A Theory of Timed Automata [1]

Rajeev Alur [2] David L. Dill [3]
Computer Science Department, Stanford University
Stanford, CA 94305.

Abstract. We propose *timed (finite) automata* to model the behavior of real-time systems over time. Our definition provides a simple, and yet powerful, way to annotate state-transition graphs with timing constraints using finitely many real-valued *clocks*. A timed automaton accepts *timed words* — infinite sequences in which a real-valued time of occurrence is associated with each symbol. We study timed automata from the perspective of formal language theory: we consider closure properties, decision problems, and subclasses. We consider both nondeterministic and deterministic transition structures, and both Büchi and Muller acceptance conditions. We show that nondeterministic timed automata are closed under union and intersection, but not under complementation, whereas deterministic timed Muller automata are closed under all Boolean operations. The main construction of the paper is an (PSPACE) algorithm for checking the emptiness of the language of a (nondeterministic) timed automaton. We also prove that the universality problem and the language inclusion problem are solvable only for the deterministic automata: both problems are undecidable (Π^1_1-hard) in the nondeterministic case and PSPACE-complete in the deterministic case. Finally, we discuss the application of this theory to automatic verification of real-time requirements of finite-state systems.

Keywords: Real-time systems, automatic verification, formal languages and automata theory.

---

[1] Preliminary versions of this paper appear in the *Proceedings of the 17th International Colloquium on Automata, Languages, and Programming* (1990), and in the *Proceedings of the REX workshop "Real-time: theory in practice"* (1991).

[2] Current address: AT&T Bell Laboratories, 600 Mountain Avenue, Room 2D-144, Murray Hill, NJ 07974.

[3] Supported by the National Science Foundation under grant MIP-8858807, and by the United States Navy, Office of the Chief of Naval Research under grant N00014-91-J-1901. This publication does not necessarily reflect the position or the policy of the U.S. Government, and no official endorsement of this work should be inferred.

1 Introduction

Modal logics and ω-automata for *qualitative* temporal reasoning about concurrent systems have been studied in great detail (selected references: [36, 32, 16, 28, 47, 44, 37, 11]). These formalisms abstract away from time, retaining only the sequencing of events. In the *linear time model,* it is assumed that an execution can be completely modeled as a sequence of states or system events, called an *execution trace* (or just *trace*). The *behavior* of the system is a set of such execution sequences. Since a set of sequences is a formal language, this leads naturally to the use of automata for the specification and verification of systems. When the systems are finite-state, as many are, we can use finite automata, leading to effective constructions and decision procedures for automatically manipulating and analyzing system behavior. The universal acceptance of finite automata as the canonical model of finite-state computation can be attributed to the robustness of the model and the appeal of its theory. In particular, a variety of competing formalisms — nondeterministic Büchi automata, deterministic and nondeterministic Muller automata, ω-regular expressions, modal formulas of (extended) temporal logic, and second-order formulas of the monadic theory of one successor (S1S) — have the same expressiveness, and define the class of *ω-regular languages* [7, 9, 33, 46, 42]. Consequently many verification theories are based on the theory of ω-regular languages.

Although the decision to abstract away from quantitative time has had many advantages, it is ultimately counterproductive when reasoning about systems that must interact with physical processes; the correct functioning of the control system of airplanes and toasters depends crucially upon *real-time* considerations. We would like to be able to specify and verify models of real-time systems as easily as qualitative models. Our goal is to modify finite automata for this task and develop a theory of *timed* finite automata, similar in spirit to the theory of ω-regular languages. We believe that this should be the first step in building theories for the real-time verification problem.

For simplicity, we discuss models that consider executions to be infinite sequences of events, not states (the theory with state-based models differs only in details). Within this framework, it is possible to add timing to an execution trace by pairing it with a sequence of times, where the i'th element of the time sequence gives the time of occurrence of the i'th event. At this point, however, a fundamental question arises: what is the nature of time?

Modeling time

One alternative, which leads to the *discrete-time* model, requires the time sequence to be a monotonically increasing sequence of integers. This model is appropriate for certain kinds of synchronous digital circuits, where signal changes are considered to have changed exactly when a clock signal arrives. One of the advantages of this model is that it can be transformed easily into an ordinary formal language. Each timed trace can be expanded into a trace where the times increase by exactly one at each step, by inserting a special *silent* event as many times as necessary between events in the original trace. Once this transformation has been performed, the time of each event is the same as its position, so the time sequence can be discarded, leaving an ordinary string. Hence, discrete time behaviors can be manipulated using ordinary finite automata. Of course, in physical processes events do not always happen at integer-valued times. The discrete-time model requires that continuous time be approximated by choosing some fixed quantum *a priori*, which limits the accuracy with which physical systems can be modeled.

The *fictitious-clock model* is similar to the discrete time model, except that it only requires the sequence of integer times to be non-decreasing. The interpretation of a timed execution trace in this model is that events occur in the specified order at real-valued times, but only the (integer) readings of the actual times with respect to a digital clock are recorded in the trace. This model is also easily transformed into a conventional formal language. First, add to the set of events a new one, called *tick*. The untimed trace corresponding to a timed trace will include all of the events from the timed trace, in the same order, but with $t_{i+1} - t_i$ number of *ticks* inserted between the i'th and the $(i+1)$'th events (note that this number may be 0). Once again, it is conceptually simple to manipulate these behaviors using finite automata, but the compensating disadvantage is that it represents time only in an approximate sense.

We prefer a *dense-time* model, in which time is a dense set, because it is a more natural model for physical processes operating over continuous time. In this model, the times of events are real numbers, which increase monotonically without bound. Dealing with dense time in a finite-automata framework is more difficult than the other two cases, because it is not obvious how to transform a set of dense-time traces into an ordinary formal language. Instead, we have developed a theory of *timed* formal languages and *timed automata* to support automated reasoning about such systems.

Overview

To augment finite ω-automata with timing constraints, we propose the formalism of *timed automata*. Timed automata accept *timed words* — infinite sequences in which a real-valued time of occurrence is associated with each symbol. A timed automaton is a finite automaton with a finite set of real-valued *clocks*. The clocks can be reset to 0 (independently of each other) with the transitions of the automaton, and keep track of the time elapsed since the last reset. The transitions of the automaton put certain constraints on the clock values: a transition may be taken only if the current values of the clocks satisfy the associated constraints. With this mechanism we can model timing properties such as "the channel delivers every message within 3 to 5 time units of its receipt". Timed automata can capture several interesting aspects of real-time systems: qualitative features such as liveness, fairness, and nondeterminism; and quantitative features such as periodicity, bounded response, and timing delays.

We study timed automata from the perspective of formal language theory. We consider both deterministic and nondeterministic varieties, and for acceptance criteria we consider both Büchi and Muller conditions. We show that nondeterministic timed automata are closed under union and intersection, but surprisingly, not under complementation. The closure properties for the deterministic classes are similar to their untimed counterparts: deterministic timed Muller automata are closed under all Boolean operations, whereas deterministic timed Büchi automata are closed under only the positive Boolean operations. These results imply that, unlike the untimed case, deterministic timed Muller automata are strictly less expressive than their nondeterministic counterparts.

We study a variety of decision problems for the different types of timed automata. The main positive result is an *untiming* construction for timed automata. Due to the real-valued clock variables, the state space of a timed automaton is infinite, and the untiming algorithm constructs a finite quotient of this space. This is used to prove that the set of untimed words consistent with the timing constraints of a timed automaton forms an ω-regular set. It also leads to a PSPACE decision procedure for testing emptiness of the language of a timed automaton. We also show that the dual problem of testing whether a timed automaton accepts all timed words (i.e., the universality question) is undecidable (Π^1_1-hard) for nondeterministic automata. This also implies the undecidability of the language inclusion problem. However, both these problems can be solved in PSPACE for the deterministic versions.

Finally, we show how to apply the theory of timed automata to prove correctness of finite-state real-time systems. We give a PSPACE verification algorithm to test whether a system modeled as a product of timed automata satisfies its specification given as a deterministic timed Muller automaton.

Related work

Different ways of incorporating timing constraints in the qualitative models of a system have been proposed recently, however, no attempt has been made to develop a theory of timed languages and no algorithms for checking real-time properties in the dense-time model have been developed.

Perhaps the most standard way of introducing timing information in a process model is by associating lower and upper bounds with transitions. Examples of these include timed Petri nets [38], timed transition systems [35, 21], timed I/O automata [31], and Modecharts [25]. In a timed automaton, unlike these other models, a bound on the time taken to traverse a path in the automaton, not just the time interval between the successive transitions, can be directly expressed. Our model is based on an earlier model proposed by Dill that employs timers [13]. A model similar to Dill's was independently proposed and studied by Lewis [30]. He defines *state-diagrams*, and gives a way of translating a circuit description to a state-diagram. A state-diagram is a finite-state machine where every edge is annotated with a matrix of intervals constraining various delays. Lewis also develops an algorithm for checking consistency of the timing information for a special class of state-diagrams; the ones for which there exists a constant K such that at most K transitions can happen in a time interval of unit length. Our untiming construction does not need the latter assumption, and has a better worst-case complexity. We note that the decidability and lower bound results presented here carry over to his formalism also.

There have been a few attempts to extend temporal logics with quantitative time [6, 24, 26, 35, 17, 5, 20]. Most of these logics employ the discrete-time or the fictitious-clock semantics. In the case of the dense-time model the only previously known result is an undecidability result: in [5] it is shown that the satisfiability problem for a real-time extension of the linear-time temporal logic PTL is undecidable (Σ^1_1-hard) in the dense-time model.

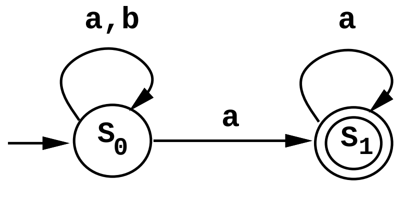


Figure 1: Büchi automaton accepting $(a+b)^*a^\omega$

2 ω-automata

In this section we will briefly review the relevant aspects of the theory of *ω-regular languages.*

The more familiar definition of a formal language is as a set of finite words over some given (finite) alphabet (see, for example, [23]). As opposed to this, an ω-language consists of infinite words. Thus an ω-language over a finite alphabet Σ is a subset of Σ^ω — the set of all infinite words over Σ. ω-automata provide a finite representation for certain types of ω-languages. An ω-automaton is essentially the same as a nondeterministic finite-state automaton, but with the acceptance condition modified suitably so as to handle infinite input words. Various types of ω-automata have been studied in the literature [7, 33, 9, 42]. We will mainly consider two types of ω-automata: Büchi automata and Muller automata.

A *transition table* $\mathcal{A}$ is a tuple $\langle \Sigma, \mathrm{S}, \mathrm{S}_0, \mathrm{E}\rangle$, where Σ is an input alphabet, S is a finite set of automaton states, $\mathrm{S}_0 \subseteq \mathrm{S}$ is a set of start states, and $\mathrm{E} \subseteq \mathrm{S} \times \mathrm{S} \times \Sigma$ is a set of edges. The automaton starts in an initial state, and if $\langle s, s', a\rangle \in \mathrm{E}$ then the automaton can change its state from s to s' reading the input symbol a.

For a word $\sigma = \sigma_1\sigma_2\ldots$ over the alphabet Σ, we say that

$$r \;:\; s_0 \xrightarrow{\sigma_1} s_1 \xrightarrow{\sigma_2} s_2 \xrightarrow{\sigma_3} \cdots$$

is a *run* of $\mathcal{A}$ over σ, provided $s_0 \in \mathrm{S}_0$, and $\langle s_{i-1}, s_i, \sigma_i\rangle \in \mathrm{E}$ for all $i \geq 1$. For such a run, the set *inf*(r) consists of the states $s \in \mathrm{S}$ such that $s = s_i$ for infinitely many $i \geq 0$.

Different types of ω-automata are defined by adding an acceptance condition to the definition of the transition tables. A *Büchi automaton* $\mathcal{A}$ is a transition table $\langle \Sigma, \mathrm{S}, \mathrm{S}_0, \mathrm{E}\rangle$ with an additional set $\mathrm{F} \subseteq \mathrm{S}$ of accepting states. A run r of $\mathcal{A}$ over a word $\sigma \in \Sigma^\omega$ is an *accepting run* iff *inf*$(r) \cap \mathrm{F} \neq \emptyset$. In other words, a run r is accepting iff some state from the set F repeats infinitely often along r. The language $L(\mathcal{A})$ accepted by $\mathcal{A}$ consists of the words $\sigma \in \Sigma^\omega$ such that $\mathcal{A}$ has an accepting run over σ.

Example 2.1 Consider the 2-state automaton of Figure 1 over the alphabet $\{a, b\}$. The state s_0 is the start state and s_1 is the accepting state. Every accepting run of the automaton has the form

$$r \;:\; s_0 \xrightarrow{\sigma_1} s_0 \xrightarrow{\sigma_2} \cdots \xrightarrow{\sigma_n} s_0 \xrightarrow{a} s_1 \xrightarrow{a} s_1 \xrightarrow{a} \cdots$$

with $\sigma_i \in \{a, b\}$ for $1 \leq i \leq n$ for some $n \geq 1$. The automaton accepts all words with only a finite number of b's; that is, the language $L_0 = (a+b)^*a^\omega$. ■

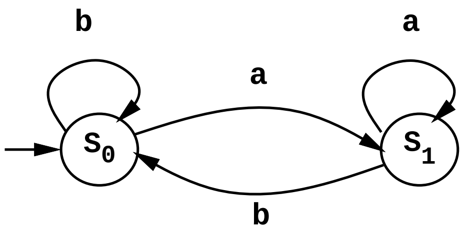


Figure 2: Deterministic Muller automaton accepting $(a+b)^* a^\omega$

An ω-language is called *ω-regular* iff it is accepted by some Büchi automaton. Thus the language L_0 of Example 2.1 is an ω-regular language.

The class of ω-regular languages is closed under all the Boolean operations. Language intersection is implemented by a product construction for Büchi automata [9, 47]. There are known constructions for complementing Büchi automata [41, 40].

When Büchi automata are used for modeling finite-state concurrent processes, the verification problem reduces to that of language inclusion. The inclusion problem for ω-regular languages is decidable. To test whether the language of one automaton is contained in the other, we check for emptiness of the intersection of the first automaton with the complement of the second. Testing for emptiness is easy; we only need to search for a cycle that is reachable from a start state and includes at least one accepting state. In general, complementing a Büchi automaton involves an exponential blow-up in the number of states, and the language inclusion problem is known to be PSPACE-complete [41]. However, checking whether the language of one automaton is contained in the language of a *deterministic* automaton can be done in polynomial time [27].

A transition table $\mathcal{A} = \langle \Sigma, \mathrm{S}, \mathrm{S}_0, \mathrm{E} \rangle$ is *deterministic* iff (i) there is a single start state, that is, $|\mathrm{S}_0| = 1$, and (ii) the number of a-labeled edges starting at s is at most one for all states $s \in \mathrm{S}$ and for all symbols $a \in \Sigma$. Thus, for a deterministic transition table, the current state and the next input symbol determine the next state uniquely. Consequently, a deterministic automaton has at most one run over a given word. Unlike the automata on finite words, the class of languages accepted by deterministic Büchi automata is strictly smaller than the class of ω-regular languages. For instance, there is no deterministic Büchi automaton which accepts the language L_0 of Example 2.1. Muller automata (defined below) avoid this problem at the cost of a more powerful acceptance condition.

A *Muller automaton* $\mathcal{A}$ is a transition table $\langle \Sigma, \mathrm{S}, \mathrm{S}_0, \mathrm{E} \rangle$ with an *acceptance family* $\mathcal{F} \subseteq 2^{\mathrm{S}}$. A run r of $\mathcal{A}$ over a word $\sigma \in \Sigma^\omega$ is an *accepting run* iff $\mathit{inf}(r) \in \mathcal{F}$. That is, a run r is accepting iff the set of states repeating infinitely often along r equals some set in $\mathcal{F}$. The language accepted by $\mathcal{A}$ is defined as in case of Büchi automata.

The class of languages accepted by Muller automata is the same as that accepted by Büchi automata, and also equals that accepted by deterministic Muller automata.

Example 2.2 The deterministic Muller automaton of Figure 2 accepts the language L_0 consisting of all words over $\{a, b\}$ with only a finite number of b's. The Muller acceptance family is $\{\{s_1\}\}$. Thus every accepting run can visit the state s_0 only finitely often. ■

Thus deterministic Muller automata form a strong candidate for representing ω-regular languages: they are as expressive as their nondeterministic counterpart, and they can be complemented in polynomial time. Algorithms for constructing the intersection of two Muller automata and for checking language inclusion are known [10].

3 Timed automata

In this section we define timed words by coupling a real-valued time with each symbol in a word. Then we augment the definition of ω-automata so that they accept timed words, and use them to develop a theory of timed regular languages analogous to the theory of ω-regular languages.

3.1 Timed languages

We define timed words so that a behavior of a real-time system corresponds to a timed word over the alphabet of events. As in the case of the dense-time model, the set of nonnegative real numbers, R, is chosen as the time domain. A word σ is coupled with a time sequence τ as defined below:

Definition 3.1 A *time sequence* $\tau = \tau_1 \tau_2 \cdots$ is an infinite sequence of time values $\tau_i \in \mathsf{R}$ with $\tau_i > 0$, satisfying the following constraints:

1. *Monotonicity:* τ increases strictly monotonically; that is, $\tau_i < \tau_{i+1}$ for all $i \geq 1$.
2. *Progress:* For every $t \in \mathsf{R}$, there is some $i \geq 1$ such that $\tau_i > t$.

A *timed word* over an alphabet Σ is a pair (σ, τ) where $\sigma = \sigma_1 \sigma_2 \ldots$ is an infinite word over Σ and τ is a time sequence. A *timed language* over Σ is a set of timed words over Σ. ■

If a timed word (σ, τ) is viewed as an input to an automaton, it presents the symbol σ_i at time τ_i. If each symbol σ_i is interpreted to denote an event occurrence then the corresponding component τ_i is interpreted as the time of occurrence of σ_i. Under certain circumstances it may be appropriate to allow the same time value to be associated with many consecutive events in the sequence. To accommodate this possibility one could use a slightly different definition of timed words by requiring a time sequence to increase only monotonically (i.e., require $\tau_i \leq \tau_{i+1}$ for all $i \geq 1$). All our results continue to hold in this alternative model also.

Let us consider some examples of timed languages.

Example 3.2 Let the alphabet be $\{a, b\}$. Define a timed language L_1 to consist of all timed words (σ, τ) such that there is no b after time 5.6. Thus the language L_1 is given by

$$L_1 \;=\; \{(\sigma, \tau) \mid \forall i. ((\tau_i > 5.6) \rightarrow (\sigma_i = a))\}.$$

Another example is the language L_2 consisting of timed words in which a and b alternate, and for the successive pairs of a and b, the time difference between a and b keeps increasing. The language L_2 is given as

$$L_2 \;=\; \{((ab)^\omega, \tau) \mid \forall i. ((\tau_{2i} - \tau_{2i-1}) < (\tau_{2i+2} - \tau_{2i+1}))\}. \qquad \blacksquare$$

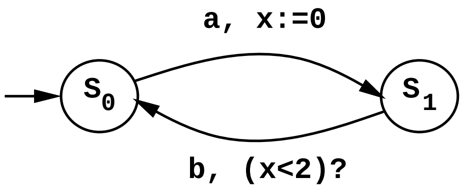


Figure 3: Example of a timed transition table

The language-theoretic operations such as intersection, union, complementation are defined for timed languages as usual. In addition we define the *Untime* operation which discards the time values associated with the symbols, that is, it considers the projection of a timed trace (σ, τ) on the first component.

