The Theory of Timed Automata

Rajeev Alur
AT&T Bell Laboratories
Murray Hill, NJ 07974, USA.

David Dill [1]
Department of Computer Science
Stanford University, Stanford, CA 94305, USA.

Abstract. We propose *timed 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* — strings 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 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 Introduction

Modal logics and ω-automata for *qualitative* temporal reasoning about concurrent systems have been studied in great detail (selected references: [Pnu77, MP81, EC82, Lam83, WVS83, Var87, Pnu86, CES86]) 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 linear sequence of states or system events, called an *execution trace* (or just *trace*). The *behavior* of the system is a set of 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. Even when automata are not used directly, they are never far away; for example, automata theory proves useful in developing the basic decision procedures for propositional linear temporal logic.

[1]Supported by the NSF under grant MIP-8858807.

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.

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?

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 more realistic physically. 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.

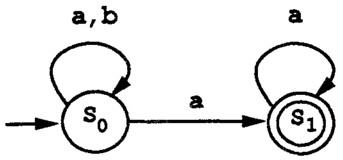

Figure 1: Büchi automaton accepting $(a+b)^*a^\omega$

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 alphabet (see, for example, [HU79]). As opposed to this, an ω-language consists of infinite words. Thus an ω-language over an 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 [Büc62, McN66, Cho74, Tho90]. 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 $\sigma \in \Sigma^\omega$, 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 $\mathit{inf}(r)$ 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 $\mathit{inf}(r) \cap \mathrm{F} \neq \emptyset$. 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

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

The automaton accepts all words with only a finite number of b's; that is, the language $L_0 = (a+b)^*a^\omega$. ■

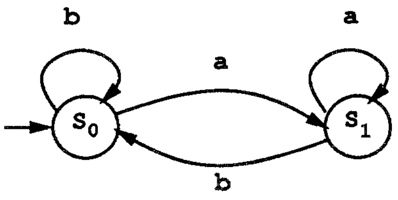

Figure 2: Deterministic Muller automaton accepting $(a+b)^*a^\omega$

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 [Cho74, WVS83]. There are known constructions for complementing Büchi automata [SVW87, Saf88].

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 [SVW87]. However, checking whether the language of one automaton is contained in the language of a *deterministic* automaton can be done in polynomial time [Kur87].

A transition table $\mathcal{A} = \langle \Sigma, S, S_0, E \rangle$ is *deterministic* iff (i) there is a single start state, that is, $|S_0| = 1$, and (ii) the number of a-labeled edges starting at s is at most one for all states $s \in S$ 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 $\langle \Sigma, S, S_0, E \rangle$ with an *acceptance family* $\mathcal{F} \subseteq 2^S$. A run r of $\mathcal{A}$ over a word $\sigma \in \Sigma^\omega$ is an *accepting run* iff $inf(r) \in \mathcal{F}$. 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, more importantly, 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 the language inclusion are known [CDK89].

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 $\tau_i \in$ 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 $t \in$ 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. 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 the time difference between the successive pairs of a and b keeps increasing. L_2 is given as

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

■

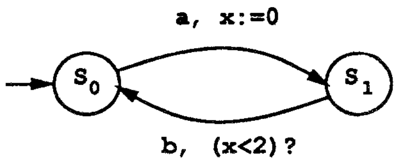

Figure 3: Example of a timed transition table

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

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

■

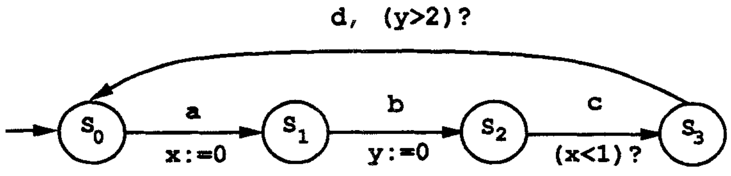

Figure 4: Timed transition table with 2 clocks

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 loops between 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 bounds on the time difference between a and the following b, or c and the following d. This is an important advantage of having access to 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 view 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. Allowing more complex constraints, such as those involving addition of clock values, makes any sort of analysis impossible, for instance, the emptiness problem becomes undecidable.

