3

VERIFICATION OF TIMED SYSTEMS

Time has no divisions to mark its passage, there is never a thunderstorm or blare of trumpets to announce the beginning of a new month or year.

—Thomas Mann - The Magic Mountain, 1924

Time is such a simple, almost primitive idea. It is just a means of material differentiation, a way of uniting us all; for in our external, material lives we value the synchronized efforts of individual people.

—Andrei Tarkovsky - Time Within Time: The Diaries, 1989

Summary

This chapter reviews the previous work on the formal verification of timed systems. The attention is focused in those modeling, specification and analysis alternatives more widely used. For a deeper insight, the interested reader is addressed to the provided references.

The correctness of timed systems depends on their timing properties. As a consequence, quantitative time information is essential for their analysis. The main paradigms for the incorporation of quantitative timing information to the system's models are first reviewed. A widely used approach is that of *continuous-time* for which the most popular representative, the *timed automata* modeling formalism, is analyzed.

The concept of timed temporal logic is introduced as an appropriate mechanism for the specification of timing-related properties. Several alternatives are briefly commented.

Since most verification methods rely on the analysis of the timed state space of the system, the reachability problem on timed systems is analyzed.

Finally, a brief review is provided of the approaches that use Petri nets for the timing analysis and the verification of timed systems.

3.1 Introduction

Three major ingredients are required to accomplish the verification task successfully:

- A model of the system, which is capable to capture those behaviors of the system that are relevant for the verification.
- A specification language, expressive enough to state the properties of interest.
- A verification methodology, which is suitable to be used in conjunction with the modeling and the specification formalisms.

Several modeling formalisms have been already introduced in Chapter 2, namely those used for the research presented in this work. Among the formalisms used by other researchers, *timed automata* [AD94] deserve particular attention since they are the model of choice in many verification methodologies.

Regarding the specification formalisms, there exists a wide spectrum which can be roughly divided into two categories: logic-based and automata-based approaches.

In the logic-based approach, originally introduced in [Pnu77], the properties under verification are stated as formulas using a *temporal logic* (see [Eme90] for an overview). Despite of a number of derivatives, two main families of temporal logics exist: *linear temporal logic* (LTL) pioneered by [Pnu77, OL82], and *computational tree logic* (CTL) pioneered by [BAPM81, EC82]. Provided the specification in the form of a set of formulas of the logic, the state space of the system is explored checking whether each formula is satisfied in all possible behaviors of the system. The resulting verification methods, *i.e.* the so-called *model-checking* methods, were pioneered by [LP85, CES86, BCM$^+$92, GW91] among others.

In automata-based approaches, the same formalism is used for describing both the system and the specification containing the properties of interest. Then, the verification consists in showing that all behaviors of the system are also part of the specification. This is often achieved by showing an *implementation relation* between the system and its specification in terms of *language containment* [Tho81], *simulation* [DHWT91] or an *homomorphism* [Kur94], for example.

The following sections review the major approaches used for the modeling, the specification and the analysis of timed systems. First, several alternatives for the representation of quantitative timing information are reviewed. Next, timed automata are presented as the most commonly used model for timed systems, whereas timed temporal logic is introduced as a specification formalism used to state properties in which a quantitative notion of time is required. Then, some strategies for the representation of the system's timed state space are analyzed. Finally, some approaches that use Petri nets for the verification of timed systems are reviewed.

3.2 Quantitative timing information

Most of the early works in formal verification were only focused at verifying the functional properties of systems (see Section 1.2.1). In those works, time was present into the models (mostly finite automata) and into the specifications (mostly temporal logic), only as a qualitative notion. In such cases, properties only assert, for example, that a certain condition is always true or that a expected response of the system eventually occurs. While qualitative modeling of time allows the efficient verification of certain properties, it is not satisfactory for verifying the correctness of systems that depend crucially on timing: combinatorial circuits must meet some given clock requirements, embedded controllers must respond to interrupts within some time interval, etc. As a more precise example, consider the statement "*trigger the alarm upon detection of an intruder*" referred to a security system. This temporal sequencing carries no quantitative information on the delay allowable between the detection and the alarm action. Hence, it is not possible to directly model the triggering of the alarm "*less that 5 seconds after detecting the intruder*".

For those systems whose correctness depends on a proper timing, often called *time-critical* or *real-time* systems, a quantitative notion of time must be incorporated both into the system models and also into the specification formalisms. The way time is represented has a crucial impact on the size of the resulting timed state space. Three main approaches exist for that purpose: *discrete-time*, *fictitious-clock* and *continuous-time*.

Formalisms based on the *discrete-time* notion (*e.g.* [AK83, JM86, EMSS90, BMPY97]) map time onto the integer domain. This approach is appropriate to describe the behavior of *synchronous* systems where all components are driven by a common global clock. However, in order to model *asynchronous* systems they require to discretize time by choosing some fixed *time quantum*, so that the separation of two events in the timed domain is always a multiple of such quantum. The main advantage of this approach is that the timing analysis and timed state exploration techniques are generally simpler than their counterparts for continuous-time. However, the main drawback is that determining the time quantum *a priori* may not be easy and therefore may compromise the accuracy of the resulting model. In this sense, in [BS91] it is shown that the reachability problem for asynchronous circuits with bounded delays cannot be solved correctly using the discrete-time model. Also, the choice of a sufficiently small time quantum to model the system accurately enough, may blow up the timed state space so that verification becomes a no longer feasible task. Figure 3.1 (a) illustrates the concept of discrete-time, where events can only occur at instants multiple of the fixed time quantum.

