Automata-theoretic Verification of Real-Time Systems

Rajeev Alur
Computing Science Research
AT&T Bell Labs
Murray Hill, NJ 07974.
email: alur@research.att.com

David Dill
Computer Science Department
Stanford University
Stanford, CA 94305
email: dill@cs.stanford.edu

November 2, 1995

1 Introduction

Formal methods for specifying, analyzing, and manipulating the behavior of concurrent systems become much more attractive in practical use if they can be automated. A number of methods based on *finite-state* representations have achieved considerable success in practical applications such as protocol and hardware verification, precisely because many problems are decidable for finite-state representations. Finite-state verification methods include checking equivalences (such as bisimulation), preorders (such as simulation), temporal logic properties (eg. CTL model-checking), and inclusion of the language of one automaton in another.

Until recently, temporal logics and finite automata were primarily concerned with *qualitative* temporal reasoning about systems. For example, whether a system deadlocks or livelocks, whether a property is always true, or whether some response eventually occurs. More recently, ways of extending finite-state techniques to timed systems have been discovered, which retain many of the desirable properties of conventional finite representations.

In this chapter, we will concentrate on *linear-time models*, although finite-state real-time techniques can also be applied to *branching-time* problems, such as (timed) CTL model-checking and bisimulation checking. In the *linear time model,* it is assumed that an execution can be completely modeled as a sequence of states or system events, called a *trace*. The *behavior* of the system is a set of such traces. 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, we can use finite automata, leading to effective constructions and decision procedures for *automatically* manipulating and analyzing system behavior.

In qualitative models, it is useful to describe non-terminating executions, so that *liveness* properties, such as "if a request occurs infinitely often, so does the response" can be expressed. Consequently many verification theories are based on the theory of ω-regular languages, which reasons about sets of infinite strings, instead of the finite strings usually considered in ordinary regular languages (e.g. the system COSPAN [Kur94] or the system HSIS [ABB+94]). In our linear real-time model, an execution is an infinite trace of events, and time is added by pairing each event of a trace with a time value. Time values are chosen from the set of reals. Such a model is called a *dense-time* model. The alternative *discrete-time* model uses integer time values, and requires that continuous time be approximated by choosing some fixed quantum *a priori*, which limits the accuracy with which physical systems can be modeled. Dealing with dense time in a finite-automata framework is more difficult than dealing with discrete time, because the transformation from a set of dense-time traces into an ordinary formal language is not obvious. Instead, we have developed a theory of *timed* formal languages and *timed automata* to support automated reasoning about such systems. The study of timed finite automata has yielded interesting theoretical results, and, if progress continues at its current rate, is likely to succeed in practice just as qualitative finite-state methods have.

Overview

We begin with an overview of ω-automata and verification for untimed systems (Section 2). Then, we define timed automata by augmenting ω-automata with a set of real-valued variables called *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 accept *timed words* — infinite sequences in which a real-valued time of occurrence is associated with each symbol. 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 present an overview of the formal language theory for timed automata. Due to the real-valued clock variables, the state space of a timed automaton is infinite. The *untiming* algorithm, discussed in detail in Section 5, constructs a finite quotient of this space, and is the key to algorithmic solutions to decision problems for timed automata.

Section 6 outlines the application of timed automata to verification of timed systems. A timed system is modeled as a collection of timed automata representing the various components of the system. The specification to be checked is given as a deterministic timed automaton representing the correct behaviors. The system satisfies the property if the language of the product of the automata modeling the components is contained in the language of the specification automaton. We present an algorithmic solution to the verification problem. To alleviate the high computational complexity of the verification algorithm, different verification tools use different heuristics. We discuss some of the implemented solutions.

2 Automata-theoretic Verification of Untimed Systems

In this section we will briefly review the relevant aspects of the theory of *ω-regular languages*, and its application to modeling and automatic verification of untimed systems. We refer the reader to [Tho90] for a summary of the theory of ω-regular languages, and to [Kur94] for its application to verification.

2.1 Büchi automata

The more familiar definition of a formal language is as a set of finite words over some given (finite) alphabet. As opposed to this, an ω-language consists of infinite words. Thus an ω-language over

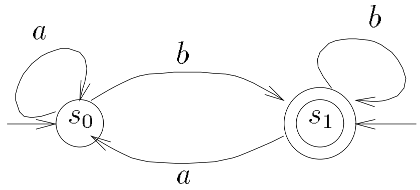


Figure 1: Büchi automaton accepting words with infinitely many b's

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.

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. Formally, for an infinite word $\overline{\sigma} = \sigma_1 \sigma_2 \ldots$ 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 the transition table $\mathcal{A}$ 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. We will use Büchi acceptance condition (alternatives such as Streett acceptance or Muller acceptance lead to expressively equivalent definitions, see [Tho90]). 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 the Büchi automaton $\mathcal{A}$ 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 $\{a, b\}$. Both states are start states and s_1 is the accepting state. The automaton accepts all words with an infinite number of b's. Thus, the automaton expresses the constraint that every a is followed by b. ■

An ω-language is called *ω-regular* iff it is accepted by some Büchi automaton. The class of ω-regular languages is closed under all the Boolean operations. Language intersection is implemented by a product construction for Büchi automata. There are known constructions for complementing Büchi automata.

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

A transition table $\mathcal{A} = \langle \Sigma, \mathrm{S}, \mathrm{S}_0, \mathrm{E}\rangle$ is *deterministic* iff there is a single start state, and 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, and this allows an efficient way to complement.

2.2 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.

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. Formally, given a set A of events, a *trace* $\overline{\sigma} = \sigma_1\sigma_2\ldots$ 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 2.2 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 $\overline{\sigma}_P = \{a\}, \{b\}, \{a\}, \{b\}\ldots$ Often we will denote the singleton set $\{a\}$ by the symbol a, and infinite repetition $abababa\ldots$ by $(ab)^\omega$. 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 $\overline{\sigma} \in \mathcal{P}(A)^\omega$ onto $B \subseteq A$ (written $\overline{\sigma}\lceil B$) is formed by intersecting each event set in $\overline{\sigma}$ with B and deleting all the empty sets from the sequence. For instance, in Example 2.2 $\overline{\sigma}_P\lceil\{a\}$ 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 $\{P_i = (A_i, X_i) \mid i = 1, 2, \ldots n\}$,

$$\|_i\, P_i \;=\; (\, \cup_i A_i,\; \{\overline{\sigma} \in \mathcal{P}(\cup_i A_i)^\omega \mid \overline{\sigma}\lceil A_i \in X_i \text{ for } i = 1, \ldots n\}.$$

Thus $\overline{\sigma}$ is a trace of $\|_i\, P_i$ iff $\overline{\sigma}\lceil A_i$ 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 2.3 Consider another channel Q connected to the channel P of Example 2.2. 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 a *language inclusion problem:* is the language of the implementation automaton a subset of the language of the specification automaton? Intuitively, the specification automaton gives the set of allowed behaviors, so the implementation is included in the specification if and only if every actual behavior of the implementation is allowed. 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 2.4 Consider the channels of Example 2.3. 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 $ab(acb)^\omega$. ■

2.3 ω-automata and verification

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 $\mathcal{A}_I$ for the implementation I using the construction for language intersection for Büchi automata.