Definition 3.3 For a timed language L over Σ, $Untime(L)$ is the ω-language consisting of $\sigma \in \Sigma^\omega$ such that $(\sigma, \tau) \in L$ for some time sequence τ. ■

For instance, referring to Example 3.2, $Untime(L_1)$ is the ω-language $(a+b)^* a^\omega$, and $Untime(L_2)$ consists of a single word $(ab)^\omega$.

3.2 Transition tables with timing constraints

Now we extend transition tables to *timed transition tables* so that they can read timed words. When an automaton makes a state-transition, the choice of the next state depends upon the input symbol read. In case of a timed transition table, we want this choice to depend also upon the time of the input symbol relative to the times of the previously read symbols. For this purpose, we associate a finite set of (real-valued) *clocks* with each transition table. A clock can be set to zero simultaneously with any transition. At any instant, the reading of a clock equals the time elapsed since the last time it was reset. With each transition we associate a clock constraint, and require that the transition may be taken only if the current values of the clocks satisfy this constraint. Before we define the timed transition tables formally, let us consider some examples.

Example 3.4 Consider the timed transition table of Figure 3. The start state is s_0. There is a single clock x. An annotation of the form $x := 0$ on an edge corresponds to the action of resetting the clock x when the edge is traversed. Similarly an annotation of the form $(x < 2)$? on an edge gives the clock constraint associated with the edge.

The automaton starts in state s_0, and moves to state s_1 reading the input symbol a. The clock x gets set to 0 along with this transition. While in state s_1, the value of the clock x shows the time elapsed since the occurrence of the last a symbol. The transition from state s_1 to s_0 is enabled only if this value is less than 2. The whole cycle repeats when the automaton moves back to state s_0. Thus the timing constraint expressed by this transition table is that the delay between a and the following b is always less than 2; more formally, the language is

$$\{((ab)^\omega, \tau) \mid \forall i.\, (\tau_{2i} < \tau_{2i-1} + 2)\}.$$

■

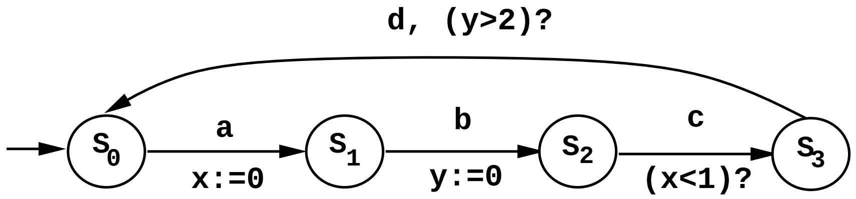


Figure 4: Timed transition table with 2 clocks

Thus to constrain the delay between two transitions e_1 and e_2, we require a particular clock to be reset on e_1, and associate an appropriate clock constraint with e_2. Note that clocks can be set asynchronously of each other. This means that different clocks can be restarted at different times, and there is no lower bound on the difference between their readings. Having multiple clocks allows multiple concurrent delays, as in the next example.

Example 3.5 The timed transition table of Figure 4 uses two clocks x and y, and accepts the language

$$L_3 \;=\; \{((abcd)^\omega, \tau) \mid \forall j.\, ((\tau_{4j+3} < \tau_{4j+1} + 1) \,\wedge\, (\tau_{4j+4} > \tau_{4j+2} + 2))\}.$$

The automaton cycles among the states s_0, s_1, s_2 and s_3. The clock x gets set to 0 each time it moves from s_0 to s_1 reading a. The check $(x < 1)$? associated with the c-transition from s_2 to s_3 ensures that c happens within time 1 of the preceding a. A similar mechanism of resetting another independent clock y while reading b and checking its value while reading d, ensures that the delay between b and the following d is always greater than 2. ■

Notice that in the above example, to constrain the delay between a and c and between b and d the automaton does not put any explicit bounds on the time difference between a and the following b, or c and the following d. This is an important advantage of having multiple clocks which can be set independently of each other. The above language L_3 is the intersection of the two languages L_3^1 and L_3^2 defined as

$$\begin{aligned} L_3^1 &= \{((abcd)^\omega, \tau) \mid \forall j.\, (\tau_{4j+3} < \tau_{4j+1} + 1)\}, \\ L_3^2 &= \{((abcd)^\omega, \tau) \mid \forall j.\, (\tau_{4j+4} > \tau_{4j+2} + 2)\}. \end{aligned}$$

Each of the languages L_3^1 and L_3^2 can be expressed by an automaton which uses just one clock; however to express their intersection we need two clocks.

We remark that the clocks of the automaton do not correspond to the local clocks of different components in a distributed system. All the clocks increase at the uniform rate counting time with respect to a fixed global time frame. They are fictitious clocks invented to express the timing properties of the system. Alternatively, we can consider the automaton to be equipped with a finite number of stop-watches which can be started and checked independently of one another, but all stop-watches refer to the same clock.

3.3 Clock constraints and clock interpretations

To define timed automata formally, we need to say what type of clock constraints are allowed on the edges. The simplest form of a constraint compares a clock value with a time constant. We allow only the Boolean combinations of such simple constraints. Any value from Q, the set of nonnegative rationals, can be used as a time constant. Later, in Section 5.5, we will show that allowing more complex constraints, such as those involving addition of clock values, leads to undecidability.

Definition 3.6 For a set X of clock variables, the set $\Phi(X)$ of *clock constraints* δ is defined inductively by

$$\delta := x \leq c \mid c \leq x \mid \neg\delta \mid \delta_1 \wedge \delta_2,$$

where x is a clock in X and c is a constant in Q. ■

Observe that constraints such as **true**, $(x = c)$, $x \in [2, 5)$ can be defined as abbreviations.

A *clock interpretation* ν for a set X of clocks assigns a real value to each clock; that is, it is a mapping from X to R. We say that a clock interpretation ν for X satisfies a clock constraint δ over X iff δ evaluates to true using the values given by ν.

For $t \in \mathsf{R}$, $\nu + t$ denotes the clock interpretation which maps every clock x to the value $\nu(x) + t$, and the clock interpretation $t\cdot\nu$ assigns to each clock x the value $t\cdot\nu(x)$. For $Y \subseteq X$, $[Y \mapsto t]\nu$ denotes the clock interpretation for X which assigns t to each $x \in Y$, and agrees with ν over the rest of the clocks.

3.4 Timed transition tables

Now we give the precise definition of timed transition tables.

Definition 3.7 A *timed transition table* $\mathcal{A}$ is a tuple $\langle \Sigma, \mathrm{S}, \mathrm{S}_0, \mathrm{C}, \mathrm{E}\rangle$, where

- Σ is a finite alphabet,
- S is a finite set of states,
- $\mathrm{S}_0 \subseteq \mathrm{S}$ is a set of start states,
- C is a finite set of clocks, and
- $\mathrm{E} \subseteq \mathrm{S} \times \mathrm{S} \times \Sigma \times 2^{\mathrm{C}} \times \Phi(\mathrm{C})$ gives the set of transitions. An edge $\langle s, s', a, \lambda, \delta\rangle$ represents a transition from state s to state s' on input symbol a. The set $\lambda \subseteq \mathrm{C}$ gives the clocks to be reset with this transition, and δ is a clock constraint over C.

■

Given a timed word (σ, τ), the timed transition table $\mathcal{A}$ starts in one of its start states at time 0 with all its clocks initialized to 0. As time advances, the values of all clocks change, reflecting the elapsed time. At time τ_i, $\mathcal{A}$ changes state from s to s' using some transition of the form $\langle s, s', \sigma_i, \lambda, \delta\rangle$ reading the input σ_i, if the current values of clocks satisfy δ. With this transition the clocks in λ are reset to 0, and thus start counting time with respect to the time of occurrence of this transition. This behavior is captured by defining *runs* of timed transition tables. A run records the state and the values of all the clocks at the transition points. For a time sequence $\tau = \tau_1\tau_2\ldots$ we define $\tau_0 = 0$.

Definition 3.8 A run r, denoted by $(\overline{s}, \overline{\nu})$, of a timed transition table $\langle \Sigma, \mathrm{S}, \mathrm{S}_0, \mathrm{C}, \mathrm{E}\rangle$ over a timed word (σ, τ) is an infinite sequence of the form

$$r \ : \ \langle s_0, \nu_0\rangle \ \xrightarrow[\tau_1]{\sigma_1} \ \langle s_1, \nu_1\rangle \ \xrightarrow[\tau_2]{\sigma_2} \ \langle s_2, \nu_2\rangle \ \xrightarrow[\tau_3]{\sigma_3} \ \cdots$$

with $s_i \in \mathrm{S}$ and $\nu_i \in [\mathrm{C} \rightarrow \mathsf{R}]$, for all $i \geq 0$, satisfying the following requirements:

- *Initiation:* $s_0 \in \mathrm{S}_0$, and $\nu_0(x) = 0$ for all $x \in \mathrm{C}$.
- *Consecution:* for all $i \geq 1$, there is an edge in E of the form $\langle s_{i-1}, s_i, \sigma_i, \lambda_i, \delta_i\rangle$ such that $(\nu_{i-1} + \tau_i - \tau_{i-1})$ satisfies δ_i and ν_i equals $[\lambda_i \mapsto 0](\nu_{i-1} + \tau_i - \tau_{i-1})$.

The set $\mathit{inf}(r)$ consists of those states $s \in \mathrm{S}$ such that $s = s_i$ for infinitely many $i \geq 0$. ■

Example 3.9 Consider the timed transition table of Example 3.5. Consider a timed word

$$(a, 2) \ \rightarrow \ (b, 2.7) \ \rightarrow \ (c, 2.8) \ \rightarrow \ (d, 5) \ \rightarrow \ \cdots$$

Below we give the initial segment of the run. A clock interpretation is represented by listing the values $[x, y]$.

$$\langle s_0, [0, 0]\rangle \xrightarrow[2]{a} \langle s_1, [0, 2]\rangle \xrightarrow[2.7]{b} \langle s_2, [0.7, 0]\rangle \xrightarrow[2.8]{c} \langle s_3, [0.8, 0.1]\rangle \xrightarrow[5]{d} \langle s_0, [3, 2.3]\rangle \cdots$$

■

Along a run $r = (\overline{s}, \overline{\nu})$ over (σ, τ), the values of the clocks at time t between τ_i and τ_{i+1} are given by the interpretation $(\nu_i + t - \tau_i)$. When the transition from state s_i to s_{i+1} occurs, we use the value $(\nu_i + \tau_{i+1} - \tau_i)$ to check the clock constraint; however, at time τ_{i+1}, the value of a clock that gets reset is defined to be 0.

Note that a transition table $\mathcal{A} = \langle \Sigma, \mathrm{S}, \mathrm{S}_0, \mathrm{E}\rangle$ can be considered to be a timed transition table $\mathcal{A}'$. We choose the set of clocks to be the empty set, and replace every edge $\langle s, s', a\rangle$ by $\langle s, s', a, \emptyset, \mathbf{true}\rangle$. The runs of $\mathcal{A}'$ are in an obvious correspondence with the runs of $\mathcal{A}$.

3.5 Timed regular languages

We can couple acceptance criteria with timed transition tables, and use them to define timed languages.

Definition 3.10 A *timed Büchi automaton* (in short TBA) is a tuple $\langle \Sigma, \mathrm{S}, \mathrm{S}_0, \mathrm{C}, \mathrm{E}, \mathrm{F}\rangle$, where $\langle \Sigma, \mathrm{S}, \mathrm{S}_0, \mathrm{C}, \mathrm{E}\rangle$ is a timed transition table, and $\mathrm{F} \subseteq \mathrm{S}$ is a set of *accepting* states.

A run $r = (\overline{s}, \overline{\nu})$ of a TBA over a timed word (σ, τ) is called an *accepting run* iff $\mathit{inf}(r) \cap \mathrm{F} \neq \emptyset$.

For a TBA $\mathcal{A}$, the language $L(\mathcal{A})$ of timed words it accepts is defined to be the set $\{(\sigma, \tau) \mid \ \mathcal{A}$ has an accepting run over $(\sigma, \tau)\}$. ■

In analogy with the class of languages accepted by Büchi automata, we call the class of timed languages accepted by TBAs timed regular languages.

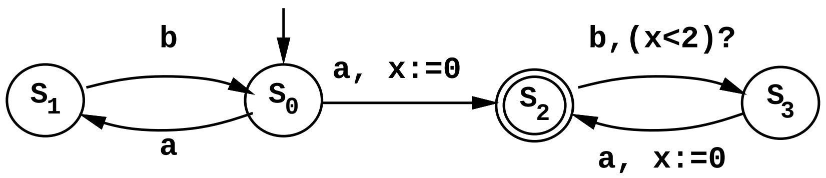


Figure 5: Timed Büchi automaton accepting L_{crt}

Definition 3.11 A timed language L is a *timed regular language* iff $L = L(\mathcal{A})$ for some TBA $\mathcal{A}$. ■

Example 3.12 The language L_3 of Example 3.5 is a timed regular language. The timed transition table of Figure 4 is coupled with the acceptance set consisting of all the states.

For every ω-regular language L over Σ, the timed language $\{(\sigma, \tau) \mid \sigma \in L\}$ is regular.

A typical example of a nonregular timed language is the language L_2 of Example 3.2. It requires that the time difference between the successive pairs of a and b form an increasing sequence.

Another nonregular language is $\{(a^\omega, \tau) \mid \forall i. (\tau_i = 2^i)\}$. ■

The automaton of Example 3.13 combines the Büchi acceptance condition with the timing constraints to specify an interesting convergent response property:

Example 3.13 The automaton of Figure 5 accepts the timed language L_{crt} over the alphabet $\{a, b\}$.

$$L_{\text{crt}} \;=\; \{((ab)^\omega, \tau) \mid \exists i. \forall j \geq i. (\tau_{2j} < \tau_{2j-1} + 2)\}.$$

The start state is s_0, the accepting state is s_2, and there is a single clock x. The automaton starts in state s_0, and cycles between the states s_0 and s_1 for a while. Then, nondeterministically, it moves to state s_2 setting its clock x to 0. While in the cycle between the states s_2 and s_3, the automaton resets its clock while reading a, and ensures that the next b is within 2 time units. Interpreting the symbol b as a response to a request denoted by the symbol a, the automaton models a system with a *convergent response time*; the response time is "eventually" always less than 2 time units. ■

The next example shows that timed automata can specify periodic behavior also.

Example 3.14 The automaton of Figure 6 accepts the following language over the alphabet $\{a, b\}$.

$$\{(\sigma, \tau) \mid \forall i. \exists j. (\tau_j = 3i \wedge \sigma_j = a)\}$$

The automaton has a single state s_0, and a single clock x. The clock gets reset at regular intervals of period 3 time units. The automaton requires that whenever the clock equals 3 there is an a symbol. Thus it expresses the property that a happens at all time values that are multiples of 3. ■

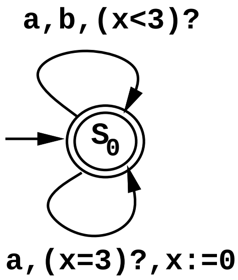


Figure 6: Timed automaton specifying periodic behavior

3.6 Properties of timed regular languages

The next theorem considers some closure properties of timed regular languages.

Theorem 3.15 The class of timed regular languages is closed under (finite) union and intersection.

PROOF. Consider TBAs $\mathcal{A}_i = \langle \Sigma, \mathrm{S}_i, \mathrm{S}_{i_0}, \mathrm{C}_i, \mathrm{E}_i, \mathrm{F}_i \rangle$, $i = 1, 2, \ldots n$. Assume without loss of generality that the clock sets C_i are disjoint. We construct TBAs accepting the union and intersection of $L(\mathcal{A}_i)$.

Since TBAs are nondeterministic the case of union is easy. The required TBA is simply the disjoint union of all the automata.

Intersection can be implemented by a trivial modification of the standard product construction for Büchi automata [9]. The set of clocks for the product automaton $\mathcal{A}$ is $\cup_i \mathrm{C}_i$. The states of $\mathcal{A}$ are of the form $\langle s_1, \ldots s_n, k \rangle$, where each $s_i \in \mathrm{S}_i$, and $1 \leq k \leq n$. The i-th component of the tuple keeps track of the state of $\mathcal{A}_i$, and the last component is used as a counter for cycling through the accepting conditions of all the individual automata. Initially the counter value is 1, and it is incremented from k to $(k+1)$ (modulo n) iff the current state of the k-th automaton is an accepting state. Note that we choose the value of n *mod* n to be n.

The initial states of $\mathcal{A}$ are of the form $\langle s_1, \ldots s_n, 1 \rangle$ where each s_i is a start state of $\mathcal{A}_i$. A transition of $\mathcal{A}$ is obtained by coupling the transitions of the individual automata having the same label. Let $\{\langle s_i, s'_i, a, \lambda_i, \delta_i \rangle \in \mathrm{E}_i \mid i = 1, \ldots n\}$ be a set of transitions, one per each automaton, with the same label a. Corresponding to this set, there is a joint transition of $\mathcal{A}$ out of each state of the form $\langle s_1, \ldots s_n, k \rangle$ labeled with a. The new state is $\langle s'_1, \ldots s'_n, j \rangle$ with $j = (k+1)\ mod\ n$ if $s_k \in \mathrm{F}_k$, and $j = k$ otherwise. The set of clocks to be reset with this transition is $\cup_i \lambda_i$, and the associated clock constraint is $\wedge_i \delta_i$.

The counter value cycles through the whole range $1, \ldots n$ infinitely often iff the accepting conditions of all the automata are met. Consequently, we define the accepting set for $\mathcal{A}$ to consist of states of the form $\langle s_1, \ldots s_n, n \rangle$, where $s_n \in \mathrm{F}_n$. ∎

In the above product construction, the number of states of the resulting automaton is $n \cdot \Pi_i |\mathrm{S}_i|$. The number of clocks is $\Sigma_i |\mathrm{C}_i|$, and the size of the edge set is $n \cdot \Pi_i |\mathrm{E}_i|$. Note that $|\mathrm{E}|$ includes the length of the clock constraints assuming binary encoding for the constants.

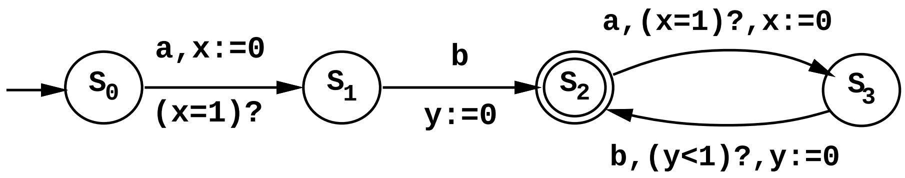


Figure 7: Timed automaton accepting $L_{converge}$

Observe that even for the timed regular languages arbitrarily many symbols can occur in a finite interval of time. Furthermore, the symbols can be arbitrarily close to each other. Consider the following example.

