Timed Transition Systems[1]

Thomas A. Henzinger
Computer Science Department, Cornell University
Ithaca, NY 14853, U.S.A.

Zohar Manna
Computer Science Department, Stanford University
Stanford, CA 94305, U.S.A.
and
Department of Applied Mathematics, Weizmann Institute of Science
Rehovot 76100, Israel

Amir Pnueli
Department of Applied Mathematics, Weizmann Institute of Science
Rehovot 76100, Israel

Abstract. We incorporate time into an interleaving model of concurrency. In timed transition systems, the qualitative fairness requirements of traditional transition system are replaced (and superseded) by quantitative lower-bound and upper-bound timing constraints on transitions. The purpose of this paper is to explore the scope of applicability for the abstract model of timed transition systems. We demonstrate that the model can represent a wide variety of phenomena that routinely occur in conjunction with the timed execution of concurrent processes. Our treatment covers both processes that are executed in parallel on separate processors and communicate either through shared variables or by message passing, and processes that time-share a limited number of processors under a given scheduling policy. Often it is this scheduling policy that determines if a system meets its real-time requirements. Thus we explicitly address such questions as time-outs, interrupts, static and dynamic priorities.

Keywords: Transition systems, concurrency, real time.

[1] This research was supported in part by an IBM graduate fellowship, by the National Science Foundation grants CCR-89-11512 and CCR-89-13641, by the Defense Advanced Research Projects Agency under contract N00039-84-C-0211, by the United States Air Force Office of Scientific Research under contract AFOSR-90-0057, and by the European Community ESPRIT Basic Research Action project 3096 (SPEC).

1 Introduction

In [HMP91], we extended the specification language of temporal logic and the corresponding verification framework to prove timing properties of reactive systems. To model the timed execution of reactive systems, we generalized the computational model of transition systems conservatively by imposing timing requirements on the transitions. We claimed that a wide variety of real-time phenomena that are encountered in practice can be represented and analyzed within the model of timed transition systems. In this paper, we substantiate that claim. We use timed transition systems to model the timed execution of both multiprocessing and multiprogramming systems. Specifically, we address issues that routinely occur in real-time process communication, such as time-outs, and issues of real-time process control, including typical time-sharing strategies. By doing so, we enlargen the scope of applicability of the temporal proof methodologies for verifying timing properties of reactive systems that were presented in [HMP91].

In our model, we assume a global, fictitious, real-valued clock, whose actions advance time by nonuniform amounts. The clock actions are interleaved with the system actions, which have no duration in time. In some other work aimed at the formal analysis of real-time systems, it has been claimed that while this *interleaving* model of computation may be adequate for the qualitative analysis of reactive systems, it is inappropriate for the real-time analysis of programs, and a more realistic model, such as *maximal parallelism*, is needed [KSdR+88]. One of the points that we demonstrate in this paper is a refutation of that claim. We show that by a careful incorporation of time into the interleaving model, we can still model adequately most of the phenomena that occur in the timed execution of programs. Yet we retain the important economic advantage of interleaving models, namely, that at any point only one transition can occur and has to be analyzed.

We define the formal semantics of a real-time system as a set of timed execution sequences. This is done in two steps. First, in Section 2, we review the *abstract* computational model of timed transition systems and identify the possible timed execution sequences (computations) of any such system. Then, we consider *concrete* real-time systems and show how the concrete constructs can be interpreted within the abstract model. We begin, in Section 3, with the representation of real-time processes that are executed in parallel and communicate either through a shared memory or by message passing. Although the timeless interleaving of concurrent activities identifies true parallelism with (sequential) nondeterminism, when time is of the essence, we can no longer ignore the difference between *multiprocessing*, where each parallel task is executed on a separate machine, and *multiprogramming*, where several tasks reside on the same machine. This is because questions of priorities, interrupts, and scheduling of tasks may strongly influence the ability of a system to meet its timing constraints. These issues in modeling time-sharing systems are discussed in Section 4.

2 Timing Constraints for Transition Systems

Timed transition systems generalize the basic computational model of transition systems [Kel76, Pnu77] by associating minimal and maximal time delays with the transitions. We use the real line as time domain and adapt the definition of timed transition systems that was given for a discrete time domain in [HMP91]. Similar notions of transition systems with timing constraints have been defined, for integer time, in [Har88, PH88, HMP90, Ost90].

A *transition system* $S = \langle V, \Sigma, \Theta, \mathcal{T} \rangle$ consists of four components:

1. a finite set V of *variables*.

2. a set Σ of *states*. Every state $\sigma \in \Sigma$ is an interpretation of V; that is, it assigns to every variable $x \in V$ a value $\sigma(x)$ in its domain.

3. a subset $\Theta \subseteq \Sigma$ of *initial states*.

4. a finite set $\mathcal{T}$ of *transitions*, including the idle transition τ_I. Every transition $\tau \in \mathcal{T}$ is a binary relation on Σ; that is, it defines for every state $\sigma \in \Sigma$ a (possibly empty) set of τ-successors $\tau(\sigma) \subseteq \Sigma$. We say that the transition τ is *enabled* on a state σ iff $\tau(\sigma) \neq \emptyset$. In particular, the *idle* (stutter) transition

$$\tau_I \;=\; \{(\sigma, \sigma) \mid \sigma \in \Sigma\}$$

is enabled on every state.

An infinite sequence $\sigma = \sigma_0 \sigma_1 \ldots$ of states is a *computation* (execution sequence, run) of the transition system $S = \langle V, \Sigma, \Theta, \mathcal{T} \rangle$ iff it satisfies the following two requirements:

Initiality $\sigma_0 \in \Theta$.

Consecution For all $i \geq 0$, there is a transition $\tau \in \mathcal{T}$ such that $\sigma_{i+1} \in \tau(\sigma_i)$ (which is also denoted by $\sigma_i \xrightarrow{\tau} \sigma_{i+1}$). We say that the transition τ is *taken* at position i of the computation σ.

We incorporate time into the transition system model by assuming that all transitions happen "instantaneously," while real-time constraints restrict the times at which transitions may occur. The timing constraints are classified into two categories: *lower-bound* and *upper-bound* requirements. They ensure that transitions occur neither too early nor too late, respectively. All of our time bounds are nonnegative integers N. The absence of a lower-bound requirement is modeled by a lower bound of 0; the absence of an upper-bound requirement, by an upper bound of ∞. For notational convenience, we assume that $\infty \geq n$ for all $n \in \mathsf{N}$.

A *timed transition system* $S = \langle V, \Sigma, \Theta, \mathcal{T}, l, u \rangle$ consists of an underlying transition system $S^- = \langle V, \Sigma, \Theta, \mathcal{T} \rangle$ as well as

5. a *minimal delay* $l_\tau \in \mathsf{N}$ for every transition $\tau \in \mathcal{T}$. We require that $l_{\tau_I} = 0$.

6. a *maximal delay* $u_\tau \in \mathsf{N} \cup \{\infty\}$ for every transition $\tau \in \mathcal{T}$. We require that $u_\tau \geq l_\tau$ for all $\tau \in \mathcal{T}$, and that $u_\tau = \infty$ if τ is enabled on any initial state in Θ. In particular, $u_{\tau_I} = \infty$.