The specification is given as a Büchi automaton $\mathcal{A}_S$ over the alphabet $\mathcal{P}(A)$. The implementation meets the specification iff $L(\mathcal{A}_I) \subseteq L(\mathcal{A}_S)$. 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 provably computationally expensive, namely, 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 hence, in practice, either deterministic automata are used for specifications, or the user provides complement of the specification (i.e. the automaton that accepts "bad" traces).

A variety of heuristics are used to implement the verification strategy outlined above. Note that testing language inclusion corresponds to finding cycles in the product of the automata $\mathcal{A}_i$ together with the complement $\mathcal{A}_S^c$. There is no need to explicitly construct the product automaton, and thus the search is done on-the-fly. The search may be done by enumerating states in a depth-first fashion, or by manipulating sets of states in a breadth-first fashion. The latter technique—also called symbolic model checking—sometimes turns out to be effective even for systems with a large number of states (see [McM93] for an overview of symbolic model checking using binary decision diagrams). For complex problems, the verification algorithm is used in conjunction with compositional and hierarchical proof methods that allow a systematic decomposition of the verification problem (see [Dil89b, Kur94, LT87] for some of the methodologies, and also the article by Lynch in this volume).

2.4 Train-Gate Controller

We consider an example of an automatic controller that opens and closes a gate at a railroad crossing. The system is composed of three components: TRAIN, GATE and CONTROLLER as shown in Figure 2. All of them are modeled as Büchi automata. The example is simplified, however, it suffices to illustrate the basic concepts in automata-theoretic automated verification. Note that, for now, we model only the sequencing of events within each component, and timing will be added to the model later.

The event set for the train automaton is $\{$*approach, exit, in, out,* $id_T\}$. 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 event set for the gate automaton 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 can take its idling transition id_G in states s_0 or s_2 forever. Finally, the event set for the controller 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. Whenever it receives the signal *exit*, it responds with a signal *raise* to the gate.

The entire system is then TRAIN $\|$ GATE $\|$ 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 automaton $\mathcal{A}_I$ specifying the entire system is obtained by composing the above three automata.

The safety correctness requirement for the system is that whenever the train is inside the gate, the gate should be closed. The safety property is specified by the automaton of Figure 3. 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.

To verify the safety requirement, we need to check whether the language of $\mathcal{A}_I$ is contained in the language of the safety automaton. This can be done using an automated tool such as COSPAN. The desired inclusion does not hold, and the verification tool reports a trace of $\mathcal{A}_I$ that violates the safety property (the trace consists of only two events, *approach* followed by *in*). We need to introduce sufficient delay between the events *approach* and *in* so that the safety property is

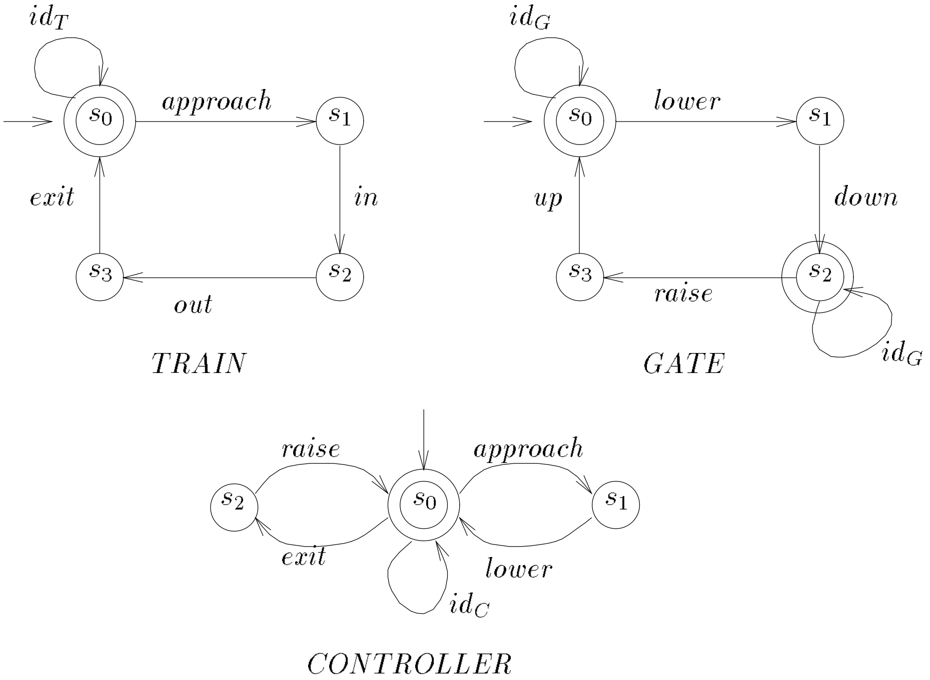


Figure 2: Train-gate controller

satisfied.

3 Timed Languages

To introduce time in trace semantics, we define timed words by coupling a real-valued time with each symbol in a word.

3.1 Timed languages

The set of nonnegative real numbers, R^+, is chosen as the time domain. A *time sequence* $\overline{\tau} = \tau_1\tau_2\cdots$ is an infinite sequence of time values $\tau_i \in \mathsf{R}^+$ with $\tau_i > 0$, satisfying the following constraints:

1. *Monotonicity:* $\overline{\tau}$ 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 $(\overline{\sigma}, \overline{\tau})$ consisting of an infinite word $\overline{\sigma} = \sigma_1\sigma_2\ldots$ over Σ and a time sequence $\overline{\tau}$. A *timed language* over Σ is a set of timed words over Σ. 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. The progress requirement ensures that we disallow infinitely many events to occur within a finite interval of time. Let us consider some examples of timed languages.

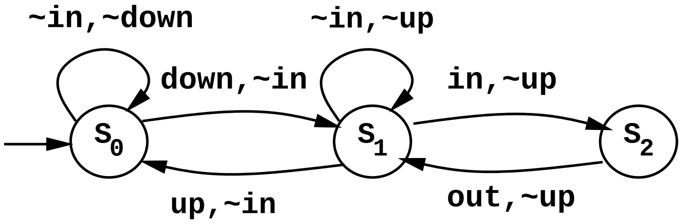


Figure 3: Safety property

Example 3.1 Let the alphabet be $\{a, b\}$. Define a timed language L_1 to consist of all timed words $(\overline{\sigma}, \overline{\tau})$ such that there is no b after time 5.6. Thus the language L_1 is given by

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

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

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

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 $(\overline{\sigma}, \overline{\tau})$ on the first component: for a timed language L over Σ, $\mathit{Untime}(L)$ is the ω-language consisting of words $\overline{\sigma}$ such that $(\overline{\sigma}, \overline{\tau}) \in L$ for some time sequence $\overline{\tau}$. For instance, referring to Example 3.1, $\mathit{Untime}(L_1)$ is the ω-language with words that contain only finitely many b's, and $\mathit{Untime}(L_2)$ consists of a single word $(ab)^\omega$.

3.2 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 2.2 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.

