Chapter 4

Model Checking Real-Time Temporal Logic

Time-critical systems

Temporal logics like PLTL and CTL facilitate the specification of properties that focus on the temporal order of events. This temporal order is a qualitative notion; time is not considered quantitatively. For instance, let proposition p correspond to the occurrence of event A, and q correspond to the occurrence of event B. Then the PLTL-formula $\mathsf{G}\,[p \;\Rightarrow\; \mathsf{F}\,q]$ states that event A is always eventually followed by event B, but it does not state anything about how long the period between the occurrences of A and B will be. This absent notion of quantitative time is essential for the specification of *time-critical systems.* According to the "folk definition":

> *Time-critical systems are those systems in which correctness depends not only on the logical result of the computation but also on the time at which the results are produced.*

That is, time-critical systems must satisfy not only functional correctness requirements, but also timeliness requirements. Such systems are typically characterized by quantitative timing properties relating occurrences of events.

A typical example of a time-critical system is a train crossing: once the approach of a train is detected, the crossing needs to be closed within a certain time bound in order to halt car and pedestrian traffic before the train reaches the crossing. Communication protocols are another typical example of time-critical systems: after the transmission of a datum, a retransmission must be initiated if an acknowledgement is not received within a certain time bound. A third example of a time-critical system is a radiation machine where patients are imposed a high dosis of radiation during a limited time-period; a small exceed of this period is dangerous and can cause the patient's death.

Quantitative timing information is not only essential for time-critical systems, but is also a necessary ingredient in order to analyze and specify the performance aspects of systems. Typical performance-related statements like "job arrivals take place with an inter-arrival time of 30 seconds", can only be stated if one can *measure* the amount of elapsed time.

Quantitative time in temporal logics

In this chapter we treat the extension of a branching-time temporal logic with a quantitative notion of time. We have seen in the two previous chapters that linear and branching temporal logics are interpreted in terms of models where the notion of states plays a central role. The most important change when switching to quantitative time is the ability to measure time. That is, how is the notion of time incorporated in the state-based model and how is the relationship between states and time? As indicated by Koymans (1989) there are several issues that need to be addressed when incorporating time. The following issues, for example, need to be addressed:

- how should time elements be represented (explicitly or implicitly)?
- what is the notion of time reference (absolute or relative)?
- what is the semantical time domain (discrete or continuous)?
- how is time measure presented (additive or metric)?

As a result of the various ways in which these questions can be answered, a whole spectrum of real-time temporal logics has been developed. Some important real-time temporal logics are: Real-Time Temporal Logic (RTTL, Ostroff and Wonham, 1985), Metric Temporal Logic (Koymans, 1990), Explicit Clock Temporal Logic (Harel, Lichtenstein and Pnueli, 1990), Timed Propositional Temporal Logic (Alur and Henzinger, 1991), Timed Computational Tree Logic (Alur and Dill, 1989) and Duration Calculus (Chaochen, Hoare and Ravn, 1991). The first five are timed extensions of linear temporal logics, the one-but-last a real-time branching temporal logic, and the last an interval temporal logic, a timed extension of temporal logic that is suited for reasoning about time intervals. The purpose of these lecture notes is neither to discuss all possible variants nor to compare the merits or drawbacks of these approaches.

Choice of time domain

One of the most important issues is the choice of the semantical time domain. Since time — as in Newtonian physics — is of continuous nature, the most obvious choice is a continuous time domain like the real numbers. For synchronous systems, for instance, in which components proceed in a "lock-step" fashion, discrete time domains are appropriate. A prominent example of discrete-time temporal logics is RTTL. For synchronous systems one could alternatively consider to use ordinary temporal logic where the next operator is used as a means to "measure" the discrete advance of time. In a nutshell this works as follows. The basic idea is that a computation can be considered as a sequence of states where each transition from one state to the next state in the sequence can be thought of as a single tick of some computation clock. That is, $\mathsf{X}\,\phi$ is valid if after one tick of the clock ϕ holds. Let X^k be a sequence of k consecutive next operators defined by $\mathsf{X}^0\phi = \phi$ and $\mathsf{X}^{k+1}\phi = \mathsf{X}^k(\mathsf{X}\,\phi)$ for $k \geqslant 0$. The requirement that the maximal delay between event A and event B is at most 32 time-units then amounts to specifying

$$\mathsf{G}\,[p \;\Rightarrow\; \mathsf{X}^{<32}q]$$

where $\mathsf{X}^{<k}\phi$ is an abbreviation of the finite disjunction $\mathsf{X}^0\phi \;\vee\; \ldots \;\vee\; \mathsf{X}^{k-1}\phi$ and p, q correspond to the occurrences of A and B, respectively. (As an aside remark, the introduction of the X^k-operator makes the satisfiability problem for timed linear temporal logic EXPSPACE-hard.) Since we do not want to restrict ourselves to synchronous systems we consider a *real-valued* time-domain in these lecture notes. This allows for a more accurate specification of non-synchronous systems.

Model checking timed CTL

Suitability as specification language has originally been the driving force for the development of most real-time temporal logics. With the exception of the work by Alur and co-authors in the early nineties, automatic verification of real-time properties has not directly been the main motivation. This interest has increased significantly later (and is still increasing). Introducing time into a temporal logic must be carried out in a careful way in order to keep the model-checking problem be decidable. We concentrate on the temporal logic Timed CTL (TCTL, for short), for which model checking is decidable (Alur, Courcoubetis & Dill, 1993). This is a real-time branching temporal logic that represents time elements implicitly, supports relative time references, and is interpreted in terms of an additive, continuous time domain (i.e. $\mathbb{R}^+$, the non-negative real numbers). The basic idea of Timed CTL is to allow simple time constraints as parameters of the usual CTL temporal operators. In Timed CTL we can, for instance, specify that the maximal delay between event A and event B is 32 time-units by

$$\mathsf{AG}\,[p \;\Rightarrow\; \mathsf{AF}_{<32}\,q]$$

where as before proposition p (q) corresponds to the occurrence of event A (B). Timed CTL is a real-time extension of CTL for which model checking algorithms and several tools do exist.

The main difficulty of model-checking models in which the time domain is continuous is that the model to be checked has *infinitely* many states — for each time value the system can be in a certain state, and there are infinitely many of those values! This suggests that we are faced with an undecidable problem. In fact, the major question in model checking real-time temporal logics that must be dealt with is *how to cope with this infinite state space?* Since the solution to this key problem is rather similar for linear and branching temporal logics we treat one case, Timed CTL.

The essential idea to perform model checking on a continuous time-domain is to realize a discretization of this domain on-demand, i.e. depending on the property to be proven and the system model.

As a specification formalism for real-time systems we introduce the notion of *timed automata*, an extension of finite-state automata with clocks that are used to measure time. Since their conception (Alur & Dill, 1994), timed automata have been used for the specification of various types of time-critical systems, ranging from communication protocols to safety-critical systems. In addition, several model checking tools have been developed for timed automata. The central topic of the chapter will be to develop a model checking algorithm that determines the truth of a TCTL-formula with respect to a timed automaton.

Model checking real-time linear temporal logics

As we will see in this chapter the model-checking algorithm is strongly based on the labelling procedures that we introduced for automatically verifying CTL in Chapter 3. Although we will not deal with real-time *linear* temporal logics here, we like to mention that model checking of such logics — if decidable — is possible using the same paradigm as we have used for untimed linear temporal logics, namely Büchi automata. Alur and Henzinger (1989) have shown that using a timed extension of Büchi automata, model checking of an appropriate timed propositional linear temporal logic is feasible. Based on these theoretical results, Tripakis and Courcoubetis (1996) have constructed a prototype of a real-time extension of SPIN, the tool that we have used to illustrate model checking PLTL in Chapter 2. The crux of model checking real-time PLTL is to transform a timed Büchi automaton A_{sys} into a discretized variant, called a region automaton,

$\mathcal{R}(A_{sys})$. Then the scheme proceeds as for PLTL: a timed Büchi automaton $\mathcal{A}_{\neg\phi}$ is constructed for the property ϕ to be checked, and subsequently the emptiness of the product region automaton $\mathcal{R}(A_{sys} \otimes A_{\neg\phi})$ is checked. This algorithm uses polynomial space. The transformation of a Büchi automaton into its region automaton is based on an equivalence relation that is identical to the one used in the branching-time approach in this chapter. If A has n states (actually, locations) and k clocks, then $\mathcal{R}(A)$ has $n \cdot 2^{\mathcal{O}(k \log c)} \cdot \mathcal{O}(2^k \cdot k!)$ states, where c is the largest integer constant occurring in A with which clocks are compared.

4.1 Timed automata

Timed automata are used to model finite-state real-time systems. A timed automaton is in fact a finite-state automaton equipped with a finite set of real-valued *clock variables*, called clocks for short. Clocks are used to measure the elapse of time.

Definition 37. (**Clock**)
A clock is a variable ranging over $\mathbb{R}^+$.

In the sequel we will use x, y and z as clocks. A state of a timed automaton consists of the current *location* of the automaton plus the current values of all clock variables.[1] Clocks can be initialized (to zero) when the system makes a transition. Once initialized, they start incrementing their value implicitly. All clocks proceed at the same rate. The value of a clock thus denotes the amount of time that has been elapsed since it has been initialized. Conditions on the values of clocks are used as enabling conditions of transitions: only if the clock constraint is fulfilled, the transition is enabled, and can be taken; otherwise, the transition is blocked. Invariants on clocks are used to limit the amount of time that may be spent in a location. Enabling conditions and invariants are constraints over clocks:

Definition 38. (**Clock constraints**)

[1]Note the (deliberate) distinction between location and state.

For set C of clocks with $x, y \in C$, the set of *clock constraints* over C, $\Psi(C)$, is defined by

$$\alpha ::= x \prec c \;\big|\; x - y \prec c \;\big|\; \neg\, \alpha \;\big|\; (\alpha \;\wedge\; \alpha)$$

where $c \in \mathbb{N}$ and $\prec\, \in \{ <, \leqslant \}$.

Throughout this chapter we use typical abbreviations such as $x \geqslant c$ for $\neg\,(x < c)$ and $x = c$ for $x \leqslant c \;\wedge\; x \geqslant c$, and so on. We could allow addition of constants in clock constraints like $x \leqslant c{+}d$ for $d \in \mathbb{N}$, but addition of clock variables, like in $x{+}y < 3$, would make model checking undecidable. Also if c could be a real number, then model checking a timed temporal logic that is interpreted in a dense time domain — as in our case — becomes undecidable. Therefore, c is required to be a natural. Decidability is not affected if the constraint is relaxed such that c is allowed to be a rational number. In this case the rationals in each formula can be converted into naturals by suitable scaling, see also Example 34.

Definition 39. (**(Safety) timed automaton**)
A *timed automaton* $\mathcal{A}$ is a tuple $(L, l_0, E, \mathit{Label}, C, \mathit{clocks}, \mathit{guard}, \mathit{inv})$ with

- L, a non-empty, finite set of locations with initial location $l_0 \in L$
- $E \subseteq L \times L$, a set of edges
- $\mathit{Label} : L \longrightarrow 2^{AP}$, a function that assigns to each location $l \in L$ a set $\mathit{Label}(l)$ of atomic propositions
- C, a finite set of clocks
- $\mathit{clocks} : E \longrightarrow 2^{C}$, a function that assigns to each edge $e \in E$ a set of clocks $\mathit{clocks}(e)$
- $\mathit{guard} : E \longrightarrow \Psi(C)$, a function that labels each edge $e \in E$ with a clock constraint $\mathit{guard}(e)$ over C, and
- $\mathit{inv} : L \longrightarrow \Psi(C)$, a function that assigns to each location an *invariant*.

The function *Label* has the same role as for models for CTL and PLTL and associates to a location the set of atomic propositions that are valid in that location. As we will see, this function is only relevant for defining the satisfaction of atomic propositions in the semantics of TCTL. For edge e, the set $\mathit{clocks}(e)$ denotes the set of clocks that are to be reset when traversing e, and $\mathit{guard}(e)$ denotes the enabling condition that specifies when e can be taken. For location l, $\mathit{inv}(l)$ constrains the amount of time that can be spent in l.

For depicting timed automata we adopt the following conventions. Circles denote locations and edges are represented by arrows. Invariants are indicated inside locations, unless they equal 'true', i.e. if no constraint is imposed on delaying. Arrows are equipped with labels that consist of an optional clock constraint and an optional set of clocks to be reset, separated by a straight horizontal line. If the clock constraint equals 'true' and if there are no clocks to be reset, the arrow label is omitted. The horizontal line is omitted if either the clock constraint is 'true' or if there are no clocks to be reset. We will often omit the labelling *Label* in the sequel. This function will play a role when discussing the model checking algorithm only, and is of no importance for the other concepts to be introduced.

Example 27. *Figure 4.1(a) depicts a simple timed automaton with one clock x and one location l with a self-loop. The self-transition can be taken if clock x has at least the value 2, and when being taken, clock x is reset. Initially, clock x has the value 0. Figure 4.1(b) gives an example execution of this timed automaton, by depicting the value of clock x. Each time the clock is reset to 0, the automaton moves from location l to location l. Due to the invariant 'true' in l, time can progress without bound while being in l.*

Changing the timed automaton of Figure 4.1(a) slightly by incorporating an invariant $x \leqslant 3$ in location l, leads to the effect that x cannot progress without bound anymore. Rather, if $x \geqslant 2$ (enabling constraint) and $x \leqslant 3$ (invariant) the outgoing edge must be taken. This is illustrated in Figure 4.1(c) and (d).

Observe that the same effect is not obtained when strengthening the enabling constraint in Figure 4.1(a) into $2 \leqslant x \leqslant 3$ while keeping the invariant 'true' in l. In that case, the outgoing edge can only be taken when $2 \leqslant x \leqslant 3$ (as in the previous scenario), but is not forced to be taken, i.e. it can simply be ignored by

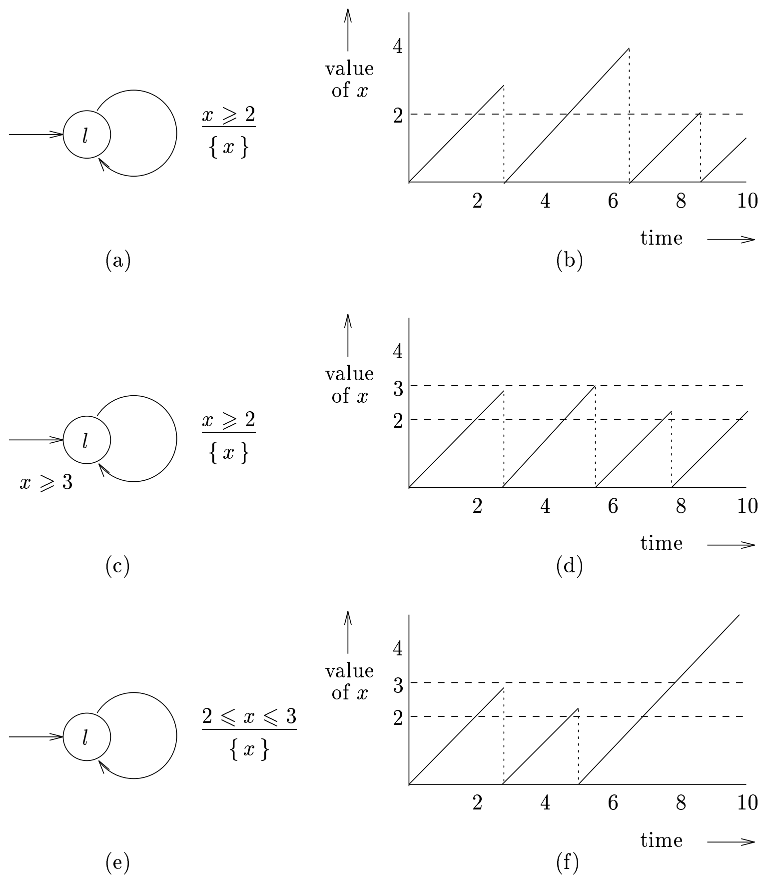


Figure 4.1: A timed automaton with one clock and a sample evolution of it

letting time pass while staying in l. This is illustrated in Figure 4.1(e) and (f). *(End of example.)*

We like to emphasize that different clocks can be started at different times, and hence there is no lower bound on their difference. This is, for instance, not possible in a discrete-time model, where the difference between two concurrent clocks is always a multiple of one unit of time. Having multiple clocks thus allows to model multiple concurrent delays. This is exemplified in the following example.