Definition 3.6 For a set X of clocks, 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 $t \in$ 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* is a tuple $\langle \Sigma, S, S_0, C, E \rangle$, where

- Σ is a finite alphabet,
- S is a finite set of states,
- $S_0 \subseteq S$ is a set of start states,
- C is a finite set of clocks, and
- $E \subseteq S \times S \times \Sigma \times 2^C \times \Phi(C)$ 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 $\lambda \subseteq C$ 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 it. 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 $(\bar{s}, \bar{\nu})$, 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 $s \in \mathrm{S}$ 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 $r = (\bar{s}, \bar{\nu})$ 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 $r = (\bar{s}, \bar{\nu})$ 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 $\{(\sigma, \tau) \mid \mathcal{A} \text{ has an accepting run over } (\sigma, \tau)\}$. ■

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

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

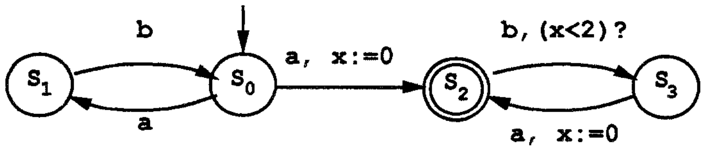

Figure 5: Timed Büchi automaton accepting L_{crt}

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 loops 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 loop 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. ■

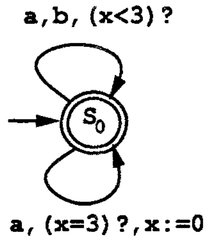

Figure 6: Timed automaton specifying periodic behavior

3.6 Properties of timed regular languages

The study of formal languages has been greatly enriched by the consideration of closure properties and decision problems. Here, we summarize the answers to some of the basic questions of these types, especially those that are relevant to verification.

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. ■

The construction for union is trivial, since we are considering nondeterministic automata. The construction for intersection is a straightforward product of automata. While constructing the product of n TBAs $\mathcal{A}_i = \langle\Sigma, S_i, S_{0_i}, C_i, E_i, F_i\rangle$, $i = 1, 2 \ldots n$, the number of states of the resulting automaton is $n \cdot \Pi_i|S_i|$. The number of clocks is $\Sigma_i|C_i|$, and the size of the edge set is $n \cdot \Pi_i|E_i|$. Note that $|E|$ includes the length of the clock constraints assuming binary encoding for the constants.

The main result on timed automata is an algorithm for checking the emptiness of the language. 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 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 (see [Alu91] for a detailed description of this construction).

Theorem 3.16 Given a TBA $\mathcal{A} = \langle\Sigma, S, S_0, C, E, F\rangle$, there exists a Büchi automaton over Σ which accepts $\mathit{Untime}[L(\mathcal{A})]$. ■

Theorem 3.16 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 $\mathit{Untime}(L)$ is ω-regular.

The untiming construction is interesting by itself, but also gives an immediate solution to the problem of testing a timed automaton for emptiness, since the timed language is empty iff the untimed language is empty. For a timed automaton $\mathcal{A} = \langle\Sigma, S, S_0, C, E, F\rangle$, the Büchi automaton constructed by Theorem 3.16 has size $O[(|S| + |E|) \cdot 2^{\delta(\mathcal{A})}]$, where $\delta(\mathcal{A})$ denotes the length of the timing constraints labeling the edges of $\mathcal{A}$ (assuming binary encoding for the constants). Consequently, 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. One need not construct the untimed automaton explicitly, and the emptiness test can be implemented in polynomial space. This blow-up in complexity seems unavoidable; we reduce the acceptance problem for linear bounded automata, a known PSPACE-complete problem [HU79], to the emptiness question for TBAs to prove the PSPACE lower bound for the emptiness problem. This gives the following theorem.

Theorem 3.17 The problem of deciding the emptiness of the language of a given timed automaton $\mathcal{A}$, is PSPACE-complete. ∎

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.

Surprisingly, several problems that are decidable for finite automata are undecidable for timed automata. The most basic of these is the *universality problem:* Does the language of a given automaton over Σ comprise of all the timed words over Σ? Equivalently: Is the complement of the language of the automaton empty? The next theorem gives this undecidability result.

Theorem 3.18 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. ∎

The above theorem is proved by reducing a Π^1_1-hard problem of 2-counter machines to the universality question. The class Π^1_1 consists of highly undecidable problems, including some nonarithmetical sets (for an exposition of the analytical hierarchy consult, for instance, [Rog67]). Part of the surprise value of this result is the continuous versions of many other problems are *easier* to solve than the discrete versions: for example, number theory is undecidable but the theory of the reals is decidable, and linear programming can be solved in polynomial time, but integer programming is NP-complete[2].