A *timed trace* over a set of events A is a pair $(\overline{\sigma}, \overline{\tau})$ where $\overline{\sigma}$ is a trace over A, and $\overline{\tau}$ is a time sequence. In a timed trace $(\overline{\sigma}, \overline{\tau})$, 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 $\tau_0 = 0$. 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 3.2 Consider the channel P of Example 2.2 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 $\rho_P \;=\; (a, 1), (b, 2), (a, 4), (b, 5) \ldots$ and it is represented as a timed process $P^T = (\{a, b\}, \{\rho_P\})$. $\blacksquare$

The operations on untimed processes are extended in the obvious way to timed processes. To get the projection of $(\overline{\sigma}, \overline{\tau})$ onto $B \subseteq A$, we first intersect each event set in $\overline{\sigma}$ 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 the 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: the parallel composition of two timed traces is either a single timed trace or is empty.

Example 3.3 As in Example 2.3 consider another channel Q connected to P. For Q, as before, the only possible trace is $\overline{\sigma}_Q = (bc)^\omega$. 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

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

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

$$(a, 1), (b, 2), (c, 3.8), (a, 4), (b, 5), (c, 6.02) \ldots \blacksquare$$

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 $\overline{\sigma}$ such that $(\overline{\sigma}, \overline{\tau}) \in L$ for some time sequence $\overline{\tau}$.

Note that

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

However, as Example 3.4 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 3.4 For the channels of Example 3.3, $Untime(P^T) = P$ and $Untime(Q^T) = Q$. The composition $P^T \parallel Q^T$ has a unique untimed trace $(abc)^\omega$, but $P \parallel Q$ has infinitely many traces: between every pair of b events all possible orderings of an event a and an event c are admissible. $\blacksquare$

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

Example 3.5 Consider the verification problem of Example 2.4 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 $(\{a, b, c\}, \{((abc)^\omega, \overline{\tau})\})$. 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. $\blacksquare$

4 Timed automata

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.

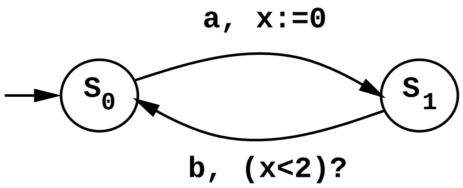


Figure 4: Example of a timed transition table

4.1 Transition tables with timing constraints

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 4.1 Consider the timed transition table of Figure 4. 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. ■

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 4.2 The timed transition table of Figure 5 uses two clocks x and y, and accepts the language

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

The clock x gets set to 0 each time the automaton 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

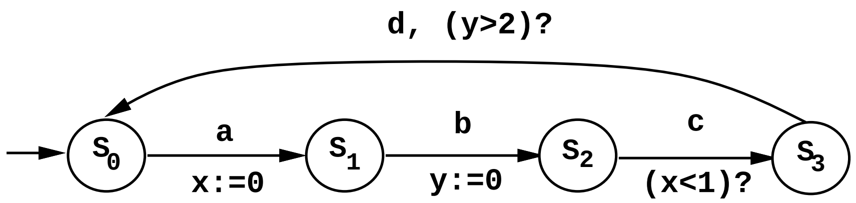


Figure 5: Timed transition table with 2 clocks

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 one another. 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.

4.2 Clock constraints and clock interpretations

To define timed automata formally, we need to say what type of clock constraints are allowed on the edges. An atomic constraint compares a clock value with a time constant, and a clock constraint is a conjunction of atomic constraints. Any value from Q, the set of nonnegative rationals, can be used as a time constant. Formally, for a set X of clock variables, the set $\Phi(X)$ of *clock constraints* δ is defined inductively

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

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

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 $\nu(x)+t$. 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.

4.3 Timed transition tables

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,
- $S_0 \subseteq S$ 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 $(\overline{\sigma}, \overline{\tau})$, 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 r, denoted by $(\overline{s}, \overline{\nu})$, of a timed transition table $\langle \Sigma, S, S_0, C, E \rangle$ over a timed word $(\overline{\sigma}, \overline{\tau})$ 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, for all $i \geq 0$, $s_i \in S$ and ν_i is a clock interpretation for C, satisfying the following requirements:

- *Initiation:* $s_0 \in S_0$, and $\nu_0(x) = 0$ for all $x \in C$.
- *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 $inf(r)$ consists of those states $s \in S$ such that $s = s_i$ for infinitely many $i \geq 0$.

Example 4.3 Consider the timed transition table of Example 4.2, and the word $(a, 2), (b, 2.7), (c, 2.8), (d, 5) \ldots$ Below we give the initial segment of the run. A clock interpretation is represented by listing the values $[x, y]$.

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

4.4 Timed regular languages

We can couple acceptance criteria with timed transition tables, and use them to define timed languages. A *timed Büchi automaton* (in short TBA) is a tuple $\langle \Sigma, S, S_0, C, E, F \rangle$, where $\langle \Sigma, S, S_0, C, E \rangle$ is a timed transition table, and $F \subseteq S$ is a set of *accepting* states. A run $r = (\overline{s}, \overline{\nu})$ of a TBA over a timed word $(\overline{\sigma}, \overline{\tau})$ is called an *accepting run* iff $inf(r) \cap F \neq \emptyset$. For a TBA $\mathcal{A}$, the language $L(\mathcal{A})$ of timed words it accepts is defined to be the set $\{(\overline{\sigma}, \overline{\tau}) \mid \mathcal{A}$ has an accepting run over $(\overline{\sigma}, \overline{\tau})\}$.

In analogy with the class of languages accepted by Büchi automata, we call the class of timed languages accepted by TBAs timed regular languages: a timed language L is a *timed regular language* iff $L = L(\mathcal{A})$ for some TBA $\mathcal{A}$.

Example 4.4 The language L_3 of Example 4.2 is a timed regular language (the timed transition table of Figure 5 coupled with the acceptance set consisting of all the states, accepts L_3).

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

A typical example of a nonregular timed language is the language L_2 of Example 3.1. It requires that the time difference between the successive pairs of a and b form an increasing sequence. Another nonregular language is $\{(a^\omega, \overline{\tau}) \mid \forall i.\, (\tau_i = 2^i)\}$. ■

4.5 Properties of Timed Automata

The closure properties and decision problems for timed automata play an important role in their application to verification. We mention the relevant results here, and refer the reader to [AD94] for details.

The class of timed regular languages is closed under intersection. That is, given TBAs $\mathcal{A}_i$, it is possible to construct a TBA that accepts the intersection of $L(\mathcal{A}_i)$'s. The construction is a modification of the product construction for Büchi automata.

Consider TBAs $\mathcal{A}_i = \langle \Sigma, \mathrm{S}_i, \mathrm{S}_{i_0}, \mathrm{C}_i, \mathrm{E}_i, \mathrm{F}_i \rangle$, $i = 1, 2, \ldots n$ with disjoint clock sets. 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 $0 \leq k < n$. 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 0, and it is incremented from k to $(k+1)$ (modulo n) iff the current state of the k-th automaton is an accepting state.