Let $\mathcal{T}_0 \subseteq \mathcal{T}$ be the set of transitions with the maximal delay 0. To allow time to progress, we put a restriction on these transitions. We require that there is no sequence

$$\sigma_0 \xrightarrow{\tau_0} \sigma_1 \xrightarrow{\tau_1} \cdots \xrightarrow{\tau_{n-1}} \sigma_n$$

of states and transitions such that $n > |\mathcal{T}_0|$ and $\tau_i \in \mathcal{T}_0$ for all $0 \leq i < n$. This condition ensures the operationality (machine-closure) of timed transition systems [Hen91a].

Timed state sequences

We model time by the real numbers R. A *timed state sequence* $\rho = (\sigma, \mathsf{T})$ consists of an infinite sequence σ of states $\sigma_i \in \Sigma$, where $i \geq 0$, and an infinite sequence T of corresponding times $\mathsf{T}_i \in \mathsf{R}$ that satisfy the following two conditions:

Monotonicity For all $i \geq 0$,

either $\mathsf{T}_{i+1} = \mathsf{T}_i$,
or $\mathsf{T}_{i+1} > \mathsf{T}_i$ and $\sigma_{i+1} = \sigma_i$;

that is, time never decreases. It may increase, by any amount, only between two consecutive states that are identical. The case that the time stays the same between two identical states is referred to as a *stutter step*; the case that the time increases is called a *time step*.

Progress For all $t \in \mathsf{R}$, there is some $i \geq 0$ such that $\mathsf{T}_i \geq t$; that is, time never converges. Since the time domain R contains no maximal element, there are infinitely many time steps in every timed state sequence.

It follows that a timed state sequence alternates *state* activities with *time* activities. Throughout state activities, time does not advance; throughout time activities, the state does not change. Since all state activities are represented by finite subsequences, timed state sequences observe the condition of *finite variability*, namely, that the state changes only finitely often throughout any finite interval of time [BKP86].

A set Π of timed state sequences is *closed under stuttering* iff (1) the sequence

$$\cdots \longrightarrow (\sigma_i, \mathsf{T}_i) \longrightarrow \cdots$$

is in Π precisely when the sequence

$$\cdots \longrightarrow (\sigma_i, \mathsf{T}_i) \longrightarrow (\sigma_i, \mathsf{T}_i) \longrightarrow \cdots$$

is in Π, and (2) the sequence

$$\cdots \longrightarrow (\sigma_i, \mathsf{T}_i) \longrightarrow (\sigma_i, \mathsf{T}_{i+1}) \longrightarrow \cdots$$

is in Π precisely when the sequence

$$\cdots \longrightarrow (\sigma_i, \mathsf{T}_i) \longrightarrow (\sigma_i, t) \longrightarrow (\sigma_i, \mathsf{T}_{i+1}) \longrightarrow \cdots$$

is in Π for $\mathsf{T}_i \leq t \leq \mathsf{T}_{i+1}$. To close a set of timed state sequences under stuttering, then, we must (1) add and delete finitely many stutter steps, and (2) split and merge finitely many time steps.

Timed execution sequences

Just as the execution sequences of transition systems are infinite state sequences, we model the execution sequences of timed transition systems by timed state sequences. The timed state sequence $\rho = (\sigma, \mathsf{T})$ is a *computation* of the timed transition system $S = \langle V, \Sigma, \Theta, \mathcal{T}, l, u \rangle$ iff the state sequence σ is a computation of the underlying transition system S^- and

Lower bound For every transition $\tau \in \mathcal{T}$ and all positions $i \geq 0$ and $j \geq i$ with $\mathsf{T}_j < \mathsf{T}_i + l_\tau$,

> if τ is taken at position j of σ,
> then τ is enabled on σ_i.

In other words, once enabled, τ is delayed for at least l_τ time units; it can be taken only after being continuously enabled for l_τ time units. Any transition that is enabled initially, on the first state of a timed state sequence, can be taken immediately (as if it has been enabled forever).

Upper bound For every transition $\tau \in \mathcal{T}$ and position $i \geq 0$, there is some position $j \geq i$ with $\mathsf{T}_j \leq \mathsf{T}_i + u_\tau$ such that

> either τ is not enabled on σ_j,
> or τ is taken at position j of σ.

In other words, once enabled, τ is delayed for at most u_τ time units; it cannot be continuously enabled for more than u_τ time units without being taken. Since the maximal delay of every transition that is enabled initially must be ∞, the first state change of a computation may occur at any time.

Note that at both stutter steps and time steps, the idle transition τ_I is taken. The idle transition marks phases of time activity throughout a computation; the other transitions are interleaved during phases of state activity. We consider all computations of a timed transition system to be infinite. Finite (terminating as well as deadlocking) computations are represented by infinite extensions that add only idle transitions; that is, a final, infinite phase of time activity.

It is not difficult to check that the computations of any timed transition system S are closed (1) under *stuttering* and (2) under *shifting the origin of time*. The former property ensures that timed transition systems can be refined [Hen91b]; the latter property means that timed transition systems cannot refer to absolute time. Specifically, the addition of a real constant to all times of a timed state sequence does not alter the property of being a computation of S. Thus we will often assume, without loss of generality, that the time of the first state change of a computation is 0.

Since the state component of any computation of S is a computation of the underlying untimed transition system S^-, ordinary timeless reasoning is sound for timed transition systems: every untimed property of infinite state sequences that is satisfied by all computations of S^-, is also satisfied by all computations of S. The converse, however, is generally not true. The timing constraints of S can be viewed as filters that prohibit certain possible behaviors of the untimed transition system S^-. Special cases are a minimal delay 0 and a maximal delay ∞ for a transition τ. While the former does not rule out any computations of S^-, the latter adds to S^- a *weak-fairness* (justice) assumption in the sense of [MP89]: τ cannot be continuously enabled without being taken.

The choice of the real numbers as time domain is taken for the sake of concreteness. Indeed, any total ordering with an appropriate definition of addition by a unit (i.e., $t \leq t+1$, and $t \leq t'$ implies $t+1 \leq t'+1$) can be chosen as a time domain. In the case that the time domain is any one-element set, timed transition systems are easily seen to coincide with ordinary fair transition systems (with a weak-fairness requirement for every transition).

3 Modeling Time-critical Multiprocessing Systems

The concrete real-time systems we consider first consist of a fixed number of sequential real-time programs that are executed in parallel, on separate processors, and communicate through a shared memory. We show how time-outs and real-time response can be programmed in this language. Then we add message passing primitives for process synchronization and communication.

3.1 Syntax: Timed transition diagrams

A *shared-variables multiprocessing system* P has the form

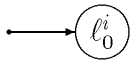
$$\{\theta\}[P_1 \| \dots \| P_m].$$

Each *process* P_i, for $1 \leq i \leq m$, is a sequential nondeterministic real-time program over the finite set U_i of *private* (local) *data variables* and the finite set U_s of *shared data variables*. The formula θ, called the *data precondition* of P, restricts the initial values of the variables in

$$U \;=\; U_s \cup \bigcup_{1 \leq i \leq m} U_i.$$

The real-time programs P_i can be alternatively presented in a textual programming language or as transition diagrams. We shall use the latter, graphical, representation. For this purpose, we extend the untimed transition diagram language by labeling transitions with minimal and maximal time delays.