Example 28. *Figure 4.2(a) depicts a timed automaton with two clocks, x and y. Initially, both clocks start running from value 0 on, until one time-unit has passed. From this point in time, both edges are enabled and can be taken non-deterministically. As a result, either clock x or clock y is reset, while the other clock continues. It is not difficult to see that the difference between clocks x and y is arbitrary. An example evolution of this timed automaton is depicted in part (b) of the figure.* *(End of example.)*

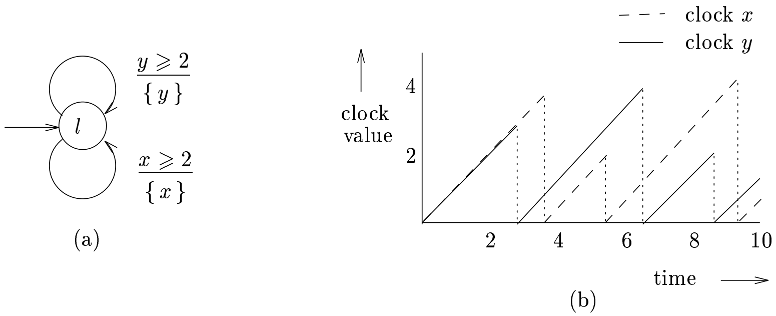


Figure 4.2: A timed automaton with two clocks and a sample evolution of it

Example 29. *Figure 4.3 shows a timed automaton with two locations, named* off *and* on*, and two clocks x and y. All clocks are initialized to 0 if the initial location* off *is entered. The timed automaton in Figure 4.3 models a switch that controls a light. The switch may be turned on at any time instant since the light has been switched on for at least two time units, even if the light is still on. It*

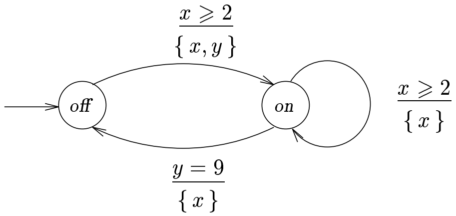


Figure 4.3: A timed automaton example: the switch

switches automatically off exactly 9 time-units after the most recent time the light has turned from off to on. Clock x is used to keep track of the delay since the last time the light has been switched on. The edges labelled with the clock constraint $x \geqslant 2$ model the on-switching transitions. Clock y is used to keep track of the delay since the last time that the light has moved from location off to on, and controls switching the light off. *(End of example.)*

In the above examples we have already shown how clocks evolve. The values of clocks are formally defined by *clock valuations*.

Definition 40. (**Clock valuation**)
A *clock valuation* v for a set of clocks C is a function $v : C \longrightarrow \mathbb{R}^+$, assigning to each clock $x \in C$ its current value $v(x)$.

Let $V(C)$ denote the set of all clock valuations over C. A *state* of $\mathcal{A}$ is a pair (l, v) with l a location in $\mathcal{A}$ and v a valuation over C, the set of clocks of $\mathcal{A}$.

Example 30. Consider the timed automaton of Figure 4.3. Some states of this timed automaton are the pairs (off, v) *with* $v(x) = v(y) = 0$ *and* (off, v') *with* $v'(x) = 4$ *and* $v'(y) = 13$ *and* (on, v'') *with* $v''(x) = Label$ *and* $v''(y) = 3$. *Note that the latter state is not reachable.* *(End of example.)*

Let v be a clock valuation on set of clocks C. Clock valuation $v{+}t$ denotes that all clocks are increased by t with respect to valuation v. It is defined by $(v{+}t)(x) = v(x) + t$ for all clocks $x \in C$. Clock valuation $\mathsf{reset}\ x\ \mathsf{in}\ v$, valuation

v with clock x reset, is defined by

$$(\mathsf{reset}\ x\ \mathsf{in}\ v)(y) = \begin{cases} v(y) & \text{if } y \neq x \\ 0 & \text{if } y = x. \end{cases}$$

Nested occurrences of reset are typically abbreviated. For instance, $\mathsf{reset}\ x\ \mathsf{in}$ $(\mathsf{reset}\ y\ \mathsf{in}\ v)$ is simply denoted $\mathsf{reset}\ x, y\ \mathsf{in}\ v$.

Example 31. Consider the clock valuations v and v' of the previous example. Valuation $v{+}9$ is defined by $(v{+}9)(x) = (v{+}9)(y) = 9$. *In clock valuation* $\mathsf{reset}\ x\ \mathsf{in}\ (v{+}9)$, *clock x has value 0 and clock y reads 9. Clock valuation v' now equals* $(\mathsf{reset}\ x\ \mathsf{in}\ (v{+}9)) + 4$. *(End of example.)*

We can now formally define what it means for a clock constraint to be valid or not. This is done in a similar way as characterizing the semantics of a temporal logic, namely by defining a satisfaction relation. In this case the satisfaction relation $\models$ is a relation between clock valuations (over set of clocks C) and clock constraints (over C).

Definition 41. (**Evaluation of clock constraints**)
For $x, y \in C$, $v \in V(C)$ and let $\alpha, \beta \in \Psi(C)$ we have

$$\begin{array}{ll} v \models x \prec c & \text{iff } v(x) \prec c \\ v \models x - y \prec c & \text{iff } v(x) - v(y) \prec c \\ v \models \neg\, \alpha & \text{iff } v \not\models \alpha \\ v \models \alpha \ \wedge\ \beta & \text{iff } v \models \alpha \ \wedge\ v \models \beta. \end{array}$$

For negation and conjunction, the rules are identical to those for propositional logic. In order to check whether $x \prec c$ is valid in v, it is simply checked whether $v(x) \prec c$. Similarly, constraints on differences of clocks are treated.

Example 32. Consider clock valuation v, $v{+}9$ and $\mathsf{reset}\ x\ \mathsf{in}\ (v + 9)$ *of the previous example and suppose we want to check the validity of $\alpha = x \leqslant 5$ and of $\beta = (x - y = 0)$. It follows $v \models \alpha$ and $v \models \beta$, since $v(x) = v(y) = 0$. We have*

$v{+}9 \not\models \alpha$ *since* $(v{+}9)(x) = 9 \not\leqslant 5$ *and* $v{+}9 \models \beta$ *since* $(v{+}9)(x) = (v{+}9)(y) = 9$. *The reader is invited to check that* $\mathsf{reset}\ x\ \mathsf{in}\ (v{+}9) \models \alpha$, *but* $\mathsf{reset}\ x\ \mathsf{in}\ (v{+}9) \not\models \beta$. *(End of example.)*

4.2 Semantics of timed automata

The interpretation of timed automata is defined in terms of an infinite transition system $(S, \longrightarrow)$ where S is a set of states, i.e. pairs of locations and clock valuations, and $\longrightarrow$ is the transition relation that defines how to evolve from one state to another. There are two possible ways in which a timed automaton can proceed: by traversing an edge in the automaton, or by letting time progress while staying in the same location. Both ways are represented by a single transition relation $\longrightarrow$.

Definition 42. (**Transition system underlying a timed automaton**)
The transition system associated to timed automaton $\mathcal{A}$, denoted $\mathcal{M}(\mathcal{A})$, is defined as $(S, s_0, \longrightarrow)$ where:

- $S = \{ (l, v) \in L \times V(C) \mid v \models inv(l) \}$
- $s_0 = (l_0, v_0)$ where $v_0(x) = 0$ for all $x \in C$
- the transition relation $\longrightarrow \subseteq S \times (\mathbb{R}^+ \cup \{ * \}) \times S$ is defined by the rules:
  1. $(l, v) \xrightarrow{*} (l', \mathsf{reset}\ clocks(e)\ \mathsf{in}\ v)$ if the following conditions hold:
     (a) $e = (l, l') \in E$
     (b) $v \models guard(e)$, and
     (c) $(\mathsf{reset}\ clocks(e)\ \mathsf{in}\ v) \models inv(l')$
  2. $(l, v) \xrightarrow{d} (l, v{+}d)$, for positive real d, if the following condition holds:
     $\forall\, d' \leqslant d.\, v{+}d' \models inv(l)$.

The set of states is the set of pairs (l, v) such that l is a location of $\mathcal{A}$ and v is a clock valuation over C, the set of clocks in $\mathcal{A}$, such that v does not invalidate

the invariant of l. Note that this set may include states that are unreachable. For a transition that corresponds to (a) traversing edge e in the timed automaton it must be that (b) v satisfies the clock constraint of e (otherwise the edge is disabled), and (c) the new clock valuation, that is obtained by resetting all clocks associated to e in v, satisfies the invariant of the target location l' (otherwise it is not allowed to be in l'). Idling in a location (second clause) for some positive amount of time is allowed if the invariant is respected while time progresses. Notice that it does not suffice to only require $v{+}d \models inv(l)$, since this could invalidate the invariant for some $d' < d$. For instance, for $inv(l) = (x \leqslant 2) \vee (x > 4)$ and state (l, v) with $v(x) = 1.5$ it should not be allowed to let time pass with 3 time-units: although the resulting valuation $v{+}3 \models inv(l)$, the intermediate valuation $v{+}2$, for instance, invalidates the clock constraint.

Definition 43. (**Path**)
A *path* is an infinite sequence $s_0\, a_0\, s_1\, a_1\, s_2\, a_2\, \ldots$ of states alternated by transition labels such that $s_i \xrightarrow{a_i} s_{i+1}$ for all $i \geqslant 0$.

Note that labels can either be * in case of traversing an edge in the timed automaton, or (positive) real numbers in case of delay transition. The set of paths starting in a given state is defined as before. A *position* of a path is a pair (i, d) such that d equals 0 if $a_i = *$, i.e. in case of an edge-transition, and equals a_i otherwise. The set of positions of the form (i, d) characterizes the set of states visited by σ while going from state s_i to the successor s_{i+1}. Let $Pos(\sigma)$ denotes the set of positions in σ. For convenience, let $\sigma(i, d)$ denote the state $(s_i, v_i{+}d)$. A total order on positions is defined by:

$$(i, d) \ll (j, d') \text{ if and only if } i < j \ \vee\ (i = j \ \wedge\ d \leqslant d').$$

That is, position (i, d) precedes (j, d') if l_i is visited before l_j in σ, or if the positions point to the same location and d is at most d'.

In order to "measure" the amount of time that elapses on a path, we introduce:

Definition 44. (**Elapsed time on a path**)
For path $\sigma = s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \ldots$ and natural i, the time elapsed from s_0 to s_i,

denoted $\Delta(\sigma, i)$ is defined by:

$$\begin{array}{rcl} \Delta(\sigma,0) & = & 0 \\ \Delta(\sigma,i{+}1) & = & \Delta(\sigma,i) + \begin{cases} 0 & \text{if } a_i = * \\ a_i & \text{if } a_i \in \mathbb{R}^+. \end{cases} \end{array}$$

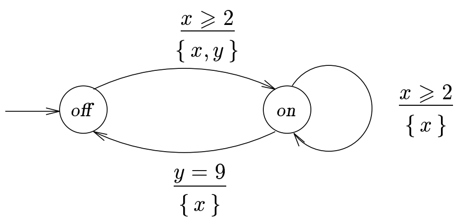
Path σ is called *time-divergent* if $\lim_{i\to\infty} \Delta(\sigma,i) = \infty$. The set of time-divergent paths starting at state s is denoted $P^{\infty}_{\mathcal{M}}(s)$. An example of a non time-divergent path is a path that visits an infinite number of states in a bounded amount of time. For instance, the path

$$s_0 \xrightarrow{2^{-1}} s_1 \xrightarrow{2^{-2}} s_2 \xrightarrow{2^{-3}} s_3 \ldots$$

is not time-divergent, since an infinite number of states is visited in the bounded interval $[\frac{1}{2}, 1]$. Since such paths are not realistic, usually non-Zeno timed automata are considered.

Definition 45. (**Non-Zeno timed automaton**)
A timed automaton $\mathcal{A}$ is called $non-Zeno$ if from any of its states some time-divergent path can start.

Example 33. *Recall the light switch from Example 29:*



A prefix of an example path of the switch is

$$\begin{array}{rcl} \sigma & = & (\text{off}, v_0) \xrightarrow{3} (\text{off}, v_1) \xrightarrow{*} (\text{on}, v_2) \xrightarrow{4} (\text{on}, v_3) \xrightarrow{*} (\text{on}, v_4) \\ & & \xrightarrow{1} (\text{on}, v_5) \xrightarrow{2} (\text{on}, v_6) \xrightarrow{2} (\text{on}, v_7) \xrightarrow{*} (\text{off}, v_8) \ldots \end{array}$$

with $v_0(x) = v_0(y) = 0$, $v_1 = v_0{+}3$, $v_2 = \textsf{reset}\ x, y\ \textsf{in}\ v_1$, $v_3 = v_2{+}4$, $v_4 = \textsf{reset}\ x\ \textsf{in}\ v_3$, $v_5 = v_4{+}1$, $v_6 = v_5{+}2$, $v_7 = v_6{+}2$ *and* $v_8 = \textsf{reset}\ x\ \textsf{in}\ v_7$. *These clock valuations are summarized by the following table:*

	v_0	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8
x	*0*	*3*	*0*	*4*	*0*	*1*	*3*	*5*	*0*
y	*0*	*3*	*0*	*4*	*4*	*5*	*7*	*9*	*9*

The transition $(\text{off}, v_1) \xrightarrow{*} (\text{on}, v_2)$ *is, for instance, possible since (a) there is an edge e from off to on, (b)* $v_1 \models x \geqslant 2$ *since* $v_1(x) = 3$, *and (c)* $v_2 \models \text{inv}(\text{on})$. *The positions of* σ *are (0,3), (1,0), (2,4), (3,0), (4,1), (5,2), (6,2), (7,0), respectively. We have, for instance,* $(1,0) \ll (5,2)$. *In addition* $\Delta(\sigma,3) = 7$ *and* $\Delta(\sigma,6) = 12$.

Another possible evolution of the switch is to stay infinitely long in location off by making infinitely many delay transitions. Although at some point, i.e. if $v(x) \geqslant 2$ *the edge to location on is enabled, it can be ignored continuously. Similarly, the switch may stay arbitrarily long in location on. These behaviors are caused by the fact that* $\text{inv}(\text{off}) = \text{inv}(\text{on}) = true$. *If we modify the switch such that* $\text{inv}(\text{off})$ *becomes* $y \leqslant 9$ *while* $\text{inv}(\text{on})$ *remains true, the aforementioned path* σ *is still legal. In addition, the light may stay infinitely long in location off — while awaiting a person that pushes the button — it* must *switch off automatically if during 9 time-units the automaton has not switched from off to on. (End of example.)*

4.3 Syntax of TCTL

Let $\mathcal{A}$ be a timed automaton, AP a set of atomic propositions and D a non-empty set of clocks that is disjoint from the clocks of $\mathcal{A}$, i.e. $C \cap D = \varnothing$. $z \in D$

is sometimes called a *formula clock*. Since clocks form a part of the syntax of the logic, the logic is sometimes referred to as explicit-clock temporal logic.

Definition 46. (**Syntax of timed computation tree logic**)
For $p \in AP$, $z \in D$, and $\alpha \in \Psi(C \cup D)$, the set of TCTL-formulas is defined by:

$$\boxed{\phi ::= p \;\big|\; \alpha \;\big|\; \neg\,\phi \;\big|\; \phi \,\vee\, \phi \;\big|\; z \;\mathsf{in}\; \phi \;\big|\; \mathsf{E}\,[\phi\,\mathsf{U}\,\phi] \;\big|\; \mathsf{A}\,[\phi\,\mathsf{U}\,\phi].}$$

Notice that α is a clock constraint over formula clocks and clocks of the timed automaton; it allows, for instance, the comparison of a formula clock and an automaton clock. The boolean operators true, false, $\wedge$, $\Rightarrow$ and $\Leftrightarrow$ are defined in the usual way (see Chapter 2). Clock z in z in ϕ is called a *freeze identifier* and bounds the formula clock z in ϕ. The intuitive interpretation is that z in ϕ is valid in state s if ϕ holds in s where clock z starts with value 0 in s. For instance, z in $(z = 0)$ is a valid statement, while z in $(z > 1)$ is not. The use of this freeze identifier is very useful in combination with until-constructions to specify typical timeliness requirements like punctuality, bounded response, and so on. Typically, one is not interested in formulas where formula clocks from D are free, that is, not bound. For simplicity we therefore consider all TCTL-formulas to be closed from now on. This means, for instance, that formulas like $x \leqslant 2$ and z in $(z - y = 4)$ are illegal if x, y are formula clocks (i.e. $x, y \in D$); x in $(x \leqslant 2)$ and z in $(y$ in $(z - y = 4))$ are legal formulas, however.

Like for CTL, the basic operators of TCTL are until-formulas with existential and universal quantification over paths. EF, EG, and so on, are derivatives of these until-formulas like before. Notice that there are no timed variants of EX and AX in TCTL. The reason for this will become clear when we treat the semantics of the logic. The binding of the operators is identical to that of CTL.