The initial states of $\mathcal{A}$ are of the form $\langle s_1, \ldots s_n, 0\rangle$ 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 $s_{k+1} \in \mathrm{F}_{k+1}$, 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 $0, \ldots (n-1)$ 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 $\langle s_1, \ldots s_n, n-1\rangle$, where $s_n \in \mathrm{F}_n$.

The class of timed regular languages is also closed under union. However, it is not closed under complement.

Example 4.5 The language accepted by the automaton of Figure 6 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. ■

In Section 5, we give an algorithm for testing whether the language of a TBA is empty. Recall that to test whether the language of one Büchi automaton is contained in another, we test the emptiness of the language of the product of the first automaton with the complement of the latter. This strategy cannot be used for TBAs, as it is not possible to automatically complement a TBA. In fact, there is no algorithm for testing whether the language of one TBA is contained in another

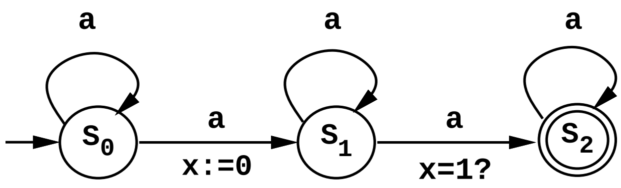


Figure 6: Noncomplementable automaton

(the language inclusion problem is undecidable). The language inclusion problem is solvable if we use deterministic TBAs as specification automata.

4.6 Deterministic timed automata

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. 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. Deterministic timed automata can be easily complemented because a deterministic timed transition table has at most one run over a given timed word. The algorithm for checking emptiness can be used to test whether the language of one TBA is included in the language of a deterministic TBA. More details regarding deterministic TBAs can be found in [AD94].

5 Checking emptiness

In this section we describe 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.

Recall that our definition of timed automata allows clock constraints which involve comparisons with rational constants. If the clock constraints of the given automaton $\mathcal{A}$ involve rational constants, we can multiply each constant by the least common multiple of denominators of all the constants appearing in the clock constraints of $\mathcal{A}$. This transformation leaves the untimed language unchanged. Consequently, for checking emptiness, we can restrict ourselves to timed automata whose clock constraints involve only integer constants.

5.1 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: 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. For each $x \in \mathrm{C}$, let c_x be the largest integer c such that x is compared with c in 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:

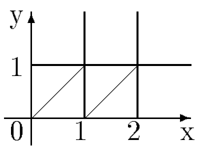
1. For all $x \in \mathrm{C}$, either $\lfloor \nu(x) \rfloor$ and $\lfloor \nu'(x) \rfloor$ are the same, or both $\nu(x)$ and $\nu'(x)$ exceed c_x.
2. For all $x, y \in \mathrm{C}$ with $\nu(x) \leq c_x$ and $\nu(y) \leq c_y$, $\mathit{fract}(\nu(x)) \leq \mathit{fract}(\nu(y))$ iff $\mathit{fract}(\nu'(x))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

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 5.1 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 7. ■

The role of the region equivalence can be understood by defining a (time-abstract) transition relation over the extended states. For two extended states $\langle s, \nu \rangle$ and $\langle s', \nu' \rangle$, and an alphabet



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 7: Clock regions

symbol a, define $\langle s, \nu\rangle \xrightarrow{a} \langle s', \nu'\rangle$ iff there exists a time increment $t \in \mathsf{R}^+$ and an edge $\langle s, s', a, \lambda, \delta\rangle$ such that $\nu + t$ satisfies δ and $\nu' = [\lambda \mapsto 0](\nu + t)$. Thus, $\langle s, \nu\rangle \xrightarrow{a} \langle s', \nu'\rangle$ iff the automaton in extended state $\langle s, \nu\rangle$ can let some time elapse, and read the input symbol a to transition to $\langle s', \nu'\rangle$. The crucial property of the equivalence relation $\sim$ is the following:

> If $\nu_1 \sim \nu_2$ and $\langle s, \nu_1\rangle \xrightarrow{a} \langle s', \nu_1'\rangle$ then there exists a clock interpretation ν_2' such that $\nu_1' \sim \nu_2'$ and $\langle s, \nu_2\rangle \xrightarrow{a} \langle s', \nu_2'\rangle$.

Due to this property, the equivalence relation $\sim$ is called a time-abstract bisimulation.

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 conclude that the number of clock regions is bounded by $[|\mathrm{C}|! \cdot 2^{|\mathrm{C}|} \cdot \Pi_{x \in \mathrm{C}}(2c_x + 2)]$. Thus, the number of clock regions is exponential in the encoding of the clock constraints.

5.2 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 $\nu' \in \alpha'$.

For a timed transition table $\mathcal{A} = \langle \Sigma, \mathrm{S}, \mathrm{S}_0, \mathrm{C}, \mathrm{E}\rangle$, 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 $s \in \mathrm{S}$ and α is a clock region.
- The initial states are of the form $\langle s_0, [\nu_0]\rangle$ where $s_0 \in \mathrm{S}_0$ and $\nu_0(x) = 0$ for all $x \in \mathrm{C}$.
- $R(\mathcal{A})$ has an edge $\langle\langle s, \alpha\rangle, \langle s', \alpha'\rangle, a\rangle$ iff $\langle s, \nu\rangle \xrightarrow{a} \langle s', \nu'\rangle$ for some $\nu \in \alpha$ and some $\nu' \in \alpha'$.

Example 5.2 Consider the timed automaton $\mathcal{A}_0$ shown in Figure 8. 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. ■

Let us establish a correspondence between the runs of $\mathcal{A}$ and the runs of $R(\mathcal{A})$. For a run $r = (\overline{s}, \overline{\nu})$ of $\mathcal{A}$ of the form

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

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

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

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: a run $r = (\overline{s}, \overline{\alpha})$ of the region automaton $R(\mathcal{A})$ is *progressive* iff for each clock $x \in \mathrm{C}$, there are infinitely many $i \geq 0$ such that α_i satisfies $[(x = 0) \vee (x > c_x)]$. The correspondence between the runs of $\mathcal{A}$ and the runs of $R(\mathcal{A})$ can be made precise now: 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 $(\overline{\sigma}, \overline{\tau})$ such that r equals $[r']$.