A *timed transition diagram* for the process P_i is a finite directed graph whose vertices $L_i = \{\ell_0^i, \dots \ell_{n_i}^i\}$ are called *locations*. The *entry* location — usually ℓ_0^i — is indicated as follows:



The intended meaning of the entry location ℓ_0^i is that the control of the process P_i starts at the location ℓ_0^i. The component processes of a system are not required to start synchronously (i.e., at the same time). Each edge in the graph is labeled by a guarded instruction, a *minimal delay* $l \in \mathsf{N}$ and a *maximal delay* $u \in \mathsf{N} \cup \{\infty\}$ such that $u \geq l$:

$$\ell^i_j \xrightarrow[{[l,u]}]{c\ \to\ \bar{x} := \bar{e}} \ell^i_k$$

where the guard c is a boolean expression, $\bar{x}$ is a vector of variables, and $\bar{e}$ an equally typed vector of expressions (the guard *true* and the delay interval $[0, \infty]$ are usually suppressed; for the empty vector *nil*, the instruction $c \to nil := nil$ is abbreviated to $c?$). We require that every cycle in the graph consists of no fewer than two edges, at least one of which is labeled by a positive (nonzero) maximal delay.

The intended operational meaning of the given edge is as follows. The minimal delay l guarantees that whenever the control of the process P_i has resided at the location ℓ^i_j for at least l time units during which the guard c has been continuously true, then P_i *may* proceed to the location ℓ^i_k. The maximal delay u ensures that whenever the control of the process P_i has resided at ℓ^i_j for u time units during which the guard c has been continuously true, then P_i *must* proceed to ℓ^i_k (or, more precisely, time cannot advance before either the guard c becomes false, which may be caused by process parallel to P_i, or the process P_i proceeds). In doing so, the control of P_i moves to the location ℓ^i_k "instantaneously," and the current values of $\bar{e}$ are assigned to the variables $\bar{x}$. In general, a process may have to proceed via several edges all of whose guards have been continuously true for their corresponding maximal delays. In this case, any such edge is chosen nondeterministically.

It follows that the control of a process P_i may remain at a location ℓ^i_j forever only in one of two situations: if ℓ^i_j has no outgoing edges, we say that P_i has *terminated*; if each of the guards that are associated with the outgoing edges of the location ℓ^i_j is false infinitely often, we say that P_i has *deadlocked*. The second condition is necessary (although not sufficient) for stagnation, because if one guard is true forever, then the corresponding maximal delay $u \leq \infty$ guarantees the progress of P_i.

3.2 Semantics: Timed transition systems

The operational view of timed transition diagrams can be captured by a simple translation into the abstract model of timed transition systems. With the given shared-variables multiprocessing system

$$P: \quad \{\theta\}[P_1 \| \ldots \| P_m],$$

we associate the following timed transition system $S_P = \langle V, \Sigma, \Theta, \mathcal{T}, l, u \rangle$:

1. $V = U \cup \{\pi_1, \ldots \pi_m\}$. Each *control variable* π_i, where $1 \leq i \leq m$, ranges over the set $L_i \cup \{\perp\}$. The value of π_i indicates the location of the control for the process P_i; it is $\perp$ (undefined) before the process P_i starts.

2. Σ contains all interpretations of V.

3. Θ is the set of all states $\sigma \in \Sigma$ such that θ is true in σ and $\sigma(\pi_i) = \perp$ for all $1 \leq i \leq m$.

4. $\mathcal{T}$ contains, in addition to the idle transition τ_I, an *entry transition* τ^i_0 for every process P_i, where $1 \leq i \leq m$, as well as a transition τ_E for every edge E in the timed transition diagrams for $P_1, \ldots P_m$. In particular, $\sigma' \in \tau^i_0(\sigma)$ iff

$\sigma(\pi_i) = \perp$ and $\sigma'(\pi_i) = \ell_0^i$,
$\sigma'(y) = \sigma(y)$ for all $y \in V - \{\pi_i\}$.

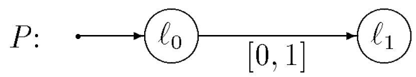
If E connects the source location ℓ_j^i to the target location ℓ_k^i and is labeled by the instruction $c \to \bar{x} := \bar{e}$, then $\sigma' \in \tau_E(\sigma)$ iff

$\sigma(\pi_i) = \ell_j^i$ and $\sigma'(\pi_i) = \ell_k^i$,
c is true in σ and $\sigma'(\bar{x}) = \sigma(\bar{e})$,
$\sigma'(y) = \sigma(y)$ for all $y \in V - \{\pi_i, \bar{x}\}$.

5. If τ is an entry transition, then $l_\tau = 0$. For every edge E labeled by the minimal delay l, let $l_{\tau_E} = l$.

6. If τ is an entry transition, then $u_\tau = \infty$. For every edge E labeled by the maximal delay u, let $u_{\tau_E} = u$.

This translation defines the set of possible computations of the real-time system P as a set of timed state sequences. For instance, the computations of the trivial system P that consists of a single process with the timed transition diagram



are exactly the timed state sequences that result from closing all sequences of the form

$$(\perp, 0) \longrightarrow (\ell_0, 0) \longrightarrow (\ell_0, t) \longrightarrow (\ell_1, t) \longrightarrow (\ell_1, 1) \longrightarrow (\ell_1, 2) \longrightarrow \cdots$$

where $0 \le t \le 1$, under stuttering and shifting the origin of time. The condition on timed transition diagrams that every cycle contains at least one positive (nonzero) maximal delay ensures that the timed transition system S_P is operational. (The condition that every cycle contains at least two edges guarantees that once a transition is taken, it cannot stay enabled, which simplifies the reasoning about timed transition systems [HMP91].)

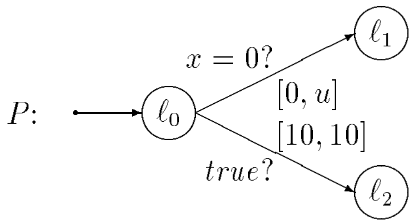
We remark that our semantics of shared-variables multiprocessing systems is conservative over the untimed case. Suppose that the system P contains no delay labels (recall that, in this case, all minimal delays are 0 and all maximal delays are ∞). Then the state components of the computations of S_P are precisely the legal execution sequences of P, as defined in the interleaving model of concurrency, that are weakly fair with respect to every transition [MP89]: no process can stop when one of its transitions is continuously enabled. Weak fairness for every individual transition and, consequently, progress for every process is guaranteed by the maximal delays ∞.

3.3 Examples: Time-out and timely response

We now model two typical real-time constructs as shared-variables multiprocessing systems to demonstrate the scope of the timed transition diagram language. In the first example (*time-out*), a process checks if an external event happens within a certain amount of time. In the second example (*traffic light*), a process reacts to an external event and is required to do so within a certain amount of time. A third example combines several processes.

Time-out

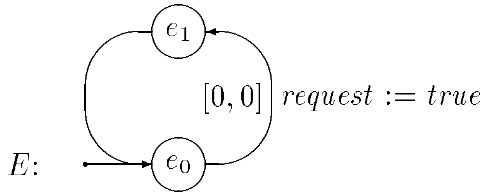
To see how a time-out situation can be programmed, consider the process P with the following timed transition diagram:



We assume that the variable x may be altered by some other process that is executed in parallel to P. When at the location ℓ_0, the process P attempts, for 10 time units, to proceed to the location ℓ_1 by checking the value of x. If the value of x is not found to be 0, then P does not succeed and proceeds to the alternative location ℓ_2 after 10 time units. The choice of the maximal delay u determines how often P checks the value of x. For example, if $u \geq 10$, then P may not check the value of x at all before timing out after 10 time units. If $0 < u < 10$, then P has to check the value of x at least once every u time units. Consequently, if the value of x is 0 for more than u time units, it will be detected. On the other hand, the value of x being 0 may go undetected if it fluctuates too frequently, even in the case of $u = 0$.