Using the freeze identifier, temporal operators of CTL like $\mathsf{E}\,[\phi\,\mathsf{U}\,\psi]$, $\mathsf{EF}\,\phi$ and so on, can be equipped with a time constraint in a succinct way. For instance, the formula

$$\mathsf{A}\,[\phi\,\mathsf{U}_{\leqslant 7}\,\psi]$$

intuitively means that along any path starting from the current state, the property ϕ holds continuously until within 7 time-units ψ becomes valid. It can be defined by

$$z \;\mathsf{in}\; \mathsf{A}\,[(\phi \;\wedge\; z \leqslant 7)\,\mathsf{U}\,\psi].$$

Alternatively, the formula

$$\mathsf{EF}_{<5}\,\phi$$

means that along some path starting from the current state, the property ϕ becomes valid within 5 time-units, and is defined by

$$z \;\mathsf{in}\; \mathsf{EF}\,(z < 5 \;\wedge\; \phi)$$

where EF is defined as before. Stated otherwise, $\mathsf{EF}_{<c}\,\phi$ denotes that a ϕ-state is reachable within c time-units. The dual expression, $\mathsf{AF}_{<c}\,\phi$ is valid if all executions lead to a ϕ-state within c time-units.

Example 34. Let $AP = \{\, b = 1, b < 2, b \geqslant 3 \,\}$ *be a set of atomic propositions. Example* TCTL*-formulas are:* $\mathsf{E}\,[(b < 2)\,\mathsf{U}_{\leqslant 21}\,(b \geqslant 3)]$, $\mathsf{AF}_{\geqslant 0}\,(b < 2)$, *and* $\mathsf{EF}_{<7}\,[\mathsf{EF}_{<3}\,(b = 1)]$. *Formula* $\mathsf{AX}_{<2}(b = 1)$ *is not a* TCTL*-formula since there is no next operator in the logic. The formula* $\mathsf{AF}_{\leqslant\pi}\,(b < 2)$ *is not a legal* TCTL*-formula, since* $\pi \notin \mathbb{N}$. *For the same reason* $\mathsf{AF}_{=\frac{2}{3}}\,[\mathsf{AG}_{<\frac{4}{5}}\,(b < 2)]$ *is also not allowed. Using appropriate scaling, however, this formula can be converted into the legal* TCTL*-formula* $\mathsf{AF}_{=5}\,[\mathsf{AG}_{<6}\,(b < 2)]$. *(End of example.)*

Notice that it is not allowed to write a bounded response time property like "there exists some unknown time t such that if ϕ holds, then before time t property ψ holds". For instance,

$$\exists\, t.\, z \;\mathsf{in}\; (\mathsf{AG}_{\geqslant 0}\,[(b = 1) \;\Rightarrow\; \mathsf{AF}\,(z < t \;\wedge\; b \geqslant 3)]).$$

Such quantifications over time makes model checking undecidable.

4.4 Semantics of TCTL

Recall that the interpretation of temporal logic is defined in terms of a model. For PLTL such a model consists of a (non-empty) set S of states, a total successor function R on states, and an assignment *Label* of atomic propositions to states. For CTL the function R is changed into a relation in order to take the branching nature into account. For TCTL, in addition, the notion of real-time has to be incorporated. This is needed in order, for instance, to be able to obtain a formal interpretation of formulas of the form $z \mathsf{in} \phi$ that contain time constraints. The basic idea is to consider a *state*, i.e. a location and a clock valuation. The location determines which atomic propositions are valid (using the labelling *Label*), while the clock valuation is used to determine the validity of the clock constraints in the formula at hand. Since clock constraints may contain besides clocks of the automaton, formula clocks, an additional clock valuation is used to determine the validity of statements about these clocks. Thus, the validity is defined for a state $s = (l, v)$ and formula clock valuation w. We will use $v \cup w$ to denote the valuation that for automata clock x equals $v(x)$ and for formula clock z equals $w(z)$.

The semantics of TCTL is defined by a satisfaction relation (denoted $\models$) that relates a transition system $\mathcal{M}$, state (i.e. a location plus a clock valuation over the clocks in the automaton), a clock valuation over the formula clocks occurring in ϕ, and a formula ϕ. We write $(\mathcal{M}, (s, w), \phi) \in \models$ by $\mathcal{M}, (s, w) \models \phi$. We have $\mathcal{M}, (s, w) \models \phi$ if and only if ϕ is valid in state s of model $\mathcal{M}$ under formula clock valuation w.

State $s = (l, v)$ satisfies ϕ if the "extended" state (s, w) satisfies ϕ where w is a clock valuation such that $w(z) = 0$ for all formula clocks z.

Definition 47. (**Semantics of** TCTL)
Let $p \in AP$ be an atomic proposition, $\alpha \in \Psi(C \cup D)$ a clock constraint over $C \cup D$, $\mathcal{M} = (S, \longrightarrow)$ an infinite transition system, $s \in S$, $w \in V(D)$, and ϕ, ψ

TCTL-formulae. The satisfaction relation $\models$ is defined by:

$$
\begin{array}{ll}
s, w \models p & \text{iff } p \in \mathit{Label}(s) \\
s, w \models \alpha & \text{iff } v \cup w \models \alpha \\
s, w \models \neg\, \phi & \text{iff } \neg\,(s, w \models \phi) \\
s, w \models \phi \,\vee\, \psi & \text{iff } (s, w \models \phi) \,\vee\, (s, w \models \psi) \\
s, w \models z \ \mathsf{in}\ \phi & \text{iff } s, \mathsf{reset}\ z\ \mathsf{in}\ w \models \phi \\
s, w \models \mathsf{E}\,[\phi \,\mathsf{U}\, \psi] & \text{iff } \exists\, \sigma \in P^{\infty}_{\mathcal{M}}(s).\, \exists\, (i, d) \in \mathit{Pos}(\sigma). \\
& \quad (\sigma(i, d), w{+}\Delta(\sigma, i) \models \psi \ \wedge \\
& \quad (\forall\, (j, d') \ll (i, d).\, \sigma(j, d'), w{+}\Delta(\sigma, j) \models \phi \,\vee\, \psi)) \\
s, w \models \mathsf{A}\,[\phi \,\mathsf{U}\, \psi] & \text{iff } \forall\, \sigma \in P^{\infty}_{\mathcal{M}}(s).\, \exists\, (i, d) \in \mathit{Pos}(\sigma). \\
& \quad ((\sigma(i, d), w{+}\Delta(\sigma, i)) \models \psi \ \wedge \\
& \quad (\forall\, (j, d') \ll (i, d).\, (\sigma(j, d'), w{+}\Delta(\sigma, j)) \models \phi \,\vee\, \psi)).
\end{array}
$$

For atomic propositions, negations and disjunctions, the semantics is defined in a similar way as before. Clock constraint α is valid in (s, w) if the values of the clocks in v, the valuation in s, and w satisfy α. Formula $z \ \mathsf{in}\ \phi$ is valid in (s, w) if ϕ is valid in (s, w') where w' is obtained from w by resetting z. The formula $\mathsf{E}\,[\phi \,\mathsf{U}\, \psi]$ is valid in (s, w) if there exists a time-divergent path σ starting in s, that satisfies ψ at some position, and satisfies $\phi \vee \psi$ at all preceding positions. The reader might wonder why it is not required — like in the untimed variants of temporal logic we have seen before — that just ϕ holds at all preceding positions. The reason for this is the following. Assume $i = j$, $d \leqslant d'$ and suppose ψ does not contain any clock constraint. Then, by definition $(i, d) \ll (j, d')$. Moreover, if $i = j$ it means that the positions (i, d) and (j, d') point to the same location in the timed automaton. But then, the same atomic propositions are valid in (i, d) and (j, d'), and, since ψ does not contain any clock constraint, the validity of ψ in (i, d) and (j, d') is the same. Thus, delaying in locations forces us to use this construction.

Consider now, for instance, the formula $\mathsf{E}\,[\phi \,\mathsf{U}_{<12}\, \psi]$ and state s in $\mathcal{M}$. This formula means that there exists an s-path which has an initial prefix that lasts at most 12 units of time, such that ψ holds at the end of the prefix and ϕ holds at all intermediate states. Formula $\mathsf{A}\,[\phi \,\mathsf{U}_{<12}\, \psi]$ requires this property to be valid for all paths starting in s. The formulae $\mathsf{E}\,[\phi \,\mathsf{U}_{\geqslant 0}\, \psi]$ and $\mathsf{A}\,[\phi \,\mathsf{U}_{\geqslant 0}\, \psi]$ are abbreviated by

$\mathsf{E}\,[\phi\,\mathsf{U}\,\psi]$ and $\mathsf{A}\,[\phi\,\mathsf{U}\,\psi]$, respectively. This corresponds to the fact that the timing constraint $\geqslant 0$ is not restrictive.

For PLTL the formula $\mathsf{F}\,(\phi\ \wedge\ \psi)$ implies $\mathsf{F}\,\phi\ \wedge\ \mathsf{F}\,\psi$, but the reverse implication does not hold (the interested reader is invited to check this). Similarly, in CTL we have that $\mathsf{AF}\,(\phi\ \wedge\ \psi)$ implies $\mathsf{AF}\,\phi\ \wedge\ \mathsf{AF}\,\psi$, but not the other way around. Due to the possibility of requiring punctuality, the validity of a property at a precise time instant, we obtain for $\mathsf{AF}_{=c}\,(\phi\ \wedge\ \psi)$ also the implication in the other direction:

$$\mathsf{AF}_{=c}\,(\phi\ \wedge\ \psi) \equiv (\mathsf{AF}_{=c}\,\phi\ \wedge\ \mathsf{AF}_{=c}\,\psi)$$

This can be formally proven as follows:

$$\begin{array}{ll}
& s, w \models \mathsf{AF}_{=c}\,(\phi\ \wedge\ \psi) \\
\Leftrightarrow & \{ \text{ definition of } \mathsf{AF}_{=c} \ \} \\
& s, w \models \mathsf{A}\,[\text{true}\,\mathsf{U}_{=c}\,(\phi\ \wedge\ \psi)] \\
\Leftrightarrow & \{ \text{ definition of } \ \mathsf{U}_{=c}\,;\ \text{let } z \text{ be a `fresh' clock } \} \\
& s, w \models z \ \mathsf{in}\ \mathsf{A}\,[\text{true}\,\mathsf{U}\,(z = c\ \wedge\ \phi\ \wedge\ \psi)] \\
\Leftrightarrow & \{ \text{ semantics of } z \ \mathsf{in}\ \phi \ \} \\
& s, \mathsf{reset}\ z\ \mathsf{in}\ w \models \mathsf{A}\,[\text{true}\,\mathsf{U}\,(z = c\ \wedge\ \phi\ \wedge\ \psi)] \\
\Leftrightarrow & \{ \text{ semantics of } \mathsf{A}\,[\phi\,\mathsf{U}\,\psi];\ \text{true is valid in any state } \} \\
& \forall\,\sigma \in P^{\infty}_{\mathcal{M}}(s).\,\exists\,(i,d) \in Pos(\sigma). \\
& \qquad (\sigma(i,d), (\mathsf{reset}\ z\ \mathsf{in}\ w) + \Delta(\sigma,i) \models (z = c\ \wedge\ \phi\ \wedge\ \psi)) \\
\Leftrightarrow & \{ \text{ predicate calculus } \} \\
& \forall\,\sigma \in P^{\infty}_{\mathcal{M}}(s).\,\exists\,(i,d) \in Pos(\sigma). \\
& \qquad (\sigma(i,d), (\mathsf{reset}\ z\ \mathsf{in}\ w) + \Delta(\sigma,i) \models (z = c\ \wedge\ \phi)\ \wedge\ (z = c\ \wedge\ \psi)) \\
\Leftrightarrow & \{ \text{ semantics of } \wedge \ \} \\
& \forall\,\sigma \in P^{\infty}_{\mathcal{M}}(s).\,\exists\,(i,d) \in Pos(\sigma). \\
& \qquad\qquad ((\sigma(i,d), (\mathsf{reset}\ z\ \mathsf{in}\ w) + \Delta(\sigma,i) \models (z = c\ \wedge\ \phi)) \\
& \qquad\ \wedge\ (\sigma(i,d), (\mathsf{reset}\ z\ \mathsf{in}\ w) + \Delta(\sigma,i) \models (z = c\ \wedge\ \psi)) \\
\Leftrightarrow & \{ \text{ semantics of } \mathsf{A}\,[\phi\,\mathsf{U}\,\psi];\ \text{true is valid in any state } \} \\
& \quad s, \mathsf{reset}\ z\ \mathsf{in}\ w \models \mathsf{A}\,[\text{true}\,\mathsf{U}\,(z = c\ \wedge\ \phi)]
\end{array}$$

$$\begin{array}{ll}
& \quad \wedge\ \ s, \mathsf{reset}\ z\ \mathsf{in}\ w \models \mathsf{A}\,[\text{true}\,\mathsf{U}\,(z = c\ \wedge\ \psi)] \\
\Leftrightarrow & \{ \text{ semantics of } z \ \mathsf{in}\ \phi \ \} \\
& s, w \models z\ \mathsf{in}\ \mathsf{A}\,[\text{true}\,\mathsf{U}\,(z = c\ \wedge\ \phi)]\ \wedge\ s, w \models z\ \mathsf{in}\ \mathsf{A}\,[\text{true}\,\mathsf{U}\,(z = c\ \wedge\ \psi)] \\
\Leftrightarrow & \{ \text{ definition of } \ \mathsf{U}_{=c}\,;\ \text{definition of } \mathsf{AF}_{=c} \ \} \\
& s, w \models \mathsf{AF}_{=c}\,\phi\ \wedge\ s, w \models \mathsf{AF}_{=c}\,\psi \\
\Leftrightarrow & \{ \text{ semantics of } \wedge \ \} \\
& s, w \models (\mathsf{AF}_{=c}\,\phi\ \wedge\ \mathsf{AF}_{=c}\,\psi).
\end{array}$$

The reader is invited to check that this equivalence only holds for the timing constraint $= c$, but not for the others. Intuitively this stems from the fact that in $\mathsf{AF}_{=c}\,(\phi\ \wedge\ \psi)$ we not only require that eventually $\phi\ \wedge\ \psi$ hold, but also at exactly the same time instant. But then also $\mathsf{AF}_{=c}\,\phi$ and $\mathsf{AF}_{=c}\,\psi$. Since conditions like $< c$, $\geqslant c$, and so on, do not single out one time instant a similar reasoning does not apply to the other cases.

Extending the expressivity of CTL with real-time aspects in a dense time-setting has a certain price to pay. Recall that the satisfiability problem of a logic is as follows: does there exist a model $\mathcal{M}$ (for that logic) such that $\mathcal{M} \models \phi$ for a given formula ϕ? For PLTL and CTL satisfiability is decidable, but for TCTL it turns out that the satisfiability problem is *highly undecidable*. Actually, for most real-time logics the satisfiability problem is undecidable: Alur & Henzinger (1993) showed that only a very weak arithmetic over a discrete time domain can result in a logic for which satisfiability is decidable.

Another difference with the untimed variant is the extension with *past* operators, operators that refer to the history rather than to the future (like F, U and G). Some example past operators are previous (the dual of next), and since (the dual of until). It is known that extending CTL and PLTL with past operators does not enhance their expressivity; the major reason for considering these operators for these logics is to improve specification convenience — some properties are easier to specify when past operators are incorporated. For TCTL, however, this is not the case: extending TCTL (actually, any temporal logic interpreted over a continuous domain) with past operators extends its expressivity (Alur & Henzinger, 1992). Model checking such logics is however still possible using a natural accommodation, and the worst case complexity is not increased.

4.5 Specifying timeliness properties in TCTL

In this section we treat some typical timeliness requirements for time-critical systems and formally specify these properties in TCTL.

1. *Promptness* requirement: specifies a maximal delay between the occurrence of an event and its reaction. For example, every transmission of a message is followed by a reply within 5 units of time. Formally,

   $$\mathsf{AG}\,[\mathit{send}(m) \;\Rightarrow\; \mathsf{AF}_{<5}\,\mathit{receive}\,(r_m)]$$

   where it is assumed that m and r_m are unique, and that $\mathit{send}(m)$ and $\mathit{receive}(r_m)$ are valid if m is being sent, respectively if r_m is being received, and invalid otherwise.

2. *Punctuality* requirement: specifies an exact delay between events. For instance, there exists a computation during which the delay between transmitting m and receiving its reply is exactly 11 units of time. Formally:

   $$\mathsf{EG}\,[\mathit{send}(m) \;\Rightarrow\; \mathsf{AF}_{=11}\,\mathit{receive}\,(r_m)]$$

3. *Periodicity* requirement: specifies that an event occurs with a certain period. For instance, consider a machine that puts boxes on a moving belt that moves with a constant speed. In order to maintain an equal distance between successive boxes on the belt, the machine is required to put boxes periodically with a period of 25 time-units, say. Naively, one is tempted to specify this periodic behavior by

   $$\mathsf{AG}\,[\mathsf{AF}_{=25}\,\mathit{putbox}]$$

   where the atomic proposition *putbox* is valid if the machine puts a box on the belt, and invalid otherwise. This formulation does, however, allow to put additional boxes on the belt in between two successive box placings that have a spacing of 25 time-units. For instance, a machine that puts boxes at time instant 25, 50, 75, ... and at time 10, 35, 60, ... on the belt, satisfies the above formulation. A more accurate formulation of our required periodicity requirement is:

   $$\mathsf{AG}\,[\mathit{putbox} \;\Rightarrow\; \neg\,\mathit{putbox}\,\mathsf{U}_{=25}\,\mathit{putbox}]$$

   This requirement specifies a delay of 25 time-units between two successive box placings, and in addition, requires that no such event occurs in between[2].