The *fictitious-clock* approach introduces a special *tick* event into the model (*e.g.* [AH89, Bur89, Ost90, HLP90]). Thus, time is understood as a global state variable that ranges over the domain of natural numbers, and is incremented with every *tick* event. Generally, this paradigm allows arbitrarily many events of any process between two successive *tick*

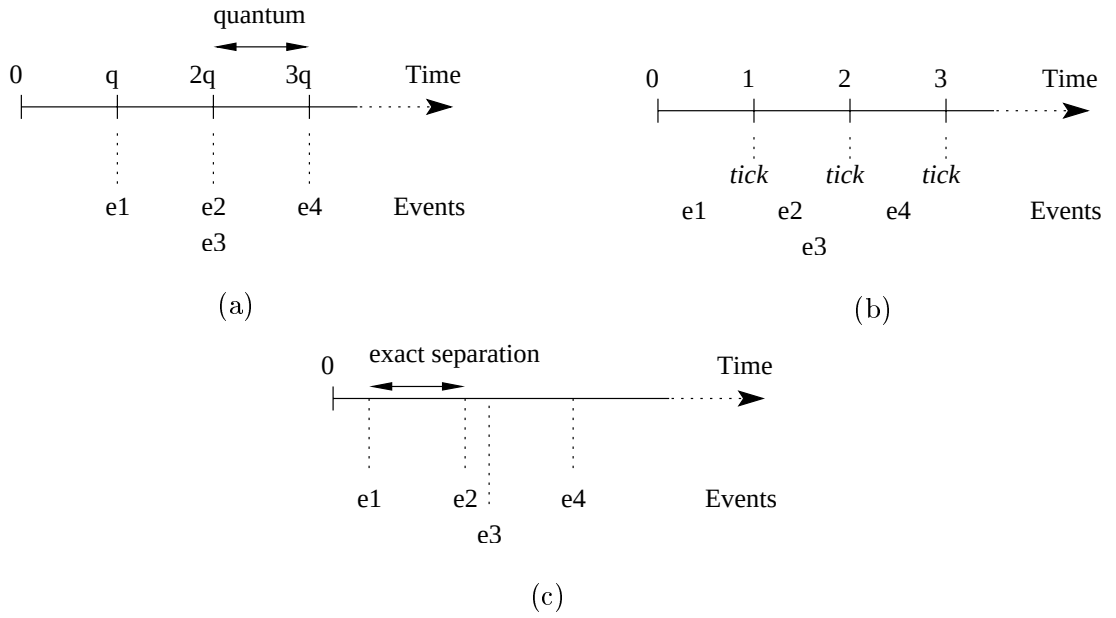

Figure 3.1 Three representations of time: (a) discrete-time, (b) fictitious-clock and (c) continuous-time.

events. The timing delay between two events is measured by counting the number of *ticks* between them. When it is required that there be k *ticks* between two events, it can only be inferred that the actual delay between them is at least $k-1$ time units and at most $k+1$ time units. Therefore, it is impossible to determine precisely some typical and simple requirements on the delays between events, *e.g.* "*the delay between the detection and the alarm equals 2 seconds*". In general, the models based in the fictitious-clock approach require a somewhat cumbersome encoding mechanism to measure time intervals. This reduces the readability of the model and makes modifying the model a tricky and prone to errors task. As a result, ensuring that the model obtained is a good characterization of the actual system is often very difficult. Finally, notice that the discrete-time approach can be seen as a special case of the fictitious-clock approach where the events occur only in lock-step with the *ticks*. Figure 3.1 (b) illustrates the fictitious-clock approach. Although events may occur at any time, the precise occurrence instant can only be approximated to be in between of two *tick* events.

The third approach for the modeling of real-time behavior, models time more realistically as a continuous magnitude. Some examples of the use of the so-called *continuous-time* or *dense-time* can be found in [Dil89b, Koy90, ACD90, HMP92a, HNSY92, YSSC93, RM94, LPY95, SB97]. These approaches associate a non-negative real value to each event of the system, and therefore to each reachable state. Continuous-time differs from the

other time models because the exact bounds on the actual delays between the events can be expressed. Moreover, the use of continuous-time allows a more precise modeling of analog or asynchronous systems, as well as systems that operate at different clock frequencies. Figure 3.1 (c) illustrates the continuous-time notion.

Since this approach does not rely on the use of a discretization constant, one possible drawback of using the reals as time domain is the added complexity. However, it has been shown that with appropriate techniques (*e.g.* [AIKY92, HMP92b, ABH+97, TKY+98]), the analysis of continuous-time models does not increase in complexity, if compared to the discrete-time counterparts. The main idea behind such techniques consists in breaking the infinite continuous timed state space into equivalence classes, such that all states in the same class lead to the same behavior and can be analyzed together.

3.3 Timed automata

With the wide adoption of the continuous-time paradigm, the *timed automata* framework, pioneered by [Dil89b, ACD90], has become one of the most popular choices to incorporate quantitative time into the system's models. Several timing verification tools use this formalism as their basis: COSPAN [AK95], KRONOS [Yov97], UPAAL [BLL+95], MOCHA [AHM+98], among others.

A timed automaton is a classical finite automaton augmented with a finite set of real-valued *clocks*. That is, a timed automaton is built from two elements: a finite automaton which describes the (control) states or *locations*, and the transitions between them; and a set of clocks used to specify the quantitative time constraints. Transitions are assumed to happen instantaneously, whereas time can elapse when the automaton is at a given state. In the initial location all clock values are set to zero. Then, the clocks evolve at a uniform rate taking non-negative real values. At any instant, *reading* a clock tells how much time has elapsed since the last time the clock was reset.

Besides the source and target locations, a *transition* is formed by other three elements: a *guard*, also called *clock constraint* or *firing condition*, such that the transition cannot be taken unless the current values of the clocks satisfy the guard; a *label*, or action name; and a set of clocks that must be reset after the transition has been taken.

A clock constraint is often associated to each location of the automaton. This type of constraint, called the *invariant* of the location, forces that time can elapse in the location only as long as the invariant is still satisfied.

In order to provide a formal definition of a timed automaton, clock constraints must be precisely defined first. Let X be a set of real clocks, the set $\Phi(X)$ of clock constraints φ allowable as location invariants and enabling conditions, is defined as:

- All inequalities of the form $x < c$, $x \leq c$, $c < x$, $c \leq x$ are in $\Phi(X)$ where x is a clock and c is a non-negative real number.

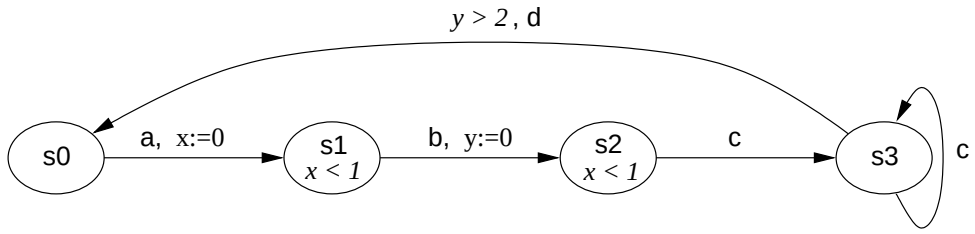

Figure 3.2 An example of timed automaton.