Example 5.3 Consider the region automaton $R(\mathcal{A}_0)$ of Figure 8. 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). ■

5.3 The untiming construction

For a timed automaton $\mathcal{A}$, its region automaton can be used to recognize *Untime*$[L(\mathcal{A})]$. For this purpose, we need to add acceptance conditions so that only progressive runs satisfy the accepting conditions. This leads to the main theorem for timed automata:

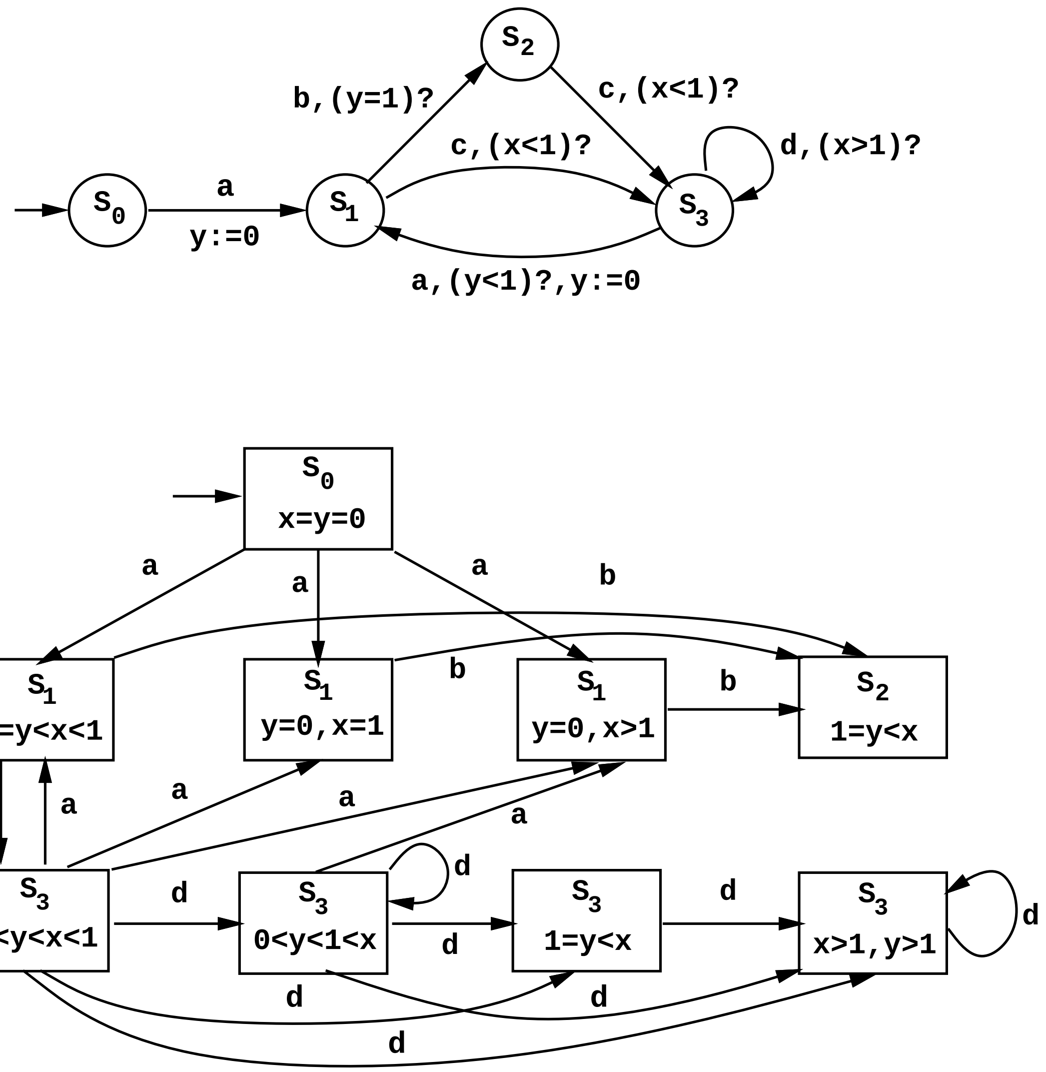


Figure 8: Automaton $\mathcal{A}_0$ and its region automaton

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* $Untime[L(\mathcal{A})]$,

or equivalently,

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

Example 5.4 Let us consider the region automaton $R(\mathcal{A}_0)$ of Example 5.2 again. Since all states of $\mathcal{A}_0$ are accepting, from the description of the progressive runs in Example 5.3 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

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

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. 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 $O[(|\mathrm{S}| + |\mathrm{E}|) \cdot 2^{|\delta(\mathcal{A})|}]$, where $|\delta(\mathcal{A})|$ is the length of the encoding of the clock constraints of $\mathcal{A}$. This blow-up in the length of clock constraints seems unavoidable; technically, the problem of checking emptiness of a TBA is PSPACE-complete. Note that the source of this complexity is not the choice of R^+ to model time. The PSPACE lower bound holds even if we leave the syntax of timed automata unchanged, but use the discrete domain N to model time.

6 Verification

In this section we discuss how to use the theory of timed automata to prove correctness of finite-state real-time systems.

6.1 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.

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 $[\|_i\, P_i]$. 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 $L(\mathcal{A}_I) \subseteq S$. If S is given as a deterministic TBA $\mathcal{A}_S$, then we can solve this algorithmically. 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 deterministic TBA $\mathcal{A}_S = \langle \mathcal{P}(A), \mathrm{S}_0, \mathrm{S}_{0_0}, \mathrm{C}_0, \mathrm{E}_0, \mathrm{F}_0 \rangle$. 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. The transitions of the region automaton $R(\mathcal{A})$ are defined from the edges of $\mathcal{A}$ as described in Section 5. To test the desired inclusion, the algorithm searches for a cycle in the region automaton such that

(1) it is accessible from an initial state of $R(\mathcal{A})$,
(2) it satisfies the progressiveness 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 cycle does not contain a state whose 0-th component belongs to the accepting set F_0.

The desired inclusion does not hold iff a cycle with all the above conditions can be found.

6.2 Verification example

Let us revisit the railroad controller example. We introduce the following timing characteristics to the model of Section 2.4. The train is required to send the signal *approach* at least 2 minutes before it enters the crossing. Furthermore, we know that the maximum delay between the signals *approach* and *exit* is 5 minutes. The gate responds to the signal *lower* by closing within 1 minute, and responds to the signal *raise* within 1 to 2 minutes. The response time of the controller to the *approach* signal is 1 minute, and to the signal *exit* is at most 1 minute. These constraints can easily be expressed using clocks, and the revised model is shown in Figure fig:timedgrc. As before, the implementation timed automaton $\mathcal{A}_I$ is the parallel composition TRAIN $\|$ GATE $\|$ CONTROLLER.

In addition to the safety requirement, we can now consider the real-time liveness requirement that the gate is never closed at a stretch for more than 10 minutes. The real-time liveness property is specified by the timed automaton of Figure 10. 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 6. Observe that though the safety property is purely a qualitative property, it does not hold if we discard the timing requirements. In case of both properties, the region automaton of