Example 3.16 The language accepted by the automaton in Figure 7 is

$$L_{converge} \;=\; \{((ab)^{\omega}, \tau) \mid \forall i.\, (\tau_{2i-1} = i \,\wedge\, (\tau_{2i} - \tau_{2i-1} > \tau_{2i+2} - \tau_{2i+1}))\}.$$

Every word accepted by this automaton has the property that the sequence of time differences between a and the following b is strictly decreasing. A sample word accepted by the automaton is

$$(a, 1) \rightarrow (b, 1.5) \rightarrow (a, 2) \rightarrow (b, 2.25) \rightarrow (a, 3) \rightarrow (b, 3.125) \rightarrow \cdots$$

■

This example illustrates that the model of reals is indeed different from the discrete-time model. If we require all the time values τ_i to be multiples of some fixed constant ϵ, however small, the language accepted by the automaton of Figure 7 will be empty.

On the other hand, timed automata do not distinguish between the set of reals R and the set of rationals Q. Only the denseness of the underlying domain plays a crucial role. In particular, Theorem 3.17 shows that if we require all the time values in time sequences to be rational numbers, the untimed language $Untime[L(\mathcal{A})]$ of a timed automaton $\mathcal{A}$ stays unchanged.

Theorem 3.17 Let L be a timed regular language. For every word σ, $\sigma \in Untime(L)$ iff there exists a time sequence τ such that $\tau_i \in \mathsf{Q}$ for all $i \geq 1$, and $(\sigma, \tau) \in L$.

PROOF. Consider a timed automaton $\mathcal{A}$, and a word σ. If there exists a time sequence τ with all rational time values such that $(\sigma, \tau) \in L(\mathcal{A})$, then clearly, $\sigma \in Untime[L(\mathcal{A})]$.

Now suppose for an arbitrary time sequence τ, $(\sigma, \tau) \in L(\mathcal{A})$. Let $\epsilon \in \mathsf{Q}$ be such that every constant appearing in the clock constraints of $\mathcal{A}$ is an integral multiple of ϵ. Let $\tau'_0 = 0$, and $\tau_0 = 0$. If $\tau_i = \tau_j + n\epsilon$ for some $0 \leq j < i$ and $n \in \mathsf{N}$, then choose $\tau'_i = \tau'_j + n\epsilon$. Otherwise choose $\tau'_i \in \mathsf{Q}$ such that for all $0 \leq j < i$, for all $n \in \mathsf{N}$, $(\tau'_i - \tau'_j) < n\epsilon$ iff $(\tau_i - \tau_j) < n\epsilon$. Note that because of the denseness of Q such a choice of τ'_i is always possible.

Consider an accepting run $r = (\overline{s}, \overline{\nu})$ of $\mathcal{A}$ over (σ, τ). Because of the construction of τ', if a clock x is reset at the i-th transition point, then its possible values at the j-th

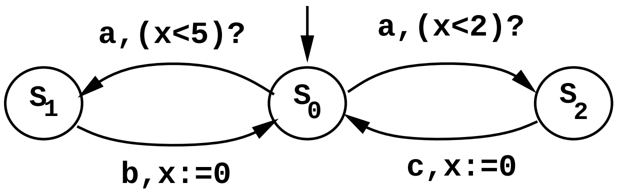


Figure 8: Timed Muller automaton

transition point along the two time sequences, namely, $(\tau_j - \tau_i)$ and $(\tau'_j - \tau'_i)$, satisfy the same set of clock constraints. Consequently it is possible to construct an accepting run $r' = (\overline{s}, \overline{\nu}')$ over (σ, τ') which follows the same sequence of edges as r. In particular, choose $\nu'_0 = \nu_0$, and if the i-th transition along r is according to the edge $\langle s_{i-1}, s_i, \sigma_i, \lambda_i, \delta_i \rangle$, then set $\nu'_i = [\lambda_i \mapsto 0](\nu'_{i-1} + \tau'_i - \tau'_{i-1})$. Consequently, $\mathcal{A}$ accepts (σ, τ'). ■

3.7 Timed Muller automata

We can define timed automata with Muller acceptance conditions also.

Definition 3.18 A *timed Muller automaton* (TMA) is a tuple $\langle \Sigma, \mathrm{S}, \mathrm{S}_0, \mathrm{C}, \mathrm{E}, \mathcal{F} \rangle$, where $\langle \Sigma, \mathrm{S}, \mathrm{S}_0, \mathrm{C}, \mathrm{E} \rangle$ is a timed transition table, and $\mathcal{F} \subseteq 2^{\mathrm{S}}$ specifies an acceptance family.

A run $r = (\overline{s}, \overline{\nu})$ of the automaton over a timed word (σ, τ) is an accepting run iff $\mathit{inf}(r) \in \mathcal{F}$.

For a TMA $\mathcal{A}$, the language $L(\mathcal{A})$ of timed words it accepts is defined to be the set $\{(\sigma, \tau) \mid \mathcal{A} \text{ has an accepting run over } (\sigma, \tau)\}$. ■

Example 3.19 Consider the automaton of Figure 8 over the alphabet $\{a, b, c\}$. The start state is s_0, and the Muller acceptance family consists of a single set $\{s_0, s_2\}$. So any accepting run should cycle between states s_0 and s_1 only finitely many times, and between states s_0 and s_2 infinitely many times. Every word (σ, τ) accepted by the automaton satisfies: (1) $\sigma \in (a(b+c))^*(ac)^\omega$, and (2) for all $i \geq 1$, the difference $(\tau_{2i-1} - \tau_{2i-2})$ is less than 2 if the $(2i)$-th symbol is c, and less than 5 otherwise. ■

Recall that untimed Büchi automata and Muller automata have the same expressive power. The following theorem states that the same holds true for TBAs and TMAs. Thus the class of timed languages accepted by TMAs is the same as the class of timed regular languages. The proof of the following theorem closely follows the standard argument that an ω-regular language is accepted by a Büchi automaton iff it is accepted by some Muller automaton.

Theorem 3.20 A timed language is accepted by some timed Büchi automaton iff it is accepted by some timed Muller automaton.

Proof. Let $\mathcal{A} = \langle \Sigma, \mathrm{S}, \mathrm{S}_0, \mathrm{C}, \mathrm{E}, \mathrm{F} \rangle$ be a TBA. Consider the TMA $\mathcal{A}'$ with the same timed transition table as that of $\mathcal{A}$, and with the acceptance family $\mathcal{F} = \{\mathrm{S}' \subseteq \mathrm{S} : \mathrm{S}' \cap \mathrm{F} \neq \emptyset\}$. It is easy to check that $L(\mathcal{A}) = L(\mathcal{A}')$. This proves the "only if" part of the claim.

In the other direction, given a TMA, we can construct a TBA accepting the same language using the simulation of Muller acceptance condition by Büchi automata. Let $\mathcal{A}$ be a TMA given as $\langle \Sigma, \mathrm{S}, \mathrm{S}_0, \mathrm{C}, \mathrm{E}, \mathcal{F} \rangle$. First note that $L(\mathcal{A}) = \cup_{\mathrm{F} \in \mathcal{F}} L(\mathcal{A}_\mathrm{F})$ where $\mathcal{A}_\mathrm{F} = \langle \Sigma, \mathrm{S}, \mathrm{S}_0, \mathrm{C}, \mathrm{E}, \{\mathrm{F}\} \rangle$, so it suffices to construct, for each acceptance set F, a TBA $\mathcal{A}'_\mathrm{F}$ which accepts the language $L(\mathcal{A}_\mathrm{F})$. Assume $\mathrm{F} = \{s_1, \ldots s_k\}$. The automaton $\mathcal{A}'_\mathrm{F}$ uses nondeterminism to guess when the set F is entered forever, and then uses a counter to make sure that every state in F is visited infinitely often. States of $\mathcal{A}'_\mathrm{F}$ are of the form $\langle s, i \rangle$, where $s \in \mathrm{S}$ and $i \in \{0, 1, \ldots k\}$. The set of initial states is $\mathrm{S}_0 \times \{0\}$. The automaton simulates the transitions of $\mathcal{A}$, and at some point nondeterministically sets the second component to 1. For every transition $\langle s, s', a, \lambda, \delta \rangle$ of $\mathcal{A}$, the automaton $\mathcal{A}'_\mathrm{F}$ has a transition $\langle \langle s, 0 \rangle, \langle s', 0 \rangle, a, \lambda, \delta \rangle$, and, in addition, if $s' \in \mathrm{F}$ it also has a transition $\langle \langle s, 0 \rangle, \langle s', 1 \rangle, a, \lambda, \delta \rangle$.

While the second component is nonzero, the automaton is required to stay within the set F. For every $\mathcal{A}$-transition $\langle s, s', a, \lambda, \delta \rangle$ with both s and s' in F, for each $1 \leq i \leq k$, there is an $\mathcal{A}'_\mathrm{F}$-transition $\langle \langle s, i \rangle, \langle s', j \rangle, a, \lambda, \delta \rangle$ where $j = (i+1)$ *mod* k, if s equals s_i, else $j = i$. The only accepting state is $\langle s_k, k \rangle$. ∎

4 Checking emptiness

In this section we develop an algorithm for checking the emptiness of the language of a timed automaton. The existence of an infinite accepting path in the underlying transition table is clearly a necessary condition for the language of an automaton to be nonempty. However, the timing constraints of the automaton rule out certain additional behaviors. We will show that a Büchi automaton can be constructed that accepts exactly the set of untimed words that are consistent with the timed words accepted by a timed automaton.

4.1 Restriction to integer constants

Recall that our definition of timed automata allows clock constraints which involve comparisons with rational constants. The following lemma shows that, for checking emptiness, we can restrict ourselves to timed automata whose clock constraints involve only integer constants. For a timed sequence τ and $t \in \mathbb{Q}$, let $t \cdot \tau$ denote the timed sequence obtained by multiplying all τ_i by t.

Lemma 4.1 Consider a timed transition table $\mathcal{A}$, a timed word (σ, τ), and $t \in \mathbb{Q}$. $(\overline{s}, \overline{\nu})$ is a run of $\mathcal{A}$ over (σ, τ) iff $(\overline{s}, t \cdot \overline{\nu})$ is a run of $\mathcal{A}_t$ over $(\sigma, t \cdot \tau)$, where $\mathcal{A}_t$ is the timed transition table obtained by replacing each constant d in each clock constraint labeling the edges of $\mathcal{A}$ by $t \cdot d$.

Proof. The lemma can be proved easily from the definitions using induction. ∎

Thus there is an isomorphism between the runs of $\mathcal{A}$ and the runs of $\mathcal{A}_t$. If we choose t to be the least common multiple of denominators of all the constants appearing in the clock constraints of $\mathcal{A}$, then the clock constraints for $\mathcal{A}_t$ use only integer constants. In this translation, the values of the individual constants grow at most with the product of the denominators of all the original constants. We assume binary encoding for the constants.

Let us denote the length of the clock constraints of $\mathcal{A}$ by $|\delta(\mathcal{A})|$. It is easy to prove that $|\delta(\mathcal{A}_t)|$ is bounded by $|\delta(\mathcal{A})|^2$. Observe that this result depends crucially on the fact that we encode constants in binary notation; if we use unary encoding then $|\delta(\mathcal{A}_t)|$ can be exponential in $|\delta(\mathcal{A})|$.

Observe that $L(\mathcal{A})$ is empty iff $L[\mathcal{A}_t]$ is empty. Hence, to decide the emptiness of $L(\mathcal{A})$ we consider $\mathcal{A}_t$. Also $Untime[L(\mathcal{A})]$ equals $Untime[L(\mathcal{A}_t)]$. In the remainder of the section we assume that the clock constraints use only integer constants.

4.2 Clock regions

At every point in time the future behavior of a timed transition table is determined by its state and the values of all its clocks. This motivates the following definition:

Definition 4.2 For a timed transition table $\langle \Sigma, \mathrm{S}, \mathrm{S}_0, \mathrm{C}, \mathrm{E} \rangle$, an *extended state* is a pair $\langle s, \nu \rangle$ where $s \in \mathrm{S}$ and ν is a clock interpretation for C. ■

Since the number of such extended states is infinite (in fact, uncountable), we cannot possibly build an automaton whose states are the extended states of $\mathcal{A}$. But if two extended states with the same $\mathcal{A}$-state agree on the integral parts of all clock values, and also on the ordering of the fractional parts of all clock values, then the runs starting from the two extended states are very similar. The integral parts of the clock values are needed to determine whether or not a particular clock constraint is met, whereas the ordering of the fractional parts is needed to decide which clock will change its integral part first. For example, if two clocks x and y are between 0 and 1 in an extended state, then a transition with clock constraint $(x = 1)$ can be followed by a transition with clock constraint $(y = 1)$, depending on whether or not the current clock values satisfy $(x < y)$.

The integral parts of clock values can get arbitrarily large. But if a clock x is never compared with a constant greater than c, then its actual value, once it exceeds c, is of no consequence in deciding the allowed paths.

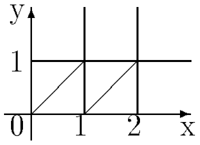
Now we formalize this notion. For any $t \in \mathsf{R}$, $\mathit{fract}(t)$ denotes the fractional part of t, and $\lfloor t \rfloor$ denotes the integral part of t; that is, $t = \lfloor t \rfloor + \mathit{fract}(t)$. We assume that every clock in C appears in some clock constraint.

Definition 4.3 Let $\mathcal{A} = \langle \Sigma, \mathrm{S}, \mathrm{S}_0, \mathrm{C}, \mathrm{E} \rangle$ be a timed transition table. For each $x \in \mathrm{C}$, let c_x be the largest integer c such that $(x \leq c)$ or $(c \leq x)$ is a subformula of some clock constraint appearing in E.

The equivalence relation $\sim$ is defined over the set of all clock interpretations for C; $\nu \sim \nu'$ iff all the following conditions hold:

1. For all $x \in \mathrm{C}$, either $\lfloor \nu(x) \rfloor$ and $\lfloor \nu'(x) \rfloor$ are the same, or both $\nu(x)$ and $\nu'(x)$ are greater than c_x.
2. For all $x, y \in \mathrm{C}$ with $\nu(x) \leq c_x$ and $\nu(y) \leq c_y$, $\mathit{fract}(\nu(x)) \leq \mathit{fract}(\nu(y))$ iff $\mathit{fract}(\nu'(x)) \leq \mathit{fract}(\nu'(y))$.
3. For all $x \in \mathrm{C}$ with $\nu(x) \leq c_x$, $\mathit{fract}(\nu(x)) = 0$ iff $\mathit{fract}(\nu'(x)) = 0$.

A *clock region* for $\mathcal{A}$ is an equivalence class of clock interpretations induced by $\sim$. ■



6 Corner points: e.g. [(0,1)]
14 Open line segments: e.g. $[0 < x = y < 1]$
8 Open regions: e.g. $[0 < x < y < 1]$

Figure 9: Clock regions

We will use $[\nu]$ to denote the clock region to which ν belongs. Each region can be uniquely characterized by a (finite) set of clock constraints it satisfies. For example, consider a clock interpretation ν over two clocks with $\nu(x) = 0.3$ and $\nu(y) = 0.7$. Every clock interpretation in $[\nu]$ satisfies the constraint $(0 < x < y < 1)$, and we will represent this region by $[0 < x < y < 1]$. The nature of the equivalence classes can be best understood through an example.

Example 4.4 Consider a timed transition table with two clocks x and y with $c_x = 2$ and $c_y = 1$. The clock regions are shown in Figure 9. ■

Note that there are only a finite number of regions. Also note that for a clock constraint δ of $\mathcal{A}$, if $\nu \sim \nu'$ then ν satisfies δ iff ν' satisfies δ. We say that a clock region α satisfies a clock constraint δ iff every $\nu \in \alpha$ satisfies δ. Each region can be represented by specifying

(1) for every clock x, one clock constraint from the set

$$\{x = c \mid c = 0, 1, \ldots c_x\} \cup \{c - 1 < x < c \mid c = 1, \ldots c_x\} \cup \{x > c_x\},$$

(2) for every pair of clocks x and y such that $c - 1 < x < c$ and $d - 1 < y < d$ appear in (1) for some c, d, whether $\mathit{fract}(x)$ is less than, equal to, or greater than $\mathit{fract}(y)$.

By counting the number of possible combinations of equations of the above form, we get the upper bound in the following lemma.

Lemma 4.5 The number of clock regions is bounded by $[|\mathrm{C}|! \cdot 2^{|\mathrm{C}|} \cdot \Pi_{x \in \mathrm{C}}(2c_x + 2)]$. ■

Remember that $|\delta(\mathcal{A})|$ stands for the length of the clock constraints of $\mathcal{A}$ assuming binary encoding, and hence the product $\Pi_{x \in \mathrm{C}}(2c_x + 2)$ is $O[2^{|\delta(\mathcal{A})|}]$. Since the number of clocks $|\mathrm{C}|$ is bounded by $|\delta(\mathcal{A})|$, henceforth, we assume that the number of regions is $O[2^{|\delta(\mathcal{A})|}]$. Note that if we increase $\delta(\mathcal{A})$ without increasing the number of clocks or the size of the largest constants the clocks are compared with, then the number of regions does not grow with $|\delta(\mathcal{A})|$. Also observe that a region can be represented in space linear in $|\delta(\mathcal{A})|$.

4.3 The region automaton

The first step in the decision procedure for checking emptiness is to construct a transition table whose paths mimic the runs of $\mathcal{A}$ in a certain way. We will denote the desired transition table by $R(\mathcal{A})$, the *region automaton* of $\mathcal{A}$. A state of $R(\mathcal{A})$ records the state of the timed transition table $\mathcal{A}$, and the equivalence class of the current values of the clocks. It is of the form $\langle s, \alpha \rangle$ with $s \in \mathrm{S}$ and α being a clock region. The intended interpretation is that whenever the extended state of $\mathcal{A}$ is $\langle s, \nu \rangle$, the state of $R(\mathcal{A})$ is $\langle s, [\nu] \rangle$. The region automaton starts in some state $\langle s_0, [\nu_0] \rangle$ where s_0 is a start state of $\mathcal{A}$, and the clock interpretation ν_0 assigns 0 to every clock. The transition relation of $R(\mathcal{A})$ is defined so that the intended simulation is obeyed. It has an edge from $\langle s, \alpha \rangle$ to $\langle s', \alpha' \rangle$ labeled with a iff $\mathcal{A}$ in state s with the clock values $\nu \in \alpha$ can make a transition on a to the extended state $\langle s', \nu' \rangle$ for some $\nu' \in \alpha'$.

The edge relation can be conveniently defined using a *time-successor* relation over the clock regions. The time-successors of a clock region α are all the clock regions that will be visited by a clock interpretation $\nu \in \alpha$ as time progresses.

Definition 4.6 A clock region α' is a time-successor of a clock region α iff for each $\nu \in \alpha$, there exists a positive $t \in \mathrm{R}$ such that $\nu + t \in \alpha'$. ■

Example 4.7 Consider the clock regions shown in Figure 9 again. The time-successors of a region α are the regions that can be reached by moving along a line drawn from some point in α in the diagonally upwards direction (parallel to the line $x = y$). For example, the region $[(1 < x < 2), (0 < y < x - 1)]$ has, other than itself, the following regions as time-successors: $[(x = 2), (0 < y < 1)]$, $[(x > 2), (0 < y < 1)]$, $[(x > 2), (y = 1)]$ and $[(x > 2), (y > 1)]$. ■

Now let us see how to construct all the time-successors of a clock region. Recall that a clock region α is specified by giving (1) for every clock x, a constraint of the form $(x = c)$ or $(c - 1 < x < c)$ or $(x > c_x)$, and (2) for every pair x and y such that $(c - 1 < x < c)$ and $(d - 1 < y < d)$ appear in (1), the ordering relationship between $\mathit{fract}(x)$ and $\mathit{fract}(y)$. To compute all the time-successors of α we proceed as follows. First observe that the time-successor relation is a transitive relation. We consider different cases.