- If φ_1 and φ_2 are in $\Phi(X)$ then the constraint $\varphi_1 \wedge \varphi_2$ is in $\Phi(X)$.

Notice that if X contains k clocks, then each clock constraint delimits a convex region in a k-dimensional Euclidean space. This observation provides a way for representing the timed state space of a timed automaton (see Section 3.5).

The formal definition of a timed automaton follows:

DEFINITION 3.1 (TIMED AUTOMATA) *[AD94]*

A timed automaton *is a 6-tuple* $A = \langle \Sigma, S, S_o, X, I, T \rangle$ *such that:* Σ *is a finite* alphabet*;* S *is a finite set of* locations *(states);* $S_o \subseteq S$ *is a set of* initial locations*;* X *is a set of* clocks*;* $I : S \longrightarrow \Phi(X)$ *is the* location invariant*; and* $T \subseteq S \times \Sigma \times \Phi(X) \times 2^X \times S$ *is a set of* transitions.

The 5-tuple $\langle \mathsf{s}, \mathsf{a}, \varphi, \lambda, \mathsf{s}' \rangle \in T$ *is a transition from location* s *to location* s' *corresponding to the action labeled as* a*. The clock constraint* φ *specifies when the transition is enabled, and* $\lambda \subseteq X$ *is the set of clocks that are reset when the transition is taken.*

■ 3.1

EXAMPLE 3.1 *Consider the timed automaton in Figure 3.2.*

When the system switches from the initial location s_0 *to location* s_1 *by the action* a*, the clock* x *is reset to* 0*. Therefore, in all the other locations, the value of clock* x *shows the time elapsed since the last occurrence of action* a*.*

The invariant $x < 1$ *associated to locations* s_1 *and* s_2*, ensures that the* c*-labeled switch from location* s_2 *to* s_3 *happens within time* 1 *of the preceding* a*. Resetting the other independent clock* y *together with the* b*-labeled switch from* s_1 *to* s_2*, and checking its value on the* d*-labeled switch from* s_3 *to* s_0 *ensures that the delay between* b *and the following* d *is always greater that* 2*.*

Notice that locations s_0 *and* s_3 *have no invariant constraint. This means that the system can spend an arbitrary amount of time in such locations. As a consequence, there is no guarantee that the* a*-labeled switch from* s_0*, or the* d*-labeled switch from* s_3 *are taken at some time instant.*

■ 3.1

The semantics of a timed automaton A is defined by associating a transition system, $\mathcal{T}(A)$, to it. At any time, the *configuration* or global state of the system modeled by the timed automaton is given by a location, s, of the automaton and a clock interpretation, v, that assigns a real value to each clock. Thus, a configuration is a pair (s, v) where $\mathsf{s} \in S$ and $v : X \longrightarrow \mathbb{R}^+$. The set of *initial configurations* is given by the set $\{(\mathsf{s}, v) \mid \mathsf{s} \in S_o \wedge \forall x \in X\ [v(x) = 0]\}$, *i.e.* the set of initial locations in which all the clocks are set to 0.

The system changes from one configuration to another by means of two types of transitions:

- *Delay transition*: which lets a time delay $\delta \in \mathbb{R}$ to elapse, *i.e.* increasing the value of all clocks by δ. Then, the system moves from configuration (s, v) to configuration (s, v'), written as $(\mathsf{s}, v) \xrightarrow{\delta} (\mathsf{s}, v')$, where $\forall x \in X\ v'(x) = v(x) + \delta$.

- *Action transition*: which executes an actual transition $\langle \mathsf{s}, \mathsf{a}, \varphi, \lambda, \mathsf{s}' \rangle \in T$ of the automaton. This is written as $(\mathsf{s}, v) \xrightarrow{\mathsf{a}} (\mathsf{s}', v')$, such that v satisfies the guard φ and $v' = v[\lambda := 0]$.

Thus, the timed state space of a timed automaton can be seen as an *infinite* transition system $\mathcal{T}(A) = \langle Q, \Sigma \cup \mathbb{R}, R, Q_o \rangle$, where: Q and Q_o are the set of configurations and the initial configurations, respectively; the original alphabet Σ is augmented with the real numbers to include the delay transitions; and R is the *transition relation* obtained by combining the delay and the action transitions.

EXAMPLE 3.1 (CONT.) *Let the timed automaton in Figure 3.2 be called A.*

The state-space of $\mathcal{T}(A)$ is given by $Q \subseteq \{\mathsf{s}_0, \mathsf{s}_1, \mathsf{s}_2, \mathsf{s}_3\} \times \mathbb{R}^2$. A sequence of possible transitions is, for example:

$$(\mathsf{s}_0, 0, 0) \xrightarrow{1.2} (\mathsf{s}_0, 1.2, 1.2) \xrightarrow{\mathsf{a}} (\mathsf{s}_1, 0, 1.2) \xrightarrow{0.7} (\mathsf{s}_1, 0.7, 1.9) \xrightarrow{\mathsf{b}} (\mathsf{s}_2, 0.7, 0) \xrightarrow{0.1} (\mathsf{s}_2, 0.8, 0.1) \xrightarrow{0.1} \cdots$$

where the numbers at each configuration are (from left to right) the values of the clocks x and y. ■ 3.1

Solving the reachability problem for a timed automaton is a nontrivial task since the number of potential configurations is infinite. In order to solve the task, finite representations of the infinite state space are required (see Section 3.5). However, even using such representations, the state explosion problem often limits the practical applicability of the algorithms and tools that rely on the reachability set. More precisely, the problem is PSPACE-hard [AD94]. Moreover, in [CY91] it was proved that both sources of complexity, the number of clocks and the magnitude of the constraints yield to PSPACE-hardness independently of each other.

3.4 Timed specifications

Given the model of a timed system, the next step is to state and then verify properties of such system. Some of the properties are just temporal *e.g.* "*when the gate is opened the alarm is always triggered*". Other properties involve quantitative delay information, *e.g.* "*the alarm is triggered if the gate keeps opened more than 30 seconds*". For the first type of properties, *temporal logic* may be a good choice. For the second type of properties it is more suitable to use a *timed logic*, which is an extension of a temporal logic with primitives expressing conditions on the duration of events.