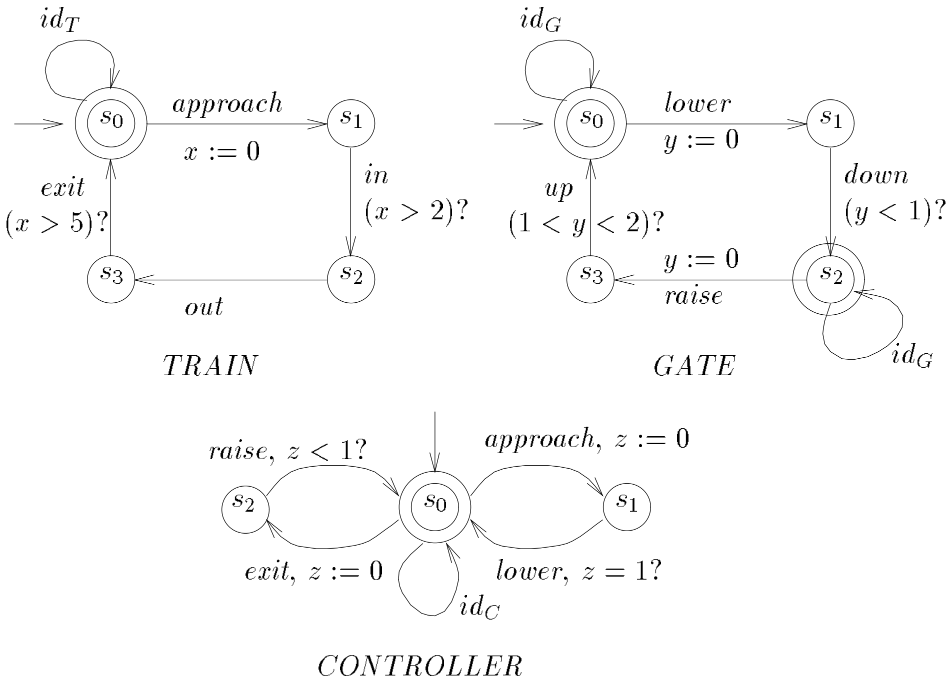


Figure 9: Train-gate controller with timing

the product has only a few reachable vertices (approximately 100), and all the tools can verify the property quickly (less than 1 minute).

6.3 Heuristics

The number of regions in a region automaton is exponential in the total number of clocks, and is proportional to the magnitudes of constants in the clock constraints. To alleviate this blow-up a variety of heuristics have been proposed.

Manipulating with Zones

The first heuristic attempts to group regions together. Consider a timed transition table $\mathcal{A} = \langle \Sigma, \mathrm{S}, \mathrm{S}_0, \mathrm{C}, \mathrm{E} \rangle$. A clock zone is a union of one or more clock regions. The zone automaton $Z(\mathcal{A})$ is a transition table over the alphabet Σ

- The states of $Z(\mathcal{A})$ are of the form $\langle s, \alpha \rangle$ where $s \in \mathrm{S}$ and α is a clock zone.
- The initial states are of the form $\langle s_0, [\nu_0] \rangle$ where $s_0 \in \mathrm{S}_0$ and $\nu_0(x) = 0$ for all $x \in \mathrm{C}$.
- $Z(\mathcal{A})$ has an edge $\langle s, \alpha \rangle \xrightarrow{a} \langle s', \alpha' \rangle$ iff the zone α' contains all clock interpretations ν' such that $\langle s, \nu \rangle \xrightarrow{a} \langle s', \nu' \rangle$ for some $\nu \in \alpha$.

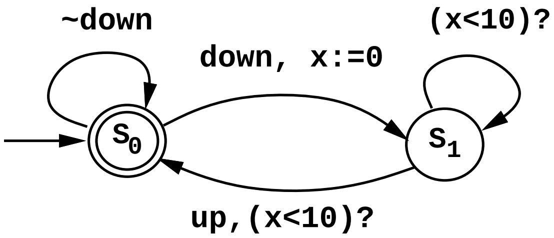


Figure 10: Real-time liveness property

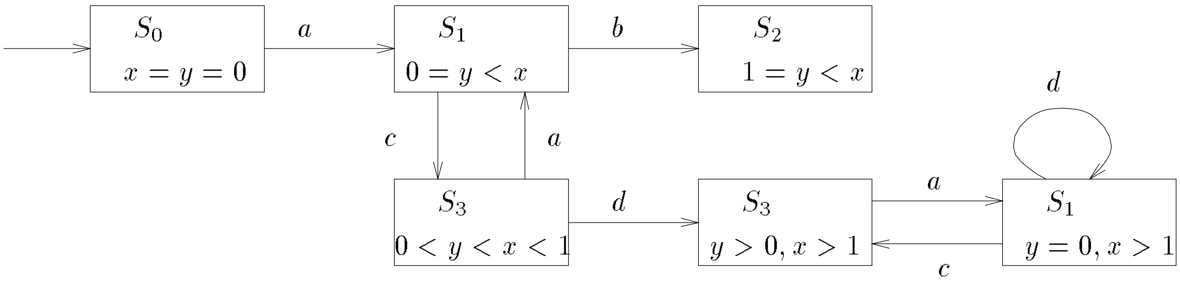


Figure 11: Reachable zone automaton

Example 6.1 Let us revisit the region construction of Example 5.2 (see Figure 8). The reachable part of the zone automaton is shown in Figure 6.3. Note that, unlike the region automaton, in the zone automaton, each vertex has at most one successor per input symbol. The number of vertices of $Z(\mathcal{A}_0)$ is less than the number of vertices of $R(\mathcal{A}_0)$. ■

The emptiness of the language of a timed automaton $\mathcal{A}$ can be checked by searching for cycles in the zone automaton $Z(\mathcal{A})$. Theoretically, the number of zones is exponential in the number of regions, and thus, the zone automaton may be exponentially bigger than the region automaton. However, in practice, the zone automaton has fewer reachable vertices, and thus, leads to an improved performance. Furthermore, while the number of clock regions grows with the magnitudes of the constants used in the clock constraints, experience indicates that the number of reachable zones is relatively insensitive to the magnitudes of constants.

Observe that if the timed transition table $\mathcal{A}$ has n clocks, then each zone is a subset of the n-dimensional euclidean space R^{+n}. Each zone can be represented by linear inequalities over the clock variables. Let us call a zone *simple* if it can be described as a conjunction of clock constraints and formulas of the form $x - y \leq c$, $x - y \geq c$, $x - y < c$, and $x - y > c$ for clocks x and y, and constant c. Thus, a simple zone is convex, and is described by comparing either a clock value, or difference of two clocks, with constants. The structure of timed transition tables ensures that

If a vertex $\langle s, \alpha \rangle$ of the zone automaton $Z(\mathcal{A})$ is reachable from an initial state, then the zone α is a (finite) union of simple zones.

A simple zone can be nicely represented by a structure called *difference-bound-matrix* (DBM) (see [Dil89a] for details). While searching for cycles in the zone automaton, each zone is maintained as a union of DBMs, and the edges of the zone automaton are computed on the fly. The DBM representation allows efficient computation of the successors of a vertex. Furthermore, the DBM representation is canonical, hence, testing equality between two zones is easy.

The tool KRONOS allows verification of timed automata based on search using zones. Instead of deterministic timed automata as specifications, KRONOS uses timed μ-calculus as the specification language. We refer the reader to [HNSY94] for the theory underlying KRONOS, and to [NOSY93] for the description of the tool and its applications.

Approximations

A variety of abstract interpretation techniques can be used to improve the performance of searching in the zone automaton. As we indicated, manipulating with simple zones is efficient. The vertices of the zone automaton contain unions of simple zones. The search can be speeded up by replacing a union of simple zones by the smallest simple zone containing them. For instance, the zone $x < 1 \vee 2 < x < 3$ is replaced by the simple zone $x < 3$.