Traffic light

To give another typical real-time application of embedded systems, let us design a traffic light controller that turns a pedestrian light green within 5 time units after a button is pushed. The environment is given by the following process E. Whenever the request button is pushed, the shared boolean variable *request* is set to *true*:



Recall that the edge labels *true*? and $[0, \infty]$ are suppressed; thus we have no knowledge about the frequency of requests.

We wish to design a traffic light controller Q that controls the status of the traffic light through the variable *light*, whose value is either *green* or *red*. As unit of time we take the amount of time it takes to switch the light; for simplicity, we also assume that the time needed for local operations within Q is negligible. Now let us specify the desired process Q. The controller Q should behave in such a way that the combined system

$$P: \quad \{request = false,\ light = red\}\ [E \| Q]$$

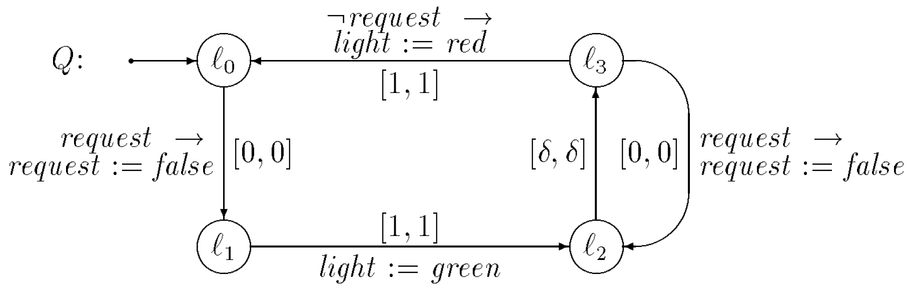
satisfies the following two correctness conditions:

(A) Whenever *request* is *true*, then *light* is *green* within 5 time units for at least 5 time units.

(B) Whenever *request* has been *false* for 25 time units, then *light* is *red*.

The first condition, (A), ensures that no pedestrian has to wait for more than 5 time units to cross the road and is given another 5 time units to do so. The second condition, (B), prevents the light from being green indefinitely (under the assumption that any controller Q resets the variable *request* to *false* whenever it is read and found to be *true*).

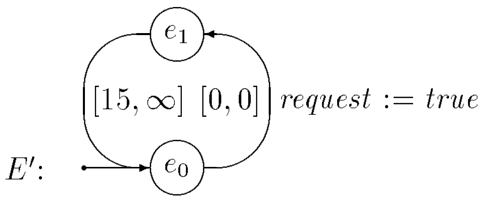
It is not hard to convince ourselves that, once it is started, the following process Q satisfies the specification:



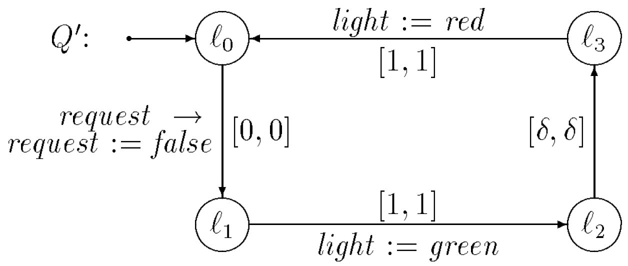
for any delay $4 \leq \delta \leq 23$. This implementation of the traffic light controller turns the light green as soon as possible after a request is received and then waits for δ time units before turning the light red again. If the request button has been pushed in the meantime, the light stays green for another δ time units.

Multiple traffic lights

We now generalize the *traffic light* example and design a system that reacts to several external events. We wish to do so by composing, in parallel, processes that are similar to Q. At this point it is convenient to accept some additional assumptions about the frequencies of the external events. In our example, we suppose that the distance between any two requests is at least 15 time units; that is,



Under this assumption, we can simplify the traffic light controller to

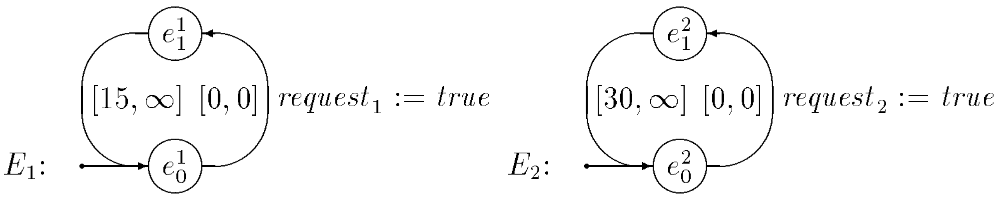


for any delay $4 \leq \delta \leq 17$. The combined system

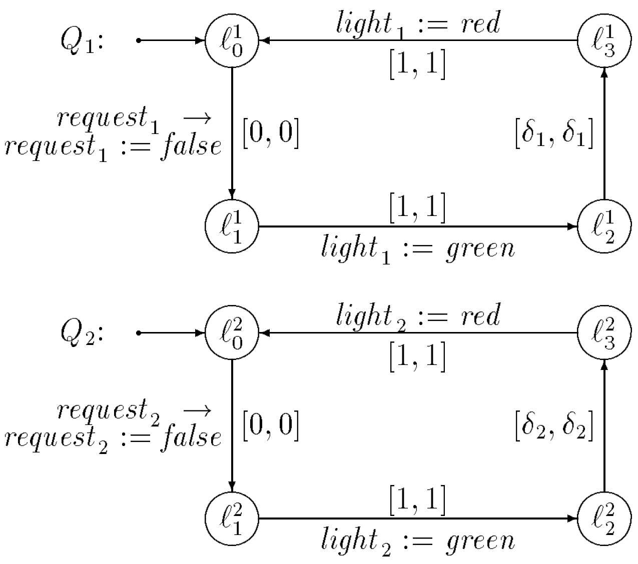
$$P' : \quad \{request = false,\ light = red\}\ [E' \| Q']$$

still satisfies both correctness requirements (A) and (B).

Now consider a more complex traffic light configuration, with two lights and two request buttons. In particular, we assume that the second light is designed for the special convenience of pedestrians in a hurry: it is required to turn green within 3 time units of a request but, on the other hand, has to stay green for only 3 time units. While pedestrians arrive at the first light with a frequency of at most one pedestrian every 15 time units, we assume that the more urgent requests are less frequent — at most one every 30 time units:



The controller for both lights executes the following two processes:



If the combined traffic light controller makes use of *two* processors and the processes Q_1 and Q_2 are executed in a truly concurrent fashion, then the correctness of the entire system

$$P_{\|} : \quad \{request_1 = request_2 = false,\ light_1 = light_2 = red\}\ [E_1 \| E_2 \| Q_1 \| Q_2]$$

follows from the correctness of its parts. Specifically, if $4 \leq \delta_1 \leq 17$ and $2 \leq \delta_2 \leq 23$, then all runs of $P_{\|}$ satisfy the following conditions:

(A_1) Whenever $request_1$ is *true*, then $light_1$ is *green* within 5 time units for 5 time units.

(A_2) Whenever $request_2$ is *true*, then $light_2$ is *green* within 3 time units for 3 time units.

(B_1) Whenever $request_1$ has been *false* for 25 time units, then $light_1$ is *red*.

(B_2) Whenever $request_2$ has been *false* for 25 time units, then $light_2$ is *red*.