3.4.1 Temporal logic

Temporal logic is a form of logic specifically tailored to state and reason about the notion of *order* in time, using a simple and clear notation. Time is represented as an implicit magnitude by means of constructs that mimic the time adverbs of natural language (*e.g.* "always", "until", etc.). The remaining of this section assumes the reader has some familiarity with temporal logic. A good survey about the theoretical foundations behind temporal logic can be found in [Eme90].

Each type of temporal logic offers its own temporal operators which can deal with time according to two basic paradigms. A *linear time model* assumes that for each time instance there exists exactly one successor time point. This model is particularly well suited for *physical time*. The resulting *linear temporal logic* (LTL) was originally pioneered by [Pnu77, OL82]. Conversely, a *branching time model* allows several successors of each time instance. This model is appropriate to capture *computations*, where different execution traces can be selected at a certain step of the ongoing calculation. Hence, time is modeled by tree-like structures where the different possible successor computation paths are chosen non-deterministically. The so-called *computational tree logic* (CTL) [BAPM81, EC82], follows the branching time paradigm.

In order to establish a comparison between both paradigms, the more expressive logic CTL* [EH86] is briefly introduced first. Despite of the boolean propositions used as the atomic formulas of the logic, CTL* contains both LTL and CTL, thus provides operators for linear and branching time.

Linear time operators make statements about a single *computation path* (a sequence of states) which starts in the actual state. Thus, the G (*always*) operator indicates that a formula must hold for all successor states on the path; the the F (*sometimes*) operator indicates that a formula must hold in some successor state (without telling which one) on the path; the X (*next*) operator indicates that a formula must hold in the immediate successor state on the path; and the U (*until*) operator which combines two formulas, where the first one must hold along the path until the second formula becomes true.

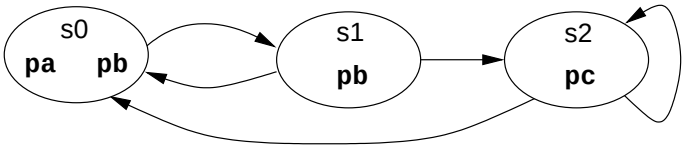

Figure 3.3 A simple automaton with atomic propositions.

The previous operators deal with a single execution path from a given state. Branching time provides two quantifiers over sets of executions which allow to express formulas about the many possible executions starting from a given state. Thus, the quantifiers A and E indicate respectively that, *for all* paths out of the current state a given formula holds, and that there *exists* at least one path where the formula holds. It is important not to confuse A and G : the formula $\mathsf{A}\phi$ states that all the possible executions from the current state satisfy ϕ, whereas $\mathsf{G}\phi$ indicates that ϕ holds at every state of a particular execution being considered.

The aforementioned constructs are summarized in the following grammar for CTL* :

$$\begin{array}{lll} \phi, \psi ::= & P_1 \mid P_2 \mid \ldots & \text{(atomic propositions)} \\ & \mid \neg\phi \mid \phi \wedge \psi \mid \phi \Rightarrow \psi \mid \ldots & \text{(boolean operators)} \\ & \mid \mathsf{F}\phi \mid \mathsf{G}\phi \mid \mathsf{X}\phi \mid \phi\mathsf{U}\psi & \text{(temporal operators)} \\ & \mid \mathsf{A}\phi \mid \mathsf{E}\phi & \text{(path quantifiers)} \end{array}$$

EXAMPLE 3.2 *Let us consider some properties related to the automaton in Figure 3.3 and how they can be expressed using temporal logic. The automaton consists of three states and three atomic propositions simply called* pa, pb *and* pc.

In any execution of the automaton, either the proposition pa *holds infinitely often or the automaton ultimately remains forever in state* s_2 *where* pc *holds. This can be expressed with the formula:* $\mathsf{GF\ pa} \vee \mathsf{FG\ pc}$.

Notice that there is one execution from s_0 *which does not satisfy the formula* $\mathsf{pb\ U\ pc}$, i.e. *that execution in which states* s_0 *and* s_1 *alternate forever.*

Notice also that all executions out of s_0 *visit state* s_1. *Since in one step from* s_1 *a state satisfying* pc *is reachable, any execution out of* s_0 *satisfies the formula* $\mathsf{FEX\ pc}$. *Observe that the* E *quantifier is important in this formula, since the execution in which* s_0 *and* s_1 *alternate does not satisfy* $\mathsf{FX\ pc}$. *Thanks to the* E *quantifier the executions in which* s_2 *follows* s_1 *are also covered.* ■ 3.2

Although the origins of LTL and CTL differ, both can be seen as subsets of the more expressive logic CTL*. LTL is obtained from CTL* by subtracting the A and E path quantifiers. Thus, a formula in LTL cannot cover the possible alternative executions

which split at every state. Similarly, CTL is the subset of CTL* in which the use of a temporal operator must be under the immediate scope of a path quantifier. The basic valid combinations are: AF, EF, AG, EG, AX, EX, A_U_ and E_U_ .

From a syntactical point of view, there are LTL formulas that cannot be expressed in CTL, and vice versa. Moreover, there are CTL* formulas which cannot be expressed in neither of both. The analysis of the differences between LTL and CTL from a semantical point of view requires a more detailed study of the types of properties that can be expressed with each (see [BBF+01] for more details).

Reachability properties state that some particular situation *can be reached* by the system. CTL to models reachability properties in a natural way by means of the EF construct. Thus, $\mathsf{EF}\phi$ can be read as "*there exists a path, from the current state, along which some state satisfies* ϕ". In order to state a reachability property from all reachable states, the AG and EF constructs must be nested, *e.g.* $\mathsf{AG}(\mathsf{EF}\phi)$. Conversely, and since LTL implicitly quantifies for all executions of the system, only the negation of reachability can be expressed in LTL. That is, "*something is never reachable*", *e.g.* $\mathsf{G}(\neg\phi)$. However, this type of property is often seen as a safety property.

Safety properties express that, under certain conditions, something *never occurs*. Safety can be expressed naturally in both LTL and CTL, by means of the expressions $\mathsf{G}\phi$ and $\mathsf{AG}\phi$, respectively.

