrates that the computational effort may be largely independent of the specified bounds. On the other hand, observe that exploring $P \otimes P_D$ is quite expensive for these bounds.

When we run the algorithm with the bounds $a = 0$, $b = 3$, $c = 1$, and $d = 8$, the counter-example obtained in the very first iteration is timing consistent, and thus the protocol is proved to be incorrect; the data is shown in Case 4 of Figure 9.

Mutual exclusion protocol II

Now let us consider a slightly more complex protocol for mutual exclusion from [AT92]. As before, we have three competing processes P_1, P_2, and P_3. Each P_i repeats the following algorithm that employs 3 shared variables x, y, and z; the shared variables y and z are initially 0.

(0) *remainder region*
(1) $x := i$;
(2) **await** $(y = 0)$;
(3) $y := i$;
(4) **if** $x \neq i$ **then** (5) *delay*;
    (6) **if** $y \neq i$ **then goto** (1);
    (7) **await** $(z = 0)$
  else (8) $z := 1$;
(9) *critical section*;
(10) $z := 0$;
(11) **if** $y = i$ **then** (12) $y := 0$.

The correctness of the protocol again depends on the assumptions about the time taken to execute each read and write operation. For each process P_i, with each labeled statement j, except $j = 0$ and $j = 9$, we associate a delay δ_i^j. The unconstrained product has 71593 reachable states. The property to be verified is again the mutual exclusion property. As before, we assume that each read or write operation of each process has lower bound a and upper bound b, and the *delay* statement has lower bound c and upper bound d.

Suppose we set $a = 0$, $b = 2$, $c = 5$, and $d = 8$. For the specified delay constraint D, the product $P \otimes P_D$ has 477960 reachable states. When we run our iterative scheme, termination is obtained after 15 iterations (see Case 5 of Figure 9). Out of the original 33 delays, the final delay constraint D^* consists of 15 delays corresponding to statements 3, 4, 5, 8, and 12. The new bounds are given by $a' = 0$, $b' = 1$, $c' = 2$, and $d' = \infty$. This is indeed the optimal delay constraint. The correctness of the protocol does require that there be an upper bound on the assignments 3, 8, and 12, and also on the read statement 4, and the delay at statement 5 should be at least twice as much as this upper bound. Since the analysis of this protocol is quite tricky, it is quite encouraging that our algorithm found the tightest constraints. In the final iteration, the product $P \otimes P_{D^*}$ has 55167 reachable states, much smaller than the number of reachable states of $P \otimes P_D$, and also smaller than that of P itself. If we set $a = 0$, $b = 2$, $c = 3$, and $d = \infty$, the protocol is proved to be incorrect after 11 iterations (see Case 6 of Figure 9).

Mutual exclusion protocol III

Now let us consider another protocol for mutual exclusion from [LS92]. For this protocol, unlike the previous two, the property of mutual exclusion does not depend upon the timing assumptions, but the property of deadlock freedom does. The protocol uses two shared variables x and y, both initially 0. The algorithm for process P_i is as follows.

(0) *remainder region*
(1) **await** $(x = 0)$;
(2) $x := i$;
(3) *delay*;
(4) **if** $x \neq i$ **then goto** (1);
(5) **if** $y \neq 0$ **then goto** (1);
(6) $y := 1$;
(7) **if** $x \neq i$ **then goto** (1);
(8) *critical section*;
(9) $y := 0$;
(10) $x := 0$;

As before, we assume that each read or write operation of each process has lower bound a and upper bound b, and the *delay* statement has lower bound c and upper bound d. The unconstrained product has 37504 reachable states. The property to be verified is again the mutual exclusion property for $a = 0$, $b = 2$, $c = 5$, and $d = 8$. For the specified delay constraint D, the product $P \otimes P_D$ has 49409 reachable states. When we run our iterative scheme, termination is obtained in the first iteration (see data in Case 7 of Figure 9). Thus, the protocol satisfies mutual exclusion in absence of any timing assumptions.

Next we consider the property of deadlock freedom, namely, if some process leaves the remainder region then eventually some process enters the critical section. This example differs from all the previous ones, as the counterexample reported in each iteration is an infinite sequence. The termination is obtained in 6 iterations. Out of the original 27 delays, the final delay constraint D^* consists of 6 delays corresponding to statements 2 and 3. The new bounds are given by $a' = 0$, $b' = 1$, $c' = 2$, and $d' = \infty$. Again, the iterative scheme helps to identify the constraints necessary to prove the property, and performs better than the exploration of $P \otimes P_D$ (see Case 8 of Figure 9).

Acknowledgements

We thank Edith Cohen, David Johnson, and Jeff Lagarias for helpful discussions.

References

[ACD93] R. Alur, C. Courcoubetis, and D.L. Dill. Model-checking in dense real-time. *Information and Computation*, 104(1):2–34, 1993. Also appears in the Proc. of 5th LICS, 1990.