A more interesting case is obtained if only a single processor is available to control both lights and the two processes Q_1 and Q_2 have to share it. Using the *interleaving* (shuffle) operator of [Hoa85], we denote the resulting system by the expression

$$P_{|||}: \quad \{request_1 = request_2 = false,\ light_1 = light_2 = red\}\ [E_1 \| E_2 \| (Q_1 ||| Q_2)].$$

Note that the behavior of the environment $E_1 \| E_2$ is still truly concurrent to the behavior of the traffic light controller $Q_1 ||| Q_2$, which executes both processes Q_1 and Q_2 on a single processor in an interleaved fashion.

Let us assume that $\delta_1 = 10$ and $\delta_2 = 2$, a case in which the multiprocessing $P_{||}$ is correct. However, if we have no knowledge about the strategy by which the processes Q_1 and Q_2 are scheduled on the processor they share, other than that it is fair (i.e., the turn of each process will come eventually), then the time-sharing system $P_{|||}$ does not satisfy the specification consisting of the requirements (A_1), (A_2), (B_1), and (B_2). This is easy to see. Suppose that the process Q_1 is, for some time, given priority over the process Q_2, and the traffic light controller receives a request for the second light only 1 time unit after it has received a request for the first light. Then it will serve the first request by turning $light_1$ green and (busy) waiting for 10 time units, thus violating (A_2). On the other hand, if the process Q_2 that serves the more urgent yet less frequent requests is always given priority over the process Q_1, then the system $P_{|||}$ is correct. This is because of the low frequency of requests for the second light only one such request can interrupt the service of a request for the first light.

Before we discuss the modeling of time-sharing, priorities, and interrupts in greater detail, let us first stay with truly concurrent processes and introduce message-passing operations.

3.4 Message passing

Asynchronous message passing can be modeled by shared variables that represent message channels. In this subsection, we extend our timed transition diagram language by a primitive for *synchronous* (CSP-style) message passing, which can be used for the synchronization and communication of parallel processes.

Syntax

A (*message-passing*) *multiprocessing system* P has the form

$$\{\theta\}[P_1 \| \ldots \| P_m],$$

where θ is a data precondition and each process P_i, for $1 \leq i \leq m$, is a sequential nondeterministic real-time program over the finite set $U_i \cup U_s$ of data variables (in the case of pure message-passing systems, $U_s = \emptyset$). We employ again timed transition diagrams to represent processes, but enrich the repertoire of instructions by guarded send and receive operations. The *send* operation $\alpha!e$ outputs the value of the expression e on the channel α. The *receive* operation $\alpha?x$ reads an input value from the channel α and assigns it to the variable x. A send instruction and a receive instruction *match* iff they belong to different processes and address the same channel:

$$P_i\colon\quad \ell^i_j \xrightarrow[{[l,u]}]{c \;\to\; \alpha!e} \ell^i_k$$

$$P_{i'}\colon\quad \ell^{i'}_{j'} \xrightarrow[{[l',u']}]{c' \;\to\; \alpha?x} \ell^{i'}_{k'}$$

For any two matching communication instructions with the delay intervals $[l, u]$ and $[l', u']$, respectively, we require that $max(l, l') \leq min(u, u')$.

Since we use the paradigm of synchronous message passing, a send operation can be executed only jointly with a matching receive operation. Thus the intended operational meaning of the given two edges is as follows. Suppose that, for $max(l, l')$ time units, the control of the process P_i has resided at the location ℓ^i_j and the control of the process $P_{i'}$ has resided at the location $\ell^{i'}_{j'}$ and the guards c and c' have been continuously true. Then P_i and $P_{i'}$ *may* proceed, synchronously, to the locations ℓ^i_k and $\ell^{i'}_{k'}$, respectively. On the other hand, if P_i has resided at ℓ^i_j and $P_{i'}$ has resided at $\ell^{i'}_{j'}$ and the guards c and c' have been continuously true for $min(u, u')$ time units, then both processes *must* proceed. In doing so, the current value of e is assigned to x.

Semantics

Synchronous message passing can be modeled formally by timed transition systems. We define the timed transition system $S_P = \langle V, \Sigma, \Theta, \mathcal{T}, l, u\rangle$ that is associated with the given message-passing multiprocessing system P as in the shared-variables case, only that $\mathcal{T}$ contains an additional transition for every matching pair of communication instructions. Suppose that the two edges E (from ℓ^i_j to ℓ^i_k) and E' (from $\ell^{i'}_{j'}$ to $\ell^{i'}_{k'}$) in the timed transition diagrams for P_i and $P_{i'}$ are labeled by the matching instructions $c \to e!\alpha$ and $c' \to \alpha?x$, respectively. Then

- $\mathcal{T}$ contains, for the matching edges E and E', a transition $\tau_{E,E'}$ such that $\sigma' \in \tau_{E,E'}(\sigma)$ iff

  $\sigma(\pi_i) = \ell^i_j$ and $\sigma'(\pi_i) = \ell^i_k$,
  $\sigma(\pi_{i'}) = \ell^{i'}_{j'}$ and $\sigma'(\pi_{i'}) = \ell^{i'}_{k'}$,
  c and c' are true in σ and $\sigma'(x) = \sigma(e)$,
  $\sigma'(y) = \sigma(y)$ for all $y \in V - \{\pi_i, \pi_{i'}, x\}$.

- If the matching edges E and E' are labeled by the minimal delays l and l', respectively, let $l_{\tau_{E,E'}} = max(l, l')$.

- If the matching edges E and E' are labeled by the maximal delays u and u', respectively, let $u_{\tau_{E,E'}} = min(u, u')$.

This translation defines the set of possible computations of any distributed real-time system P whose processes communicate either through shared variables or by message passing.

Process synchronization

Recall that the component processes of the multiprocessing system $P_1 \| P_2$ may start at arbitrary, even vastly different, times. An important application of synchronous message passing is the synchronization of parallel processes. Let P_1 and P_2 be two real-time processes whose timed transition diagrams have the entry locations ℓ_0^1 and ℓ_0^2, respectively, and let α be a channel. Now consider the two processes $\hat{P}_1$ and $\hat{P}_2$ whose timed transition diagrams are obtained from the transition diagrams for P_1 and P_2 by adding new entry locations:

$$\hat{P}_1: \quad \longrightarrow \hat{\ell}_0^1 \xrightarrow[{[0,0]}]{\alpha!} \ell_0^1$$

$$\hat{P}_2: \quad \longrightarrow \hat{\ell}_0^2 \xrightarrow[{[0,0]}]{\alpha?} \ell_0^2$$

The added message-passing operations have the effect of synchronizing the start of the two processes P_1 and P_2 (whenever message passing is used for the purpose of process synchronization only, the data that is passed between processes is immaterial and the data components of the send and receive instructions are usually suppressed). It follows that the component processes of the multiprocessing system $\hat{P}_1 \| \hat{P}_2$ start synchronously, at the exact same (arbitrary) time.

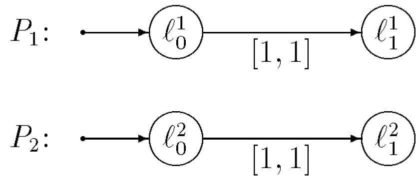
From now on, we shall write $P_1 \|_s P_2$ for the system P whose component processes P_1 and P_2 start synchronously; that is, the notation $P_1 \|_s P_2$ is an abbreviation for the message-passing system $\hat{P}_1 \| \hat{P}_2$. Equivalently, we can directly define the formal semantics S_P of the *synchronous* multiprocessing system $P_1 \|_s P_2$ as containing a single entry transition $\tau_0^{1,2}$ for both processes P_1 and P_2; namely, $\sigma' \in \tau_0^{1,2}(\sigma)$ iff

$$\begin{array}{l}
\sigma(\pi_1) = \sigma(\pi_2) = \bot, \\
\sigma'(\pi_1) = \ell_0^1 \text{ and } \sigma'(\pi_2) = \ell_0^2, \\
\sigma'(y) = \sigma(y) \text{ for all } y \in V - \{\pi_1, \pi_2\}.
\end{array}$$

It is not hard to generalize our notion of synchronous message passing to synchronous broadcasting, which allows arbitrarily many parallel processes to synchronize simultaneously on joint transitions.