Recall that the language inclusion problem for Büchi automata can be solved in PSPACE. However, it follows from Theorem 3.18 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 3.19 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. ∎

It is equally difficult to decide whether the languages of two timed automata are the same. It can be proved by using these decidability results that, unlike regular languages, timed regular languages are not closed under complementation. The (non-constructive) proof depends on the Π^1_1-hardness of the inclusion problem.

[2]Thanks to Moshe Vardi for this observation

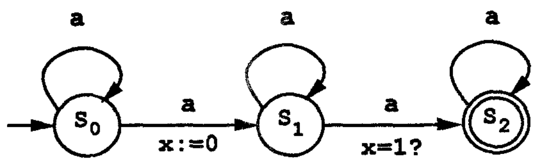

Figure 7: Noncomplementable automaton

Corollary 3.20 The class of timed regular languages is not closed under complementation. ■

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

Example 3.21 The language accepted by the automaton of Figure 7 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 past 1 time unit, would require an unbounded number of clocks. ■

3.7 Comparison of dense and discrete time

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. The following example shows that there is a timed regular language L such that for every $(\sigma, \tau) \in L$, there exists some $\epsilon \geq 0$ such that the sequence of time differences $(\tau_{i+1} - \tau_i)$ converges to the limit ϵ.

Example 3.22 The language accepted by the automaton in Figure 8 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 converges. A sample word accepted by the automaton is

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

■

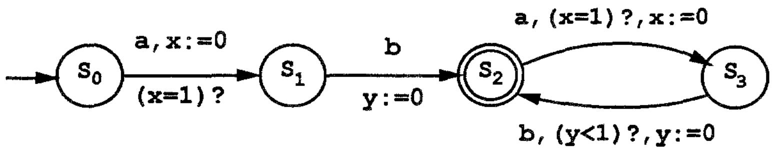

Figure 8: Timed automaton accepting $L_{converge}$

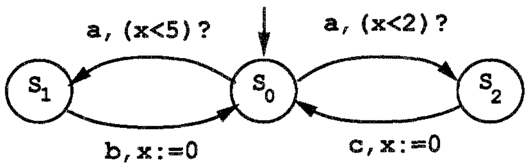

Figure 9: Timed Muller automaton

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 8 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 3.23 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 3.23 Let L be a timed regular language. For every word σ, $\sigma \in Untime(L)$ iff there exists a time sequence τ such that $\tau_i \in$ Q for all $i \geq 1$, and $(\sigma, \tau) \in L$. ∎

3.8 Timed Muller automata

We can define timed automata with Muller acceptance conditions also.

Definition 3.24 A *timed Muller automaton* (TMA) is a tuple $\langle \Sigma, S, S_0, C, E, \mathcal{F} \rangle$, where $\langle \Sigma, S, S_0, C, E \rangle$ is a timed transition table, and $\mathcal{F} \subseteq 2^S$ 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 $inf(r) \in \mathcal{F}$.

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 3.25 Consider the automaton of Figure 9 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 loop 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 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 3.26 A timed language is accepted by some timed Büchi automaton iff it is accepted by some timed Muller automaton. ■

4 Deterministic timed automata

The results of Section 3 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 deterministic timed Muller automata (DTMA) is closed under all the Boolean operations.

4.1 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 a state, the values for all the clocks, and the next input symbol *along with its time of occurrence*, 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 4.1 A timed transition table $\langle \Sigma, S, S_0, C, E \rangle$ is called *deterministic* iff

1. it has only one start state, $|S_0| = 1$, and
2. for all $s \in S$, 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.

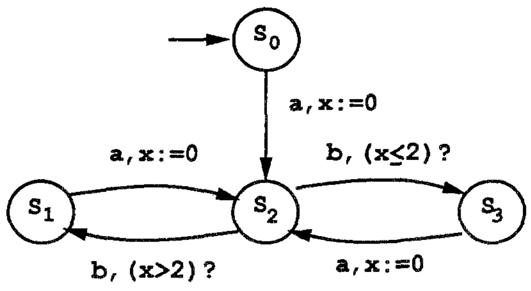

Figure 10: Deterministic timed Muller automaton

Example 4.2 The DTMA of Figure 10 accepts the language L_{crt} of Example 3.13

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

The Muller acceptance family is given by $\{\{s_2, s_3\}\}$. The state s_2 has two mutually exclusive outgoing transitions on b. The acceptance condition requires that the transition with the clock constraint $(x > 2)$ is taken only finitely often. ■

Observe that a deterministic timed transition table has at most one run over a given timed word. Consequently, deterministic timed automata can be easily complemented.