First suppose that α satisfies the constraint $(x > c_x)$ for every clock x. The only time-successor of α is itself. This is the case for the region $[(x > 2), (y > 1)]$ in Figure 9.

Now suppose that the set C_0 consisting of clocks x such that α satisfies the constraint $(x = c)$ for some $c \leq c_x$, is nonempty. In this case, as time progresses the fractional parts of the clocks in C_0 become nonzero, and the clock region changes immediately. The time-successors of α are same as the time-successors of the clock region β specified as below:

(1) For $x \in \mathrm{C}_0$, if α satisfies $(x = c_x)$ then β satisfies $(x > c_x)$, otherwise if α satisfies $(x = c)$ then β satisfies $(c < x < c+1)$. For $x \notin \mathrm{C}_0$, the constraint in β is the same as that in α.

(2) For clocks x and y such that $x < c_x$ and $y < c_y$ holds in α, the ordering relationship in β between their fractional parts is the same as in α.

For instance, in Figure 9, the time-successors of $[(x = 0), (0 < y < 1)]$ are same as the time-successors of $[0 < x < y < 1]$.

If both the above cases do not apply, then let C_0 be the set of clocks x for which α does not satisfy $(x > c_x)$ and which have the maximal fractional part; that is, for all clocks y for which α does not satisfy $(y > c_y)$, $\mathit{fract}(y) \leq \mathit{fract}(x)$ is a constraint of α. In this case, as time progresses, the clocks in C_0 assume integer values. Let β be the clock region specified by

(1) For $x \in C_0$, if α satisfies $(c - 1 < x < c)$ then β satisfies $(x = c)$. For $x \notin C_0$, the constraint in β is same as that in α.
(2) For clocks x and y such that $(c - 1 < x < c)$ and $(d - 1 < y < d)$ appear in (1), the ordering relationship in β between their fractional parts is same as in α.

In this case, the time-successors of α include α, β, and all the time-successors of β. For instance, in Figure 9, time-successors of $[0 < x < y < 1]$ include itself, $[(0 < x < 1), (y = 1)]$, and all the time-successors of $[(0 < x < 1), (y = 1)]$.

Now we are ready to define the region automaton.

Definition 4.8 For a timed transition table $\mathcal{A} = \langle \Sigma, S, S_0, C, E \rangle$, the corresponding region automaton $R(\mathcal{A})$ is a transition table over the alphabet Σ.

- The states of $R(\mathcal{A})$ are of the form $\langle s, \alpha \rangle$ where $s \in S$ and α is a clock region.
- The initial states are of the form $\langle s_0, [\nu_0] \rangle$ where $s_0 \in S_0$ and $\nu_0(x) = 0$ for all $x \in C$.
- $R(\mathcal{A})$ has an edge $\langle \langle s, \alpha \rangle, \langle s', \alpha' \rangle, a \rangle$ iff there is an edge $\langle s, s', a, \lambda, \delta \rangle \in E$ and a region α'' such that (1) α'' is a time-successor of α, (2) α'' satisfies δ, and (3) $\alpha' = [\lambda \mapsto 0]\alpha''$.

■

Example 4.9 Consider the timed automaton $\mathcal{A}_0$ shown in Figure 10. The alphabet is $\{a, b, c, d\}$. Every state of the automaton is an accepting state. The corresponding region automaton $R(\mathcal{A}_0)$ is also shown. Only the regions reachable from the initial region $\langle s_0, [x = y = 0] \rangle$ are shown. Note that $c_x = 1$ and $c_y = 1$. The timing constraints of the automaton ensure that the transition from s_2 to s_3 is never taken. The only reachable region with state component s_2 satisfies the constraints $[y = 1, x > 1]$, and this region has no outgoing edges. Thus the region automaton helps us in concluding that no transitions can follow a b-transition. ■

From the bound on the number of regions, it follows that the number of states in $R(\mathcal{A})$ is $O[|S| \cdot 2^{|\delta(\mathcal{A})|}]$. An inspection of the definition of the time-successor relation shows that every region has at most $\Sigma_{x \in C}[2c_x + 2]$ successor regions. The region automaton has at most one edge out of $\langle s, \alpha \rangle$ for every edge out of s and every time-successor of α. It follows that the number of edges in $R(\mathcal{A})$ is $O[|E| \cdot 2^{|\delta(\mathcal{A})|}]$. Note that computing the time-successor relation is easy, and can be done in time linear in the length of the representation of the region. Constructing the edge relation for the region automaton is also relatively easy; in addition to computing the time-successors, we also need to determine whether the clock

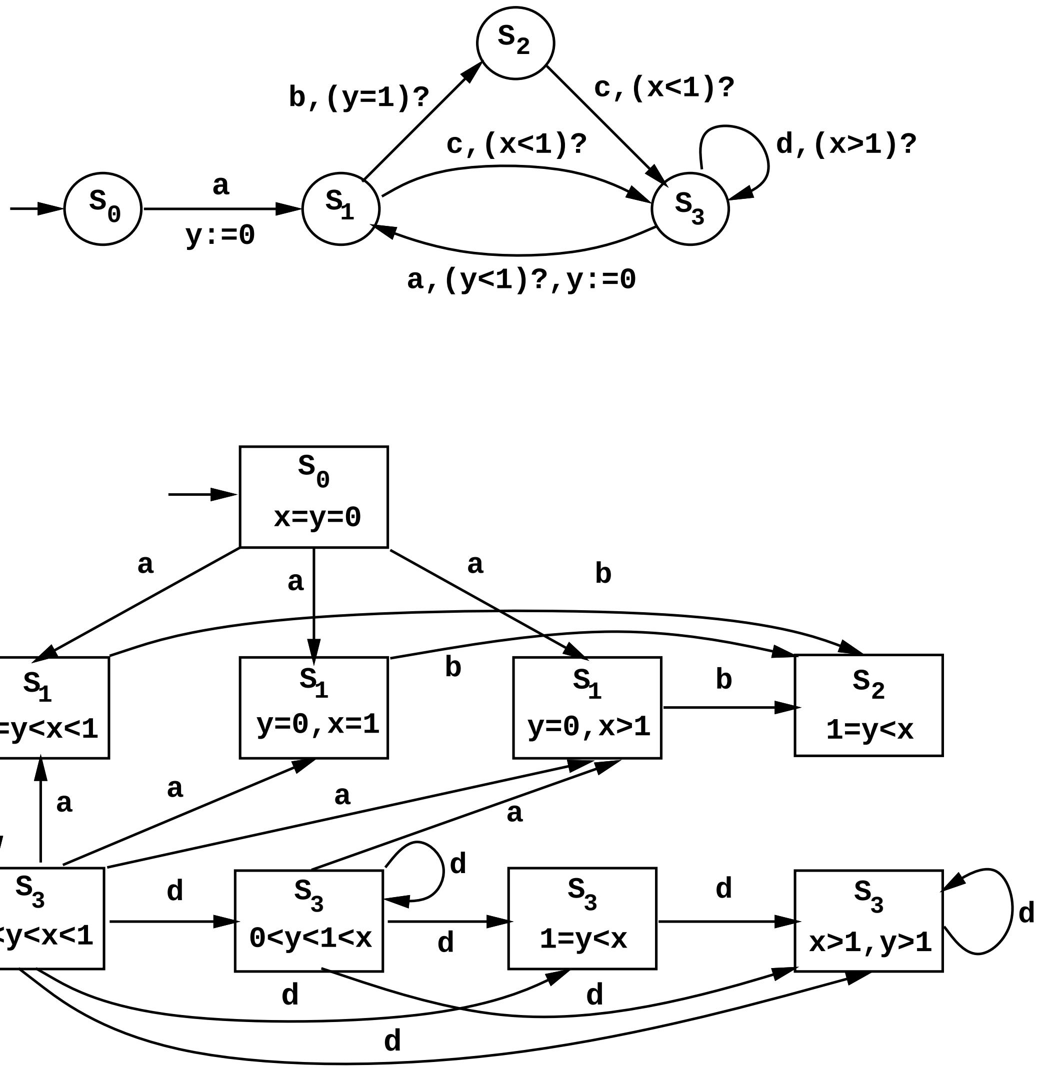


Figure 10: Automaton $\mathcal{A}_0$ and its region automaton

constraint labeling a particular $\mathcal{A}$-transition is satisfied by a clock region. The region graph can be constructed in time $O[(|\mathrm{S}| + |\mathrm{E}|)\cdot 2^{|\delta(\mathcal{A})|}]$.

Now we proceed to establish a correspondence between the runs of $\mathcal{A}$ and the runs of $R(\mathcal{A})$.

Definition 4.10 For a run $r = (\overline{s}, \overline{\nu})$ of $\mathcal{A}$ of the form

$$r \quad : \quad \langle s_0, \nu_0 \rangle \ \xrightarrow[\tau_1]{\sigma_1} \ \langle s_1, \nu_1 \rangle \ \xrightarrow[\tau_2]{\sigma_2} \ \langle s_2, \nu_2 \rangle \ \xrightarrow[\tau_3]{\sigma_3} \ \cdots$$

define its projection $[r] = (\overline{s}, [\overline{\nu}])$ to be the sequence

$$[r] \quad : \quad \langle s_0, [\nu_0] \rangle \ \xrightarrow{\sigma_1} \ \langle s_1, [\nu_1] \rangle \ \xrightarrow{\sigma_2} \ \langle s_2, [\nu_2] \rangle \ \xrightarrow{\sigma_3} \ \cdots$$

■

From the definition of the edge relation for $R(\mathcal{A})$, it follows that $[r]$ is a run of $R(\mathcal{A})$ over σ. Since time progresses without bound along r, every clock $x \in \mathrm{C}$ is either reset infinitely often, or from a certain time onwards it increases without bound. Hence, for all $x \in \mathrm{C}$, for infinitely many $i \geq 0$, $[\nu_i]$ satisfies $[(x = 0) \vee (x > c_x)]$. This prompts the following definition:

Definition 4.11 A run $r = (\overline{s}, \overline{\alpha})$ of the region automaton $R(\mathcal{A})$ of the form

$$r : \langle s_0, \alpha_0 \rangle \xrightarrow{\sigma_1} \langle s_1, \alpha_1 \rangle \xrightarrow{\sigma_2} \langle s_2, \alpha_2 \rangle \xrightarrow{\sigma_3} \cdots$$

is *progressive* iff for each clock $x \in \mathrm{C}$, there are infinitely many $i \geq 0$ such that α_i satisfies $[(x = 0) \vee (x > c_x)]$. ■

Thus for a run r of $\mathcal{A}$ over (σ, τ), $[r]$ is a progressive run of $R(\mathcal{A})$ over σ. The following Lemma 4.13 implies that progressive runs of $R(\mathcal{A})$ precisely correspond to the projected runs of $\mathcal{A}$. Before we prove the lemma let us consider the region automaton of Example 4.9 again.

Example 4.12 Consider the region automaton $R(\mathcal{A}_0)$ of Figure 10. Every run r of $R(\mathcal{A}_0)$ has a suffix of one of the following three forms: (i) the automaton cycles between the regions $\langle s_1, [y = 0 < x < 1] \rangle$ and $\langle s_3, [0 < y < x < 1] \rangle$, (ii) the automaton stays in the region $\langle s_3, [0 < y < 1 < x] \rangle$ using the self-loop, or (iii) the automaton stays in the region $\langle s_3, [x > 1, y > 1] \rangle$.

Only the case (iii) corresponds to the progressive runs. For runs of type (i), even though y gets reset infinitely often, the value of x is always less than 1. For runs of type (ii), even though the value of x is not bounded, the clock y is reset only finitely often, and yet, its value is bounded. Thus every progressive run of $\mathcal{A}_0$ corresponds to a run of $R(\mathcal{A}_0)$ of type (iii). ■

Lemma 4.13 If r is a progressive run of $R(\mathcal{A})$ over σ then there exists a time sequence τ and a run r' of $\mathcal{A}$ over (σ, τ) such that r equals $[r']$.

Proof. Consider a progressive run $r = (\overline{s}, \overline{\alpha})$ of $R(\mathcal{A})$ over σ. We construct the run r' and the time sequence τ step by step. As usual, r' starts with $\langle s_0, \nu_0 \rangle$. Now suppose that the extended state of $\mathcal{A}$ is $\langle s_i, \nu_i \rangle$ at time τ_i with $\nu_i \in \alpha_i$. There is an edge in $R(\mathcal{A})$ from $\langle s_i, \alpha_i \rangle$ to $\langle s_{i+1}, \alpha_{i+1} \rangle$ labeled with σ_{i+1}. From the definition of the region automaton it follows that there is an edge $\langle s_i, s_{i+1}, \sigma_{i+1}, \lambda_{i+1}, \delta_{i+1} \rangle \in \mathrm{E}$ and a time-successor α'_{i+1} of α_i such that α'_{i+1} satisfies δ_{i+1} and $\alpha_{i+1} = [\lambda_{i+1} \mapsto 0]\alpha'_{i+1}$. From the definition of time-successor, there exists a time τ_{i+1} such that $(\nu_i + \tau_{i+1} - \tau_i) \in \alpha'_{i+1}$. Now it is clear the next transition of $\mathcal{A}$ can be at time τ_{i+1} to an extended state $\langle s_{i+1}, \nu_{i+1} \rangle$ with $\nu_{i+1} \in \alpha_{i+1}$. Using this construction repeatedly we get a run $r' = (\overline{s}, \overline{\nu})$ over (σ, τ) with $[r'] = r$.

The only problem with the above construction is that τ may not satisfy the progress condition. Suppose that τ is a converging sequence. We use the fact that r is a progressive run to construct another time sequence τ' satisfying the progress requirement and show that the automaton can follow the same sequence of transitions as r' but at times τ'_i.

Let C_0 be the set of clocks reset infinitely often along r. Since τ is a converging sequence, after a certain position onwards, every clock in C_0 gets reset before it reaches the value 1. Since r is progressive, every clock x not in C_0, after a certain position onwards, never gets reset, and continuously satisfies $x > c_x$. This ensures that there exists $j \geq 0$ such that (1) after the j-th transition point each clock $x \notin \mathrm{C}_0$ continuously satisfies $(x > c_x)$, and each clock $x \in \mathrm{C}_0$ continuously satisfies $(x < 1)$, and (2) for each $k > j$, $(\tau_k - \tau_j)$ is less than 0.5.

Let $j < k_1 < k_2, \ldots$ be an infinite sequence of integers such that each clock x in C_0 is reset at least once between the k_i-th and k_{i+1}-th transition points along r. Now we construct another sequence $r'' = (\overline{s}, \overline{\nu}')$ with the sequence of transition times τ' as follows. The sequence of transitions along r'' is same as that along r'. If $i \notin \{k_1, k_2 \ldots\}$ then we require the $(i+1)$-th transition to happen after a delay of $(\tau_{i+1} - \tau_i)$, otherwise we require the delay to be 0.5. Observe that along r'' the delay between the k_i-th and k_{i+1}-th transition points is less than 1. Consequently, in spite of the additional delays, the value of every clock in C_0 remains less than 1 after the j-th transition point. So the truth of all the clock constraints and the clock regions at the transition points remain unchanged (as compared to r'). From this we conclude that r'' satisfies the consecution requirement, and is a run of $\mathcal{A}$. Furthermore, $[r''] = [r'] = r$.

Since τ' has infinitely many jumps each of duration 0.5, it satisfies the progress requirement. Hence r'' is the run required by the lemma. ■

4.4 The untiming construction

For a timed automaton $\mathcal{A}$, its region automaton can be used to recognize $Untime[L(\mathcal{A})]$. The following theorem is stated for TBAs, but it also holds for TMAs.

Theorem 4.14 Given a TBA $\mathcal{A} = \langle \Sigma, \mathrm{S}, \mathrm{S}_0, \mathrm{C}, \mathrm{E}, \mathrm{F} \rangle$, there exists a Büchi automaton over Σ which accepts $Untime[L(\mathcal{A})]$.

Proof. We construct a Büchi automaton $\mathcal{A}'$ as follows. Its transition table is $R(\mathcal{A})$, the region automaton corresponding to the timed transition table $\langle \Sigma, \mathrm{S}, \mathrm{S}_0, \mathrm{C}, \mathrm{E} \rangle$. The accepting set of $\mathcal{A}'$ is $\mathrm{F}' = \{\langle s, \alpha \rangle \mid s \in \mathrm{F}\}$.

If r is an accepting run of $\mathcal{A}$ over (σ, τ), then $[r]$ is a progressive and accepting run of $\mathcal{A}'$ over σ. The converse follows from Lemma 4.13. Given a progressive run r of $\mathcal{A}'$ over σ, the lemma gives a time sequence τ and a run r' of $\mathcal{A}$ over (σ, τ) such that r equals $[r']$. If r is an accepting run, so is r'. It follows that $\sigma \in Untime[L(\mathcal{A})]$ iff $\mathcal{A}'$ has a progressive, accepting run over it.

For $x \in \mathrm{C}$, let $\mathrm{F}_x = \{\langle s, \alpha \rangle \mid \alpha \models [(x = 0) \vee (x > c_x)]\}$. Recall that a run of $\mathcal{A}'$ is progressive iff some state from each F_x repeats infinitely often. It is straightforward to construct another Büchi automaton $\mathcal{A}''$ such that $\mathcal{A}'$ has a progressive and accepting run over σ iff $\mathcal{A}''$ has an accepting run over σ.

The automaton $\mathcal{A}''$ is the desired automaton; $L(\mathcal{A}'')$ equals $Untime[L(\mathcal{A})]$. ■

Example 4.15 Let us consider the region automaton $R(\mathcal{A}_0)$ of Example 4.9 again. Since all states of $\mathcal{A}_0$ are accepting, from the description of the progressive runs in Example 4.12 it follows that the transition table $R(\mathcal{A}_0)$ can be changed to a Büchi automaton by choosing the accepting set to consist of a single region $\langle s_3, [x > 1, y > 1] \rangle$. Consequently

$$Untime[L(\mathcal{A}_0)] \;=\; L[R(\mathcal{A}_0)] \;=\; ac\,(ac)^*\,d^\omega.$$

■

Theorem 4.14 says that the timing information in a timed automaton is "regular" in character; its consistency can be checked by a finite-state automaton. An equivalent formulation of the theorem is

If a timed language L is timed regular then $Untime(L)$ is ω-regular.

Furthermore, to check whether the language of a given TBA is empty, we can check for the emptiness of the language of the corresponding Büchi automaton constructed by the proof of Theorem 4.14. The next theorem follows.

Theorem 4.16 Given a timed Büchi automaton $\mathcal{A} = \langle \Sigma, \mathrm{S}, \mathrm{S}_0, \mathrm{C}, \mathrm{E}, \mathrm{F} \rangle$ the emptiness of $L(\mathcal{A})$ can be checked in time $O[(|\mathrm{S}| + |\mathrm{E}|) \cdot 2^{|\delta(\mathcal{A})|}]$.