Liveness properties express that, under certain conditions, something *will ultimately occur*. Despite of the discussion of whether liveness properties are useful in practice (see [BBF+01]) it is not easy to formally capture such notion. Two types of liveness properties are often distinguished: simple liveness or *progress*, and repetitive liveness or *fairness*. Progress is generally easier to formalize, as in the following typical example. The property "*any request will ultimately be satisfied*" is expressed as $\mathsf{AG}(\texttt{req} \Rightarrow \mathsf{AF}\texttt{sat})$ in CTL and as $\mathsf{G}(\texttt{req} \Rightarrow \texttt{Fsat})$ in LTL. Regarding *fairness*, in [EH86] it is shown that in contrast to LTL fairness properties cannot be expressed in CTL.

Deadlock-freeness is a special property relevant in systems which are supposed to operate indefinitely. Although deadlock-freeness is often seen as a safety property ("*something undesirable will never happen*"), theoretically it is not clear if it is actually a liveness property. Anyway, deadlock-freeness can be expressed in CTL as AG EX `true`, *i.e.* "*whatever the state reached is* (AG), *there exists an immediate successor state* (EX `true`)".

Finally, remark that both linear and branching time have their strengths and weaknesses. Thus the resulting logics, LTL and CTL (and their derivatives) are better suited for a particular subset of properties and also for a particular class of systems. Moreover, the verification methodology is often tailored to a specific logic thus gaining in aspects like efficiency, for example. See [Kro99, BBF+01] for more details about this discussion.

3.4.2 Timed temporal logic

In order to state and verify timing properties, the simplest way is to express them in terms of the reachability (or non-reachability) of some sets of configurations of the automaton. For more complicated properties *observer automata* can be used. For example, given a property ϕ and a timed automaton A, a new automaton A_ϕ is built and synchronized with A. Then, verifying ϕ is reduced to testing reachability of some particular states in the resulting composed automaton A_ϕ.

Another possibility is to use a *timed temporal logic*, which consists in extending with timing constraints the operators of temporal logic. The timing constraints are often expressed in terms of one-sided inequalities or time intervals, and may take integer, rational or real values. Thus, for example the formula $\mathsf{EF}_{<5}\ \phi$ indicates that "*there exists a state satisfying* ϕ *along some execution within* 5 *time units*". Similarly, the formula $\mathsf{F}_{[5,10]}\ \phi$ indicates that "*some state in which* ϕ *holds is actually reachable, after a minimum of* 5 *time units, and never later than* 10 *time units*". Also, the formula $\phi \mathsf{U}_{<2} \psi$ states that proposition ϕ holds until proposition ψ becomes true, and that ψ will become true within two time units.

To provide an example, the grammar of the timed version of CTL (TCTL) [Koy90] is the following:

$$
\begin{array}{lll}
\phi, \psi ::= & P_1 \mid P_2 \mid \ldots & \text{(atomic propositions)} \\
& \mid \neg\phi \mid \phi \ \wedge\ \psi \mid \phi \ \Rightarrow\ \psi \mid \ldots & \text{(boolean operators)} \\
& \mid \mathsf{EF}_{(\sim k)}\phi \mid \mathsf{EG}_{(\sim k)}\phi \mid \mathsf{E}\phi\mathsf{U}_{(\sim k)}\psi & \text{(temporal operators)} \\
& \mid \mathsf{AF}_{(\sim k)}\phi \mid \mathsf{AG}_{(\sim k)}\phi \mid \mathsf{A}\phi\mathsf{U}_{(\sim k)}\psi &
\end{array}
$$

where $\sim$ is a comparison operator from the set $\{<, \leq, =, \geq, >\}$ and k is a rational number.

A wide range of timed temporal logics have been developed along the years by extending in several ways the original LTL and CTL. Corresponding verification algorithms have been also developed, mostly under the paradigm of model-checking.

As derivatives of LTL, the following remarkable examples can be cited [Hen98]: TPTL [AH89] and MTL [Koy90] whose formulas can be verified in exponential time if discrete-time is assumed, but are undecidable if continuous-time is chosen; MITL [AFH91] which can be verified in exponential time regardless of the time paradigm; and ECL [HRS98] which can be always verified in polynomial time. For discrete-time, all these logics are equally expressive. For continuous-time, TPTL is more expressive than MTL, which in turn is more expressive than MITL and ECL.

As derivatives of CTL, RTCTL (real-time CTL) [EMSS90] and TCTL (timed CTL) [Koy90, ACD93] are the most common representatives. Both use the continuous-time domain, are very similar syntactically and semantically, and their verification has PSPACE-complexity. TCTL, for example is supported by the verification tool KRONOS [Yov97].

3.5 Verification of timed systems

Most approaches for the verification of timed systems rely on the construction of the timed reachability space. However, the number of timed states is infinite (in fact uncountable). For example, in the case of timed automata such infiniteness has two sources: the clock values are potentially unbounded, and even when they are restricted to a bounded interval, the set of real valuations is dense. Therefore, typical model-checking algorithms are no longer feasible. In order to overcome such complexity, finite representations of the timed state space must be provided.

Although the remaining of this section refers to timed automata, the presented techniques are generally applicable to the reachability problem in timed systems.

3.5.1 Clock regions

The main idea to overcome the aforementioned complexity is the use of *clock regions* [ACD90] which we introduce intuitively as follows. Consider two configurations (s, v) and (s, v') of a timed automaton, where the clock valuations v and v' are *very close*. Assume, for example, that there is a single clock x and that $v(x) = 1.2347$ and $v'(x) = 1.235$. Given a certain notion of closeness for configurations (see below), the automata will behave in roughly the same way from either of both configurations, and hence the same properties will be satisfied.

If the the clock constraints only contain integer numbers, an equivalence relation [ACD90] can be defined on the space of configurations, that equates two configurations if: they correspond to the same location, they agree on the integral part of the clock valuations, and they agree on the ordering of the fractional part of the clock valuations. The integral parts of the clocks are needed to decide if 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. Although the integral part of a clock x can get arbitrarily large values, if x is never compared with a constant greater that c_x, the actual value of x beyond c_x does not affect the behavior of the automaton.