4. *Minimal delay* requirement: specifies a minimal delay between events. For instance, in order to ensure the safety of a railway system, the delay between two trains at a crossing should be at least 180 time units. Let *tac* be an atomic proposition that holds when the train is at the crossing. This minimal delay statement can be formalized by:

   $$\mathsf{AG}\,[\mathit{tac} \;\Rightarrow\; \neg\,\mathit{tac}\,\mathsf{U}_{\geqslant 180}\,\mathit{tac}]$$

   The reason for using the until-construct is similar to the previous case for periodicity.

5. *Interval delay* requirement: specifies that an event must occur within a certain interval after another event. Suppose, for instance, that in order to improve the throughput of the railway system one requires that trains should have a maximal distance of 900 time-units. The safety of the train system must be remained. Formally, we can easily extend the previous minimal delay requirement:

   $$\mathsf{AG}\,[\mathit{tac} \;\Rightarrow\; (\neg\,\mathit{tac}\,\mathsf{U}_{\geqslant 180}\,\mathit{tac} \;\wedge\; \neg\,\mathit{tac}\,\mathsf{U}_{\leqslant 900}\,\mathit{tac})]$$

   Alternatively, we could write

   $$\mathsf{AG}\,[\mathit{tac} \;\Rightarrow\; \neg\,\mathit{tac}\,\mathsf{U}_{=180}\,(\mathsf{AF}_{\leqslant 720}\,\mathit{tac})]$$

   It specifies that after a train at the crossing it lasts 180 time units (the safety requirement) before the next train arrives, and in addition this next train arrives within 720+180=900 time-units (the throughput requirement).

[2] The reader might wonder why such construction is not used for the punctuality requirement. This is due to the fact that r_m is unique, so $\mathit{receive}(r_m)$ can be valid at most once. This is not true for the predicate *putbox*, since putting boxes on the belt is supposed to be a repetitive activity.

4.6 Clock equivalence: the key to model checking real time

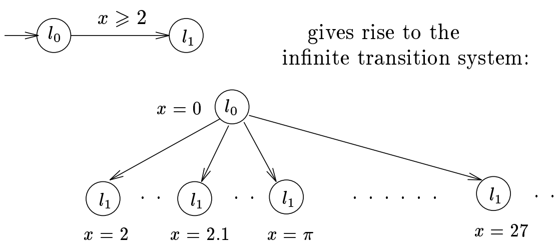
The satisfaction relation $\models$ for TCTL is not defined in terms of timed automata, but in terms of infinite transition systems. The model checking problem for timed automata can easily be defined in terms of $\mathcal{M}(\mathcal{A})$, the transition system underlying $\mathcal{A}$ (cf. Definition 42). Let the initial state of $\mathcal{M}(\mathcal{A})$ be $s_0 = (l_0, v_0)$ where l_0 is the initial location of $\mathcal{A}$ and v_0 the clock valuation that assigns value 0 to all clocks in $\mathcal{A}$. We can now formalize what it means for a timed automaton to satisfy a TCTL-formula:

Definition 48. (**Satisfiability for a timed automaton**)
For TCTL-formula ϕ and timed automaton $\mathcal{A}$ let

$$\mathcal{A} \models \phi \text{ if and only if } \mathcal{M}(\mathcal{A}), (s_0, w_0) \models \phi$$

where $w_0(y) = 0$ for all formula clocks y.

The model checking problem that we are interested in is to check whether a given timed automaton $\mathcal{A}$ satisfies some TCTL-formula ϕ. According to the last definition, this amounts to check whether $\mathcal{M}(\mathcal{A}), (s_0, w_0) \models \phi$. The basis for the model checking of $\mathcal{A}$ is thus the transition system $\mathcal{M}(\mathcal{A})$. The main problem, however, is that the state space of $\mathcal{M}(\mathcal{A})$, the set $L \times V(C)$, is *infinite*! This is exemplified by means of the following example:



How can we effectively check whether a given formula is valid for $\mathcal{M}(\mathcal{A})$, if an infinite number of states has to be considered?

From the above treatment of timed automata it is evident that the number of states of a timed automaton is infinite. The key idea by Alur and Dill (1990) that facilitates model checking of timed automata (and similar structures) is to introduce an appropriate equivalence relation, $\approx$ say, on clock valuations such that

Correctness: equivalent clock valuations satisfy the same TCTL-formulas, and

Finiteness: the number of equivalence classes under $\approx$ is finite.

The second property allows to replace an infinite set of clock valuations by a finite set of (sets of) clock valuations. In addition, the first property guarantees that for model $\mathcal{M}$ these clock valuations satisfy the same TCTL-formulas. That is to say, there does not exist a TCTL-formula that distinguishes the infinite-state and its equivalent finite-state clock valuation. Formally, for any timed automaton $\mathcal{A}$, this amounts to:

$$\mathcal{M}(\mathcal{A}), ((l, v), w) \models \phi \text{ if and only if } \mathcal{M}(\mathcal{A}), ((l, v'), w') \models \phi$$

for $v \cup w \approx v' \cup w'$.

The key observation that leads to the definition of this equivalence relation $\approx$ is that paths of timed automaton $\mathcal{A}$ starting at states which

- agree on the integer parts of all clock values, and
- agree on the ordering of the fractional parts of all clocks

are very similar. In particular, since time constraints in TCTL-formulas only refer to natural numbers, there is no TCTL-formula that distinguishes between these "almost similar" runs. Roughly speaking, two clock valuations are equivalent if they satisfy the aforementioned two constraints. Combined with the observation that

- if clocks exceed the maximal constant with which they are compared, their precise value is not of interest

the amount of equivalence classes thus obtained is not only denumerable, but also finite!

Since the number of equivalence classes is finite, and the fact that equivalent TCTL-models of timed automata satisfy the same formulas, this suggests to perform model checking on the basis of these classes. The finite-state model that thus results from timed automaton $\mathcal{A}$ is called its *region automaton*. Accordingly, equivalence classes under $\approx$ are called *regions*. Model checking a timed automaton thus boils down to model checking its associated finite region automaton. This reduced problem can be solved in a similar way as we have seen for model checking the branching temporal logic CTL in the previous chapter — by iteratively labelling states with sub-formulas of the property to be checked, as we will see later on. Thus, roughly speaking

Model checking a timed automaton against a TCTL*-formula amounts to model checking its region automaton against a* CTL*-formula.*

In summary we obtain the scheme in Table 4.1 for model checking the TCTL-formula ϕ over the timed automaton $\mathcal{A}$. Here we denote the equivalence class of clock valuation v under $\approx$ by $[v]$. (As we will see later on, the region automaton also depends on the clock constraints in the formula to be checked, but this dependency is omitted here for reasons of simplicity.)

1.	*Determine the equivalence classes (i.e. regions) under* $\approx$
2.	*Construct the region automaton* $\mathcal{R}(\mathcal{A})$
3.	*Apply the* CTL*-model checking procedure on* $\mathcal{R}(\mathcal{A})$
4.	$\mathcal{A} \models \phi$ *if and only if* $[s_0, w_0] \in Sat^R(\phi)$.

Table 4.1: Basic recipe for model checking TCTL over timed automata

In the next section we address the construction of the region automaton $\mathcal{R}(\mathcal{A})$. We start by discussing in detail the notion of equivalence $\approx$ indicated above. Since

this notion of equivalence is of crucial importance for the approach we extensively justify its definition by a couple of examples.

In the sequel we adopt the following notation for clock valuations. For clock valuation $v : C \longrightarrow \mathbb{R}^+$ and clock $x \in C$ let $v(x) = \lfloor x \rfloor + \mathit{frac}(x)$, that is, $\lfloor x \rfloor$ is the integral part of the clock value of x and $\mathit{frac}(x)$ the fractional part. For instance, for $v(x) = 2.134$ we have $\lfloor x \rfloor = 2$ and $\mathit{frac}(x) = 0.134$. The justification of the equivalence relation $\approx$ on clock valuations proceeds by four observations, that successively lead to a refinement of the notion of equivalence. Let v and v' be two clock valuations defined on the set of clocks C.

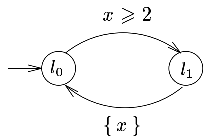


Figure 4.4: Clock valuations that agree on integer parts are equivalent

First observation: consider the timed automaton depicted in Figure 4.4 and the following states of this automaton: (l_0, v) and (l_0, v') with $v(x) = 3.2$ and $v'(x) = 3.7$. Clearly, in both states the edge from location l_0 to l_1 is enabled for any time instant exceeding 2. The fractional parts of $v(x)$ and $v'(x)$ do not determine its enabled-ness. Similarly, if $v(x) = 1.2$ and $v'(x) = 1.7$ the edge is disabled for both clock valuations and once more the fractional parts are irrelevant. Since in general clock constraints are of the form $x \sim c$ with $c \in \mathbb{N}$, only the integral parts of the clocks seem to be important. This leads to the first suggestion for clock equivalence:

$$v \approx v' \text{ if and only if } \lfloor v(x) \rfloor = \lfloor v'(x) \rfloor \text{ for all } x \in C. \tag{4.1}$$

This notion is rather simple, leads to a denumerable (but still infinite) number of equivalence classes, but is too coarse. That is, it identifies clock valuations that can be distinguished by means of TCTL-formulas.

Second observation: consider the timed automaton depicted in Figure 4.5 and the following states of this automaton: $s = (l_0, v)$ and $s' = (l_0, v')$ with $v(x) = 0.4$,

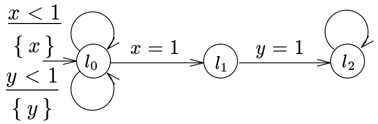


Figure 4.5: The ordering of clock valuations of different clocks is relevant

$v'(x) = 0.2$ and $v(y) = v'(y) = 0.3$. According to suggestion (4.1) we would have $v \approx v'$, since $\lfloor v(x) \rfloor = \lfloor v'(x) \rfloor = 0$ and similarly for clock y. But from state s the location l_2 can be reached, whereas this is not possible starting from state s'. This can be seen as follows. Consider s at 6 fractions of a time-unit later. Its clock valuation $v{+}0.6$ reads $v(x) = 1$ and $v(y) = 0.9$. Clearly, in this clock valuation the edge leading from l_0 to l_1 is enabled. If, after taking this edge, successively in location l_1 time progresses by 0.1 time-units, then the edge leading to location l_2 is enabled. A similar scenario does not exist for s'. In order to reach from s' location l_1 clock x needs to be increased by 0.8 time-units. But then $v'(y)$ equals 1.1, and the edge leading to l_2 will be permanently disabled. The important difference between v and v' is that $v(x) \geqslant v(y)$, but $v'(x) \leqslant v'(y)$. This suggests the following extension to suggestion (4.1):

$$v(x) \leqslant v(y) \text{ if and only if } v'(x) \leqslant v'(y) \text{ for all } x, y \in C. \tag{4.2}$$

Again, the resulting equivalence leads to a denumerable number of equivalence classes, but it is still too coarse.

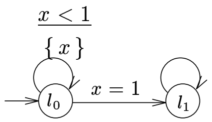


Figure 4.6: Integer-valued clocks should match

Third observation: consider the timed automaton of Figure 4.6 and the following states $s = (l_0, v)$ and $s' = (l_0, v')$ with $v(x) = 0$ and $v'(x) = 0.1$. According to suggestions (4.1) and (4.2) we would have $v \approx v'$, but, for instance, $s, w \models \mathsf{EF}_{=1}\, p$, but $s, wnot \models \mathsf{EF}_{=1}\, p$, where p is an atomic proposition that is only valid in location l_1. The main difference between v and v' is that clock x in s is exactly

0 whereas in state s' it just passed 0. This suggests to extend suggestions (4.1) and (4.2) with

$$\mathit{frac}(v(x)) = 0 \text{ if and only if } \mathit{frac}(v'(x)) = 0 \text{ for all } x \in C. \tag{4.3}$$

The resulting equivalence is not too coarse anymore, but still leads to an infinite number of equivalence classes and not to a finite one.

Fourth observation: let c_x be the maximum constant with which clock x is compared in a clock constraint (that occurs either in an enabling condition associated to an edge, or in an invariant) in the timed automaton at hand. Since c_x is the largest constant with which x is compared it follows that if $v(x) > c_x$ then the actual value of x is irrelevant; the fact that $v(x) > c_x$ suffices to decide the enabled-ness of all edges in the timed automaton.

In combination with the three conditions above, this observation leads to a *finite* number of equivalence classes: only the integral parts of clocks are of importance up to a certain bound ((4.1) plus the current observation) plus their fractional ordering (4.2) and the fact whether fractional parts are zero or not (4.3). These last two points now also are of importance only if the clocks have not exceed their bound. For instance, if $v(y) > c_y$ then the ordering with clock x is not of importance, since the value of y will not be tested anymore anyway.

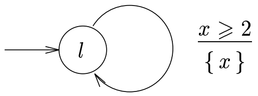
Finally, we come to the following definition of clock equivalence (Alur and Dill, 1994).

Definition 49. **(Clock equivalence)**
Let $\mathcal{A}$ be a timed automaton with set of clocks C and $v, v' \in V(C)$. Then $v \approx v'$ if and only if

1. $\lfloor v(x) \rfloor = \lfloor v'(x) \rfloor$ or $v(x) > c_x$ and $v'(x) > c_x$, for all $x \in C$, and
2. $\mathit{frac}(v(x)) \leqslant \mathit{frac}(v(y))$ iff $\mathit{frac}(v'(x)) \leqslant \mathit{frac}(v'(y))$ for all $x, y \in C$ with $v(x) \leqslant c_x$ and $v(y) \leqslant c_y$, and
3. $\mathit{frac}(v(x)) = 0$ iff $\mathit{frac}(v'(x)) = 0$ for all $x \in C$ with $v(x) \leqslant c_x$.

This definition can be modified in a straightforward manner such that $\approx$ is defined on a set of clocks C' such that $C \subseteq C'$. This would capture the case where C' can also included formula clocks (beside clocks in the automaton). For formula clock z in ϕ, the property at hand, c_z is the largest constant with which z is compared in ϕ. For instance, for formula z in $(AF[(p \wedge z \leqslant 3) \vee (q \wedge z > 5)])$, $c_z = 5$.

Example 35. *Consider the simple timed automaton depicted by:*



This automaton has a set of clocks $C = \{ x \}$ with $c_x = 2$, since the only clock constraint is $x \geqslant 2$. We gradually construct the regions of this automaton by considering each constraint of the definition of $\approx$ separately. Clock valuations v and v' are equivalent if $v(x)$ and $v'(x)$ belong to the same equivalence class on the real line. (In general, for n clocks this amounts to considering an n-dimensional hyper-space on $\mathbb{R}^+$.) For convenience let $[x \sim c]$ abbreviate $\{ x \mid x \sim c \}$ for natural c and comparison operator $\sim$.

1. *The requirement that $\lfloor v(x) \rfloor = \lfloor v'(x) \rfloor$ leads to the following partition of the real line:*

   $$[0 \leqslant x < 1], [1 \leqslant x < 2], [2 \leqslant x < 3], [3 \leqslant x < 4], \ldots$$

2. *Since $c_x = 2$, it is not interesting for the timed automaton at hand to distinguish between the valuations $v(x) = 3$ and $v'(x) = 27$. In summary, the equivalence classes that result from considering the first constraint of Definition 49 is*

   $$[0 \leqslant x < 1], [1 \leqslant x < 2], [x = 2], \text{ and } [x > 2]$$

3. *According to the second constraint, the ordering of clocks should be maintained. In our case, this constraint trivially holds since there is only one clock. So, there are no new equivalence classes introduced by considering this constraint.*

4. *The constraint that $\mathrm{frac}(v(x)) = 0$ iff $\mathrm{frac}(v'(x)) = 0$ if $v(x) \leqslant c_x$ leads to partitioning, for instance, the equivalence class $[0 \leqslant x < 1]$, into the classes $[x = 0]$ and $[0 < x < 1]$. Similarly, the class $[1 \leqslant x < 2]$ is partitioned. The class $[x > 2]$ is not partitioned any further since for this class the condition $v(x) \leqslant c_x$ is violated. As a result we obtain for the simple timed automaton the following 6 equivalence classes:*

   $$[x = 0], [0 < x < 1], [x = 1], [1 < x < 2], [x = 2], \text{ and } [x > 2].$$

(End of example.)