Proof. Let $\mathcal{A}'$ be the Büchi automaton constructed as outlined in the proof of Theorem 4.14. Recall that in Section 4.3 we had shown that the number of states in $\mathcal{A}'$ is $O[|\mathrm{S}| \cdot 2^{|\delta(\mathcal{A})|}]$, the number of edges is $O[|\mathrm{E}| \cdot 2^{|\delta(\mathcal{A})|}]$.

The language $L(\mathcal{A})$ is nonempty iff there is a cycle C in $\mathcal{A}'$ such that C is accessible from some start state of $\mathcal{A}'$ and C contains at least one state each from the set F' and each of the sets F_x. This can be checked in time linear in the size of $\mathcal{A}'$ [41]. The complexity bound of the theorem follows. ■

Recall that if we start with an automaton $\mathcal{A}$ whose clock constraints involve rational constants, we need to apply the above decision procedure on $\mathcal{A}_t$ for the least common denominator t of all the rational constants (see Section 4.1). This involves a blow-up in the size of the clock constraints; we have $\delta[\mathcal{A}_t] = O[\delta(\mathcal{A})^2]$.

The above method can be used even if we change the acceptance condition for timed automata. In particular, given a timed Muller automaton $\mathcal{A}$ we can effectively construct a Muller (or, Büchi) automaton which accepts $Untime[L(\mathcal{A})]$, and use it to check for the emptiness of $L(\mathcal{A})$.

4.5 Complexity of checking emptiness

The complexity of the algorithm for deciding emptiness of a TBA is exponential in the number of clocks and the length of the constants in the timing constraints. This blow-up in complexity seems unavoidable; we reduce the acceptance problem for linear bounded automata, a known PSPACE-complete problem [23], to the emptiness question for TBAs to prove the PSPACE lower bound for the emptiness problem. We also show the problem to be PSPACE-complete by arguing that the algorithm of Section 4.4 can be implemented in polynomial space.

Theorem 4.17 The problem of deciding the emptiness of the language of a given timed automaton $\mathcal{A}$, is PSPACE-complete.

Proof. [PSPACE-membership] Since the number of states of the region automaton is exponential in the number of clocks of $\mathcal{A}$, we cannot construct the entire transition table. But it is possible to (nondeterministically) check for nonemptiness of the region automaton by guessing a path of the desired form using only polynomial space. This is a fairly standard trick, and hence we omit the details.

[PSPACE-hardness] The question of deciding whether a given linear bounded automaton accepts a given input string is PSPACE-complete [23]. A linear bounded automaton M is a nondeterministic Turing machine whose tape head cannot go beyond the end of the input markers. We construct a TBA $\mathcal{A}$ such that its language is nonempty iff the machine M halts on a given input.

Let Γ be the tape alphabet of M and let Q be its states. Let $\Sigma = \Gamma \cup (\Gamma \times Q)$, and let $a_1, a_2, \ldots a_k$ denote the elements of Σ. A configuration of M in which the tape reads $\gamma_1\gamma_2 \ldots \gamma_n$, and the machine is in state q reading the i-th tape symbol, is represented by the string $\sigma_1, \ldots \sigma_n$ over Σ such that $\sigma_j = \gamma_j$ if $j \neq i$ and $\sigma_i = \langle \gamma_i, q \rangle$.

The acceptance corresponds to a special state q_f; after which the configuration stays unchanged. The alphabet of $\mathcal{A}$ includes Σ, and in addition, has a symbol a_0. A computation of M is encoded by the word

$$\sigma_1^1 a_0 \ldots \sigma_n^1 a_0 \sigma_1^2 a_0 \ldots \sigma_n^2 a_0 \ldots \sigma_1^j a_0 \ldots \sigma_n^j a_0 \ldots$$

such that $\sigma_1^j \ldots \sigma_n^j$ encodes the j-th configuration according to the above scheme. The time sequence associated with this word also encodes the computation: we require the time difference between successive a_0's to be $k+1$, and if $\sigma_i^j = a_l$ then we require its time to be l greater than the time of the previous a_0. The encoding in the time sequence is used to enforce the consecution requirement.

We want to construct $\mathcal{A}$ which accepts precisely the timed words encoding the halting computations of M according to the above scheme. We only sketch the construction. $\mathcal{A}$ uses $2n+1$ clocks. The clock x is reset with each a_0. While reading a_0 we require $(x = k+1)$ to hold, and while reading a_i we require $(x = i)$ to hold. These conditions ensure that the encoding in the time sequence is consistent with the word. For each tape cell i, we have two clocks x_i and y_i. The clock x_i is reset with σ_i^j, for odd values of j, and the clock y_i is reset with σ_i^j, for even values of j. Assume that the automaton has read the first j configurations, with j odd. The value of the clock x_i represents the i-th cell of the j-th configuration. Consequently, the possible choices for the values of σ_i^{j+1} are determined by examining the values of x_{i-1}, x_i and x_{i+1} according to the transition rules for M. While reading the $(j+1)$-th configuration, the y-clocks get set to appropriate values; these values are examined while reading the $(j+2)$-th configuration. This ensures proper consecution of configurations. Proper initialization and halting can be enforced in a straightforward way. The size of $\mathcal{A}$ is polynomial in n and the size of M. ∎

Note that the source of this complexity is not the choice of R to model time. The PSPACE-hardness result can be proved if we leave the syntax of timed automata unchanged, but use the discrete domain N to model time. Also this complexity is insensitive to the encoding of the constants; the problem is PSPACE-complete even if we encode all constants in unary.

5 Intractable problems

In this section we show the universality problem for timed automata to be undecidable. The universality problem is to decide whether the language of a given automaton over

Σ comprises all the timed words over Σ. Specifically, we show that the problem is Π_1^1-hard by reducing a Π_1^1-hard problem of 2-counter machines. The class Π_1^1 consists of highly undecidable problems, including some nonarithmetical sets (for an exposition of the analytical hierarchy consult, for instance, [39]). Note that the universality problem is same as deciding emptiness of the complement of the language of the automaton. The undecidability of this problem has several implications such as nonclosure under complement and undecidability of testing for language inclusion.

5.1 A Σ_1^1-complete problem

A *nondeterministic 2-counter machine* M consists of two counters C and D, and a sequence of n instructions. Each instruction may increment or decrement one of the counters, or jump, conditionally upon one of the counters being zero. After the execution of a nonjump instruction, M proceeds nondeterministically to one of the two specified instructions.

We represent a configuration of M by a triple $\langle i, c, d\rangle$, where $1 \leq i \leq n$, $c \geq 0$, and $d \geq 0$ give the values of the location counter and the two counters C and D, respectively. The consecution relation on configurations is defined in the obvious way. A *computation* of M is an infinite sequence of related configurations, starting with the initial configuration $\langle 1, 0, 0\rangle$. It is called *recurring* iff it contains infinitely many configurations in which the location counter has the value 1.

The problem of deciding whether a nondeterministic Turing machine has, over the empty tape, a computation in which the starting state is visited infinitely often, is known to be Σ_1^1-complete [19]. Along the same lines we obtain the following result.

Lemma 5.1 The problem of deciding whether a given nondeterministic 2-counter machine has a recurring computation, is Σ_1^1-hard. ■

5.2 Undecidability of the universality problem

Now we proceed to encode the computations of 2-counter machines using timed automata, and use the encoding to prove the undecidability result.

Theorem 5.2 Given a timed automaton over an alphabet Σ the problem of deciding whether it accepts all timed words over Σ is Π_1^1-hard.

PROOF. We encode the computations of a given 2-counter machine M with n instructions using timed words over the alphabet $\{b_1, \ldots b_n, a_1, a_2\}$. A configuration $\langle i, c, d\rangle$ is represented by the sequence $b_i a_1^c a_2^d$. We encode a computation by concatenating the sequences representing the individual configurations. We use the time sequence associated with a timed word σ to express that the successive configurations are related as per the requirements of the program instructions. We require that the subsequence of σ corresponding to the time interval $[j, j+1)$ encodes the j-th configuration of the computation. Note that the denseness of the underlying time domain allows the counter values to get arbitrarily large. To enforce a requirement such as the number of a_1 symbols in two intervals encoding the successive configurations is the same we require that every a_1 in the first interval has a matching a_1 at distance 1 and vice versa.

Define a timed language L_{undec} as follows. (σ, τ) is in L_{undec} iff

- $\sigma = b_{i_1} a_1^{c_1} a_2^{d_1} b_{i_2} a_1^{c_2} a_2^{d_2} \cdots$ such that $\langle i_1, c_1, d_1 \rangle, \langle i_2, c_2, d_2 \rangle \cdots$ is a recurring computation of M.
- For all $j \geq 1$, the time of b_{i_j} is j.
- For all $j \geq 1$,
  - if $c_{j+1} = c_j$ then for every a_1 at time t in the interval $(j, j+1)$ there is an a_1 at time $t+1$.
  - if $c_{j+1} = c_j + 1$ then for every a_1 at time t in the interval $(j+1, j+2)$ except the last one, there is an a_1 at time $t-1$.
  - if $c_{j+1} = c_j - 1$ then for every a_1 at time t in the interval $(j, j+1)$ except the last one, there is an a_1 at time $t+1$.

  Similar requirements hold for a_2's.

Clearly, L_{undec} is nonempty iff M has a recurring computation. We will construct a timed automaton $\mathcal{A}_{undec}$ which accepts the complement of L_{undec}. Hence $\mathcal{A}_{undec}$ accepts every timed word iff M does not have a recurring computation. The theorem follows from Lemma 5.1.

The desired automaton $\mathcal{A}_{undec}$ is a disjunction of several TBAs.

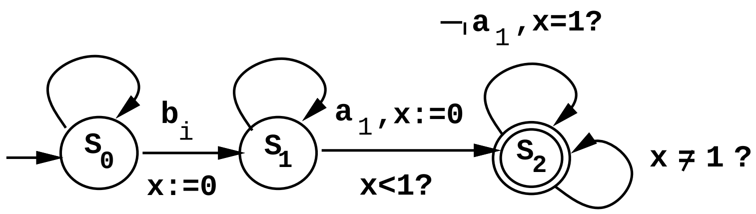
Let $\mathcal{A}_0$ be the TBA which accepts (σ, τ) iff for some integer $j \geq 1$, either there is no b symbol at time j, or the subsequence of σ in the time interval $(j, j+1)$ is not of the form $a_1^* a_2^*$. It is easy to construct such a timed automaton.

A timed word (σ, τ) in L_{undec} should encode the initial configuration over the interval $[1, 2)$. Let $\mathcal{A}_{init}$ be the TBA which requires that the subsequence of σ corresponding to the interval $[1, 2)$ is not b_1; it accepts the language $\{(\sigma, \tau) \mid (\sigma_1 \neq b_1) \vee (\tau_1 \neq 1) \vee (\tau_2 < 2)\}$.

For each instruction $1 \leq i \leq n$ we construct a TBA $\mathcal{A}_i$. $\mathcal{A}_i$ accepts (σ, τ) iff the timed word has b_i at some time t, and the configuration corresponding to the subsequence in $[t+1, t+2)$ does not follow from the configuration corresponding to the subsequence in $[t, t+1)$ by executing the instruction i. We give the construction for a sample instruction, say, "increment the counter D and jump nondeterministically to instruction 3 or 5". The automaton $\mathcal{A}_i$ is the disjunction of the following six TBAs $\mathcal{A}_i^1, \ldots \mathcal{A}_i^6$.

Let $\mathcal{A}_i^1$ be the automaton which accepts (σ, τ) iff for some $j \geq 1$, $\sigma_j = b_i$, and at time $\tau_j + 1$ there is neither b_3 nor b_5. It is easy to construct this automaton.

Let $\mathcal{A}_i^2$ be the following TBA:



In this figure, an edge without a label means that the transition can be taken on every input symbol. While in state s_2, the automaton cannot accept a symbol a_1 if the condition $(x = 1)$ holds. Thus $\mathcal{A}_i^2$ accepts (σ, τ) iff there is some b_i at time t followed by an a_1 at time $t' < (t+1)$ such that there is no matching a_1 at time $(t'+1)$.

Similarly we can construct $\mathcal{A}_i^3$ which accepts (σ, τ) iff there is some b_i at time t, and for some $t' < (t+1)$ there is no a_1 at time t' but there is an a_1 at time $(t'+1)$. The complements of $\mathcal{A}_i^2$ and $\mathcal{A}_i^3$ together ensure proper matching of a_1's.

Along similar lines we ensure proper matching of a_2 symbols. Let $\mathcal{A}_i^4$ be the automaton which requires that for some b_i at time t, there is an a_2 at some $t' < (t+1)$ with no match at $(t'+1)$. Let $\mathcal{A}_i^5$ be the automaton which says that for some b_i at time t there are two a_2's in $(t+1, t+2)$ without matches in $(t, t+1)$. Let $\mathcal{A}_i^6$ be the automaton which requires that for some b_i at time t the last a_2 in the interval $(t+1, t+2)$ has a matching a_2 in $(t, t+1)$. Now consider a word (σ, τ) such that there is b_i at some time t such that the encoding of a_2's in the intervals $(t, t+1)$ and $(t+1, t+2)$ do not match according to the desired scheme. Let the number of a_2's in $(t, t+1)$ and in $(t+1, t+2)$ be k and l respectively. If $k > l$ then the word is accepted by $\mathcal{A}_i^4$. If $k = l$, then either there is no match for some a_2 in $(t, t+1)$, or every a_2 in $(t, t+1)$ has a match in $(t+1, t+2)$. In the former case the word is accepted by $\mathcal{A}_i^4$, and in the latter case it is accepted by $\mathcal{A}_i^6$. If $k < l$ the word is accepted by $\mathcal{A}_i^5$.

The requirement that the computation be not recurring translates to the requirement that b_1 appears only finitely many times in σ. Let $\mathcal{A}_{recur}$ be the Büchi automaton which expresses this constraint.

Putting all the pieces together we claim that the language of the disjunction of $\mathcal{A}_0$, $\mathcal{A}_{init}$, $\mathcal{A}_{recur}$, and each of $\mathcal{A}_i$, is the complement of L_{undec}. ■

It is shown in [5] that the satisfiability problem for a real-time extension of the propositional linear temporal logic PTL becomes undecidable if a dense domain is chosen to model time. Thus our undecidability result is not unusual for formalisms reasoning about dense real-time. Obviously, the universality problem for TMAs is also undecidable. We have not been able to show that the universality problem is Π_1^1-complete, an interesting problem is to locate its exact position in the analytical hierarchy. In the following subsections we consider various implications of the above undecidability result.

5.3 Inclusion and equivalence

Recall that the language inclusion problem for Büchi automata can be solved in PSPACE. However, it follows from Theorem 5.2 that there is no decision procedure to check whether the language of one TBA is a subset of the other. This result is an obstacle in using timed automata as a specification language for automatic verification of finite-state real-time systems.

Corollary 5.3 Given two TBAs $\mathcal{A}_1$ and $\mathcal{A}_2$ over an alphabet Σ, the problem of checking $L(\mathcal{A}_1) \subseteq L(\mathcal{A}_2)$ is Π_1^1-hard.

PROOF. We reduce the universality problem for a given timed automaton $\mathcal{A}$ over Σ to the language inclusion problem. Let $\mathcal{A}_{univ}$ be an automaton which accepts every timed word over Σ. The automaton $\mathcal{A}$ is universal iff $L(\mathcal{A}_{univ}) \subseteq L(\mathcal{A})$. ■

Now we consider the problem of testing equivalence of two automata. A natural definition for equivalence of two automata uses equality of the languages accepted by the two. However alternative definitions exist. We will explore one such notion.

Definition 5.4 For timed Büchi automata $\mathcal{A}_1$ and $\mathcal{A}_2$ over an alphabet Σ, define $\mathcal{A}_1 \sim_1 \mathcal{A}_2$ iff $L(\mathcal{A}_1) = L(\mathcal{A}_2)$. Define $\mathcal{A}_1 \sim_2 \mathcal{A}_2$ iff for all timed automata $\mathcal{A}$ over Σ, $L(\mathcal{A}) \cap L(\mathcal{A}_1)$ is empty precisely when $L(\mathcal{A}) \cap L(\mathcal{A}_2)$ is empty. ■

For a class of automata closed under complement the above two definitions of equivalence coincide. However, these two equivalence relations differ for the class of timed regular languages because of the nonclosure under complement (to be proved shortly). In fact, the second notion is a weaker notion: $\mathcal{A}_1 \sim_1 \mathcal{A}_2$ implies $\mathcal{A}_1 \sim_2 \mathcal{A}_2$, but not vice versa. The motivation behind the second definition is that two automata (modeling two finite-state systems) should be considered different only when a third automaton (modeling the observer or the environment) composed with them gives different behaviors: in one case the composite language is empty, and in the other case there is a possible joint execution. The proof of Theorem 5.2 can be used to show undecidability of this equivalence also. Note that the problems of deciding the two types of equivalences lie at different levels of the hierarchy of undecidable problems.

Theorem 5.5 For timed Büchi automata $\mathcal{A}_1$ and $\mathcal{A}_2$ over an alphabet Σ,

1. The problem of deciding whether $\mathcal{A}_1 \sim_1 \mathcal{A}_2$ is Π_1^1-hard.
2. The problem of deciding whether $\mathcal{A}_1 \sim_2 \mathcal{A}_2$ is complete for the co-r.e. class.

PROOF. The language of a given TBA $\mathcal{A}$ is universal iff $\mathcal{A} \sim_1 \mathcal{A}_{univ}$. Hence the Π_1^1-hardness of the universality problem implies Π_1^1-hardness of the first type of equivalence.

Now we show that the problem of deciding nonequivalence, by the second definition, is recursively enumerable. If the two automata are inequivalent then there exists an automaton $\mathcal{A}$ over Σ such that only one of $L(\mathcal{A}) \cap L(\mathcal{A}_1)$ and $L(\mathcal{A}) \cap L(\mathcal{A}_2)$ is empty. Consider the following procedure P: P enumerates all the TBAs over Σ one by one. For each TBA $\mathcal{A}$, it checks for the emptiness of $L(\mathcal{A}) \cap L(\mathcal{A}_1)$ and the emptiness of $L(\mathcal{A}) \cap L(\mathcal{A}_2)$. If P ever finds different answers in the two cases, it halts saying that $\mathcal{A}_1$ and $\mathcal{A}_2$ are not equivalent.

Finally we prove that the problem of deciding the second type of equivalence is unsolvable. We use the encoding scheme used in the proof of Theorem 5.2. The only difference is that we use the *halting* problem of a *deterministic* 2-counter machine M instead of the recurring computations of a nondeterministic machine. Recall that the halting problem for deterministic 2-counter machines is undecidable. Assume that the n-th instruction is the halting instruction. We obtain $\mathcal{A}'_{undec}$ by replacing the disjunct $\mathcal{A}_{recur}$ by an automaton which accepts (σ, τ) iff b_n does not appear in σ. The complement of $L(\mathcal{A}'_{undec})$ consists of the timed words encoding the halting computation.