4 Modeling Time-critical Multiprogramming Systems

While the interleaving model for concurrency identifies true parallelism (multiprocessing) with nondeterminism (multiprogramming), the *traffic light* example of the previous section suggests that the ability of a system to meet its real-time constraints depends crucially on the number of processors that are available and on the process allocation algorithm. This dependence is already vividly demonstrated by the following trivial system consisting of the two processes P_1 and P_2:



If both processes are executed in parallel on two processors, we denote the resulting system by $P_1 \| P_2$ (or $P_1 \|_s P_2$, if the processes are started at the same time); if they share a single processor and are executed one transition at a time according to some scheduling strategy, the composite system is denoted by $P_1 ||| P_2$.

In the untimed case, it is the very essence of the interleaving semantics to identify both systems with the same set of possible (interleaved) execution sequences — the stuttering closure of the two state sequences

$$(\ell_0^1, \ell_0^2) \xrightarrow{P_1} (\ell_1^1, \ell_0^2) \xrightarrow{P_2} (\ell_1^1, \ell_1^2) \longrightarrow \cdots,$$
$$(\ell_0^1, \ell_0^2) \xrightarrow{P_2} (\ell_0^1, \ell_1^2) \xrightarrow{P_1} (\ell_1^1, \ell_1^2) \longrightarrow \cdots$$

(a state is an interpretation of the two control variables π_1 and π_2). Real time, however, can distinguish between true concurrency and sequential nondeterminism: if both processes start synchronously, then the parallel execution of P_1 and P_2 terminates within 1 time unit; on the other hand, any interleaved sequential execution of P_1 and P_2 takes 2 time units. This distinction must be captured by our model:

1. In the two-processor case $P_1 \|_s P_2$, we obtain as computations the timed state sequences that result from closing the two sequences

   $$(\perp, \perp, 0) \longrightarrow (\ell_0^1, \ell_0^2, 0) \longrightarrow (\ell_0^1, \ell_0^2, 1) \xrightarrow{P_1} (\ell_1^1, \ell_0^2, 1) \xrightarrow{P_2} (\ell_1^1, \ell_1^2, 1) \longrightarrow (\ell_1^1, \ell_1^2, 2) \longrightarrow \cdots,$$
   $$(\perp, \perp, 0) \longrightarrow (\ell_0^1, \ell_0^2, 0) \longrightarrow (\ell_0^1, \ell_0^2, 1) \xrightarrow{P_2} (\ell_0^1, \ell_1^2, 1) \xrightarrow{P_1} (\ell_1^1, \ell_1^2, 1) \longrightarrow (\ell_1^1, \ell_1^2, 2) \longrightarrow \cdots$$

   under stuttering and shifting the origin of time (the third component of every triple denotes the time). The system $P_1 \| P_2$ has more computations, because the time difference between the start of P_1 and the start of P_2 can be arbitrarily large.

2. In the time-sharing case $P_1 ||| P_2$, the set of computations will be defined to be essentially the closure of the two timed state sequences

   $$(\perp, 0) \longrightarrow (\ell_0^1, \ell_0^2, 0) \longrightarrow (\ell_0^1, \ell_0^2, 1) \xrightarrow{P_1} (\ell_1^1, \ell_0^2, 1) \longrightarrow (\ell_1^1, \ell_0^2, 2) \xrightarrow{P_2} (\ell_1^1, \ell_1^2, 2) \longrightarrow \cdots,$$
   $$(\perp, 0) \longrightarrow (\ell_0^1, \ell_0^2, 0) \longrightarrow (\ell_0^1, \ell_0^2, 1) \xrightarrow{P_2} (\ell_0^1, \ell_1^2, 1) \longrightarrow (\ell_0^1, \ell_1^2, 2) \xrightarrow{P_1} (\ell_1^1, \ell_1^2, 2) \longrightarrow \cdots$$

   under stuttering and shifting the origin of time. We write "essentially," because we will augment the states by information about the status of the two processes (either active or suspended). Also, observe that we have silently assumed that the swapping of processes is instantaneous and that neither process has priority over the other process. All of these issues will be discussed in detail.

Thus, for modeling time-critical applications, we can no longer ignore the difference between multiprocessing and multiprogramming. In this section, we first show how our model extends to concrete real-time systems that consist of a fixed number of sequential programs that are executed, by time-sharing, on a single processor. Then we use our framework to represent general, distributed multiprogramming systems, in which several processes share a pool of processors statically or dynamically.

4.1 Syntax and semantics

A *multiprogramming system* P has the form

$$\{\theta\}[P_1|||\ldots|||P_m].$$

Each process P_i, for $1 \leq i \leq m$, is again a sequential nondeterministic real-time program over the finite set U of data variables, whose initial values satisfy the data precondition θ. We represent the real-time programs P_i by timed transition diagrams as before. Note, however, that in the multiprogramming case the control of the (single) processor resides at one particular location of one particular process. Thus the intended operational meaning of the edge

$$\ell^i_j \xrightarrow[{[l,u]}]{c \;\to\; x := e} \ell^i_k$$

is as follows. The minimal delay l guarantees that whenever the control (of the single processor) has resided at the location ℓ^i_j for at least l time units and the guard c is true, then the control *may* proceed to the location ℓ^i_k. The maximal delay u ensures that whenever the control has resided at ℓ^i_j for u time units and the guard c is true, then it *must* proceed to ℓ^i_k. This is because, in the single-processor case, no other process can interfere with the active process and change the value of c.

The operational view of the concrete model is again captured formally by a translation into timed transition systems. With the given multiprogramming system P, we associate the following timed transition system $S_P = \langle V, \Sigma, \Theta, \mathcal{T}, l, u\rangle$:

1. $V = U \cup \{\mu, \pi_1, \ldots \pi_m\}$. There are two kinds of control variables: the *processor control variable* μ ranges over the set $\{1, \ldots m, \perp\}$; each *process control variable* π_i, for $1 \leq i \leq m$, ranges over the set L_i of locations of the process P_i. The value of the processor control variable μ is $\perp$ (undefined) before the (single) processor starts executing processes. Thereafter the control of the processor resides at the location π_μ of the process P_μ. We say that the process P_μ is *active*, while all other processes P_i, for $i \neq \mu$, are *suspended* (if the value of μ is undefined, then all processes are suspended). The process control variable π_i of a suspended process indicates the location at which the execution of P_i will resume when P_i (re)gains control of the processor.

2. Σ contains all interpretations of V.

3. Θ is the set of all states $\sigma \in \Sigma$ such that θ is true in σ, and $\sigma(\mu) = \perp$, and $\sigma(\pi_i) = \ell^i_0$ for all $1 \leq i \leq m$.