Formally, for a timed transition table $\mathcal{A}$, the approximate-zone automaton $Z^*(\mathcal{A})$ is a transition table as follows. A state of $Z^*(\mathcal{A})$ is a pair $\langle s, \alpha \rangle$ consisting of a state $s \in \mathrm{S}$ and a simple zone α. A state $\langle s, \alpha \rangle$ is initial if $s \in \mathrm{S}_0$ and α contains the single clock interpretation that assigns 0 to all clocks. $Z^*(\mathcal{A})$ has an edge $\langle s, \alpha \rangle \stackrel{a}{\rightarrow} \langle s', \alpha' \rangle$ iff the zone α' is the smallest simple zone that contains all clock interpretations ν' such that $\langle s, \nu \rangle \stackrel{a}{\rightarrow} \langle s', \nu' \rangle$ for some $\nu \in \alpha$. In other words, if the zone automaton has the edge $\langle s, \alpha \rangle \stackrel{a}{\rightarrow} \langle s', \alpha'' \rangle$, and α' is the convex hull of α'' then $Z^*(\mathcal{A})$ has the edge $\langle s, \alpha \rangle \stackrel{a}{\rightarrow} \langle s', \alpha' \rangle$.

The transition table $Z^*(\mathcal{A})$ approximates $Z(\mathcal{A})$. The acceptance conditions on $Z(\mathcal{A})$ are translated to the acceptance conditions on $Z^*(\mathcal{A})$. Thus, instead of searching $Z(\mathcal{A})$ we can search for cycles in $Z^*(\mathcal{A})$. Note that searching in the approximate-zone automaton $Z^*(\mathcal{A})$ can be done efficiently manipulating DBMs.

If the language of $Z^*(\mathcal{A})$ is empty then so is the language of $Z(\mathcal{A})$, and so is the language of $\mathcal{A}$ (when $\mathcal{A}$ represents the product of the components together with the complement of the specification, this means that the system satisfies its specification). However, when the language of $Z^*(\mathcal{A})$ is nonempty, we cannot conclude nonemptiness of the language of $Z(\mathcal{A})$. In this case, we may need to perform the search in $Z(\mathcal{A})$. More effective techniques that perform repeated search using only simple zones have been developed. See [Won94] for a variety of approximation techniques for zone automata, and experimental results on its applications.

Iterative Verification

Consider a TBA $\mathcal{A} = \langle \Sigma, \mathrm{S}, \mathrm{S}_0, \mathrm{C}, \mathrm{E} \rangle$. The computational complexity of the verification problem depends upon the number of clocks and magnitudes of constants in clock constraints. The iterative approximation strategy considers TBAs $\mathcal{A}_0$, $\mathcal{A}_1$, $\mathcal{A}_2, \ldots$ and in iteration i, tests the emptiness of $L(\mathcal{A}_i)$. The approximations satisfy the following property:

- Each TBA $\mathcal{A}_i$ is an approximation of $\mathcal{A}$: $L(\mathcal{A}) \subseteq L(\mathcal{A}_i)$.
- Each TBA $\mathcal{A}_i$ has the same state-transition structure as $\mathcal{A}$, but simpler clock constraints than $\mathcal{A}$ (i.e. the constraints on edges of $\mathcal{A}_i$ use less number of clocks, or constants with smaller magnitudes).
- As i increases, $\mathcal{A}_i$ is a better approximation of $\mathcal{A}$: $L(\mathcal{A}_{i+1}) \subset L(\mathcal{A}_i)$.
- The approximations converge in finite number number of iterations: for some i, $L(\mathcal{A}) = L(\mathcal{A}_i)$.

Specifically, $\mathcal{A}_0$ has the same state-transition structure as $\mathcal{A}$ except that every clock constraint of $\mathcal{A}$ is simplified to true. Testing emptiness of $\mathcal{A}_0$ is easy: we simply need to search for a reachable cycle that contains an accepting state. If $L(\mathcal{A}_0)$ is empty then so is $L(\mathcal{A})$, and we are done. If not, then there is a word $\overline{\sigma}$ and an accepting run $\overline{s}$ of $\mathcal{A}_0$ over $\overline{\sigma}$. The next step is to test whether there exists a time sequence $\overline{\tau}$ such that $\mathcal{A}$ has a run over $(\overline{\sigma}, \overline{\tau})$ that follows the state-sequence $\overline{s}$. This problem is computationally easy, and can be solved efficiently in polynomial-time. If there is such a run, then $\overline{\sigma} \in \mathit{Untime}(L(\mathcal{A}))$, and we can infer that $L(\mathcal{A})$ is nonempty. If not, then the algorithm needs to compute the next approximation $\mathcal{A}_1$. The approximation is computed by adding a minimal set of constraints of $\mathcal{A}$ to $\mathcal{A}_0$ so that $\mathcal{A}_1$ has no run corresponding to $\overline{s}$. A variety of heuristics are used for this purpose. The next example illustrates the ideas.

Example 6.2 Consider the TBA $\mathcal{A}$ shown in Figure 6.3 (the alphabet is unary, and 5 is the accepting state). The automaton uses 3 clocks, and the largest constant is 20. Thus, the region automaton has large number of vertices. Observe that $L(\mathcal{A})$ is empty.

The first approximation is $\mathcal{A}_0$. The language $L(\mathcal{A}_0)$ is nonempty, and the accepting run is $\overline{s} = 012345^\omega$. This run $\overline{s}$ is checked against the constraints of $\mathcal{A}$, and the run is found to be inconsistent (i.e. $\mathcal{A}$ cannot follow the run $\overline{s}$). The algorithm then computes a minimal set $C' \subseteq \{x, y, z\}$ of clocks such that thee constraints on the clocks in C' are sufficient for the inconsistency of $\overline{s}$. In this case, $C' = \{y, z\}$. Next, the algorithm attempts to relax the constraints: lower bounds can be decreased, and upper bounds can be increased. In particular, $y < 15$ can be replaced by $y < 20$. Also, it is possible to divide all constants by the greatest common divisor of all the constants. This leads to the approximation $\mathcal{A}_1$. The automaton $\mathcal{A}_1$ has only 2 clocks, and the constants are small. The region automaton of $\mathcal{A}_1$ is used to conclude that $L(\mathcal{A}_1)$ is empty, and this implies $L(\mathcal{A})$ is empty. ■

Thus at each step of the iteration, either we can conclude the emptiness or nonemptiness of $L(\mathcal{A})$, or need to add additional constraints (with relaxed bounds) to obtain a better approximation. For precise details of the iterative scheme, see [AIKY95]. This heuristic is implemented in the tool COSPAN (see [AIKY95] for experimental results). In many cases, the original emptiness question can be answered in a few iterations. Another advantage of the method is that in many cases the algorithm computes the tight bounds that are needed to prove the specification.

7 Discussion

In this chapter, we have shown how to extend the theory of finite automata to incorporate timing, and illustrated its application to verification of real-time systems. Various tools such as COSPAN, HSIS, and KRONOS incorporate timing verification based on these principles. For further details on the implementations and experimental results the reader is referred to [NOSY93, AIKY95, Won94].