We claim that $\mathcal{A}_{univ} \sim_2 \mathcal{A}'_{undec}$ iff the machine M does not halt. If M does not halt then $\mathcal{A}'_{undec}$ accepts all timed words, and hence, its language is the same as that of $\mathcal{A}_{univ}$. If M halts, then we can construct a timed automaton $\mathcal{A}_{halt}$ which accepts a particular timed word encoding the halting computation of M. If M halts in k steps, then $\mathcal{A}_{halt}$ uses k clocks to ensure proper matching of the counter values in successive configurations. The details are very similar to the PSPACE-hardness proof of Theorem 4.17. $L(\mathcal{A}_{halt}) \cap L(\mathcal{A}_{univ})$ is nonempty whereas $L(\mathcal{A}_{halt}) \cap L(\mathcal{A}'_{undec})$ is empty, and thus $\mathcal{A}_{univ}$ and $\mathcal{A}'_{undec}$ are inequivalent in this case. This completes the proof. ■

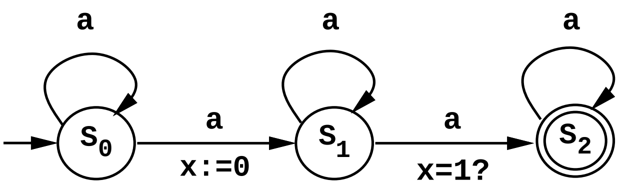


Figure 11: Noncomplementable automaton

5.4 Nonclosure under complement

The Π_1^1-hardness of the inclusion problem implies that the class of TBAs is not closed under complement.

Corollary 5.6 The class of timed regular languages is not closed under complementation.

PROOF. Given TBAs $\mathcal{A}_1$ and $\mathcal{A}_2$ over an alphabet Σ, $L(\mathcal{A}_1) \subseteq L(\mathcal{A}_2)$ iff the intersection of $L(\mathcal{A}_1)$ and the complement of $L(\mathcal{A}_2)$ is empty. Assume that TBAs are closed under complement. Consequently, $L(\mathcal{A}_1) \not\subseteq L(\mathcal{A}_2)$ iff there is a TBA $\mathcal{A}$ such that $L(\mathcal{A}_1) \cap L(\mathcal{A})$ is nonempty, but $L(\mathcal{A}_2) \cap L(\mathcal{A})$ is empty. That is, $L(\mathcal{A}_1) \not\subseteq L(\mathcal{A}_2)$ iff $\mathcal{A}_1$ and $\mathcal{A}_2$ are inequivalent according to $\sim_2$. From Theorem 5.5 it follows that the complement of the inclusion problem is recursively enumerable. This contradicts the Π_1^1-hardness of the inclusion problem. ∎

The following example provides some insight regarding the nonclosure under complementation.

Example 5.7 The language accepted by the automaton of Figure 11 over $\{a\}$ is

$$\{(a^\omega, \tau) \mid \exists i \geq 1.\, \exists j > i.\, (\tau_j = \tau_i + 1)\}.$$

The complement of this language cannot be characterized using a TBA. The complement needs to make sure that no pair of a's is separated by distance 1. Since there is no bound on the number of a's that can happen in a time period of length 1, keeping track of the times of all the a's within the past 1 time unit, would require an unbounded number of clocks. ∎

5.5 Choice of the clock constraints

In this section we consider some of the ways to modify our definition of clock constraints and indicate how these decisions affect the expressiveness and complexity of different problems. Recall that our definition of the clock constraints allows Boolean combinations of atomic formulas which compare clock values with (rational) constants. With this vocabulary, timed automata can express only constant bounds on the delays between transitions.

First suppose we extend the definition of clock constraints to allow subformulas involving two clocks such as $(x \leq y + c)$. In particular, in Definition 3.6 of the set $\Phi(X)$ of clock

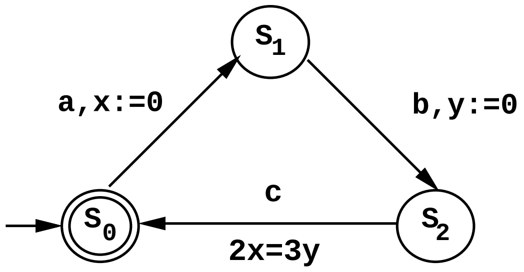


Figure 12: Automaton with clock constraints using +

constraints, we allow, as atomic constraints, the conditions $(x \leq y+c)$ and $(x+c \leq y)$, for $x, y \in X$ and $c \in \mathbb{Q}$. Thus the allowed clock constraints are quantifier-free formulas using the primitives of comparison ($\leq$) and addition by rational constants ($+c$). The untiming construction can handle this extension very easily. We need to refine the equivalence relation on clock interpretations. Now, in addition to the previous conditions, we require that two equivalent clock interpretations agree on all the subformulas appearing in the clock constraints. Also it is easy to prove that this extension of clock constraints does not add to the expressiveness of timed automata.

Next let us allow the primitive of addition in the clock constraints. Now we can write clock constraints such as $(x+y \leq x'+y')$ which allow the automaton to compare various delays. This greatly increases the expressiveness of the formalism. The language of the automaton in the following example is not timed regular.

Example 5.8 Consider the automaton of Figure 12 with the alphabet $\{a, b, c\}$. It expresses the property that the symbols a, b, and c occur cyclically, and the delay between b and c is always twice the delay between the last pair of a and b. The language is defined by

$$\{((abc)^\omega, \tau) \mid \forall j. \, [(\tau_{3j} - \tau_{3j-1}) = 2(\tau_{3j-1} - \tau_{3j-2})]\}.$$

■

Intuitively, the constraints involving addition are too powerful and cannot be implemented by finite-state systems. Even if we constrain all events to occur at integer time values (i.e., discrete-time model), to check that the delay between first two symbols is same as the delay between the next two symbols, an automaton would need an unbounded memory. Thus with finite resources, an automaton can compare delays with constants, but cannot remember delays. In fact, we can show that introducing addition in the syntax of clock constraints makes the emptiness problem for timed automata undecidable.

Theorem 5.9 Allowing the addition primitive in the syntax of clock constraints makes the emptiness problem for timed automata Π^1_1-hard.

Proof. As in the proof of Theorem 5.2 we reduce the problem of recurring computations of nondeterministic 2-counter machines to the emptiness problem for time automata using the primitive $+$. The alphabet is $\{a, b_1, \ldots b_n\}$. We say that a timed word (σ, τ) encodes a computation $\langle i_1, c_1, d_1\rangle, \langle i_2, c_2, d_2\rangle \cdots$ of the 2-counter machine iff $\sigma = b_{i_1} a b_{i_2} a b_{i_3} \cdots$ with $\tau_{2j} - \tau_{2j-1} = c_j$, and $\tau_{2j+1} - \tau_{2j} = d_j$ for all $j \geq 1$. Thus the delay between b and the following a encodes the value of the counter C, and the delay between a and the following b encodes the value of D. We construct a timed automaton which accepts precisely the timed words encoding the recurring computations of the machine. The primitive of $+$ is used to express a consecution requirement such as the value of the counter C remains unchanged. The details of the proof are quite straightforward. ■

6 Deterministic timed automata

The results of Section 5 show that the class of timed automata is not closed under complement, and one cannot automatically compare the languages of two automata. In this section we define deterministic timed automata, and show that the class of languages accepted by deterministic timed Muller automata (DTMA) is closed under all the Boolean operations.

6.1 Definition

Recall that in the untimed case a deterministic transition table has a single start state, and from each state, given the next input symbol, the next state is uniquely determined. We want a similar criterion for determinism for the timed automata: given an extended state and the next input symbol *along with its time of occurrence*, the extended state after the next transition should be uniquely determined. So we allow multiple transitions starting at the same state with the same label, but require their clock constraints to be *mutually exclusive* so that at any time only one of these transitions is enabled.

Definition 6.1 A timed transition table $\langle \Sigma, \mathrm{S}, \mathrm{S}_0, \mathrm{C}, \mathrm{E}\rangle$ is called *deterministic* iff

1. it has only one start state, $|\mathrm{S}_0| = 1$, and
2. for all $s \in \mathrm{S}$, for all $a \in \Sigma$, for every pair of edges of the form $\langle s, -, a, -, \delta_1\rangle$ and $\langle s, -, a, -, \delta_2\rangle$, the clock constraints δ_1 and δ_2 are mutually exclusive (i.e., $\delta_1 \wedge \delta_2$ is unsatisfiable).

A timed automaton is deterministic iff its timed transition table is deterministic. ■

Note that in absence of clocks the above definition matches with the definition of determinism for transition tables. Thus every deterministic transition table is also a deterministic timed transition table. Let us consider an example of a DTMA.

Example 6.2 The DTMA of Figure 13 accepts the language L_{crt} of Example 3.13:

$$L_{\mathrm{crt}} \;=\; \{((ab)^\omega, \tau) \mid \exists i.\, \forall j \geq i.\, (\tau_{2j} < \tau_{2j-1} + 2)\}.$$

The Muller acceptance family is given by $\{\{s_1, s_2\}\}$. The state s_1 has two mutually exclusive outgoing transitions on b. The acceptance condition requires that the transition with the clock constraint $(x \geq 2)$ is taken only finitely often. ■

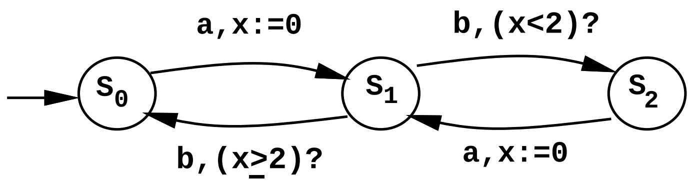


Figure 13: Deterministic timed Muller automaton

Deterministic timed automata can be easily complemented because of the following property:

Lemma 6.3 A deterministic timed transition table has at most one run over a given timed word.

PROOF. Consider a deterministic timed transition table $\mathcal{A}$, and a timed word (σ, τ). The run starts at time 0 with the extended state $\langle s_0, \nu_0 \rangle$ where s_0 is the unique start state. Suppose the extended state of $\mathcal{A}$ at time τ_{j-1} is $\langle s, \nu \rangle$, and the run has been constructed up to $(j-1)$ steps. By the deterministic property of $\mathcal{A}$, at time τ_j there is at most one transition $\langle s, s', \sigma_j, \delta, \lambda \rangle$ such that the clock interpretation at time τ_j, $\nu + \tau_j - \tau_{j-1}$, satisfies δ. If such a transition does not exist then $\mathcal{A}$ has no run over (σ, τ). Otherwise, this choice of transition uniquely extends the run to the j-th step, and determines the extended state at time τ_j. The lemma follows by induction. ■

6.2 Closure properties

Now we consider the closure properties for deterministic timed automata. Like in the untimed case, the class of languages accepted by deterministic timed Muller automata is closed under all Boolean operations.

Theorem 6.4 The class of timed languages accepted by deterministic timed Muller automata is closed under union, intersection, and complementation.

PROOF. We define a transformation on DTMAs to make the proofs easier; for every DTMA $\mathcal{A} = \langle \Sigma, \mathrm{S}, s_0, \mathrm{C}, \mathrm{E}, \mathcal{F} \rangle$ we construct another DTMA $\mathcal{A}^*$ by *completing* $\mathcal{A}$ as follows. First we add a dummy state q to the automaton. From each state s (including q), for each symbol a, we add an a-labeled edge from s to q. The clock constraint for this edge is the negation of the disjunction of the clock constraints of all the a-labeled edges starting at s. We leave the acceptance condition unchanged. This construction preserves determinism as well as the set of accepted timed words. The new automaton $\mathcal{A}^*$ has the property that for each state s and each input symbol a, the disjunction of the clock constraints of the a-labeled edges starting at s is a valid formula. Observe that $\mathcal{A}^*$ has precisely one run over any timed word. We call such an automaton *complete*. In the remainder of the proof we assume each DTMA to be complete.

Let $\mathcal{A}_i = \langle \Sigma, \mathrm{S}_i, s_{0_i}, \mathrm{C}_i, \mathrm{E}_i, \mathcal{F}_i \rangle$, for $i = 1, 2$, be two complete DTMAs with disjoint sets of clocks. First we construct a timed transition table $\mathcal{A}$ using a product construction. The set of states of $\mathcal{A}$ is $\mathrm{S}_1 \times \mathrm{S}_2$. Its start state is $\langle s_{0_1}, s_{0_2} \rangle$. The set of clocks is $\mathrm{C}_1 \cup \mathrm{C}_2$.

The transitions of $\mathcal{A}$ are defined by coupling the transitions of the two automata having the same label. Corresponding to an $\mathcal{A}_1$-transition $\langle s_1, t_1, a, \lambda_1, \delta_1\rangle$ and an $\mathcal{A}_2$-transition $\langle s_2, t_2, a, \lambda_2, \delta_2\rangle$, $\mathcal{A}$ has a transition $\langle\langle s_1, s_2\rangle, \langle t_1, t_2\rangle, a, \lambda_1 \cup \lambda_2, \delta_1 \wedge \delta_2\rangle$. It is easy to check that $\mathcal{A}$ is also deterministic. $\mathcal{A}$ has a unique run over each (σ, τ), and this run can be obtained by putting together the unique runs of $\mathcal{A}_i$ over (σ, τ).

Let $\mathcal{F}^1$ consist of the sets $\mathrm{F} \subseteq \mathrm{S}_1 \times \mathrm{S}_2$ such that the projection of F onto the first component is an accepting set of $\mathcal{A}_1$; that is,

$$\mathcal{F}^1 \;=\; \{\mathrm{F} \subseteq \mathrm{S}_1 \times \mathrm{S}_2 \mid \{s \in \mathrm{S}_1 \mid \exists s' \in \mathrm{S}_2.\, \langle s, s'\rangle \in \mathrm{F}\} \in \mathcal{F}_1\}.$$

Hence a run r of $\mathcal{A}$ is an accepting run for $\mathcal{A}_1$ iff $\mathit{inf}(r) \in \mathcal{F}^1$. Similarly define $\mathcal{F}^2$ to consist of the sets F such that $\{s' \mid \exists s \in \mathrm{S}_1.\, \langle s, s'\rangle \in \mathrm{F}\}$ is in $\mathcal{F}_2$. Now coupling $\mathcal{A}$ with the Muller acceptance family $\mathcal{F}^1 \cup \mathcal{F}^2$ gives a DTMA accepting $L(\mathcal{A}_1) \cup L(\mathcal{A}_2)$, whereas using the acceptance family $\mathcal{F}^1 \cap \mathcal{F}^2$ gives a DTMA accepting $L(\mathcal{A}_1) \cap L(\mathcal{A}_2)$.

Finally consider complementation. Let $\mathcal{A}$ be a complete DTMA $\langle \Sigma, \mathrm{S}, s_0, \mathrm{C}, \mathrm{E}, \mathcal{F}\rangle$. $\mathcal{A}$ has exactly one run over a given timed word. Hence, (σ, τ) is in the complement of $L(\mathcal{A})$ iff the run of $\mathcal{A}$ over it does not meet the acceptance criterion of $\mathcal{A}$. The complement language is, therefore, accepted by a DTMA which has the same underlying timed transition table as $\mathcal{A}$, but its acceptance condition is given by $2^{\mathrm{S}} - \mathcal{F}$. ∎

Now let us consider the closure properties of DTBAs. Recall that deterministic Büchi automata (DBA) are not closed under complement. The property that "there are infinitely many a's" is specifiable by a DBA, however, the complement property, "there are only finitely many a's" cannot be expressed by a DBA. Consequently we do not expect the class of DTBAs to be closed under complementation. However, since every DTBA can be viewed as a DTMA, the complement of a DTBA-language is accepted by a DTMA. The next theorem states the closure properties.

Theorem 6.5 The class of timed languages accepted by DTBAs is closed under union and intersection, but not closed under complement. The complement of a DTBA language is accepted by some DTMA.

PROOF. For the case of union, we construct the product transition table as in case of DTMAs (see proof of Theorem 6.4). The accepting set is $\{\langle s, s'\rangle \mid s \in \mathrm{F}_1 \vee s' \in \mathrm{F}_2\}$.

A careful inspection of the product construction for TBAs (see proof of Theorem 3.15) shows that it preserves determinism. The closure under intersection for DTBAs follows.

The nonclosure of deterministic Büchi automata under complement leads to the nonclosure for DTBAs under complement. The language $\{(\sigma, \tau) \mid \sigma \in (b^*a)^\omega\}$ is specifiable by a DBA. Its complement language $\{(\sigma, \tau) \mid \sigma \in (a+b)^*b^\omega\}$ is not specifiable by a DTBA. This claim follows from Lemma 6.7 (to be proved shortly), and the fact that the language $(a+b)^*b^\omega$ is not specifiable by a DBA.

Let $\mathcal{A} = \langle \Sigma, \mathrm{S}, s_0, \mathrm{C}, \mathrm{E}, \mathrm{F}\rangle$ be a complete deterministic automaton. (σ, τ) is in the complement of $L(\mathcal{A})$ iff the (unique) run of $\mathcal{A}$ over it does not meet the acceptance criterion of $\mathcal{A}$. The complement language is, therefore, accepted by a DTMA with the same underlying timed transition table as $\mathcal{A}$, and the acceptance family $2^{\mathrm{S}-\mathrm{F}}$. ∎

6.3 Decision problems

In this section we examine the complexity of the emptiness problem and the language inclusion problem for deterministic timed automata.

The emptiness of a timed automaton does not depend on the symbols labeling its edges. Consequently, checking emptiness of deterministic automata is no simpler; it is PSPACE-complete.

Since deterministic automata can be complemented, checking for language inclusion is decidable. In fact, while checking $L(\mathcal{A}_1) \subseteq L(\mathcal{A}_2)$, only $\mathcal{A}_2$ need be deterministic, $\mathcal{A}_1$ can be nondeterministic. The problem can be solved in PSPACE:

Theorem 6.6 For a timed automaton $\mathcal{A}_1$ and a deterministic timed automaton $\mathcal{A}_2$, the problem of deciding whether $L(\mathcal{A}_1)$ is contained in $L(\mathcal{A}_2)$ is PSPACE-complete.

PROOF. PSPACE-hardness follows, even when $\mathcal{A}_1$ is deterministic, from the fact that checking for the emptiness of the language of a deterministic timed automaton is PSPACE-hard. Let $\mathcal{A}_{empty}$ be a deterministic automaton which accepts the empty language. Now for a deterministic timed automaton $\mathcal{A}$, $L(\mathcal{A})$ is empty iff $L(\mathcal{A}) \subseteq L(\mathcal{A}_{empty})$.

Observe that $L(\mathcal{A}_1) \subseteq L(\mathcal{A}_2)$ iff the intersection of $L(\mathcal{A}_1)$ with the complement of $L(\mathcal{A}_2)$ is empty. Recall that complementing the language of a deterministic automaton corresponds to complementing the acceptance condition. First we construct a timed transition table $\mathcal{A}$ from the timed transition tables of $\mathcal{A}_1$ and $\mathcal{A}_2$ using the product construction (see proof of Theorem 6.4). The size of $\mathcal{A}$ is proportional to the product of the sizes of $\mathcal{A}_i$. Then we construct the region automaton $R(\mathcal{A})$. $L(\mathcal{A}_1) \not\subseteq L(\mathcal{A}_2)$ iff $R(\mathcal{A})$ has a cycle which is accessible from its start state, meets the progressiveness requirement, the acceptance criterion for $\mathcal{A}_1$, and the complement of the acceptance criterion for $\mathcal{A}_2$. The existence of such a cycle can be checked in space polynomial in the size of $\mathcal{A}$, as in the proof of PSPACE-solvability of emptiness (Theorem 4.17). ■

6.4 Expressiveness

In this section we compare the expressive power of the various types of timed automata.