More formally, let $v(x) \in \mathbb{R}^+$ be the valuation of a clock, $\lfloor v(x) \rfloor$ denotes the integral part of the valuation whereas $fr(v(x))$ denotes its fractional part, such that $v(x) = \lfloor v(x) \rfloor + fr(v(x))$. The *region equivalence* $v_1 \cong v_2$ for two clock valuations v_1 and v_2 is defined by the following conditions:

- $\forall\, x \in X$, either $\lfloor v_1(x) \rfloor = \lfloor v_2(x) \rfloor$ or, $\lfloor v_1(x) \rfloor > c_x$ and $\lfloor v_2(x) \rfloor > c_x$
- $\forall\, x, y \in X$ with $v_1(x) \leq c_x$ and $v_1(y) \leq c_y$, $fr(v_1(x)) \leq fr(v_1(y))$ iff $fr(v_2(x)) \leq fr(v_2(y))$.
- $\forall\, x \in X$ with $v_1(x) \leq c_x$, $fr(v_1(x)) = 0$ iff $fr(v_2(x)) = 0$.

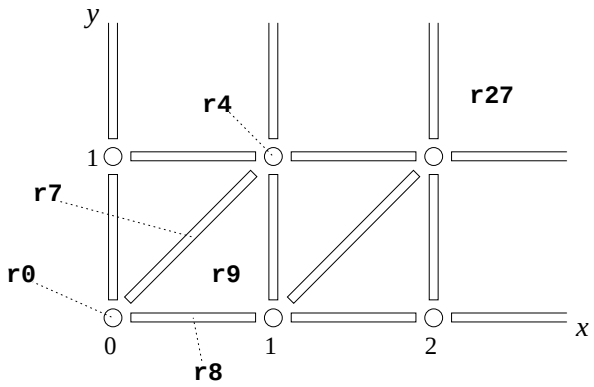

Figure 3.4 Regions for two clocks x and y, with constraints $x \sim k$ ($k \in \{0,1,2\}$) and $y \sim k$ ($k \in \{0,1\}$).

Then, a *clock region* is an equivalence class of clock valuations induced by $\cong$. Each region can be characterized by the finite set of constraints it satisfies. For example, given a clock valuation $v(x) = 0.5$ and $v(y) = 0.8$ for clock x and y, every clock valuation in the clock region for v, denoted $[v]$, satisfies the constraint $0 < x < y < 1$. The following example illustrates these ideas more intuitively.

Example 3.3 *Figure 3.4 [Kro99] shows the set of possible clock regions for two clocks x and y. Only constraints of the form $x \sim k$ with $k \in \{0,1,2\}$, and $y \sim k$ with $k \in \{0,1\}$ have been considered. Recall that $\sim \in \{<, \leq, =, \geq, >\}$.*

The example contains 28 clock regions. Some of them correspond to corner points, *like* `r0`, *characterized by the constraint $[x = y = 0]$. Other regions are* open surfaces *in the plane, like* `r9` *characterized by $[0 < y < x < 1]$, or* `r27` *characterized by $[x > 2 \wedge y > 1]$. Finally, the other regions are* open segments, *like* `r7`, *characterized by $[0 < x = y < 1]$.*

The system starts in `r0` *and as time passes, the clocks increase their values simultaneously. Thus,* `r7` *is visited next, then* `r4`, *etc. If instead of letting time to elapse, a transition that resets some of the clocks is performed, a region on the axes is reached. For example, the reset of clock y while in* `r7` *leads to* `r8`. ■ 3.3

Although the set of clock regions is finite, its cardinality grows exponentially with the number of clocks: for n clocks with constraints in which every constant k is upper bounded by K, the number of regions is $O(n!K^n)$. As a consequence, whereas determining the truth of a CTL formula has linear complexity, the problem is PSPACE-complete for a timed automaton and a TCTL formula [ACD93]. Therefore, efficient representations for handling sets of regions must be devised.

3.5.2 Region automata

The equivalence relation $\cong$ over the clock valuations can be extended over the set of possible configurations of the timed automaton. Thus, two configurations are equivalent, *i.e.* $(\mathsf{s}_1, v_1) \cong (\mathsf{s}_2, v_2)$ iff $\mathsf{s}_1 = \mathsf{s}_2$ and $v_1 \cong v_2$. The resulting equivalence classes of configurations of a timed automaton A, are captured by the so-called *region automaton* [AD94], denoted by $\mathcal{R}(A)$. A state in $\mathcal{R}(A)$ is of the form (s, α) where s is a location of A and α is a clock region.

The interpretation is that whenever the configuration in A is (s, v), the state of $\mathcal{R}(A)$ is $(\mathsf{s}, [v])$. Thus, the initial states of $\mathcal{R}(A)$ are of the form $(\mathsf{s}_o, [v_o])$ where $\mathsf{s}_o \in S_o$ and $\forall\, x \in X\ v_o(x) = 0$. Also, there is an edge $(\mathsf{s}, \alpha) \xrightarrow{\mathsf{a}} (\mathsf{s}', \alpha')$ in $\mathcal{R}(A)$ iff $(\mathsf{s}, v) \xrightarrow{\mathsf{a}} (\mathsf{s}', v')$ in A for some $v \in \alpha$ and $v' \in \alpha'$.

EXAMPLE 3.4 *Consider the timed automaton and its corresponding region automaton shown in Figure 3.5. Only the regions reachable from the initial region* $(\mathsf{s}_0, [x = y = 0])$ *are shown. Notice that the timing constraints cause that the switch from* s_2 *to* s_3 *is never taken. The only reachable region for location* s_2 *satisfies the clock constraint* $[1 = y < x]$*. This region has no outgoing edges because, in order for event* c *to happen, the constraint* $[x < 1]$ *must hold, and that is not possible.* ■ 3.4

3.5.3 Zone automata

Region automata can be easily simplified by collapsing groups of regions into *convex geometric regions* or *clock zones* [BD91, ACD+92, AD94]. For example, in the region automaton of Figure 3.5 (b), there are three regions for location s_1 with associated clock regions $[y = 0 < x < 1]$, $[y = 0, x = 1]$ and $[y = 0, x > 1]$. These regions could be collapsed to obtain the union $[y = 0 < x]$, for example. More precisely, a clock zone is formed by a conjunction of clock constraints each of which puts a lower or upper bound on a clock or a difference of two clocks. Given a timed automaton A, its zone automaton $\mathcal{Z}(A)$ can be obtained in a similar way as for the region automaton.