4. $\mathcal{T}$ contains, in addition to the idle transition τ_I, an *action* transition τ_E for every edge E in the timed transition diagrams for $P_1, \ldots P_m$. If E connects the source location ℓ^i_j to the target location ℓ^i_k and is labeled by the instruction $c \rightarrow \bar{x} := \bar{e}$, then $\sigma' \in \tau_E(\sigma)$ iff

   $\sigma(\mu) = i$,
   $\sigma(\pi_i) = \ell^i_j$ and $\sigma'(\pi_i) = \ell^i_k$,
   c is true in σ and $\sigma'(\bar{x}) = \sigma(\bar{e})$,
   $\sigma'(y) = \sigma(y)$ for all $y \in V - \{\pi_i, \bar{x}\}$.

   Furthermore, there are *scheduling* transitions $\tau \in \mathcal{T}$ that change the status of the processes by resuming a suspended process: $\sigma' \in \tau(\sigma)$ implies that

   $\sigma'(y) = \sigma(y)$ for all $y \in U$.

   The scheduling policy determines the set of scheduling transitions. A scheduling transition τ is called an *entry* transition iff it is enabled on some initial states. We restrict ourselves to scheduling policies with a single entry transition, τ_0, that is enabled on all initial states. Moreover, we require that $\sigma' \in \tau_0(\sigma)$ implies that

   $\sigma(\mu) = \perp$,
   $\sigma'(y) = \sigma(y)$ for all $y \in V - \{\mu\}$;

   that is, the entry transition τ_0 is enabled precisely on the initial states and activates, perhaps nondeterministically, one of the competing processes.

5. For every edge E labeled by the minimal delay l, let $l_{\tau_E} = l$. Furthermore, $l_{\tau_0} = 0$.

6. For every edge E labeled by the maximal delay u, let $u_{\tau_E} = u$. Furthermore, $u_{\tau_0} = \infty$.

The computations of S_P clearly depend on the scheduling transitions and their delays. In the untimed case, the scheduling issue can be reduced to fairness assumptions about the scheduling policy: correctness of an untimed multiprogramming system is generally shown for all fair scheduling strategies. It makes, however, little sense to to desire that a multiprogramming system satisfies a real-time requirement under all (fair) scheduling strategies, because the scheduling algorithm usually determines if a system meets its timing constraints. In fact, fair scheduling strategies admit *thrashing*: by switching control too often between processes, no action transition may be enabled long enough so that it has to be taken and only scheduling transitions will be performed. Thus the system may make no real progress at all and may certainly not meet any real-time deadlines. Consequently, we study the correctness of real-time multiprogramming systems always with respect to a particular given scheduling policy.

4.2 Scheduling strategies

Our selection of scheduling strategies is neither intended to be categorical nor comprehensive; we simply try to examine what we think is a representative variety of different scheduling mechanisms and, in the process, hope to convince ourselves of the utility of the timed transition system model. Throughout this subsection, we assume a fixed multiprogramming system

$$P: \quad \{\theta\}[P_1|||\ldots|||P_m]$$

and define the scheduling transitions of the associated timed transition system S_P for various scheduling algorithms.

Greedy scheduling

The simplest reasonable scheduling strategy, as well as our default strategy, is *greedy*. According to this policy, the process that is currently in control of the processor remains active until all its transitions are disabled. At this point an arbitrary other process with an enabled transition takes over. Formally, the set $\mathcal{T}$ of transitions for the timed transition system S_P contains, in addition to the entry transition τ_0, a single scheduling transition, τ_G, with $\sigma' \in \tau_G(\sigma)$ iff

$\sigma(\mu) \neq \perp$,
$\sigma'(y) = \sigma(y)$ for all $y \in V - \{\mu\}$,
$\tau_E(\sigma) = \emptyset$ for all action transitions τ_E,
$\tau_E(\sigma') \neq \emptyset$ for some action transition τ_E.

If there is no cost associated with swapping processes, then $l_{\tau_G} = u_{\tau_G} = 0$. If switching processes is not instantaneous, then the minimal and maximal delays of τ_G should be adjusted accordingly.

Scheduling instructions

More flexible scheduling strategies can be implemented with explicit scheduling operations. For this purpose, we enrich our programming language by the instruction $resume(s)$, where $s \subseteq \{1, \ldots m\}$ determines a subset of processes. The scheduling operation $resume(s)$ suspends the currently active process, say, P_i and activates, nondeterministically, one of the processes $P_{i'}$ with $i' \in s$:

$$\ell^i_j \xrightarrow[{[l,u]}]{c \;\rightarrow\; resume(s)} \ell^i_k$$

We write $resume(i')$ for $resume(\{i'\})$ and *suspend* for $resume(\{1 \leq i' \leq m \mid i' \neq i\})$; that is, the instruction *suspend* delegates the control from the currently active process to any one of the competing processes.

Formally, the set $\mathcal{T}$ of transitions of S_P contains, in addition to the entry transition τ_0, a scheduling transition τ_E for every *resume* edge E in the timed transition diagrams for $P_1, \ldots P_m$. If E connects the source location ℓ^i_j to the target location ℓ^i_k and is labeled by the instruction $c \rightarrow resume(s)$, then $\sigma' \in \tau_E(\sigma)$ iff

$\sigma(\mu) = i$ and $\sigma'(\mu) \in s$,
$\sigma(\pi_i) = \ell^i_j$ and $\sigma'(\pi_i) = \ell^i_k$,
c is true in σ,
$\sigma'(y) = \sigma(y)$ for all $y \in V - \{\mu, \pi_i\}$.

Furthermore, for every scheduling edge E labeled by the minimal delay l and the maximal delay u, let $l_{\tau_E} = l$ and $u_{\tau_E} = u$.

Delays and timers

Note that the instruction

$$\ell^i_j \xrightarrow[{[10,10]}]{} \ell^i_k$$

models a busy wait; the process P_i occupies the processor for 10 time units while waiting. To implement a nonbusy wait, in which P_i releases the processor to a competing process for 10 time units before resuming execution, we use a *timer* (alarm clock) $T_{[10,10]}$ as a parallel process:

$$\begin{tikzcd} & t_1 \arrow[d, bend right=60, "{resume(i)\ [10,10]}"'] \\ T_{[10,10]}: \arrow[r] & t_0 \arrow[u, bend right=60, "{[0,0]\ t? \ \to\ t := false}"'] \end{tikzcd}$$

We make sure that the timer $T_{[10,10]}$ is started (i.e., waiting for activation) when the process P_i becomes active, with the precondition $\{t = \mathit{false}\}$. Then the timer is activated by the process P_i with the sequence

$$\ell^i_j \xrightarrow[{[0,0]}]{t := true} \ell^i_k \xrightarrow[{[0,0]}]{resume(s)} \ell^i_n$$

which releases the control of the processor to one of the processes in the set s. After exactly 10 time units, the timer process, which operates in parallel, will return the control of the processor to the process P_i.

In general, a timer process $T_{[l,u]}$ marks nondeterministically a time period between l and u time units and is executed in parallel to the other processes of a system:

$$\{\theta\}[(P_1|||\ldots|||P_m)\|_s T_{[l,u]}].$$

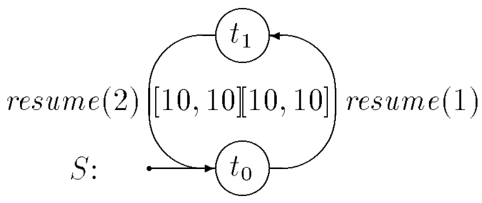
The activation of the timer $T_{[l,u]}$ is abbreviated by the *delay* instruction

$$\ell^i_j \xrightarrow[{[l,u]}]{delay(s)} \ell^i_k$$

The *delay* instruction allows us to program nonbusy delays without explicitly mentioning timers; we simply assume that there exists, implicitly, a unique timer process for every *delay* instruction in a timed transition diagram. The parameter s of the instruction *delay*(s) determines the set of processes from which a process is selected when the active process is delayed.