4.2 Closure properties

Now we consider the closure properties for deterministic timed automata. Like deterministic Muller automata, DTMAs are also closed under all Boolean operations.

Theorem 4.3 The class of timed languages accepted by deterministic timed Muller automata is closed under union, intersection, and complementation. ■

Now let us consider the closure properties of deterministic timed Büchi automata (DTBA). 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 is also a DTMA, the complement of a DTBA-language is accepted by a DTMA. The next theorem states the closure properties.

Theorem 4.4 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. ■

4.3 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 4.5 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. ■

4.4 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 4.6 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. ■

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 4.2). This language involves a combination of liveness and timing. We conjecture that no DTBA can specify this property.

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 3.21, ("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 11 which shows the inclusions between 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 12.

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

Figure 11: Classes of timed automata

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

Figure 12: Classes of ω-automata

5 Verification

In this section we discuss how to use the theory of timed automata to prove correctness of finite-state real-time systems. We start by introducing time in linear trace semantics for concurrent processes.

5.1 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 [Hoa78]. In our model, we allow several events to happen simultaneously. 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 5.1 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 5.1 $\sigma_P\lceil\{a\}$ is the trace a^ω. Notice that the projection operation may result in a finite sequence; but we will 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 5.2 Consider another channel Q connected to the channel P of Example 5.1. 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 5.3 Consider the channels of Example 5.2. 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 $ab(acb)^\omega$. ■

5.2 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 5.1 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 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.

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 5.4 Consider the channel P of Example 5.1 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

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

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 5.5 As in Example 5.2 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

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

■

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 5.6 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 5.6 Consider the channels of Example 5.5. 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 5.7 Consider the verification problem of Example 5.3 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. ∎

5.3 ω-automata and verification

We start with an overview of the application of Büchi automata to verify untimed processes [VW86, Var87]. 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 $[\parallel_i P_i]$. 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 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 [BCD+90, GW91].

5.4 Verification using timed automata

For a timed process (A, L), L is a timed language over $\mathcal{P}^+(A)$. A *timed regular process* is the 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 $P_i = (A_i, L(\mathcal{A}_i))$. The first step in the verification process is to construct a TBA $\mathcal{A}_I$ which represents the composite process $[\|_i P_i]$. To implement this, first we need to make the alphabets of various automata identical, and then take the intersection. Combining the two steps, however, reduces the size of the implementation automaton.

Theorem 5.8 Given timed processes $P_i = (A_i, L(\mathcal{A}_i))$, $i = 1, \ldots n$, represented by timed automata $\mathcal{A}_i$, there is a TBA $\mathcal{A}$ over the alphabet $\mathcal{P}^+(\cup_i A_i)$ which represents the timed process $[\|_i P_i]$. ∎

The number of states in $\mathcal{A}_I$ is $[(n+1) \cdot \Pi_i |S_i|]$. The number of clocks for $\mathcal{A}_I$ is $\Sigma_i |C_i|$, and for all clocks x, the value of c_x, the largest constant it gets compared with, remains the same.

The specification of the system is given as another timed regular language S over the alphabet $\mathcal{P}^+(A)$. 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 4.3.

Putting together all the pieces, we conclude:

Theorem 5.9 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. ■

The verification algorithm checks for a cycle with several desired properties in the untimed graph of the product of all the automata. The number of vertices in this graph is $O[|\mathcal{A}_S| \cdot \Pi_i |\mathcal{A}_i| \cdot 2^{|\delta(\mathcal{A}_S)|+\Sigma_i|\delta(\mathcal{A}_i)|}]$.

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.

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 same as contributed by the first factor, namely, exponential in the number of components.

5.5 Verification example

We consider an example of a gate controller at a railroad crossing. The system is composed of three components: TRAIN, GATE and CONTROLLER.

The automaton modeling the train is shown in Figure 13. 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.