Every DTBA can be expressed as a DTMA simply by rewriting its acceptance condition. However the converse does not hold. First observe that every ω-regular language is expressible as a DMA, and hence as a DTMA. On the other hand, since deterministic Büchi automata are strictly less expressive than deterministic Muller automata, certain ω-regular languages are not specifiable by DBAs. The next lemma shows that such languages cannot be expressed using DTBAs either. It follows that DTBAs are strictly less expressive than DTMAs. In fact, DTMAs are closed under complement, whereas DTBAs are not.

Lemma 6.7 For an ω-language L, the timed language $\{(\sigma, \tau) \mid \sigma \in L\}$ is accepted by some DTBA iff L is accepted by some DBA.

PROOF. Clearly if L is accepted by a DBA, then $\{(\sigma, \tau) \mid \sigma \in L\}$ is accepted by the same automaton considered as a timed automaton.

Now suppose that the language $\{(\sigma, \tau) \mid \sigma \in L\}$ is accepted by some DTBA $\mathcal{A}$. We construct another DTBA $\mathcal{A}'$ such that $L(\mathcal{A}') = \{(\sigma, \tau) \mid (\sigma \in L) \wedge \forall i. (\tau_i = i)\}$. $\mathcal{A}'$

Class of timed languages	Operations closed under
TMA = TBA	union, intersection
$\cup$	
DTMA	union, intersection, complement
$\cup$	
DTBA	union, intersection

Figure 14: Classes of timed automata

requires time to increase by 1 at each transition. The automaton $\mathcal{A}'$ can be obtained from $\mathcal{A}$ by introducing an extra clock x. We add the conjunct $x = 1$ to the clock constraint of every edge in $\mathcal{A}$ and require it to be reset on every edge. $\mathcal{A}'$ is also deterministic.

The next step is the untiming construction for $\mathcal{A}'$. Observe that $Untime(L(\mathcal{A}')) = L$. While constructing $R(\mathcal{A}')$ we need to consider only those clock regions which have all clocks with zero fractional parts. Since the time increase at every step is predetermined, and $\mathcal{A}'$ is deterministic, it follows that $R(\mathcal{A}')$ is a deterministic transition table. We need not check the progressiveness condition also. It follows that the automaton constructed by the untiming procedure is a DBA accepting L. ■

From the above discussion one may conjecture that a DTMA language L is a DTBA language if $Untime(L)$ is a DBA language. To answer this let us consider the convergent response property L_{crt} specifiable using a DTMA (see Example 6.2). This language involves a combination of liveness and timing. We conjecture that no DTBA can specify this property (even though $Untime(L_{\text{crt}})$ can be trivially specified by a DBA).

Along the lines of the above proof we can also show that for an ω-language L, the timed language $\{(\sigma, \tau) \mid \sigma \in L\}$ is accepted by some DTMA (or TMA, or TBA) iff L is accepted by some DMA (or MA, or BA, respectively).

Since DTMAs are closed under complement, whereas TMAs are not, it follows that the class of languages accepted by DTMAs is strictly smaller than that accepted by TMAs. In particular, the language of Example 5.7, ("some pair of a's is distance 1 apart") is not representable as a DTMA; it relies on nondeterminism in a crucial way.

We summarize the discussion on various types of automata in the table of Figure 14 which shows the inclusions among various classes and the closure properties of various classes. Compare this with the corresponding results for the various classes of ω-automata shown in Figure 15.

7 Verification

In this section we discuss how to use the theory of timed automata to prove correctness of finite-state real-time systems. We have chosen a simple formulation of the verification problem, but it suffices to illustrate the application of timed automata to verification problems. We start by introducing time in linear trace semantics for concurrent processes.

Class of ω-languages	Operations closed under
MA = BA = DMA	union, intersection, complement
$\cup$	
DBA	union, intersection

Figure 15: Classes of ω-automata

7.1 Trace semantics

In trace semantics, we associate a set of observable *events* with each process, and model the process by the set of all its *traces*. A trace is a (linear) sequence of events that may be observed when the process runs. For example, an event may denote an assignment of a value to a variable, or pressing a button on the control panel, or arrival of a message. All events are assumed to occur instantaneously. Actions with duration are modeled using events marking the beginning and the end of the action. Hoare originally proposed such a model for CSP [22].

In our model, a trace will be a sequence of sets of events. Thus if two events a and b happen simultaneously, the corresponding trace will have a set $\{a, b\}$ in our model. In the usual interleaving models, this set will be replaced by all possible sequences, namely, a followed by b and b followed by a. Also we consider only infinite sequences, which model nonterminating interaction of reactive systems with their environments.

Formally, given a set A of events, a *trace* $\sigma = \sigma_1\sigma_2\ldots$ is an infinite word over $\mathcal{P}^+(A)$ — the set of nonempty subsets of A. An *untimed process* is a pair (A, X) comprising of the set A of its observable events and the set X of its possible traces.

Example 7.1 Consider a channel P connecting two components. Let a represent the arrival of a message at one end of P, and let b stand for the delivery of the message at the other end of the channel. The channel cannot receive a new message until the previous one has reached the other end. Consequently the two events a and b alternate. Assuming that the messages keep arriving, the only possible trace is

$$\sigma_P \;:\; \{a\} \rightarrow \{b\} \rightarrow \{a\} \rightarrow \{b\} \rightarrow \cdots.$$

Often we will denote the singleton set $\{a\}$ by the symbol a. The process P is represented by $(\{a, b\}, (ab)^\omega)$. ■

Various operations can be defined on processes; these are useful for describing complex systems using the simpler ones. We will consider only the most important of these operations, namely, parallel composition. The parallel composition of a set of processes describes the joint behavior of all the processes running concurrently.

The parallel composition operator can be conveniently defined using the projection operation. The *projection* of $\sigma \in \mathcal{P}^+(A)^\omega$ onto $B \subseteq A$ (written $\sigma\lceil B$) is formed by intersecting each event set in σ with B and deleting all the empty sets from the sequence. For instance, in Example 7.1 $\sigma_P\lceil\{a\}$ is the trace a^ω. Notice that the projection operation may result in a finite sequence; but for our purpose it suffices to consider the projection of a trace σ onto B only when $\sigma_i \cap B$ is nonempty for infinitely many i.

For a set of processes $\{P_i = (A_i, X_i) \mid i = 1, 2, \ldots n\}$, their *parallel composition* $\|_i \; P_i$ is a process with the event set $\cup_i A_i$ and the trace set

$$\{\sigma \in \mathcal{P}^+(\cup_i A_i)^\omega \mid \; \wedge_i \;\; \sigma\lceil A_i \in X_i\}.$$

Thus σ is a trace of $\|_i \; P_i$ iff $\sigma\lceil A_i$ is a trace of P_i for each $i = 1, \ldots n$. When there are no common events the above definition corresponds to the unconstrained interleavings of all the traces. On the other hand, if all event sets are identical then the trace set of the composition process is simply the set-theoretic intersection of all the component trace sets.

Example 7.2 Consider another channel Q connected to the channel P of Example 7.1. The event of message arrival for Q is same as the event b. Let c denote the delivery of the message at the other end of Q. The process Q is given by $(\{b, c\}, (bc)^\omega)$.

When P and Q are composed we require them to synchronize on the common event b, and between every pair of b's we allow the possibility of the event a happening before the event c, the event c happening before a, and both occurring simultaneously. Thus $[\,P \parallel Q\,]$ has the event set $\{a, b, c\}$, and has an infinite number of traces. ∎

In this framework, the verification question is presented as an inclusion problem. Both the implementation and the specification are given as untimed processes. The implementation process is typically a composition of several smaller component processes. We say that an implementation (A, X_I) is *correct* with respect to a specification (A, X_S) iff $X_I \subseteq X_S$.

Example 7.3 Consider the channels of Example 7.2. The implementation process is $[\,P \parallel Q\,]$. The specification is given as the process $S = (\{a, b, c\}, (abc)^\omega)$. Thus the specification requires the message to reach the other end of Q before the next message arrives at P. In this case, $[P \parallel Q]$ does not meet the specification S, for it has too many other traces, specifically, the trace $ab(acb)^\omega$. ∎

Notice that according to the above definition of the verification problem, an implementation with $X_I = \emptyset$ is correct with respect to every specification. To overcome this problem, one needs to distinguish between output events (the events controlled by the system), and the input events (the events controlled by its environment), and require that the implementation should not prevent its environment from executing the input events [14]. We believe that distinguishing between input and output events and introducing timing are two orthogonal issues, and our goal in this paper is to indicate how to address the latter problem.

7.2 Adding timing to traces

An untimed process models the sequencing of events but not the actual times at which the events occur. Thus the description of the channel in Example 7.1 gives only the sequencing of the events a and b, and not the delays between them. Timing can be added to a trace by coupling it with a sequence of time values. We choose the set of reals to model time.

Recall that a *time sequence* $\tau = \tau_1\tau_2\ldots$ is an infinite sequence of time values $\tau_i \in \mathsf{R}$ satisfying the strict monotonicity and progress constraints. A *timed trace* over a set of events A is a pair (σ, τ) where σ is a trace over A, and τ is a time sequence. Note that, since different events happening simultaneously appear in a single element in a trace, there is no reason to allow the possibility of the adjacent elements in a trace having the same associated time value.

In a timed trace (σ, τ), each τ_i gives the time at which the events in σ_i occur. In particular, τ_1 gives the time of the first observable event; we always assume $\tau_1 > 0$, and define $\tau_0 = 0$. Observe that the progress condition implies that only a finite number of events can happen in a bounded interval of time. In particular, it rules out convergent time sequences such as $1/2, 3/4, 7/8, \ldots$ representing the possibility that the system participates in infinitely many events before time 1.

A *timed process* is a pair (A, L) where A is a finite set of events, and L is a set of timed traces over A.

Example 7.4 Consider the channel P of Example 7.1 again. Assume that the first message arrives at time 1, and the subsequent messages arrive at fixed intervals of length 3 time units. Furthermore, it takes 1 time unit for every message to traverse the channel. The process has a single timed trace

$$\rho_P \;=\; (a,1) \;\rightarrow\; (b,2) \;\rightarrow\; (a,4) \;\rightarrow\; (b,5) \;\rightarrow\; \cdots$$

and it is represented as a timed process $P^T = (\{a,b\}, \{\rho_P\})$. ■

The operations on untimed processes are extended in the obvious way to timed processes. To get the projection of (σ, τ) onto $B \subseteq A$, we first intersect each event set in σ with B and then delete all the empty sets along with the associated time values. The definition of parallel composition remains unchanged, except that it uses the projection for timed traces. Thus in parallel composition of two processes, we require that both the processes should participate in the common events at the same time. This rules out the possibility of interleaving: parallel composition of two timed traces is either a single timed trace or is empty.

Example 7.5 As in Example 7.2 consider another channel Q connected to P. For Q, as before, the only possible trace is $\sigma_Q = (bc)^\omega$. In addition, the timing specification of Q says that the time taken by a message for traversing the channel, that is, the delay between b and the following c, is some real value between 1 and 2. The timed process Q^T has infinitely many timed traces, and it is given by

$$[\,\{b,c\},\ \{(\sigma_Q,\tau) \mid \forall i.\,(\tau_{2i-1}+1 < \tau_{2i} < \tau_{2i-1}+2)\}\,].$$

The description of $[\,P^T \parallel Q^T\,]$ is obtained by composing ρ_P with each timed trace of Q^T. The composition process has uncountably many timed traces. An example trace is

$$(a,1) \;\rightarrow\; (b,2) \;\rightarrow\; (c,3.8) \;\rightarrow\; (a,4) \;\rightarrow\; (b,5) \;\rightarrow\; (c,6.02) \;\rightarrow\; \cdots$$

■

The time values associated with the events can be discarded by the *Untime* operation. For a timed process $P = (A, L)$, $Untime[(A, L)]$ is the untimed process with the event set A and the trace set consisting of traces σ such that $(\sigma, \tau) \in L$ for some time sequence τ.

Note that

$$Untime(P_1 \parallel P_2) \subseteq Untime(P_1) \parallel Untime(P_2).$$

However, as Example 7.6 shows, the two sides are not necessarily equal. In other words, the timing information retained in the timed traces constrains the set of possible traces when two processes are composed.

Example 7.6 Consider the channels of Example 7.5. Observe that $Untime(P^T) = P$ and $Untime(Q^T) = Q$. $[\, P^T \parallel Q^T \,]$ has a unique untimed trace $(abc)^\omega$. On the other hand, $[\, P \parallel Q \,]$ has infinitely many traces; between every pair of b events all possible orderings of an event a and an event c are admissible. ■

The verification problem is again posed as an inclusion problem. Now the implementation is given as a composition of several timed processes, and the specification is also given as a timed process.

Example 7.7 Consider the verification problem of Example 7.3 again. If we model the implementation as the timed process $[\, P^T \parallel Q^T \,]$ then it meets the specification S. The specification S is now a timed process $(\{a, b, c\}, \{((abc)^\omega, \tau)\})$. Observe that, though the specification S constrains only the sequencing of events, the correctness of $[\, P^T \parallel Q^T \,]$ with respect to S crucially depends on the timing constraints of the two channels. ■

7.3 ω-automata and verification

We start with an overview of the application of Büchi automata to verify untimed processes [45, 44]. Observe that for an untimed process (A, X), X is an ω-language over the alphabet $\mathcal{P}^+(A)$. If it is a regular language it can be represented by a Büchi automaton.

We model a finite-state (untimed) process P with event set A using a Büchi automaton $\mathcal{A}_P$ over the alphabet $\mathcal{P}^+(A)$. The states of the automaton correspond to the internal states of the process. The automaton $\mathcal{A}_P$ has a transition $\langle s, s', a \rangle$, with $a \subseteq A$, if the process can change its state from s to s' participating in the events from a. The acceptance conditions of the automaton correspond to the fairness constraints on the process. The automaton $\mathcal{A}_P$ accepts (or generates) precisely the traces of P; that is, the process P is given by $(A, L(\mathcal{A}_P))$. Such a process P is called an *ω-regular process*.

The user describes a system consisting of various components by specifying each individual component as a Büchi automaton. In particular, consider a system I comprising of n components, where each component is modeled as an ω-regular process $P_i = (A_i, L(\mathcal{A}_i))$. The implementation process is $[\|_i \; P_i]$. We can automatically construct the automaton for I using the construction for language intersection for Büchi automata. Since the event sets of various components may be different, before we apply the product construction, we need to make the alphabets of various automata identical. Let $A = \cup_i A_i$. From each $\mathcal{A}_i$, we construct an automaton $\mathcal{A}'_i$ over the alphabet $\mathcal{P}^+(A)$ such that $L(\mathcal{A}'_i) = \{\sigma \in \mathcal{P}^+(A)^\omega \mid \sigma\lceil A_i \in L(\mathcal{A}_i)\}$. Now the desired automaton $\mathcal{A}_I$ is the product of the automata $\mathcal{A}'_i$.

The specification is given as an ω-regular language S over $\mathcal{P}^+(A)$. The implementation meets the specification iff $L(\mathcal{A}_I) \subseteq S$. The property S can presented as a Büchi automaton $\mathcal{A}_S$. In this case, the verification problem reduces to checking emptiness of $L(\mathcal{A}_I) \cap L(\mathcal{A}_S)^c$.

The verification problem is PSPACE-complete. The size of $\mathcal{A}_I$ is exponential in the description of its individual components. If $\mathcal{A}_S$ is nondeterministic, taking the complement involves an exponential blow-up, and thus the complexity of verification problem is exponential in the size of the specification also. However, if $\mathcal{A}_S$ is deterministic, then the complexity is only polynomial in the size of the specification.

Even if the size of the specification and the sizes of the automata for the individual components are small, the number of components in most systems of interest is large, and in the above method the complexity is exponential in this number. Thus the product automaton $\mathcal{A}_I$ has a prohibitively large number of states, and this limits the applicability of this approach. Alternative methods which avoid enumeration of all the states in $\mathcal{A}_I$ have been proposed, and shown to be applicable to verification of some moderately sized systems [8, 18].

7.4 Verification using timed automata

For a timed process (A, L), L is a timed language over $\mathcal{P}^+(A)$. A *timed regular process* is one for which the set L is a timed regular language, and can be represented by a timed automaton.

Finite-state systems are modeled by TBAs. The underlying transition table gives the state-transition graph of the system. We have already seen how the clocks can be used to represent the timing delays of various physical components. As before, the acceptance conditions correspond to the fairness conditions. Notice that the progress requirement imposes certain fairness requirements implicitly. Thus, with a finite-state process P, we associate a TBA $\mathcal{A}_P$ such that $L(\mathcal{A}_P)$ consists of precisely the timed traces of P.

Typically, an implementation is described as a composition of several components. Each component should be modeled as a timed regular process $P_i = (A_i, L(\mathcal{A}_i))$. It is possible to construct a TBA $\mathcal{A}_I$ which represents the composite process $[\|_i\ P_i]$. To do this, first we need to make the alphabets of various automata identical, and then take the intersection. However, in the verification procedure we are about to outline, we will not explicitly construct the implementation automaton $\mathcal{A}_I$.

The specification of the system is given as another timed regular language S over the alphabet $\mathcal{P}^+(A)$, where $A = \cup_i A_i$. The system is *correct* iff $L(\mathcal{A}_I) \subset S$. If S is given as a TBA, then in general, it is undecidable to test for correctness. However, if S is given as a DTMA $\mathcal{A}_S$, then we can solve this as outlined in Section 6.3.

Putting together all the pieces, we conclude:

Theorem 7.8 Given timed regular processes $P_i = (A_i, L(\mathcal{A}_i))$, $i = 1, \ldots n$, modeled by timed automata $\mathcal{A}_i$, and a specification as a deterministic timed automaton $\mathcal{A}_S$, the inclusion of the trace set of $[\|_i\ P_i]$ in $L(\mathcal{A}_S)$ can be checked in PSPACE.

Proof. Consider TBAs $\mathcal{A}_i = \langle \mathcal{P}^+(A_i), \mathrm{S}_i, \mathrm{S}_{i_0}, \mathrm{C}_i, \mathrm{E}_i, \mathrm{F}_i \rangle$, $i = 1, \ldots n$, and the DTMA $\mathcal{A}_S = \langle \mathcal{P}^+(A), \mathrm{S}_0, \mathrm{S}_{0_0}, \mathrm{C}_0, \mathrm{E}_0, \mathcal{F} \rangle$. Assume without loss of generality that the clock sets C_i, $i = 0, \ldots n$, are disjoint.

The verification algorithm constructs the transition table of the region automaton corresponding to the product $\mathcal{A}$ of the timed transition tables of $\mathcal{A}_i$ with $\mathcal{A}_S$. The set of clocks of $\mathcal{A}$ is $\mathrm{C} = \cup_i \mathrm{C}_i$. The states of $\mathcal{A}$ are of the form $\langle s_0, \ldots s_n \rangle$ with each $s_i \in \mathrm{S}_i$. The initial states of $\mathcal{A}$ are of the form $\langle s_0, \ldots s_n \rangle$ with each $s_i \in \mathrm{S}_{i_0}$. A transition of $\mathcal{A}$ is obtained by coupling the transitions of the individual automata labeled with consistent event sets. A state $s = \langle s_0, \ldots s_n \rangle$ has a transition to state $s' = \langle s'_0, \ldots s'_n \rangle$ labeled with event set $a \in \mathcal{P}^+(A)$, clock constraint $\wedge_i \delta_i$, and the set $\cup_i \lambda_i$ of clocks, iff for each $0 \leq i \leq n$, either there is a transition $\langle s_i, s'_i, a \cap A_i, \lambda_i, \delta_i \rangle \in \mathrm{E}_i$, or the automaton $\mathcal{A}_i$ does not participate in this transition: $s'_i = s_i$, $a \cap A_i = \emptyset$, $\lambda_i = \emptyset$, and $\delta_i = \mathbf{true}$.