Example 36. *Consider the set of clocks $C = \{ x, y \}$ with $c_x = 2$ and $c_y = 1$. In Figure 4.7 we show the gradual construction of the regions by considering each constraint of the definition of $\approx$ separately. The figure depicts a partition of the two-dimensional space $\mathbb{R}^+ \times \mathbb{R}^+$. Clock valuations v and v' are equivalent if the real-valued pairs $(v(x), v(y))$ and $(v'(x), v'(y))$ are elements of the same equivalence class in the hyper-space.*

1. *The requirement that $\lfloor v(x) \rfloor = \lfloor v'(x) \rfloor$ for all clocks in C leads, for instance, to the equivalence classes $[(0 \leqslant x < 1), (0 \leqslant y < 1)]$ and $[(1 \leqslant x < 2), (0 \leqslant y < 1)]$ and so on. The constraint that $v(x) > c_x$ and $v'(x) > c_x$ for all clocks in C leads to the equivalence class $[(x > 2), (y > 1)]$. This means that for any clock valuation v for which $v(x) > 2$ and $v(y) > 1$ the exact values of x and y are irrelevant. The obtained equivalence classes are:*

   $$\begin{array}{ll} [(0 \leqslant x < 1), (0 \leqslant y < 1)] & [(1 \leqslant x < 2), (0 \leqslant y < 1)] \\ [(0 \leqslant x < 1), (y = 1)] & [(1 \leqslant x < 2), (y = 1)] \\ [(0 \leqslant x < 1), (y > 1)] & [(1 \leqslant x < 2), (y > 1)] \\ [(x = 2), (0 \leqslant y < 1)] & [(x > 2), (0 \leqslant y < 1)] \\ [(x = 2), (y = 1)] & [(x > 2), (y = 1)] \\ [(x = 2), (y > 1)] & [(x > 2), (y > 1)] \end{array}$$

   These 12 classes are depicted in Figure 4.7(a).

2. *Consider the equivalence class $[(0 \leqslant x < 1), (0 \leqslant y < 1)]$ that we obtained in the previous step. Since the ordering of the clocks now becomes important,*

this equivalence class is split into the classes $[(0 \leqslant x < 1), (0 \leqslant y < 1), (x < y)]$, $[(0 \leqslant x < 1), (0 \leqslant y < 1), (x = y)]$, *and* $[(0 \leqslant x < 1), (0 \leqslant y < 1), (x > y)]$. *A similar reasoning applies to the class* $[(1 \leqslant x < 2), (0 \leqslant y < 1)]$. *Other classes are not further partitioned. For instance, class* $[(0 \leqslant x < 1), (y = 1)]$ *does not need to be split, since the ordering of clocks* x *and* y *in this class is fixed,* $v(x) \leqslant v(y)$. *Class* $[(1 \leqslant x < 2), (y > 1)]$ *is not split, since for this class the condition* $v(x) \leqslant c_x$ *and* $v_y \leqslant c_y$ *is violated. Figure 4.7(b) shows the resulting equivalence classes.*

3. *Finally, we apply the third criterion of Definition 49. As an example consider the class* $[(0 \leqslant x < 1), (0 \leqslant y < 1), (x = y)]$ *that we obtained in the previous step. This class is now partitioned into* $[(x = 0), (y = 0)]$ *and* $[(0 < x < 1), (0 < y < 1), (x = y)]$. *Figure 4.7(c) shows the resulting 28 equivalence classes: 6 corner points, 14 open line segments, and 8 open regions.*

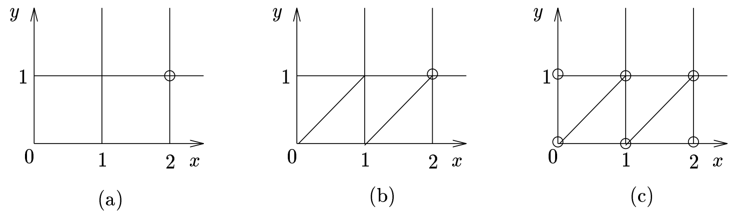


Figure 4.7: Partitioning of $\mathbb{R}^+ \times \mathbb{R}^+$ according to $\approx$ for $c_x = 2$ and $c_y = 1$

(End of example.)

4.7 Region automata

The equivalence classes under $\approx$ will be the basis for model checking timed automata. The combination of such class with a location is called a *region*.

Definition 50. (**Region**)
A *region* r is a pair $(l, [v])$ with location $l \in L$ and valuation $v \in V(C)$.

In the sequel we abbreviate $(l, [v])$ by $[s]$ for $s = (l, v)$. In addition we write $[s, w]$ as a shorthand for $(l, [v \cup w])$ for w a clock valuation over formula clocks.

Since there are finitely many equivalence classes under $\approx$, and any timed automaton possesses only a finite number of locations, the number of regions is finite. The next result says that states belonging to the same region satisfy the same TCTL-formulas. This important result for real-time model checking is due to Alur, Dill & Courcoubetis (1993).

Theorem 51.
Let $s, s' \in S$ *such that* $s, w \approx s', w'$. *For any* TCTL*-formula* ϕ, *we have:*

$$\mathcal{M}(\mathcal{A}), (s, w) \models \phi \text{ if and only if } \mathcal{M}(\mathcal{A}), (s', w') \models \phi.$$

According to this result we do not have to distinguish the equivalent states s and s', since there is no TCTL-formula that can distinguish between the two. This provides the correctness criterion for using equivalence classes under $\approx$, i.e. regions, as basis for model checking. Using regions as states we construct a finite-state automaton, referred to as the *region automaton*. This automaton consists of regions as states and transitions between regions. These transitions either correspond to the evolvement of time, or to the edges in the original timed automaton.

We first consider the construction of a region automaton by means of an example. Two types of transitions can appear between augmented regions. They are either due to (1) the passage of time (depicted as solid arrows), or (2) transitions of the timed automaton at hand (depicted as dotted arrows).

Example 37. Let us consider our simple timed automaton with a single edge, cf. Figure 4.8(a). Since the maximum constant that is compared with x *is 2, it follows* $c_x = 2$. *The region automaton is depicted in Figure 4.8(b). Since there is only one location, in each reachable region the timed automaton is in location* l. *The region automaton contains two transitions that correspond to the edge of the timed automaton. They both reach the initial region* $\mathbf{A}$. *The dotted transitions from* $\mathbf{E}$ *and* $\mathbf{F}$ *to* $\mathbf{A}$ *represent these edges. Classes in which there is a possibility* *to let time pass without bound while remaining in that class, are called unbounded classes, see below. In this example there is a single unbounded class, indicated by a grey shading, namely* **F**. *In* **F**, *the clock x can grow without bound while staying in the same region. (The reader is invited to investigate the changes to the region automaton when the simple timed automaton is changed as in Figure 4.1(c) and (e).)*

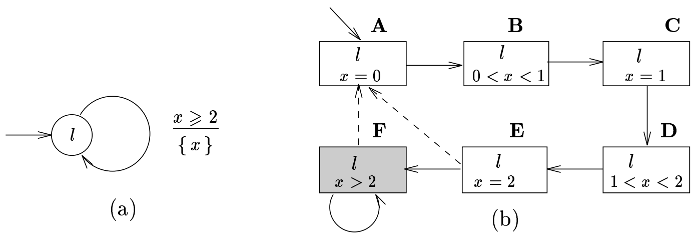


Figure 4.8: Region automaton of a simple timed automaton

Suppose now that we would like to model check a TCTL*-formula over the timed automaton of Figure 4.8(a) that contains a single formula clock z with $c_z = 2$, i.e. clock z is not compared in the formula with a constant larger than 2. The region automaton over $\{x, z\}$ is depicted in Figure 4.9. Notice that it is in fact the region automaton of before (cf. Figure 4.8(b)) extended with two "copies" of it: regions* **G** *through* **L** *and regions* **M** *through* **R**. *These "copies" are introduced for the constraints $z{-}x = 2$ and $z{-}x > 2$. Notice that the formula clock z is never reset. This is typical for a formula clock as there is no means in a formula to reset clocks once they are introduced.* *(End of example.)*

Definition 52. (**Delay-successor region**)
Let r, r' be two distinct regions (i.e. $r \neq r'$). r' is the *delay successor* of r, denoted $r' = \mathit{delsucc}(r)$, if there exists a $d \in \mathbb{R}^+$ such that for each $r = [s]$ we have $s \xrightarrow{d} s'$ and $r' = [s{+}d]$ and for all $0 \leqslant d' < d : [s{+}d'] \in r \cup r'$.

Here, for $s = (l, v)$, state $s{+}d$ denotes $(l, v{+}d)$. In words: $[s{+}d]^*$ is the delay-successor of $[s]^*$ if for some *positive* d any valuation in $[s]^*$ moves to a valuation in $[s{+}d]^*$, without having the possibility to leave these two regions at any earlier

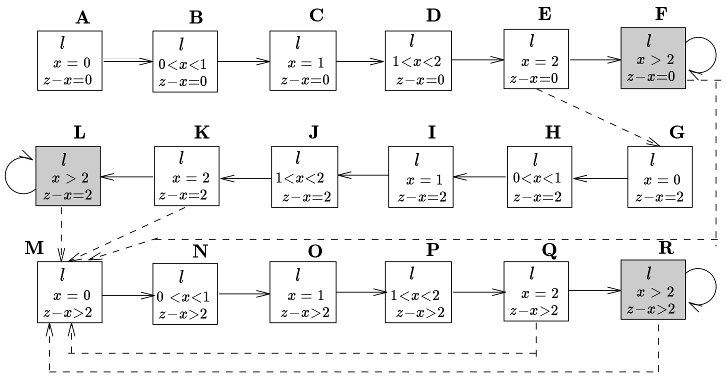


Figure 4.9: Region automaton of a simple timed automaton for formula clock z with $c_z = 2$

point in time. Observe that for each region there is at most one delay-successor. This should not surprise the reader, since delay-successors correspond to the advancing of time, and time can advance only deterministically.

Example 38. *Consider the equivalence classes of the timed automaton of Figure 4.8(a). The regions containing clock x and extra clock z (cf. Figure 4.9) are partitions of the two-dimensional real-space. Timed transitions correspond to upward moves along the diagonal line $x = z$. For instance, $[x = z = 1]$ is the delay-successor of $[(0 < x < 1), (x = z)]$, and the delay-successor of $[(1 < x < 2), (x = z)]$ has successor $[x = z = 2]$. Class $[(x = 1), (z > 2)]$ is not the delay-successor of $[(0 < x < 1), (z = 2)]$ — a unreachable region that is not depicted — since there is some real value d' such that $[(0 < x < 1), (z > 2)]$ is reached in between.* *(End of example.)*

Definition 53. (**Unbounded region**)
Region r is an *unbounded region* if for all clock valuations v such that $r = [v]$ we have $v(x) > c_x$ for all $x \in C$.

In an unbounded region all clocks have exceed the maximum constant with which they are compared, and hence all clocks may grow without bound. In

Figure 4.8(b) and 4.9 we have indicated the unbounded region by a grey shading of the region.

A region automaton now consists of a set of states (the regions), an initial state, and two transition relations: one corresponding to delay transitions, and one corresponding to the edges of the timed automaton at hand.

Definition 54. (**Region automaton**)
For timed automaton $\mathcal{A}$ and set of time constraints Ψ (over C and the formula clocks), the *region automaton* $\mathcal{R}(\mathcal{A}, \Psi)$ is the transition system $(R, r_0, \longrightarrow)$ where

- $R = S/_{\approx} = \{ [s] \mid s \in S \}$
- $r_0 = [s_0]$
- $r \longrightarrow r'$ if and only if $\exists\, s, s'.(r = [s] \;\wedge\; r' = [s'] \;\wedge\; s \xrightarrow{*} s')$
- $r \longrightarrow r'$ if and only if
  1. r is an unbounded region and $r = r'$, or
  2. $r \neq r'$ and $r' = \mathit{delsucc}(r)$.

Observe that from this definition it follows that unbounded regions are the only regions that have a self-loop consisting of a delay transition.

Example 39. To illustrate the construction of a region automaton we consider a

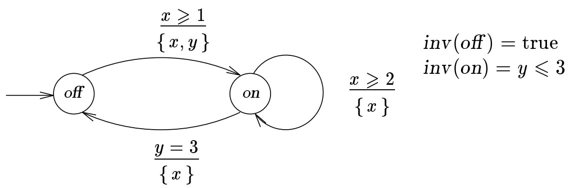


Figure 4.10: Timed automaton for modified light switch

slight variant of the light switch of Example 29, that is depicted in Figure 4.10. In order to show the effect of invariants we have equipped location on with invariant

$y \leqslant 3$. In order to keep size of the region automaton manageable we have adapted the enabling conditions of the transitions that change location; in particular we have made the constants smaller.

The region automaton $\mathcal{R}(\mathcal{A}, \Psi)$ where $\mathcal{A}$ is the modified light switch, is depicted in Figure 4.11. The set of time constraints is $\Psi = \{ x \geqslant 1, x \geqslant 2, y = 3, y \leqslant 3 \}$, i.e. the set of all enabling constraints and invariants in $\mathcal{A}$. For simplicity no formula clocks are considered. We have for example $\mathbf{D} \longrightarrow \mathbf{E}$ since there is a transition from state $s = (\mathit{off}, v)$ with $v(x) = v(y) > 1$ to state $s' = (\mathrm{on}, v')$ with $v'(x) = v'(y) = 0$ and $\mathbf{D} = [s]$, $\mathbf{E} = [s']$. There is a delay transition $\mathbf{D} \longrightarrow \mathbf{D}$, since the region $[v]$ where $v(x) = v(y) > 1$ is an unbounded region. This stems from the fact that in location off time can gro without any bound, i.e. $inv(\mathit{off}) = \mathit{true}$.

The reader is invited to check how the region automaton has to be changed when changing $\mathrm{inv}(\mathit{off})$ into $y \leqslant 4$. (End of example.)

4.8 Model checking region automata

Given the region automaton $\mathcal{R}(\mathcal{A}, \Psi)$ for timed automaton $\mathcal{A}$ and TCTL-formula ϕ with clock constraints Ψ in $\mathcal{A}$ and ϕ, the model checking algorithm now proceeds as for untimed CTL, see previous chapter. Let us briefly summarise the idea of labelling. The basic idea of the model checking algorithm is to *label* each state (i.e. augmented region) in the region automaton with the sub-formulas of ϕ that are valid in that state. This labelling procedure is performed iteratively starting by labelling the states with the sub-formulas of length 1 of ϕ, i.e. the atomic propositions (and true and false) that occur in ϕ. In the $(i{+}1)$-th iteration of the labelling algorithm sub-formulas of length $i{+}1$ are considered and the states are labelled accordingly. To that purpose the labels already assigned to states are used, being sub-formulas of ϕ of length at most i ($i \geqslant 1$). The labelling algorithm ends by considering the sub-formula of length $\mid \phi \mid$, ϕ itself. This algorithm is listed in Table 4.2.

The model checking problem $\mathcal{A} \models \phi$, or, equivalently, $\mathcal{M}(\mathcal{A}), (s_0, w_0) \models \phi$, is

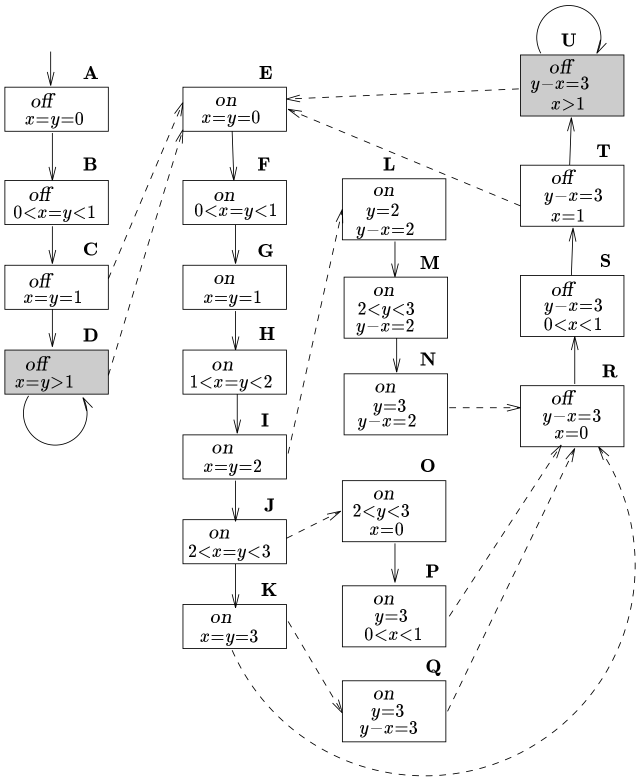


Figure 4.11: Region automaton of the modified light switch

function $Sat^R\,(\phi : Formula)$: **set of** *Region*;
(* precondition: true *)
begin
  if $\phi = \text{true} \longrightarrow$ **return** $S/_\approx$
  $[]\ \phi = \text{false} \longrightarrow$ **return** $\varnothing$
  $[]\ \phi \in AP \longrightarrow$ **return** $\{\, [s,w]_\approx \mid \phi \in Label(s) \,\}$
  $[]\ \phi = \alpha \longrightarrow$ **return** $\{\, [s,w]_\approx \mid s = (l,v) \ \wedge\ v \cup w \models \alpha \,\}$
  $[]\ \phi = \neg\,\phi_1 \longrightarrow$ **return** $S/_\approx - Sat^R(\phi_1)$
  $[]\ \phi = \phi_1 \ \vee\ \phi_2 \longrightarrow$ **return** $(Sat^R(\phi_1) \cup Sat^R(\phi_2))$
  $[]\ \phi = z.\phi_1 \longrightarrow$ **return** $\{\, [s,w]_\approx \mid (s,[z]w) \in Sat^R(\phi_1) \,\}$
  $[]\ \phi = \mathsf{E}\,[\phi_1 \,\mathsf{U}\, \phi_2] \longrightarrow$ **return** $Sat^R_{EU}(\phi_1, \phi_2)$
  $[]\ \phi = \mathsf{A}\,[\phi_1 \,\mathsf{U}\, \phi_2] \longrightarrow$ **return** $Sat^R_{AU}(\phi_1, \phi_2)$
  fi
(* postcondition: $Sat^R(\phi) = \{\, [s,w]_\approx \mid \mathcal{M}, (s,w) \models \phi \,\}$ *)
end

Table 4.2: Outline of main algorithm for model checking TCTL

now solved by considering the labelling:

$$\mathcal{M}(\mathcal{A}), (s_0, w_0) \models \phi \text{ if and only if } [s_0, w_0] \in Sat^R(\phi).$$

(The correctness of this statement follows from Theorem 55).