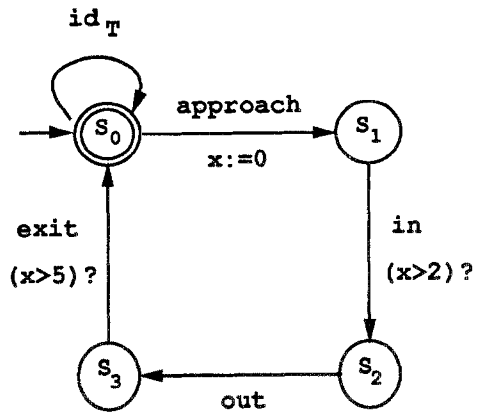

Figure 13: TRAIN

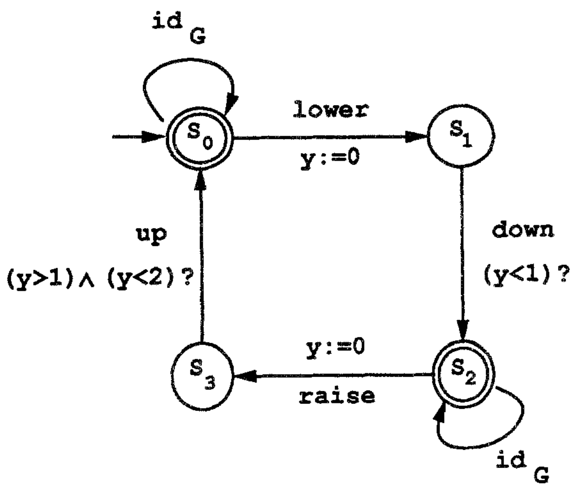

Figure 14: GATE

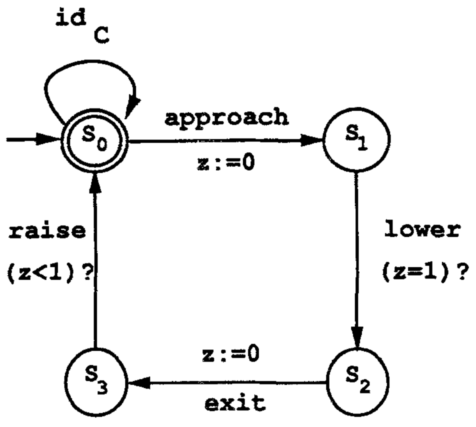

Figure 15: CONTROLLER

The automaton modeling the gate component is shown in Figure 14. 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 15 shows the automaton modeling the controller. The event set is $\{approach, exit, raise, lower, id_C\}$. 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 16. An edge label *in* stands for any event set containing *in*, and an edge label "*in*, $\sim$ *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.

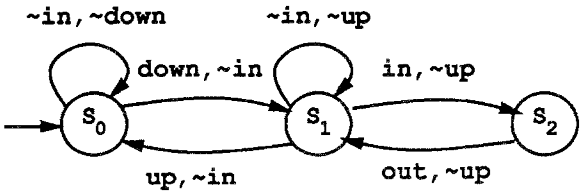

Figure 16: Safety property

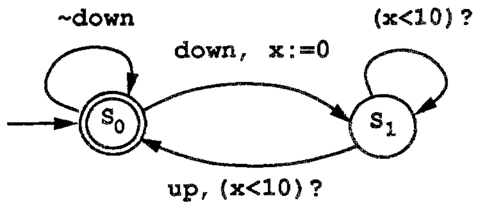

Figure 17: Real-time liveness property

The real-time liveness property is specified by the timed automaton of Figure 17. 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 5. Observe that though the safety property is purely a qualitative property, it does not hold if we discard the timing requirements.

6 Related results

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. We warn the reader that 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 [AFH91]. 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.

Alternatively, specifications can be written in branching-time temporal logics also. In [ACD90], we develop a model-checking algorithm for specifications written in TCTL — a real-time extension of the branching-time temporal logic CTL of [EC82].

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 (see the article *From ATP to Timed Graphs and Hybrid Systems* in this issue).

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.* Surprisingly, the untiming construction used to test for emptiness of a timed automaton can be used to analyze the behavior of GSMPs also. In [ACD91], 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 (see the article *Verifying Automata Specifications of Probabilistic Real-Time Systems* in this issue).