[ACH+92] R. Alur, C. Courcoubetis, N. Halbwachs, D.L. Dill, and H. Wong-Toi. Minimization of timed transition systems. In *Proceedings of the Third Conference on Concurrency Theory CONCUR '92*, Lecture Notes in Computer Science 630, pages 340–354. Springer-Verlag, 1992.

[AD90] R. Alur and D.L. Dill. Automata for modeling real-time systems. In *Automata, Languages and Programming: Proceedings of the 17th ICALP*, Lecture Notes in Computer Science 443, pages 322–335. Springer-Verlag, 1990. To appear in Theoretical Computer Science.

[AH89] R. Alur and T.A. Henzinger. A really temporal logic. In *Proceedings of the 30th IEEE Symposium on Foundations of Computer Science*, pages 164–169, 1989.

[AK83] S. Aggarwal and R.P. Kurshan. Modeling elapsed time in protocol specification. In H. Rudin and C.H. West, editors, *IFIP Protocol Specification, Testing, and Verification*, volume III, pages 51–62, 1983.

[AKS83] S. Aggarwal, R.P. Kurshan, and D. Sharma. A language for the specification and analysis of protocols. In H. Rudin and C.H. West, editors, *IFIP Protocol Specification, Testing, and Verification*, volume III, pages 35–50, 1983.

[AT92] R. Alur and G. Taubenfeld. Results about fast mutual exclusion. In *Proceedings of the 13th IEEE Real-Time Systems Symposium*, pages 12–21, 1992.

[BCDM86] M.C. Browne, E.M. Clarke, D.L. Dill, and B. Mishra. Automatic verification of sequential circuits using temporal logic. *IEEE Transactions on Computers*, C-35(12):1035–1044, 1986.

[CES86] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite-state concurrent systems using temporal-logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, 1986.

[Cho74] Y. Choueka. Theories of automata on ω-tapes: a simplified approach. *Journal of Computer and System Sciences*, 8:117–141, 1974.

[Dil89] D.L. Dill. Timing assumptions and verification of finite-state concurrent systems. In J. Sifakis, editor, *Automatic Verification Methods for Finite State Systems*, Lecture Notes in Computer Science 407, pages 197–212. Springer–Verlag, 1989.

[EMSS90] E.A. Emerson, A.K. Mok, A.P. Sistla, and J. Srinivasan. Quantitative temporal reasoning. In E.M. Clarke and R.P. Kurshan, editors, *Computer-Aided Verification, 2nd International Conference, CAV'90*, Lecture Notes in Computer Science 531, pages 136–145, 1990.

[HK90] Z. Har'El and R.P. Kurshan. Software for analytical development of communication protocols. *AT&T Technical Journal*, 1990.

[HMP91] T.A. Henzinger, Z. Manna, and A. Pnueli. Temporal proof methodologies for real-time systems. In *Proceedings of the 18th ACM Symposium on Principles of Programming Languages*, pages 353–366, 1991.

[HNSY92] T.A. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. Symbolic model-checking for real-time systems. In *Proceedings of the Seventh IEEE Symposium on Logic in Computer Science*, pages 394–406, 1992.

[Kur87] R. P. Kurshan. Reducibility in analysis of coordination. In *Lecture Notes in Computer Science*, volume 103, pages 19–39. Springer-Verlag, 1987.

[Kur90] R. P. Kurshan. Analysis of discrete event coordination. In *Lecture Notes in Computer Science*, volume 430, pages 414–453. Springer-Verlag, 1990.

[Lam87] L. Lamport. A fast mutual exclusion algorithm. *ACM Transactions on Computer Systems*, 5(1):1–11, 1987.

[Lew90] H.R. Lewis. A logic of concrete time intervals. In *Proceedings of the Fifth IEEE Symposium on Logic in Computer Science*, pages 380–389, 1990.

[LS92] N. Lynch and N. Shavit. Timing-based mutual exclusion. In *Proceedings of the 13th IEEE Real-Time Systems Symposium*, pages 2–11, 1992.

[McM93] K. McMillan. *Symbolic model checking: an approach to the state explosion problem.* Kluwer Academic Publishers, 1993.

[PS82] C.H. Papadimitriou and K. Steiglitz. *Combinatorial optimization: Algorithms and complexity.* Prentice-Hall, 1982.

[Tho90] W. Thomas. Automata on infinite objects. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 133–191. Elsevier Science Publishers, 1990.

[VW86] M.Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proceedings of the First IEEE Symposium on Logic in Computer Science*, pages 332–344, 1986.

[Yan90] M. Yannakakis. Graph-theoretic methods in database theory. In *Proceedings of the 9th ACM Symposium on Principles of Database Systems*, pages 230–242, 1990.