The computation of $Sat^R(\phi)$ is done by considering the syntactical structure of ϕ. The cases for the propositional formulas (true, false, negation, disjunction and atomic propositions) is identical to the case for CTL, cf. Chapter 3. For clock constraint α, $Sat^R(\alpha)$ is the set of regions $[s,w]$ such that α is satisfied by w and v, the clock valuation of the automata clocks in s. Region $[s,w]$ belongs to set $Sat^R(z \text{ in } \phi)$ if it satisfies ϕ for w' where $w'(z) = 0$ (start the formula clock at value 0) and $w'(x) = w(x)$ for $x \neq z$. For the until-formulas, auxiliary functions are defined. These functions are listed in Table 4.3 and 4.4. The code for the function Sat^R_{EU} is identical to the untimed case. For Sat^R_{AU} there is small, though essential, difference with the untimed algorithm for $\mathsf{A}\,[\phi \,\mathsf{U}\, \psi]$. In analogy to the case for CTL, the iteration would contain the statement:

$$Q := Q \cup \big(\{\, s \mid \forall\, s' \in Q.\, s \longrightarrow s' \,\} \cap \mathit{Sat}^R(\phi)\big)\,.$$

The correctness of this statement, however, assumes that each state s has some successor under $\longrightarrow$. This is indeed valid for CTL, since in a CTL-model each state has at least one successor. In region automata, regions may exist that have no outgoing transition, neither a delay transition nor a transition that corresponds to an edge in the timed automaton. Therefore, we take a different approach and extend Q with those regions that have at least one transition into Q, and for which all direct successors are in Q. This yields the code listed in Table 4.4. An example of a region automaton containing a region without any successors is depicted in Figure 4.12. Region **C** does not have any successors: delaying is not possible due to the invariant of l, and there is no edge that is enabled for valuation $x = 1$.

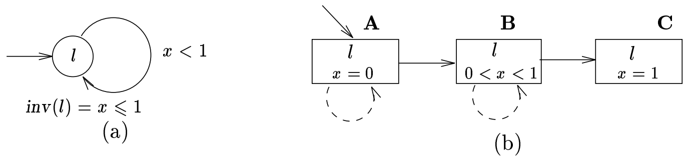


Figure 4.12: Region automaton with a region without successors

Example 40. Consider the region automaton (cf. Figure 4.11) of the modified light switch, and suppose location on is labelled with atomic proposition q and off with p. We can now check that it is impossible to reach on from off in less than one time-unit:

$$\mathcal{M}(\mathcal{A}), (s_0, w_0) \not\models \mathsf{E}\,[p\, \mathsf{U}_{<1}\, q]$$

since there is no path through the region automaton that starts in **A** *(which corresponds to $[s_0, w_0]$) to some region* **E** *through* **K***, where the light is switched on, and that lasts less than one time-unit. Formally, this can be seen as follows. Formula $\mathsf{E}\,[p\,\mathsf{U}_{<1}\, q]$ abbreviates z* in $\mathsf{E}\,[(p \,\wedge\, z < 1)\,\mathsf{U}\, q]$*. Equip the region automaton*

function $\mathit{Sat}^R_{EU}\,(\phi, \psi : \mathit{Formula})$: **set of** *Region*;
(* precondition: true *)
begin var Q, Q' : **set of** *Region*;
    $Q, Q' := \mathit{Sat}^R(\psi), \varnothing$;
    do $Q \neq Q' \longrightarrow$
        $Q' := Q$;
        $Q := Q \cup \big(\{\, s \mid \exists\, s' \in Q.\, s \longrightarrow s' \,\} \cap \mathit{Sat}^R(\phi)\big)$
    od;
    return Q
(* postcondition: $\mathit{Sat}_{EU}(\phi, \psi) = \{\, [s, w] \mid s, w \models \mathsf{E}\,[\phi\, \mathsf{U}\, \psi] \,\}$ *)
end

Table 4.3: Model checking $\mathsf{E}\,[\phi\,\mathsf{U}\,\psi]$ over regions

function $\mathit{Sat}^R_{AU}\,(\phi, \psi : \mathit{Formula})$: **set of** *Region*;
(* precondition: true *)
begin var Q, Q' : **set of** *Region*;
    $Q, Q' := \mathit{Sat}^R(\psi), \varnothing$;
    do $Q \neq Q' \longrightarrow$
        $Q' := Q$;
        $Q := Q \cup \big(\{\, s \mid \exists\, s' \in Q.\, s \longrightarrow s' \;\wedge\; (\forall\, s' \in Q.\, s \longrightarrow s') \,\} \cap \mathit{Sat}^R(\phi)\big)$
    od;
    return Q
(* postcondition: $\mathit{Sat}_{AU}(\phi, \psi) = \{\, [s, w] \mid s, w \models \mathsf{A}\,[\phi\, \mathsf{U}\, \psi] \,\}$ *)
end

Table 4.4: Model checking $\mathsf{A}\,[\phi\,\mathsf{U}\,\psi]$ over regions

with a formula clock z, that is reset in **A**, *and keeps advancing ad infinitum. The earliest time to switch the light on (i.e. to make q valid) is to take the transition from* **C** *to* **E**. *However, then z equals 1, and thus the property is invalid. (The reader is invited to check this informal reasoning by applying the algorithms to the region automaton.)* *(End of example.)*

Recall that a timed automaton is non-Zeno if from every state there is at least one time-divergent path, a path where time grows ad infinitum, does exist. Non-Zenoness is a necessary pre-condition for the model checking procedures given above, as follows from the following result (Yovine, 1998).

Theorem 55.
For $\mathcal{A}$ a non-Zeno timed automaton: $(s, w) \in Sat(\phi)$ if and only if $[s, w] \in Sat^R(\phi)$.

Thus the correctness of the model checking algorithm is only guaranteed if the timed automaton at hand is non-Zeno. Non-Zenoness of $\mathcal{A}$ can be checked by showing that from each state of $\mathcal{M}(\mathcal{A})$ time can advance by exactly one time-unit:

Theorem 56.
$\mathcal{A}$ is non-Zeno if and only if for all states $s \in S : \mathcal{M}(\mathcal{A}), s \models \mathsf{EF}_{=1}$ true.

Time complexity

The model checking algorithm labels the region automaton $\mathcal{R}(\mathcal{A}, \Psi)$ for timed automaton $\mathcal{A}$ and set of clock constraint Ψ. We know from Chapter 3 that the worst-case time complexity of such algorithm is proportional to $|\,\phi\,| \times |\,S\,|^2$, where S is the set of states of the automaton to be labelled. The number of states in $\mathcal{R}(\mathcal{A}, \Psi)$ equals the number of regions of $\mathcal{A}$ with respect to Ψ. The number of regions is proportional to the product of the number of locations in $\mathcal{A}$ and the number of equivalence classes under $\approx$. Let Ψ be a set of clock constraints over C' where $C \subseteq C'$, i.e. C' contains the clocks of $\mathcal{A}$ plus some formula clocks. For

$n = |\,C'\,|$, the number of regions is

$$\mathcal{O}\left(n! \times 2^n \times \prod_{x \in \Psi} c_x \times |\,L\,|\right).$$

Thus,

> *The worst-case time complexity of model checking* TCTL*-formula ϕ over timed automaton $\mathcal{A}$, with the clock constraints of ϕ and $\mathcal{A}$ in Ψ is:*
> $\mathcal{O}\left(|\,\phi\,| \times (n! \times 2^n \times \prod_{x \in \Psi} c_x \times |\,L\,|^2)\right).$

That is, model checking TCTL is

(i) linear in the length of the formula ϕ

(ii) exponential in the number of clocks in $\mathcal{A}$ and ϕ

(iii) exponential in the maximal constants with which clocks are compared in $\mathcal{A}$ and ϕ.

Using the techniques that we have discussed in Chapter 3, the time complexity can be reduced to being quadratic in the number of locations.

The lower-bound for the complexity of model checking TCTL for a given timed automaton is known to be PSPACE-hard (Alur, Courcoubetis & Dill, 1993). This means that at least a memory is needed of a size that is polynomial in the size of the system to be checked.

Fairness

As for untimed systems, in verifying time-critical systems we are often only interested in considering those executions in which enabled transitions are taken in a fair way, see Chapter 3. In order to be able to deal with fairness, we have seen that for the case with CTL the logic is interpreted over a fair CTL-model. A fair model has a new component, a set $\mathcal{F} \subseteq 2^L$ of fairness constraints. An analogous recipe can be followed for TCTL. We will not work out all details here, but the basic idea is to enrich a TCTL-model with a set of fairness constraints, in a similar way as for CTL. A fair path of a timed automaton is defined as a path in which for every set $F_i \in \mathcal{F}$ there are infinitely many locations in the run that are in F_i (like a generalized Büchi acceptance condition). The semantics of TCTL is now similar to the semantics provided earlier in this chapter, except that all path quantifiers are interpreted over $\mathcal{F}$-fair paths rather than over all paths. In an analogous manner as before, the region automaton is constructed. The fairness is incorporated in the region automaton by replacing each $F_i \in \mathcal{F}$ by the set $\{ (l, [v]) \mid l \in F_i \}$. The labelling algorithms can now be adapted to cover only $\mathcal{F}$-fair paths in the region automaton. Due to the consideration of fairness constraints, the worst-case time complexity of the labelling algorithm increases by a multiplicative factor that is proportional to the cardinality of $\mathcal{F}$.

Overview of TCTL model checking

We conclude this section by providing an overview of model checking a timed automaton against a TCTL-formula. Notice that the region automaton depends only on the clock constraints (i.e. the formula and automata clocks and the maximal constants with which clocks are compared) appearing in the formula ϕ, it does not depend on the formula itself. The labelled region automaton, of course, depends on the entire formula ϕ. The reader is invited to check the resemblance of this schema with the overview of CTL model checking in Chapter 3.

4.9 The model checker UPPAAL

UPPAAL is a tool suite for symbolic model checking of real-time systems (Larsen, Pettersson & Yi, 1997) developed at the University of Uppsala (Sweden) and Aalborg (Denmark). Besides model checking, it also supports simulation of timed automata and has some facilities to detect deadlocks. UPPAAL has been applied

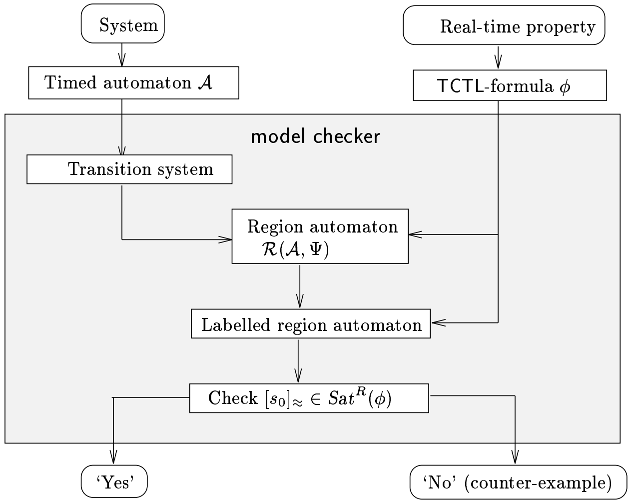


Figure 4.13: Overview of model checking TCTL over a timed automaton

to several industrial case studies such as real-time protocols, multi-media synchronization protocols and proving the correctness of electric power plants. The model checker is available under `http://www.docs.uu.se/docs/rtmv/uppaal/`. An overview of the model checker UPPAAL is given in Figure 4.14.

For reasons of efficiency, the model checking algorithms that are implemented in UPPAAL are based on (sets of) clock constraints [3], rather than on explicit (sets of) regions. By dealing with (disjoint) sets of clock constraints, a coarser partitioning of the (infinite) state space is obtained. Working with clock constraints allows to characterize $Sat(\phi)$ without explicitly building the region automaton a priori (Yovine, 1998).

UPPAAL facilitates the graphical description of timed automata by using the tool AUTOGRAPH. The output of AUTOGRAPH is compiled into textual format (using component `atg2ta`), which is checked (by `checkta`) for syntactical correctness. This textual representation is one of the inputs to UPPAAL's verifier

[3] Here, the property that each region can be expressed by a clock constraint over the clocks involved, is exploited.

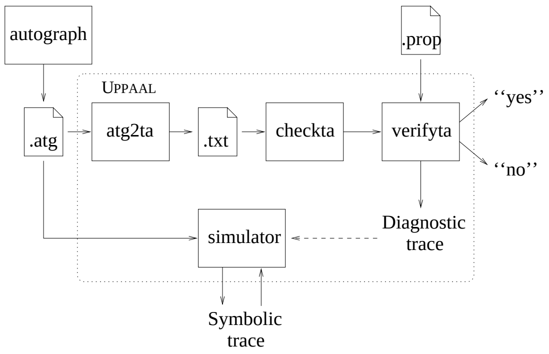


Figure 4.14: The structure of the model checker UPPAAL

`verifyta`. The verifier can be used to determine the satisfaction of a given real-time property with respect to a timed automaton. If a property is not satisfied, a diagnostic trace can be generated that indicates how the property may be violated. UPPAAL also provides a simulator that allows a graphical visualization of possible dynamic behaviors of a system description (i.e., a symbolic trace). The diagnostic trace, generated in case a property is violated, can be fed back to the simulator so that it can be analyzed with the help of the graphical presentation.

To improve the efficiency of the model checking algorithms, UPPAAL does not support the full expressivity of TCTL, but concentrates on a subset of it that is suitable for specifying safety properties. Properties are terms in the language defined by the following syntax:

$$\phi ::= \mathsf{AG}\,\psi \;\big|\; \mathsf{EF}\,\psi \qquad\qquad \psi ::= a \;\big|\; \alpha \;\big|\; \psi \wedge \psi \;\big|\; \neg\,\psi$$

where a is a location of a timed automaton (like `A.l` for automaton `A` and location `l`), and α a simple linear constraint on clocks or integer variables of the form `x` $\sim$ `n` for $\sim$ a comparison operator, `x` a clock or integer variable, and `n` an integer. Notice the restricted use of AG and EF as operators: they are only allowed as "top-level" operators, and cannot be nested. This use of path operators limits the expressiveness of the logic, but turns out to pose no significant practical restrictions and makes the model checking algorithm easier. Also notice that

formula clocks are not part of the supported logic; only automata clocks can be referred to. Bounded liveness properties can be checked by checking the timed automaton versus a test automaton.

In a nutshell, the most important ingredients for specifying systems in UPPAAL are:

- *Networks of timed automata.* The system specification taken by UPPAAL as input, consists of a *network* of timed automata. Such network is described as the composition of timed automata that can communicate via (equally labelled) channels. If $\mathcal{A}_1$ through $\mathcal{A}_N$ ($N \geqslant 0$) are timed automata then

  $$\mathcal{A} = \mathcal{A}_1 \,\|\, \mathcal{A}_2 \,\|\, \ldots \,\|\, \mathcal{A}_N$$

  specifies the network of timed automata composed of $\mathcal{A}_1$ through $\mathcal{A}_N$. The communication between components is synchronous. Send and receive statements are labels of edges of a timed automaton. A send statement over channel `a`, denoted `a!`, is enabled if the edge is enabled (that is, the associated clock constraint is satisfied in the current clock valuation), and if there is a corresponding transition (in another timed automaton) labelled with the input statement `a?` that is enabled as well. For instance, the two timed automata depicted in Figure 4.15 can communicate along channel `a` if clock `x` reads more than value 2 and clock `y` reads at most 4. If there is no execution of the two timed automata that fulfills this conjunction of constraints, the entire system — including the components that are not involved in the interaction — halts. Since communication is synchronous,

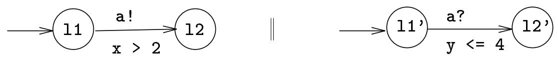
  

  Figure 4.15: Example of communication between timed automata

  in case of a synchronization both automata involved take their transition labelled `a!` and `a?` as one atomic transition, and afterwards the system is in locations `l2` and `l2'`, respectively.