EXAMPLE 3.5 *Figure 3.6 depicts the zone automaton for the timed automaton in Figure 3.5. Notice that unlike the region automaton, in the zone automaton each vertex has at most one successor per symbol. Also, the number of vertexes of* $\mathcal{Z}(A)$ *is less than those in* $\mathcal{R}(A)$*.* ■ 3.5

Theoretically, in the worst case, the number of zones is exponential with respect to the number of regions, therefore the zone automaton may be exponentially bigger than the region automaton. However, in most practical cases, the zone automaton has less reachable vertexes and provides an improvement in performance. The reason is that, while the number of clock regions depends on the magnitudes of the constants used by the clock constraints, the number of zone regions is relatively insensitive to such fact.

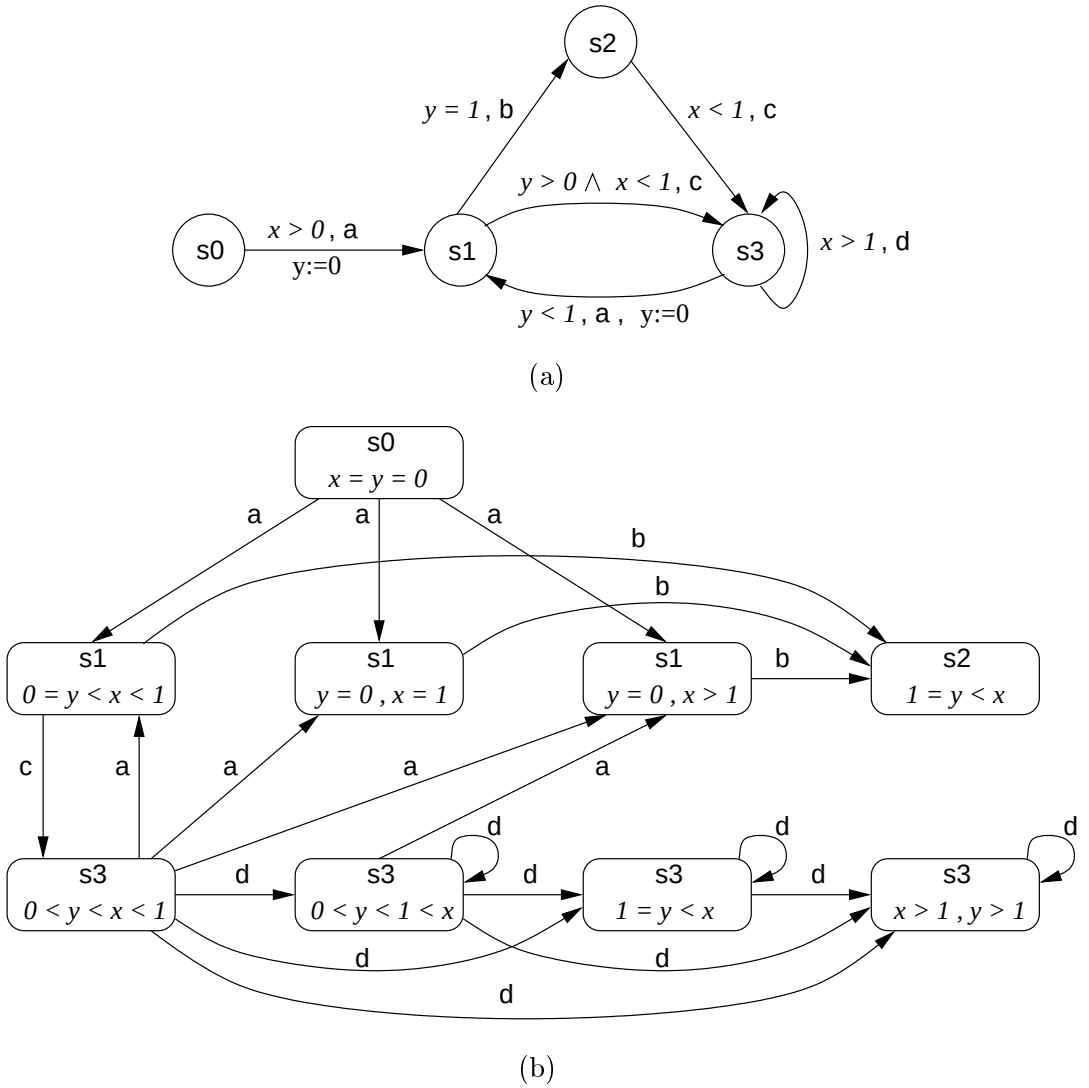

Figure 3.5 Timed automaton (a) and corresponding region automaton (b).

3.5.4 Difference-bound matrices

Clock zones can be efficiently represented by sets of linear inequalities using *difference-bound matrices* (DBM) [Dil89b]. Suppose a timed automaton has k clocks, $x_1, \ldots, x_k$. Then a clock zone can be represented by a $(k+1) \times (k+1)$ matrix D. The entry D_{i0} gives an upper bound of the clock x_i, whereas the entry D_{0i} gives a lower bound of the clock. For every pair i, j the entry D_{ij} gives an upper bound on the difference of clocks x_i and x_j. To distinguish between a strict and a non-strict bound and allow the absence of a bound, the so-called *bounds-domain* $\mathbb{D}$ for the entries of the matrix is

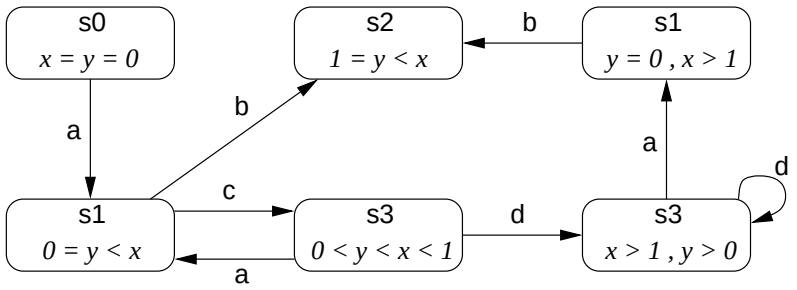

Figure 3.6 Zone automaton for the region automaton in Figure 3.5 (b).

defined to be $\mathbb{Z} \times \{0,1\} \cup \{\infty\}$: the constant ∞ denotes the absence of a bound; the bound $(c,1)$ for $c \in \mathbb{Z}$, denotes the non-strict bound $\leq c$; and the bound $(c,0)$ denotes the strict bound $< c$. A clock valuation v satisfies a DBM D iff for all $1 \leq i \leq k$, $x_i \leq D_{i0}$ and $-x_i \leq D_{0i}$, and for all $1 \leq i,j \leq k$, $x_i - x_j \leq D_{ij}$. Every DBM represents a clock zone, and every clock zone is represented by some DBM.