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 [WH91]. The problem of synthesizing schedulers from timed automata specifications is addressed in [DW90]. Courcoubetis and Yannakakis use timed automata to solve certain minimum and maximum delay problems for real-time systems [CY91]. 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

[ACD90] Rajeev Alur, Costas Courcoubetis, and David Dill. Model-checking for real-time systems. In *Proceedings of the Fifth IEEE Symposium on Logic in Computer Science*, pages 414–425, 1990.

[ACD91] Rajeev Alur, Costas Courcoubetis, and David 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.

[AFH91] Rajeev Alur, Tomás Feder, and Thomas Henzinger. The benefits of relaxing punctuality. In *Proceedings of the Tenth ACM Symposium on Principles of Distributed Computing*, pages 139–152, 1991.

[Alu91] Rajeev Alur. *Techniques for Automatic Verification of Real-Time Systems.* PhD thesis, Stanford University, 1991.

[BCD+90] J.R. Burch, E.M. Clarke, D.L. Dill, L.J. Hwang, and K. L. McMillan. Symbolic model checking: 10^{20} states and beyond. In *Proceedings of the Fifth IEEE Symposium on Logic in Computer Science*, pages 428–439, 1990.

[Büc62] Richard 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.

[CDK89] E.M. Clarke, I.A. Draghicescu, and R.P. Kurshan. A unified approach for showing language containment and equivalence between various types of ω-automata. Technical report, Carnegie Mellon University, 1989.

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

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

[CY91] Costas Courcoubetis and Mihalis Yannakakis. Minimum and maximum delay problems in real-time systems. In *Proceedings of the Third Workshop on Computer-Aided Verification, Aalborg University, Denmark*, 1991.

[DW90] David Dill and Howard Wong-Toi. Synthesizing processes and schedulers from temporal specifications. In *Proceedings of the Second Workshop on Computer-Aided Verification, Rutgers University*, 1990.

[EC82] E. Allen Emerson and Edmund M. Clarke. Using branching-time temporal logic to synthesize synchronization skeletons. *Science of Computer Programming*, 2:241–266, 1982.

[GW91] Patrice Godefroid and Pierre Wolper. A partial approach to model-checking. In *Proceedings of the Sixth IEEE Symposium on Logic in Computer Science*, pages 406–415, 1991.

[Hoa78] C.A.R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.

[HU79] John Hopcroft and Jeff Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, 1979.

[Kur87] Robert Kurshan. Complementing deterministic Büchi automata in polynomial time. *Journal of Computer and System Sciences*, 35:59–71, 1987.

[Lam83] Leslie Lamport. What good is temporal logic? In R.E.A. Mason, editor, *Information Processing 83: Proceedings of the Ninth IFIP World Computer Congress*, pages 657–668. Elsevier Science Publishers, 1983.

[McN66] Robert McNaughton. Testing and generating infinite sequences by a finite automaton. *Information and Control*, 9:521–530, 1966.

[MP81] Zohar Manna and Amir Pnueli. The temporal framework for concurrent programs. In R.S. Boyer and J.S. Moore, editors, *The correctness problem in Computer science*, pages 215–274. Academic Press, 1981.

[Pnu77] Amir Pnueli. The temporal logic of programs. In *Proceedings of the 18th IEEE Symposium on Foundations of Computer Science*, pages 46–77, 1977.

[Pnu86] Amir 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.

[Rog67] Hartley Rogers. *Theory of Recursive Functions and Effective Computability.* McGraw-Hill, 1967.

[Saf88] Shmuel Safra. On the complexity of ω-automata. In *Proceedings of the 29th IEEE Symposium on Foundations of Computer Science*, pages 319–327, 1988.

[SVW87] A. Prasad Sistla, Moshe Vardi, and Pierre Wolper. The complementation problem for Büchi automata with applications to temporal logic. *Theoretical Computer Science*, 49, 1987.

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

[Var87] Moshe 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.

[VW86] Moshe Vardi and Pierre 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.

[WH91] Howard Wong-Toi and Girard Hoffmann. The control of dense real-time discrete event systems. 1991.

[WVS83] Pierre Wolper, Moshe Vardi, and A. Prasad Sistla. Reasoning about infinite computation paths. In *Proceedings of the 24th IEEE Symposium on Foundations of Computer Science*, pages 185–194, 1983.