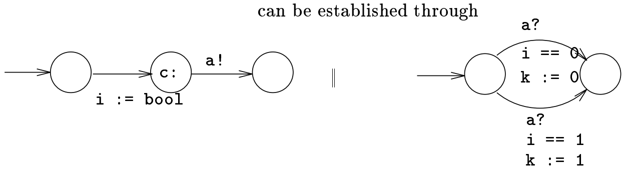
- *Clock constraints and invariants* are conjunctions of atomic constraints which have the form `x` $\sim$ `n` where `x` is a variable (a clock or an integer), `n` a non-negative integer, and $\sim \in$ { `<, <=, =, >=, >` } a comparison operator.

- *Data.* The use of data in UPPAAL (version 1.99) is restricted to clocks and integers. Variables can be either local or global, i.e. shared variables are possible. Assignments to integer variable `i` must have the form `i:=n1*i+n2`. Notice that for the latter assignments the variable on the right-hand side of the assignment should be the same as the variable on the left-hand side.

- *Committed locations* are locations that are executed *atomically* and *urgently*. Atomicity of a committed location forbids interference in the activity that is taking place involving the committed location, i.e. the entering of a committed location followed by leaving it constitutes a single atomic event: no interleaving can take place of any other transition (of any other timed automaton). Urgency means that it is not allowed to delay for any positive amount of time in a committed location. Committed locations can thus never be involved in a delay transition. Committed locations are indicated by the prefix `c:`.

- *Value passing.* UPPAAL (version 1.99) does not include mechanisms for value passing at synchronization. Value passing can be effectively modeled by means of assignments to variables. Committed locations are used to ensure that the synchronization between components and the assignment to variables — that establishes the value passing — is carried out atomically. This is exemplified by:

can be established through



Here, the distributed assignment `k := bool` is established.

Philips' bounded retransmission protocol

To illustrate the capabilities of real-time model checking we treat (part of) the model checking of an industrially relevant protocol, known as the *bounded retransmission protocol* (BRP, for short). This protocol has been developed by Philips Electronics B.V. is currently under standardization, and is used to transfer bulks of data (files) via an infra-red communication medium between audio/video equipment and a remote control unit. Since the communication medium is rather unreliable, the data to be transferred is split into small units, called chunks. These chunks are transmitted separately, and on failure of transmission — as indicated by the absence of an acknowledgement — a retransmission is issued. If, however, the number of retransmissions exceeds a certain threshold, it is assumed that there is a serious problem in the communication medium, and the transmission of the file is aborted. The timing intricacies of the protocol are twofold:

1. firstly, the sender has a timer that is used to initiate a retransmission in case an acknowledgement comes "too late", and

2. secondly, the receiver has a timer to detect the abortion (by the sender) of a transmission in case a chunk arrives "too late".

The correctness of the protocol clearly depends on the precise interpretation of "too late", and in this case study we want to determine clear and tight bounds on these timers in order for the BRP to work correctly. This case study is more extensively described in (D'Argenio, Katoen, Ruys & Tretmans, 1997).

What is the protocol supposed to do?

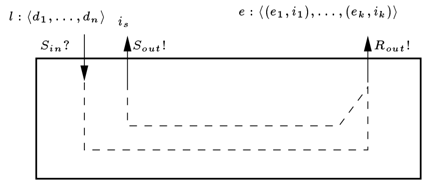
As for many transmission protocols, the service delivered by the BRP behaves like a buffer, i.e., it reads data from one client to be delivered at another one. There are two features that make the behavior much more complicated than a simple buffer. Firstly, the input is a *large file* (that can be modeled as a list), which is delivered in small chunks. Secondly, there is a *limited amount of time* for each chunk to be delivered, so we cannot guarantee an eventually successful delivery within the given time bound. It is assumed that either an initial part of the file or the whole file is delivered, so the chunks will not be garbled and their order will not be changed. Both the sender and the receiver obtain an *indication* whether the whole file has been delivered successfully or not.

The input (the list $l = d_1, \ldots, d_n$) is read on the "input" port. Ideally, each d_i is delivered on the "output" port. Each chunk is accompanied by an indication. This indication can be I_FST, I_INC, I_OK, or I_NOK. I_OK is used if d_i is the last element of the file. I_FST is used if d_i is the first element of the file *and more will follow*. All other chunks are accompanied by I_INC. However, when something goes wrong, a "not OK" indication (I_NOK) is delivered without datum. Note that the receiving client does not need a "not OK" indication before delivery of the first chunk nor after delivery of the last one.

The sending client is informed after transmission of the whole file, or when the protocol gives up. An indication is sent out on the "input" port. This indication can be I_OK, I_NOK, or I_DK. After an I_OK or an I_NOK indication, the sender can be sure, that the receiver has the corresponding indication. A "don't know" indication I_DK may occur after delivery of the last-but-one chunk d_{n-1}. This situation arises, because no realistic implementation can ensure whether the last chunk got lost. The reason is that information about a successful delivery has to be transported back somehow over the same unreliable medium. In case the last acknowledgement fails to come, there is no way to know whether the last chunk d_n has been delivered or not. After this indication, the protocol is ready to transmit a subsequent file.

This completes the informal FTS (*File Transfer Service*) description. Remark that it is unclear from this description which indication the sending client receives in case the receiving client does not receive any chunk. Since something went wrong an I_NOK indication is required, but from this indication the sending client may not deduce that the receiving client has the corresponding indication. This is because the receiving client does not receive an I_NOK indication before delivery of the first chunk. So, if the sending client receives an I_NOK either the receiving client received the same or did not receive anything at all.

Formal specification of the FTS (optional)



Schematic view of the FTS

Signatures of the input and output:

$S_{in} : l = \langle d_1, \ldots, d_n \rangle$ for $n > 0$

$S_{out} : i_s \in \{\, \text{I\_OK}, \text{I\_NOK}, \text{I\_DK} \,\}$

$R_{out} : \langle (e_1, i_1), \ldots, (e_k, i_k) \rangle$ for $0 \leqslant k \leqslant n, i_j \in \{\, \text{I\_FST}, \text{I\_INC}, \text{I\_OK}, \text{I\_NOK} \,\}$ for $0 < j \leqslant k$.

The FTS is considered to have two "service access points": one at the sender side and the other at the receiver side. The sending client inputs its file via S_{in} as a list of chunks $\langle d_1, \ldots, d_n \rangle$. We assume that $n > 0$, i.e., the transmission of empty files is not considered. The sending client receives indications i_s via S_{out}, while the receiving client receives pairs (e_j, i_j) of chunks and indications via R_{out}. We assume that all outputs with respect to previous files have been completed when a next file is input via S_{in}.

In Table 4.5 we specify the FTS in a logical way, i.e., by stating properties that should be satisfied by the service. These properties define relations between input and output. Note that a distinction is made between the case in which the receiving client receives at least one chunk ($k > 0$) and the case that it receives none ($k = 0$). A protocol conforms to the FTS if it satisfies all listed properties.

For $k > 0$ we have the following requirements. **(1.1)** states that each correctly received chunk e_j equals d_j, the chunk sent via S_{in}. In case the notification i_j indicates that an error occurred, no restriction is imposed on the accompanying

Table 4.5: Formal specification of the FTS

	$k > 0$
(1.1)	$\forall 0 < j \leqslant k : i_j \neq \text{I\_NOK} \;\Rightarrow\; e_j = d_j$
(1.2)	$n > 1 \;\Rightarrow\; i_1 = \text{I\_FST}$
(1.3)	$\forall 1 < j < k : i_j = \text{I\_INC}$
(1.4.1)	$i_k = \text{I\_OK} \;\vee\; i_k = \text{I\_NOK}$
(1.4.2)	$i_k = \text{I\_OK} \;\Rightarrow\; k = n$
(1.4.3)	$i_k = \text{I\_NOK} \;\Rightarrow\; k > 1$
(1.5)	$i_s = \text{I\_OK} \;\Rightarrow\; i_k = \text{I\_OK}$
(1.6)	$i_s = \text{I\_NOK} \;\Rightarrow\; i_k = \text{I\_NOK}$
(1.7)	$i_s = \text{I\_DK} \;\Rightarrow\; k = n$
	$k = 0$
(2.1)	$i_s = \text{I\_DK} \;\Leftrightarrow\; n = 1$
(2.2)	$i_s = \text{I\_NOK} \;\Leftrightarrow\; n > 1$

chunk e_j. **(1.2)** through **(1.4)** address the constraints concerning the received indications via R_{out}, i.e., i_j. If the number n of chunks in the file exceeds one then **(1.2)** requires i_1 to be I_FST, indicating that e_1 is the first chunk of the file and more will follow. **(1.3)** requires that the indications of all chunks, apart from the first and last chunk, equal I_INC. The requirement concerning the last chunk (e_k, i_k) consists of three parts. **(1.4.1)** requires e_k to be accompanied with either I_OK or I_NOK. **(1.4.2)** states that if i_k = I_OK then k should equal n, indicating that all chunks of the file have been received correctly. **(1.4.3)** requires that the receiving client is not notified in case an error occurs before delivery of the first chunk. **(1.5)** through **(1.7)** specify the relationship between indications given to the sending and receiving client. **(1.5)** and **(1.6)** state when the sender and receiver have corresponding indications. **(1.7)** requires a "don't know" indication to only appear after delivery of the last-but-one chunk d_{n-1}. This means that the number of indications received by the receiving client must equal n. (Either this last chunk is received correctly or not, and in both cases an indication (+ chunk) is present at R_{out}.)

For $k = 0$ the sender should receive an indication I_DK if and only if the file to be sent consists of a single chunk. This corresponds to the fact that a "don't know" indication may occur after the delivery of the last-but-one chunk only. For $k = 0$ the sender is given an indication I_NOK if and only if n exceeds one. This gives rise to **(2.1)** and **(2.2)**.

Remark that there is no requirement concerning the limited amount of time available to deliver a chunk to the receiving client as mentioned in the informal service description. The reason for this is that this is considered as a protocol requirement rather than a service requirement.

How is the protocol supposed to work?

The protocol consists of a sender S equipped with a timer T_1, and a receiver R equipped with a timer T_2 which exchange data via two unreliable (lossy) channels, K and L.

Sender S reads a file to be transmitted and sets the retry counter to 0. Then it starts sending the elements of the file one by one over K to R. Timer T_1 is set and a frame is sent into channel K. This frame consists of three bits and a datum (= chunk). The first bit indicates whether the datum is the first element of the file. The second bit indicates whether the datum is the last item of the file. The third bit is the so-called alternating bit, that is used to guarantee that data is not duplicated. After having sent the frame, the sender waits for an acknowledgement from the receiver, or for a timeout. In case an acknowledgement arrives, the timer T_1 is reset and (depending on whether this was the last element of the file) the sending client is informed of correct transmission, or the next element of the file is sent. If timer T_1 times out, the frame is resent (after the counter for the number of retries is incremented and the timer is set again), or the transmission of the file is broken off. The latter occurs if the retry counter exceeds its maximum value MAX.

Receiver R waits for a first frame to arrive. This frame is delivered at the receiving client, timer T_2 is started and an acknowledgement is sent over L to S. Then the receiver simply waits for more frames to arrive. The receiver remembers whether the previous frame was the last element of the file and the expected value of the alternating bit. Each frame is acknowledged, but it is handed over to the receiving client only if the alternating bit indicates that it is new. In this case timer T_2 is reset. Note that (only) if the previous frame was last of the file, then a fresh frame will be the first of the subsequent file and a repeated frame will still be the last of the old file. This goes on until T_2 times out. This happens if for a long time no new frame is received, indicating that transmission of the file has been given up. The receiving client is informed, provided the last element of the file has not just been delivered. Note that if transmission of the next file starts before timer T_2 expires, the alternating bit scheme is simply continued. This scheme is only interrupted after a failure.

Timer T_1 times out if an acknowledgement does not arrive "in time" at the sender. It is set when a frame is sent and reset after this frame has been acknowledged. (Assume that premature timeouts are not possible, i.e., a message must not come *after* the timer expires.)

Timer T_2 is (re)set by the receiver at the arrival of each new frame. It times out if the transmission of a file has been interrupted by the sender. So its delay must exceed MAX times the delay of T_1.[4] Assume that the sender does not start reading and transmitting the next file before the receiver has properly reacted to the failure. This is necessary, because the receiver has not yet switched its alternating bit, so a new frame would be interpreted as a repetition.

This completes the informal description of the BRP. Two significant assumptions are made in the above description, referred to as **(A1)** and **(A2)** below.

(A1) Premature timeouts are not possible

Let us suppose that the maximum delay in the channel K (and L) is TD and that timer T_1 expires if an acknowledgement has not been received within $\mathsf{T1}$ time units since the first transmission of a chunk. Then this assumption requires that $\mathsf{T1} > 2 \times \mathsf{TD} + \delta$ where δ denotes the processing time in the receiver R. **(A1)** thus requires knowledge about the processing speed of the receiver and the

[4]Later on we will show that this lower bound is not sufficient.

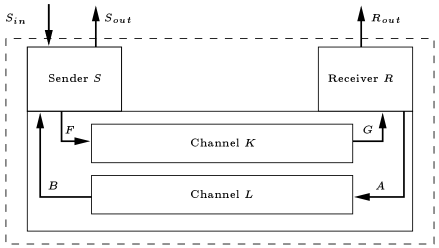
delay in the line.

(A2) In case of abortion, S waits before starting a new file until R reacted properly to abort

Since there is no mechanism in the BRP that notifies the expiration of timer T_2 (in R) to the sender S this is a rather strong and unnatural assumption. It is unclear how S "knows" that R has properly reacted to the failure, especially in case S and R are geographically distributed processes — which apparently is the case in the protocol at hand. We, therefore, consider **(A2)** as an unrealistic assumption. In the sequel we ignore this assumption and adapt the protocol slightly such that this assumption appears as a property of the protocol (rather than as an assumption!).

Modeling the protocol as a network of timed automata

The BRP consists of a sender S and a receiver R communicating through channels K and L, see the figure below. S sends chunk d_i via F to channel K accompanied with an alternating bit ab, an indication b whether d_i is the first chunk of a file (i.e., $i = 1$), and an indication b' whether d_i is the last chunk of a file (i.e., $i = n$). K transfers this information to R via G. Acknowledgements ack are sent via A and B using L.



Schematic view of the BRP.

The signatures of A, B, F, and G are:

$$F, G : (b, b', ab, d_i) \text{ with } ab \in \{\,0,1\,\}, b, b' \in \{\,\text{true}, \text{false}\,\} \text{ and } 0 < i \leqslant n$$

In addition, $A, B : ack$.

We adopt the following notational conventions. States are represented by labelled circles and the initial state as double-lined labelled circle. State invariants are denoted in brackets. Transitions are denoted by directed, labelled arrows. A list of guards denotes the conjunction of its elements. For the sake of simplicity, in this example we use value passing and variable assignments in an unrestricted way.

Channels K and L are simply modeled as first-in first-out queues of unbounded capacity with possible loss of messages. We assume that the maximum latency of both channels is TD time units.

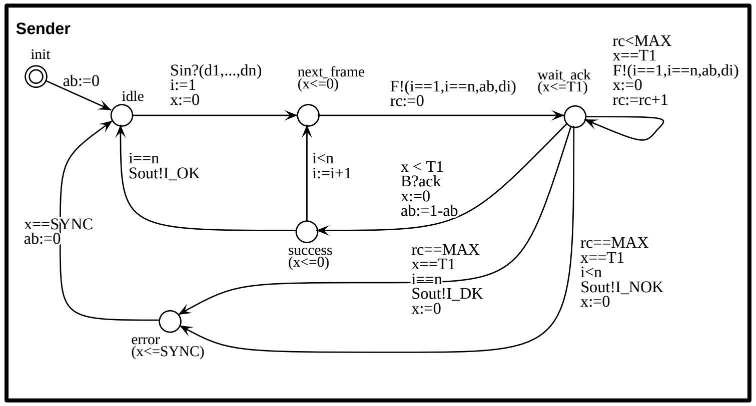


Figure 4.16: Timed automaton for sender S.