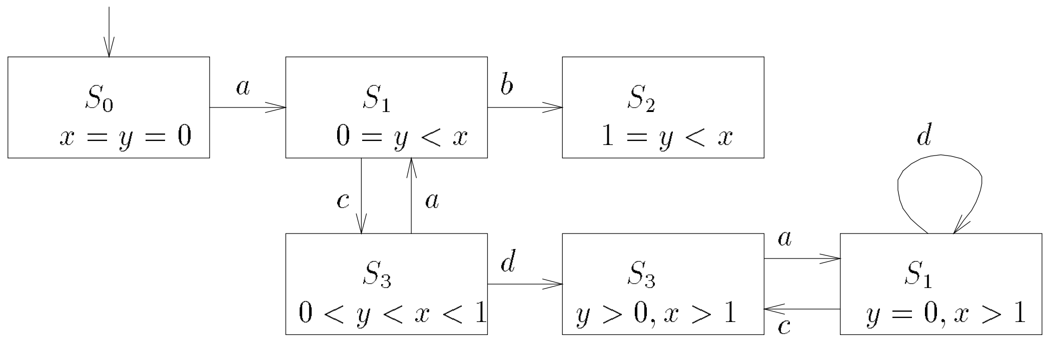


Figure 12: Iterative verification

Related work

There is an extensive literature on verification of timed systems. Examples of formalisms that admit modeling real-time systems include timed Petri nets [Ram74], timed transition systems [Ost90, HMP91], timed I/O automata [LA92], process algebras such timed CSP [RR88] and ATP [NRSV90], and Modecharts [JM87]. The algorithmic techniques developed for timed automata apply to these other models also.

In this paper, we used automata not only to describe the system, but also to write correctness requirements. Alternatively, real-time requirements can be written as formulas of timed temporal logics. Model-checking algorithms for various timed temporal logics have been developed: examples include discrete linear-time logics [JM86, Ost90, AH94], dense linear-time logics [AFH91], discrete branching-time logics [EMSS90, CC94], and dense branching-time logics [ACD93, HNSY94]

We have considered only algorithmic methods for verification that can be fully automated, and apply only to finite-state systems. Real-world problems need decomposition of the given verification problem into subproblems to which the verification algorithms can be applied. This decomposition requires a careful modeling that admits compositional and hierarchical reasoning. Such issues are discussed in, for instance, [AL91, LV92, Sha92].

Hybrid systems

Recently, the model of timed automata has been extended so that continuous variables other clocks, such as temperature and imperfect clocks, can be modeled. *Hybrid automata* are useful in modeling discrete controllers embedded within continuously changing environment. Verifying correctness of hybrid automata is computationally more expensive than of timed automata, but in simple cases, such as the railroad controller, it allows reasoning with parametric bounds. We refer the reader to [ACH+95] for an introduction to hybrid automata, and to [HH95] for an introduction to the verifier HyTech.

References

[ABB+94] A. Aziz, F. Balarin, R. Brayton, S. Cheng, R. Hojati, T. Kam, S. Krishnan, R. Ranjan, A. Sangiovanni-Vincetelli, T. Shiple, V. Singhal, S. Tasiran, and H. Wang. HSIS: A BDD-based environment for formal verification. In *Proc. Design Automation Conference*, 1994.

[ACD93] R. Alur, C. Courcoubetis, and D.L. Dill. Model-checking in dense real-time. *Information and Computation*, 104(1):2–34, 1993.

[ACH+95] R. Alur, C. Courcoubetis, T. Henzinger, P. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. The algorithmic analysis of hybrid systems. *Theoretical Computer Science*, 138:3–34, 1995.

[AD94] R. Alur and D.L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.

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

[AH94] R. Alur and T.A. Henzinger. A really temporal logic. *Journal of the ACM*, 41(1):181–204, 1994.

[AIKY95] R. Alur, A. Itai, R.P. Kurshan, and M. Yannakakis. Timing verification by successive approximation. *Information and Computation*, 118(1):142–157, 1995.

[AL91] M. Abadi and L. Lamport. An old-fashioned recipe for real time. In *Real-Time: Theory in Practice, REX Workshop*, LNCS 600, pages 1–27. Springer-Verlag, 1991.

[CC94] S. Campos and E. Clarke. Real-time symbolic model checking for discrete time models. In *Theories and experiences for real-time system development*, AMAST series in computing, 1994.

[Dil89a] D.L. Dill. Timing assumptions and verification of finite-state concurrent systems. In J. Sifakis, editor, *Automatic Verification Methods for Finite State Systems*, LNCS 407, pages 197–212. Springer–Verlag, 1989.

[Dil89b] D.L. Dill. *Trace Theory for Automatic Hierarchical Verification of Speed-independent Circuits.* ACM Distinguished Dissertation Series. MIT Press, 1989.

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

[HH95] T. Henzinger and P. Ho. Hytech: The cornell hybrid technology tool. Technical report, Cornell University, 1995.

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

[HNSY94] T.A. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. Symbolic model-checking for real-time systems. *Information and Computation*, 111(2):193–244, 1994.

[JM86] F. Jahanian and A.K. Mok. Safety analysis of timing properties in real-time systems. *IEEE Transactions on Software Engineering*, SE–12(9):890–904, 1986.

[JM87] F. Jahanian and A.K. Mok. A graph-theoretic approach for timing analysis and its implementation. *IEEE Transactions on Computers*, C-36(8):961–975, 1987.

[Kur94] R.P. Kurshan. *Computer-aided Verification of Coordinating Processes: the automata-theoretic approach.* Princeton University Press, 1994.

[LA92] N.A. Lynch and H. Attiya. Using mappings to prove timing properties. *Distributed Computing*, 6:121–139, 1992.

[LT87] N.A. Lynch and M. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *Proceedings of the Seventh ACM Symposium on Principles of Distributed Computing*, pages 137–151, 1987.

[LV92] N. Lynch and F. Vaandrager. Action transducers and timed automata. In *Proceedings of the Third Conference on Concurrency Theory CONCUR '92*, LNCS 630, pages 436–455. Springer-Verlag, 1992.

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

[NOSY93] X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. Recent results on the description and analysis of timed systems. Technical report, VERIMAG, Grenoble, France, 1993.

[NRSV90] X. Nicollin, J.-Luc Richier, J. Sifakis, and J. Voiron. ATP: an algebra for timed processes. In *Proceedings of the IFIP TC2 Working Conference on Programming Concepts and Methods, Sea of Galilee, Israel*, 1990.

[Ost90] J. Ostroff. *Temporal Logic of Real-time Systems.* Research Studies Press, 1990.

[Ram74] C. Ramchandani. Analysis of asynchronous concurrent systems by Petri nets. Technical Report MAC TR-120, Massachusetts Institute of Technology, 1974.

[RR88] G.M. Reed and A.W. Roscoe. A timed model for communicating sequential processes. *Theoretical Computer Science*, 58:249–261, 1988.

[Sha92] A.U. Shankar. A simple assertional proof system for real-time systems. In *Proceedings of the 13th IEEE Real-Time Systems Symposium*, pages 167–176, 1992.

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

[Won94] H. Wong-Toi. *Symbolic approximations for verifying real-time systems.* PhD thesis, Stanford University, 1994.