The region automaton $R(\mathcal{A})$ is defined from the product table $\mathcal{A}$ as described in Section 4. To test the desired inclusion, the algorithm searches for a cycle in the region automaton such that (1) it is accessible from the initial state of $R(\mathcal{A})$, (2) it satisfies the progessiveness condition: for each clock $x \in \mathrm{C}$, the cycle contains at least one region satisfying $[(x = 0) \vee (x > c_x)]$, (3) since our definition of the composition requires that we consider only those infinite runs in which each automaton participates infinitely many times, we require that, for each $1 \leq i \leq n$, the cycle contains a transition in which the automaton $\mathcal{A}_i$ participates, (4) the fairness requirements of all implementation automata $\mathcal{A}_i$ are met: for each $1 \leq i \leq n$, the cycle contains some state whose i-th component belongs to the accepting set F_i, (5) the fairness condition of the specification is *not* met: the projection of the states in the cycle onto the component of $\mathcal{A}_S$ does not belong to the acceptance family $\mathcal{F}$. The desired inclusion does not hold iff a cycle with all the above conditions can be found.

Each state of the region automaton can be represented in space polynomial in the description of the input automata. It follows that the inclusion test can be performed in PSPACE. ■

The number of vertices in the region automaton is $O[|\mathcal{A}_S| \cdot \Pi_i |\mathcal{A}_i| \cdot 2^{|\delta(\mathcal{A}_S)| + \Sigma_i |\delta(\mathcal{A}_i)|}]$, and the time complexity of the above algorithm is linear in this number. There are mainly three sources of exponential blow-up:

1. The complexity is proportional to the number of states in the global timed automaton describing the implementation $[\|_i \ P_i]$. This is exponential in the number of components.

2. The complexity is proportional to the product of the constants c_x, the largest constant x is compared with, over all the clocks x involved.

3. The complexity is proportional to the number of permutations over the set of all clocks.

The first factor is present in the simplest of verification problems, even in the untimed case. Since the number of components is typically large, this exponential factor has been a major obstacle in implementing model-checking algorithms.

The second factor is typical of any formalism to reason about quantitative time. The blow-up by actual constants is observed even for simpler, discrete models. Note that if the bounds on the delays of different components are relatively prime then this factor leads to a major blow-up in the complexity.

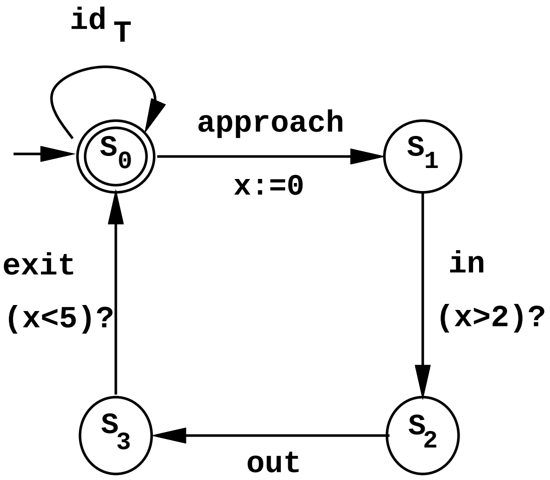


Figure 16: TRAIN

Lastly, in the untiming construction, we need to account for all the possible orderings of the fractional parts of different clocks, and this is the source of the third factor. We remark that switching to a simpler, say discrete-time, model will avoid this blow-up in complexity. However since the total number of clocks is linear in the number of independent components, this blow-up is the same as that contributed by the first factor, namely, exponential in the number of components.

7.5 Verification example

We consider an example of an automatic controller that opens and closes a gate at a railroad crossing [29]. The system is composed of three components: TRAIN, GATE and CONTROLLER.

The automaton modeling the train is shown in Figure 16. The event set is {*approach, exit, in, out,* id_T}. The train starts in state s_0. The event id_T represents its idling event; the train is not required to enter the gate. The train communicates with the controller with two events *approach* and *exit*. The events *in* and *out* mark the events of entry and exit of the train from the railroad crossing. The train is required to send the signal *approach* at least 2 minutes before it enters the crossing. Thus the minimum delay between *approach* and *in* is 2 minutes. Furthermore, we know that the maximum delay between the signals *approach* and *exit* is 5 minutes. This is a liveness requirement on the train. Both the timing requirements are expressed using a single clock x.

The automaton modeling the gate component is shown in Figure 17. The event set is {*raise, lower, up, down,* id_G}. The gate is open in state s_0 and closed in state s_2. It communicates with the controller through the signals *lower* and *raise*. The events *up* and *down* denote the opening and the closing of the gate. The gate responds to the signal *lower* by closing within 1 minute, and responds to the signal *raise* within 1 to 2 minutes. The gate can take its idling transition id_G in states s_0 or s_2 forever.

Finally, Figure 18 shows the automaton modeling the controller. The event set is

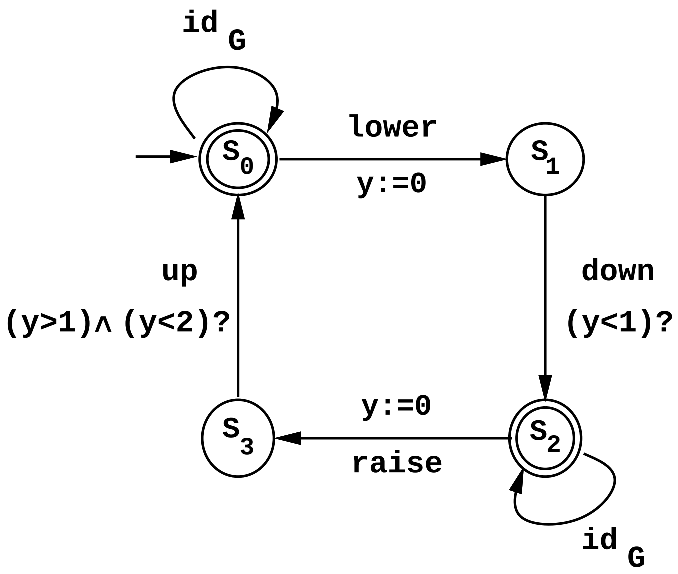


Figure 17: GATE

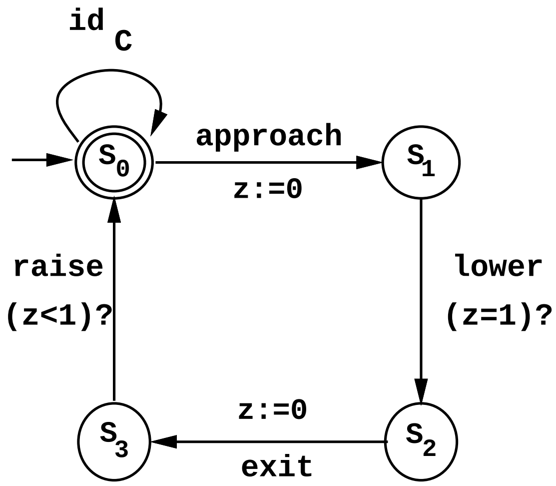


Figure 18: CONTROLLER

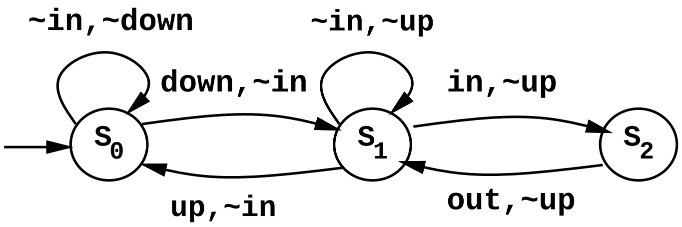


Figure 19: Safety property

$\{approach, exit, raise, lower, id_C\}$. The controller idle state is s_0. Whenever it receives the signal *approach* from the train, it responds by sending the signal *lower* to the gate. The response time is 1 minute. Whenever it receives the signal *exit*, it responds with a signal *raise* to the gate within 1 minute.

The entire system is then

$$[\text{TRAIN} \parallel \text{GATE} \parallel \text{CONTROLLER}].$$

The event set is the union of the event sets of all the three components. In this example, all the automata are particularly simple; they are deterministic, and do not have any fairness constraints (every run is an accepting run). The timed automaton $\mathcal{A}_I$ specifying the entire system is obtained by composing the above three automata.

The correctness requirements for the system are the following:

1. *Safety:* Whenever the train is inside the gate, the gate should be closed.

2. *Real-time Liveness:* The gate is never closed at a stretch for more than 10 minutes.

The specification refers to only the events *in, out, up, down.* The safety property is specified by the automaton of Figure 19. An edge label *in* stands for any event set containing *in*, and an edge label "*in*, $\neg$*out*" means any event set not containing *out*, but containing *in*. The automaton disallows *in* before *down*, and *up* before *out*. All the states are accepting states.

The real-time liveness property is specified by the timed automaton of Figure 20. The automaton requires that every *down* be followed by *up* within 10 minutes.

Note that the automaton is deterministic, and hence can be complemented. Furthermore, observe that the acceptance condition is not necessary; we can include state s_1 also in the acceptance set. This is because the progress of time ensures that the self-loop on state s_1 with the clock constraint $(x < 10)$ cannot be taken indefinitely, and the automaton will eventually visit state s_0.

The correctness of $\mathcal{A}_I$ against the two specifications can be checked separately as outlined in Section 7. Observe that though the safety property is purely a qualitative property, it does not hold if we discard the timing requirements.

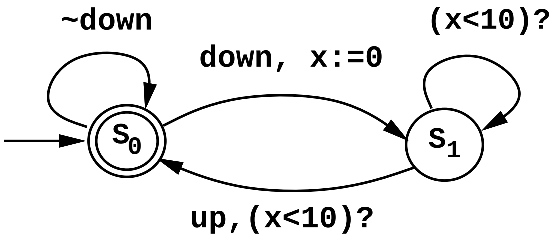


Figure 20: Real-time liveness property

8 New results on timed automata

Timed automata provide a natural way of expressing timing delays of a real-time system. In this presentation, we have studied them from the perspective of formal language theory. Now we briefly review other results about timed automata. The precise formulation of timed automata is different in different papers, but the underlying idea remains the same.

Timed automata are useful for developing a decision procedure for the logic MITL, a real-time extension of the linear temporal logic PTL [4]. The decision procedure constructs a timed automaton $\mathcal{A}_\phi$ from a given MITL-formula ϕ, such that $\mathcal{A}_\phi$ accepts precisely the satisfying models of ϕ; thereby reducing the satisfiability question for ϕ to the emptiness question for $\mathcal{A}_\phi$. This construction can also be used to check the correctness of a system modeled as a product of timed automata against MITL-specification.

The untiming construction for timed automata forms the basis for verification algorithms in the *branching-time* model also. In [1], we develop a model-checking algorithm for specifications written in TCTL — a real-time extension of the branching-time temporal logic CTL of [16]. In [43], a notion of *timed bisimulation* is defined for timed automata, and an algorithm for deciding whether two timed automata are bisimilar, is given.

Timed automata is a fairly low-level representation, and automatic translations from more structured representations such as process algebras, timed Petri nets, or high-level real-time programming languages, should exist. Recently, Sifakis et al. have shown how to translate a term of the real-time process algebra ATP to a timed automaton [34].

One promising direction of extending the process model discussed here is to incorporate probabilistic information. This is particularly relevant for systems that control and interact with physical processes. We add probabilities to timed automata by associating fixed distributions with the delays. This extension makes our processes *generalized semi-Markov processes* (GSMPs). Surprisingly, the untiming construction used to test for emptiness of a timed automaton can be used to analyze the behavior of GSMPs also. In [2], we present an algorithm that combines model-checking for TCTL with model-checking for discrete-time Markov chains. The method can also be adopted to check properties specified using deterministic timed automata [3].

Questions other than verification can also be studied using timed automata. For example, Wong-Toi and Hoffmann study the problem of supervisory control of discrete event systems when the plant and specification behaviors are represented by timed automata [48]. The problem of synthesizing schedulers from timed automata specifications is addressed in [15]. Courcoubetis and Yannakakis use timed automata to solve certain minimum and maximum delay problems for real-time systems [12]. For instance, they show how to compute the earliest and the latest time a target state can appear along the runs of an automaton from a given initial state.

References

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

[2] R. Alur, C. Courcoubetis, and D. Dill. Model-checking for probabilistic real-time systems. In *Automata, Languages and Programming: Proceedings of the 18th ICALP*, Lecture Notes in Computer Science 510, 1991.

[3] R. Alur, C. Courcoubetis, and D. Dill. Verifying automata specifications of probabilistic real-time systems. In *Proceedings of REX workshop "Real-time: theory in practice"*, Lecture Notes in Computer Science 600, pages 28–44. Springer-Verlag, 1991.

[4] R. Alur, T. Feder, and T. Henzinger. The benefits of relaxing punctuality. In *Proceedings of the Tenth ACM Symposium on Principles of Distributed Computing*, pages 139–152, 1991.

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

[6] A. Bernstein and P. Harter. Proving real-time properties of programs with temporal logic. In *Proceedings of the Eighth ACM Symposium on Operating System Principles*, pages 164–176, 1981.

[7] R. Büchi. On a decision method in restricted second-order arithmetic. In *Proceedings of the International Congress on Logic, Methodology, and Philosophy of Science 1960*, pages 1–12. Stanford University Press, 1962.

[8] J. Burch, E. Clarke, D. Dill, L. Hwang, and K. L. McMillan. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, 1992.

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

[10] E. Clarke, I. Draghicescu, and R. Kurshan. A unified approach for showing language containment and equivalence between various types of ω-automata. Technical report, Carnegie Mellon University, 1989.

[11] E. 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.

[12] C. Courcoubetis and M. Yannakakis. Minimum and maximum delay problems in real-time systems. In *Proceedings of the Third Workshop on Computer-Aided Verification*, Lecture Notes in Computer Science 575, pages 399–409, 1991.

[13] D. 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.

[14] D. Dill. *Trace Theory for Automatic Hierarchical Verification of Speed-independent Circuits.* MIT Press, 1989.

[15] D. Dill and H. Wong-Toi. Synthesizing processes and schedulers from temporal specifications. In *Proceedings of the Second Workshop on Computer-Aided Verification*, Lecture Notes in Computer Science 531, pages 272–281, 1990.

[16] E. A. Emerson and E. M. Clarke. Using branching-time temporal logic to synthesize synchronization skeletons. *Science of Computer Programming*, 2:241–266, 1982.

[17] E. A. Emerson, A. Mok, A. P. Sistla, and J. Srinivasan. Quantitative temporal reasoning. In *Proceedings of the Second Workshop on Computer-Aided Verification*, Lecture Notes in Computer Science 531, pages 136–145, 1990.

[18] P. Godefroid and P. Wolper. A partial approach to model-checking. In *Proceedings of the Sixth IEEE Symposium on Logic in Computer Science*, pages 406–415, 1991.

[19] D. Harel, A. Pnueli, and J. Stavi. Propositional dynamic logic of regular programs. *Journal of Computer and System Sciences*, 26:222–243, 1983.

[20] E. Harel, O. Lichtenstein, and A. Pnueli. Explicit-clock temporal logic. In *Proceedings of the Fifth IEEE Symposium on Logic in Computer Science*, pages 402–413, 1990.

[21] T. 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.

[22] C. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.

[23] J. Hopcroft and J. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, 1979.

[24] F. Jahanian and A. Mok. Safety analysis of timing properties in real-time systems. *IEEE Transactions on Software Engineering*, SE–12(9):890–904, 1986.

[25] F. Jahanian and A. Mok. A graph-theoretic approach for timing analysis and its implementation. *IEEE Transactions on Computers*, C-36(8):961–975, 1987.

[26] R. Koymans. Specifying real-time properties with metric temporal logic. *Journal of Real-Time Systems*, 2:255–299, 1990.

[27] R. Kurshan. Complementing deterministic Büchi automata in polynomial time. *Journal of Computer and System Sciences*, 35:59–71, 1987.

[28] L. Lamport. What good is temporal logic? In R. Mason, editor, *Information Processing 83: Proceedings of the Ninth IFIP World Computer Congress*, pages 657–668. Elsevier Science Publishers, 1983.

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

[30] H. Lewis. Finite-state analysis of asynchronous circuits with bounded temporal uncertainty. Technical Report TR-15-89, Harvard University, 1989.

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

[32] Z. Manna and A. Pnueli. The temporal framework for concurrent programs. In R. Boyer and J. Moore, editors, *The Correctness Problem in Computer Science*, pages 215–274. Academic Press, 1981.

[33] R. McNaughton. Testing and generating infinite sequences by a finite automaton. *Information and Control*, 9:521–530, 1966.

[34] X. Nicollin, J. Sifakis, and S. Yovine. From ATP to timed graphs and hybrid systems. In *Proceedings of REX workshop "Real-time: theory in practice"*, Lecture Notes in Computer Science 600, pages 549–572. Springer-Verlag, 1991.

[35] J. Ostroff. *Temporal Logic of Real-time Systems.* Research Studies Press, 1990.

[36] A. Pnueli. The temporal logic of programs. In *Proceedings of the 18th IEEE Symposium on Foundations of Computer Science*, pages 46–77, 1977.

[37] A. Pnueli. Applications of temporal logic to the specification and verification of reactive systems: a survey of current trends. In *Current Trends in Concurrency*, Lecture Notes in Computer Science 224, pages 510–584. Springer-Verlag, 1986.

[38] C. Ramchandani. Analysis of asynchronous concurrent systems by Petri nets. Technical Report MAC TR-120, Massachusetts Institute of Technology, 1974.

[39] H. Rogers. *Theory of Recursive Functions and Effective Computability.* McGraw-Hill, 1967.

[40] S. Safra. On the complexity of ω-automata. In *Proceedings of the 29th IEEE Symposium on Foundations of Computer Science*, pages 319–327, 1988.

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

[42] 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.

[43] K. Čerāns. Decidability of bisimulation equivalence for parallel timer processes. In *Proceedings of the Fourth Workshop on Computer-Aided Verifiction*, Lecture Notes in Computer Science, 1992. To appear.

[44] M. Vardi. Verification of concurrent programs – the automata-theoretic framework. In *Proceedings of the Second IEEE Symposium on Logic in Computer Science*, pages 167–176, 1987.

[45] M. 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.

[46] P. Wolper. Temporal logic can be more expressive. *Information and Control*, 56:72–99, 1983.

[47] P. Wolper, M. Vardi, and A. P. Sistla. Reasoning about infinite computation paths. In *Proceedings of the 24th IEEE Symposium on Foundations of Computer Science*, pages 185–194, 1983.

[48] H. Wong-Toi and G. Hoffmann. The control of dense real-time discrete event systems. In *Proceedings of the 30th IEEE Conference on Decision and Control*, pages 1527–1528, 1991.