Modeling the sender

The sender S (see Figure 4.16) has three system variables: $ab \in \{\,0,1\,\}$ indicating the alternating bit that accompanies the next chunk to be sent, i, $0 \leqslant i \leqslant n$, indicating the subscript of the chunk currently being processed by S, and rc, $0 \leqslant rc \leqslant \mathsf{MAX}$, indicating the number of attempts undertaken by S to retransmit d_i. Clock variable x is used to model timer T_1 and to make certain transitions urgent (see below). In the `idle` location S waits for a new file to be received via S_{in}. On receipt of a new file it sets i to one, and resets clock x. Going from location `next_frame` to `wait_ack`, chunk d_i is transmitted with the corresponding information and rc is reset. In location `wait_ack` there are several possibilities: in case the maximum number of retransmissions has been reached (i.e., $rc = \mathsf{MAX}$), S moves to an `error` location while resetting x and emitting an I_DK or I_NOK indication to the sending client (via S_{out}) depending on whether d_i is the last chunk or not; if $rc < \mathsf{MAX}$, either an `ack` is received (via B) within time (i.e., $x < \mathsf{T1}$) and S moves to the `success` location while alternating ab, or timer x expires (i.e., $x = \mathsf{T1}$) and a retransmission is initiated (while incrementing `rc`, but keeping the same alternating bit). If the last chunk has been acknowledged, S moves from location `success` to location `idle` indicating the successful transmission of the file to the sending client by I_OK. If another chunk has been acknowledged, i is incremented and x reset while moving from location `success` to location `next_frame` where the process of transmitting the next chunk is initiated.

Two remarks are in order. First, notice that transitions leaving location s, say, with location invariant $x \leqslant 0$ are executed without any delay with respect to the previous performed action, since clock x equals 0 if s is entered. Such transitions are called *urgent*. Urgent transitions forbid S to stay in location s arbitrarily long and avoid that receiver R times out without abortion of the transmission by sender S. Urgent transitions will turn out to be necessary to achieve the correctness of the protocol. They model a maximum delay on processing speed, cf. assumption **(A1)**. Secondly, we remark that after a failure (i.e., S is in location `error`) an additional delay of SYNC time units is incorporated. This delay is introduced in order to ensure that S does not start transmitting a new file before the receiver has properly reacted to the failure. This timer will make it possible to satisfy assumption **(A2)**. In case of failure the alternating bit scheme is restarted.

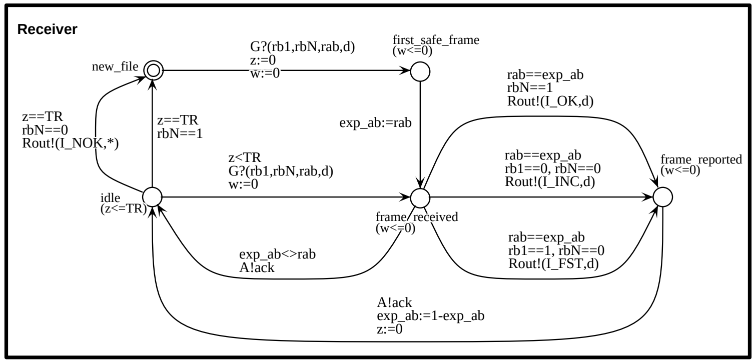


Figure 4.17: Timed automaton for receiver R.

Modeling the receiver

The receiver is depicted in Figure 4.17. System variable `exp_ab` $\in \{0, 1\}$ in receiver R models the expected alternating bit. Clock z is used to model timer T_2 that determines transmission abortions of sender S, while clock w is used to make some transitions urgent. In location `new_file`, R is waiting for the first chunk of a new file to arrive. Immediately after the receipt of such chunk `exp_ab` is set to the just received alternating bit and R enters the location `frame_received`. If the expected alternating bit agrees with the just received alternating bit (which, due to the former assignment to `exp_ab` is always the case for the first chunk) then an appropriate indication is sent to the receiving client, an `ack` is sent via A, `exp_ab` is toggled, and clock z is reset. R is now in location `idle` and waits for the next frame to arrive. If such frame arrives in time (i.e., $z < \mathsf{TR}$) then it moves to the location `frame_received` and the above described procedure is repeated; if timer z expires (i.e., $z = \mathsf{TR}$) then in case R did not just receive the last chunk of a file an indication I_NOK (accompanied with an arbitrary chunk "$*$") is sent via R_{out} indicating a failure, and in case R just received the last chunk, no failure is reported.

Most of the transitions in R are made urgent in order to be able to fulfill assumption **(A1)**. For example, if we allowed an arbitrary delay in location `frame_received` then the sender S could generate a timeout (since it takes too long for an acknowledgement to arrive at S) while an acknowledgement generated by R is possibly still to come.

Model checking the specification

An important aspect of the BRP is the question what the relationships between the timeout values, transmission delays and synchronization delays (like SYNC) are in order for the protocol to work correctly.

Premature timeouts

Assumption **(A1)** states that no premature timeouts should occur. The BRP contains two types of timeouts: one for detecting the absence of a promptly acknowledgement (by the sender), and one for detecting the abortion of the transmission (by the receiver). It can easily be seen that timer T_1 (i.e., clock x) of sender S does not violate assumption **(A1)** if it respects the two-way transmission delay (i.e., $\mathsf{T1} > 2 \times \mathsf{TD}$) plus the processing delay of the receiver R (which due to the presence of location invariants equals 0). It remains to be checked under which conditions timer T_2 of receiver R does not generate premature timeouts. This amounts to checking that R times out whenever the sender has indeed aborted the transmission of the file. Observe that a premature timeout appears in R if it moves from location `idle` to state `new_file` although there is still some frame of the previous file to come. We therefore check that in location `first_safe_frame` receiver R can only receive first chunks of a file (i.e., $\mathtt{rb1} = 1$) and not remaining ones of previous files. This is formulated in UPPAAL notation as:

$$\mathsf{AG}\,(\mathtt{R.first\_safe\_frame} \;\Rightarrow\; \mathtt{rb1} = 1) \tag{4.4}$$

Using several verifications with the verification engine `verifyta` of UPPAAL we can establish that this property holds whenever $\mathsf{TR} \geqslant 2 \times \mathsf{MAX} \times \mathsf{T1} + 3 \times \mathsf{TD}$.

In order to conclude this, the simulator of UPPAAL together with the diagnostic traces have been of big help. We were able to try different values for TD, T1, and MAX, and thus, to study the behavior of the protocol. Figure 4.18 depicts the longest trace that makes the protocol loose the connection when $\mathsf{MAX} = 2$ and $n \geqslant 2$. The $\times$ symbol indicated after sending a frame represents that it is lost in some of the channels K or L. Notice that frames are received within TD time units but they always take some time to travel through the channel. In particular, in the transmission of the first frame, the difference in time between synchronization on F and synchronization on G cannot be 0.

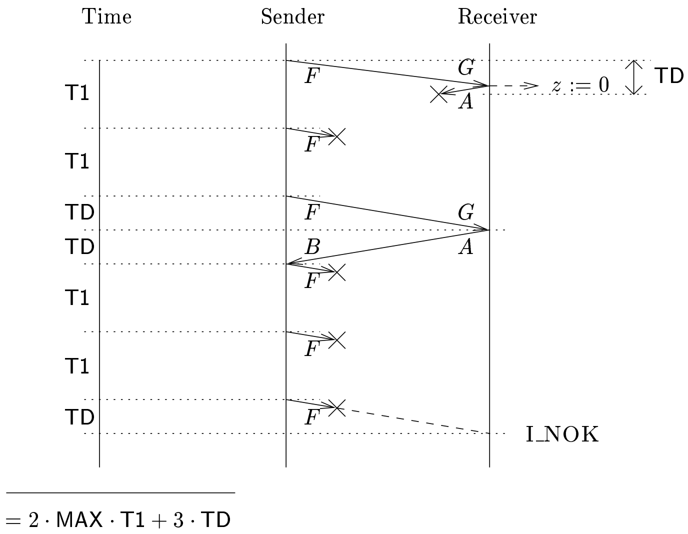


Figure 4.18: Loosing the connection

From the figure, it is clear that the receiver should not timeout (strictly) before $2 \times \mathsf{MAX} \times \mathsf{T1} + 3 \times \mathsf{TD}$ units of time, since this is the last time a frame can arrive. Premature timeouts would induce the receiver to abort the connection when there is still the possibility of some frame to arrive. As a result, property (4.4) would be violated.

Premature abortions

Assumption **(A2)** states that sender S starts the transmission of a new file only after R has properly reacted to the failure. For our model this means that if S is in location `error`, eventually, within SYNC time units, R resets and is able to receive a `new_file`. This property is expressed in TCTL as

$$\mathsf{A}\,[\texttt{S.error}\ \mathsf{U}_{\leqslant \mathsf{SYNC}}\ \texttt{R.new\_file}] \tag{4.5}$$

Unfortunately, the property language of UPPAAL does not support this type of formula. Therefore, we check the following property:

$$\mathsf{AG}\,[(\texttt{S.error}\ \wedge\ x = \mathsf{SYNC})\ \Rightarrow\ \texttt{R.new\_file}] \tag{4.6}$$

The difference between properties (4.5) and (4.6) is that (4.5) requires that `S.error` is true until `R.new_file` becomes true, while (4.6) does not take into account what happens when time passes, but considers only the instant for which $x = \mathsf{SYNC}$. Provided that S is in location `error` while clock x evolves from 0 to SYNC — which is obviously the case — (4.6) implies (4.5). Using several verifications with the verification engine **`verifyta`** of UPPAAL we can establish that property (4.6) is satisfied under the condition that $\mathsf{SYNC} \geqslant \mathsf{TR}$. This means that **(A2)** is fulfilled if this condition on the values SYNC and TR is respected.

Verification results

Summarizing, we were able to check with UPPAAL that assumptions **(A1)** and **(A2)** are fulfilled by the BRP if the following additional constraints hold

$$\mathsf{TD} > 2 \times \mathsf{TD}\ \wedge\ \mathsf{SYNC} \geqslant \mathsf{TR} \geqslant 2 \times \mathsf{MAX} \times \mathsf{T1} + 3 \times \mathsf{TD}$$

Remark that SYNC and T1 are constants in the sender S, while TR is a constant used in receiver R. These results show the importance of real-time aspects for the correctness of the BRP.

Exercises

EXERCISE 26. The intuitive meaning of the formula $\mathsf{E}\,\{\,I\,\}\phi$ is that there exists a path starting from the current state such that the *earliest* possible time at which ϕ is true lies in the interval I, for I an interval with integer boundaries. Define this operator either semantically (using $\models$), or try to formulate it in terms of existing temporal operators.

EXERCISE 27. Consider the region $r = [(2 < x < 3), (1 < y < 2), (x{-}y < 1)]$ and let clock valuation v such that $r = [v]$.

1. Characterize the region to which $[x]v$ belongs.
2. Characterize the direct time successor of r.
3. Suppose assignments of the form $x := y$ are allowed. Characterize the region r' that results from r under $x := y$.

EXERCISE 28. Let v, v' be two clock valuations over the set of clocks C such that $v \approx v'$. Prove that for any time constraint $\alpha \in \Psi(C)$ we have: $v \models \alpha$ if and only if $v' \models \alpha$.

EXERCISE 29. Consider the timed automata $\mathcal{A}_1$ and $\mathcal{A}_2$ depicted in Figure 4.19. Construct the region automata $\mathcal{R}(\mathcal{A}_1)$ and $\mathcal{R}(\mathcal{A}_2)$ for $c_z = 1$. Explain the differences and indicate for both region automata the unbounded regions.

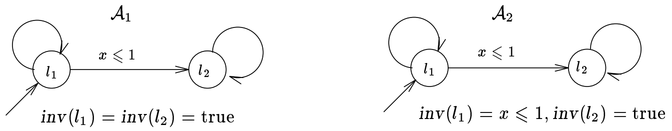
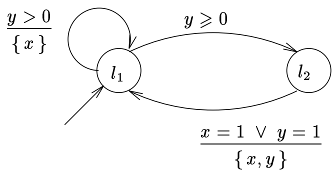


Figure 4.19: Two similar timed automata

EXERCISE 30. Consider the modified light switch of Figure 4.10 and suppose *on* is labelled with atomic proposition p and *off* is labelled with q.

1. Give a TCTL-formula that expresses the property "once the light is off, it will be switched off eventually after it has been switched on".
2. Check the property with the help of the region automaton of Figure 4.11 by indicating a path that invalidates this property or by arguing that all paths do satisfy it.
3. Give a TCTL-formula that expresses the property "there is a possibility that when the light is off, it is switched on, and switched off within 5 time-units?"
4. Check the property with the help of the region automaton of Figure 4.11 by indicating a path that satisfies this property or by arguing that there is no path that satisfies it.

EXERCISE 31. Consider the following timed automaton:



$inv(l_1) = x \leqslant 1 \ \wedge\ y \leqslant 1$

$inv(l_2) = x \leqslant 1 \ \wedge\ y \leqslant 1$

1. Construct the region automaton for this timed automaton for a formula with maximal constant 1; indicate explicitly the delay transitions and the unbounded regions.
2. Check the validity of the following formula

   $$\mathsf{E}\,[(y = 0)\;\mathsf{U}\;\mathsf{E}\,[(x > 0)\;\mathsf{U}\;(y = 0)]]$$

   and justify your answer.
3. Extend the timed automaton with a counter n that is initialized as 0 and is increased by 1 on every visit to location l_1. Check the validity of the formula

   $$z \text{ in } \mathsf{EF}\,(z \leqslant 10\ \wedge\ n > 10)$$

   and justify your answer.

4. Consider the formula $\mathsf{AF}\, p$ where p is an atomic proposition that is true if and only if the timed automaton is in location l_1.
   (a) Check the validity of this formula.
   (b) What is the validity of this formula if a notion of weak fairness (see Chapter 3) is adopted?

EXERCISE 32. Model and analyses the following soldier problem in UPPAAL. Four soldiers are heavily injured, try to flee to their home land. The enemy is chasing them and in the middle of the night they arrive at a bridge that has been damaged and can only carry two soldiers at a time. Furthermore, several land mines have been placed on the bridge and a torch is needed to move the mines out of the way. The enemy is approaching, so the soldiers know that they only have 60 minutes to cross the bridge. The soldiers only have a single torch and they are not equally injured; thus the crossing time (one-way) for the soldiers is 5 min, 10 min, 20 min and 25 minutes.

(a) Model the soldier problem in UPPAAL as a network of interacting timed automata

(b) Verify whether there exists a strategy that results in saving all soldiers.

(c) Is there such strategy if the enemy is arriving a bit earlier, say, within 55 minutes?

(This exercise is due to Ruys and Brinksma.)

4.10 Selected references

Introduction and overview of real-time temporal logics:

- R. KOYMANS. *Specifying Message Passing and Time-Critical Systems with Temporal Logic*. LNCS 651. 1992.
- J.S. OSTROFF. Formal methods for the specification and design of real-time safety critical systems. *Journal of Systems and Software*, **18**, 1992.

- T.A. HENZINGER. It's about time: real-time logics reviewed. In *Concurrency Theory*, LNCS 1466, pages 439–454, 1998.

Past operators in real-time temporal logics:

- R. ALUR AND T.A. HENZINGER. Back to the future: towards a theory of timed regular languages. In *IEEE Symp. on Foundations of Computer Science*, pages 177–186, 1992.

Overview of issues related to the incorporation of real-time in formal methods:

- R. KOYMANS. (Real) time: a philosophical perspective. In *Real-Time: Theory in Practice*, LNCS 600, pages 353–370, 1992.

Model checking of real-time systems:

- R. ALUR, C. COURCOUBETIS AND D. DILL. Model-checking in dense real-time. *Information and Computation*, **104**: 2–34, 1993.
- R. ALUR AND D. DILL. Automata-theoretic verification of real-time systems. In *Formal Methods for Real-Time Computing*, pages 55–82, 1996.
- K.G. LARSEN, B. STEFFEN AND C. WEISE. Continuous modelling of real-time and hybrid systems: from concepts to tools. *Software Tools for Technology Transfer*, **1**: 64–86, 1997.
- S. YOVINE. Model checking timed automata. In *Embedded Systems*, LNCS 1494, 1998.

Expressiveness and decidability of real-time logics:

- R. ALUR AND T. HENZINGER. Real-time logics: Complexity and expressiveness. *Information and Computation*, **104**: 35–77, 1993.

Tools for real-time model checking:

- K.G. Larsen, P. Pettersson and W. Yi. Uppaal in a nutshell. *Software Tools for Technology Transfer*, **1**: 134–153, 1997.

- S. Yovine. Kronos: a verification tool for real-time systems. *Software Tools for Technology Transfer*, **1**: 123–134, 1997.

- T.A. Henzinger, P.-H. Ho and H. Wong-Toi. HyTech: a model checker for hybrid systems. *Software Tools for Technology Transfer*, **1**: 110–123, 1997.

- S. Tripakis and C. Courcoubetis. Extending Promela and Spin for real time. In *Tools and Algorithms for the Construction and Analysis of Systems*, LNCS 1055, pages 329–348, 1996.

Bounded retransmission protocol case study:

- P.R. D'Argenio, J.-P. Katoen, T. Ruys, and G.J. Tretmans. The bounded retransmission protocol must be on time! In *Tools and Algorithms for the Construction and Analysis of Systems*, LNCS 1217, pages 416–432, 1997.