EXAMPLE 3.6 *Consider the clock zone defined by the following constraints:*

$$[0 \leq x_1 < 2] \ \wedge \ [0 < x_2 < 1] \ \wedge \ [x_1 - x_2 \geq 0]$$

It can be represented by the following difference-bound matrix:

	0	1	2
0	∞	$(0,1)$	$(0,0)$
1	$(2,0)$	∞	∞
2	$(1,0)$	$(0,1)$	∞

■ 3.6

A good source of information on the construction of difference-bound matrices can be found in [CGP00].

3.5.5 Discussion

Although many techniques have been devised to alleviate the state-explosion problem, *e.g.* partial orders [YSSC93, RM94] or approximations [HPR97], super-exponential improvements in the resulting representations and algorithms are unlikely. This fact is specially true for timed systems. In consequence, other high-level techniques (*e.g.* abstraction, compositional reasoning, induction, etc.) appear as the more promising ones for future developments.

Nevertheless, several tools exist for the verification of timed systems. The real-time extension of COSPAN [AK96] allows the analysis of timing constraints using both region

or zone automata. The state space exploration can be performed either by an on-the-fly explicit enumeration or by a BDD-based symbolic approach. With a similar approach, KRONOS [Yov97] supports model-checking of the branching real-time temporal logic TCTL, and has interfaces to a variety of process-algebraic notations. In [BMPY97], an experimental extension to KRONOS is presented, which relies on a canonical representation of discretized sets of clocks configurations using BDDs. The method takes advantage of the symbolic representation and allows to deal with systems that cannot be treated with stae-of-the-art DBM-based tools. UPAAL [BLL+95] allows the verification of safety and liveness properties on networks of communicating automata. The check relies on an on-the-fly reachability analysis of the zone automaton. Moreover, compositional reasoning techniques are used to reduce the search space [LPY95].

3.6 Petri net-based methods

Time has been incorporated to Petri nets in several ways. In *timed* PNs [Ram74] a finite fire duration is associated to each transition of the net. Thus, the firing rule is modified such that transitions must fire once they are enabled but the actual firing has a given duration. *Time* PNs [MF76] generalize this model by associating a time interval (delay bounds) inside which the transition can fire once it has been enabled. This model is more general than timed PNs and hence has been much more widely used. In contrast to these models, *orbital nets* [Rok93] associate a pair of delay bounds to the places of the net. The notion of *age* of a token is defined to capture the time elapsed since a token was put in a place. Then, transitions become enabled only when all its predecessor places are marked and the age of all the tokens belongs to the corresponding time interval. In general, PNs with time associated to places can be easily modeled by PNs with time associated to transitions, whereas the reverse is more complicated [SY96].

PNs augmented with timing information have been extensively used for the verification of timed systems. Two main areas of research can be distinguished: timing analysis, *i.e.* the computation of the separation time between the occurrence of events; and techniques to alleviate the state explosion problem.

Regarding timing analysis, [MD92] presents a polynomial algorithm for the computation of the minimum and maximum separation time between events in acyclic graphs. Although this work does not refer to PNs it is the precursor of many later works. For example, in [MM93] a polynomial algorithm is presented that estimates the minimum and maximum time differences between events in a cyclic free-choice net. The algorithm unfolds the net into an infinite acyclic graph and examines two finite acyclic sub-graphs to determine the time-separation bounds. The limitation to free-choice nets is partially overcome by the work in [HB94, Hul95]. It provides a way to compute a single exact time separation between two events in a cyclic PN with more general types of choice.

A number of approaches have been provided to alleviate the state explosion problem in timed systems. Most of them [YSSC93, KT94, SY96, VdJL96, BJLY98] rely on the use of partial order techniques derived from the original work on unfoldings of PNs [McM92]. Although these techniques allow significant improvements for highly concurrent systems, their major drawback is that they still require a time region per every sequence leading to each reachable state. To solve this problem, the works in [Rok93, RM94, MRM99] and the related verification tool ORBITS, take a different approach. They reduce the number of time regions per state by using POSETs of events rather than linear sequences, to construct the geometric regions. In turn, they can only handle a class of systems in which the firing time of an event only depends on a single predecessor event. The work started in ORBITS has been extended to deal with a wider class of systems [BM97] and improved with more efficient representations of the timed state space [BMH01]. The resulting techniques have been incorporated to the verification tool ATACS [BMH99].

On a completely different approach, recently [KBS02] have proposed a verification method for timed systems that uses the relative timing paradigm to avoid the computation of the exact timed state space. However, they restrict to a class of systems with only certain types of causality relation between the events.

3.7 Conclusions

The chapter has reviewed the most relevant approaches for the modeling, specification and verification of timed systems: timed automata, timed temporal logic and timed model-checking. Also, relevant approaches to the timing analysis and verification based on the use of Petri nets have been briefly summarized.

A number of verification tools have resulted from all these approaches, however their practical applicability is often restricted to systems with a small state space, or with a particular structure that fits well with a given verification approach. Although efficient methods for the representation of the state space have been devised, the underlying problem is still the state-explosion, which is exacerbated when timing information comes into play.

The verification methodology we propose in the next chapter uses timed transition systems as the underlying formalism to model timed systems under the continuous-time paradigm. Instead of computing the exact timed state space, the relative timing paradigm is used to abstract exact time information from the representation. Hence, LzTSs are used, which represent the ordering relations between events in the timed domain, by explicitly distinguishing between their enabling and their actual firing conditions. The approach is applicable to systems modeled by timed transition systems without restrictions. For example, no requirement is imposed about the causality relations between events or about the types of choice allowed.