Round-robin scheduling

A construction that is similar to the timer example allows us to implement a *round-robin* scheduling strategy for two processes P_1 and P_2 that share a single processor. In the system $(P_1|||P_2)\|_s S$, the scheduler



gives each of the two processes P_1 and P_2 in turn 10 time units of processor time. Needless to say, the explicit scheduling instructions give us the ability to design more sophisticated schedulers as well.

4.3 Processor allocation

Both the multiprogramming system with a timer and the multiprogramming system with a central scheduler are, in fact, combinations of multiprocessing and multiprogramming systems in which several tasks compete for some of the processors. For these systems, the question of *scheduling*, which determines the processor time that is granted to individual processes, is preceded by the question of *processor allocation*, which determines the assignment of processes to processors. This assignment can be either static, if every process is assigned to a fixed processor, or dynamic, if a set of processes competes for a pool of processors and processes may reside, over time, at different processors. We only hint how this very general notion of real-time system fits into our framework and can be modeled by timed transition systems.

A *static* (shared-variables or message-passing) system P with k processors is of the form

$$\{\theta\}[(P_{1,1}|||\ldots|||P_{1,m_1})\|\ldots\|(P_{k,1}|||\ldots|||P_{k,m_k})];$$

that is, m_i processes compete for the i-th processor. The definition of the associated timed transition system S_P is straightforward: every processor has its own process control variable μ_i, for $1 \leq i \leq k$, which ranges over the set of competing processes $\{1, \ldots m_i, \perp\}$ and designates the active process. Furthermore, every processor operates according to a local scheduling policy with a single entry transition τ_0^i for $1 \leq i \leq k$.

To model systems in which a process competes for more than one processor, we write

$$\{\theta\}[P_1, \ldots, P_m]_k$$

for the *dynamic* system in which m processes compete for k processors according to some global processor allocation and scheduling policy. To define dynamic systems, it is useful to have a more general scheduling instruction, $resume(s, x)$, which interrupts the process that is currently active on processor x and activates, on processor x, one of the processes from the set s.

4.4 Priorities and interrupts

While the scheduling instruction *resume* gives us the flexibility to design a scheduler, we often wish to adopt a simple, static scheduling strategy without having to explicitly construct a scheduler. In this subsection, we offer this possibility by generalizing the *greedy* strategy. We assign a priority to every transition, and at any point in a computation, choose only among the transitions with the highest priority. If the transition with the highest priority belongs to a suspended process, then the currently active process is interrupted and the execution of the suspended process is resumed.

A *priority* system P is a (shared-variables or message-passing, static or dynamic) system in which a priority is associated with every instruction; that is, with every edge in the timed transition diagrams for P. We use nonnegative integers as priorities (0 being the *highest* priority) and annotate an edge with a priority $p \in \mathsf{N}$ as follows:

$$\ell^i_j \xrightarrow[{[l,u]}]{p:\ c\ \rightarrow\ x := e} \ell^i_k$$

We formalize the priority semantics only for simple multiprogramming systems; the generalization to systems with several processors is straightforward. With a given priority system

$$P: \quad \{\theta\}[P_1 ||| \dots ||| P_m],$$

we associate the following timed transition system $S_P = \langle V, \Sigma, \Theta, \mathcal{T}, l, u\rangle$:

- V, Σ, and Θ are as before.
- $\mathcal{T}$ contains, in addition to τ_I, an action transition τ_E for every assignment edge E in the transition diagrams for $P_1, \dots P_m$. If E connects the source location ℓ^i_j to the target location ℓ^i_k and is labeled by the instruction p: $c \rightarrow \bar{x} := \bar{e}$, then the transition τ_E competes in state σ and would lead to state σ' (which is denoted by $\sigma \rightarrow_E \sigma'$) iff

  $\sigma(\pi_i) = \ell^i_j$ and $\sigma'(\pi_i) = \ell^i_k$,
  c is true in σ and $\sigma'(\bar{x}) = \sigma(\bar{e})$,
  $\sigma'(y) = \sigma(y)$ for all $y \in V - \{\mu, \pi_i, \bar{x}\}$.

  Then $\sigma' \in \tau_E(\sigma)$ iff

  $\sigma \rightarrow_E \sigma'$ and $\sigma(\mu) = \sigma'(\mu) = i$ and
  there is no edge E' that is labeled by a higher priority $p' < p$ such that
  $\sigma \rightarrow_{E'} \sigma''$ for some σ''.

  For any matching pair of communication edges E and E' that are labeled by the priorities p and p', respectively, we take the higher priority $min(p, p')$ for the combined transition $\tau_{E,E'}$ (although this choice is arbitrary and may be reversed, if the need arises).

  Furthermore, there is, in addition to the entry transition τ_0, a scheduling transition τ_P such that $\sigma' \in \tau_P(\sigma)$ iff

$\sigma(\mu) \neq \perp$,
$\sigma'(y) = \sigma(y)$ for all $y \in V - \{\mu\}$,
$\tau_E(\sigma) = \emptyset$ for all action transitions τ_E,
$\tau_E(\sigma') \neq \emptyset$ for some action transition τ_E.

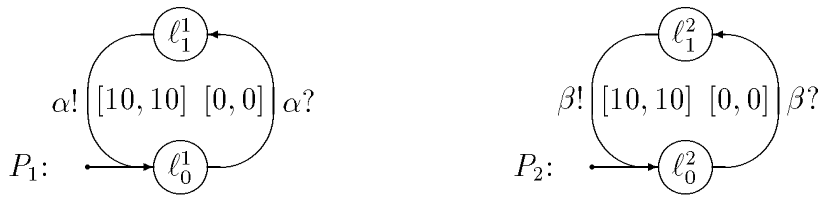
- Let l_{τ_E} and u_{τ_E} be as before, and choose l_{τ_P} and u_{τ_P} to represent the cost of swapping processes.

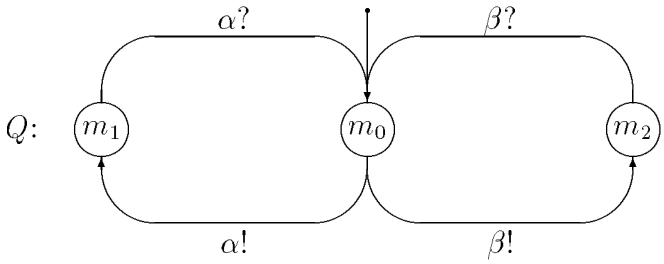
Note that if all transitions have equal priority, then the scheduling strategy is greedy (that is, $\tau_G = \tau_P$). Thus priorities generalize our previous discussion conservatively: all systems can be viewed as priority systems whose instructions have the same default priority, unless they are annotated with explicit priorities.

Dynamic priorities

Priorities can be combined with explicit scheduling operations in the obvious way. It is, however, often more convenient to model dynamic scheduling strategies, which change over time, by *dynamic* priorities, which can be modified by any process during execution. Dynamic priorities offer exciting possibilities, such as the ability of a process to increase or decrease its own priority. Moreover, they are easily incorporated into our framework. We simply use data variables that range over the nonnegative integers $\mathbb{N}$ as priorities. Instead of giving the formal semantics of dynamic priorities, which is constructed straightforwardly from the semantics of constant (static) priorities, we present an interesting real-time application of dynamic priorities.

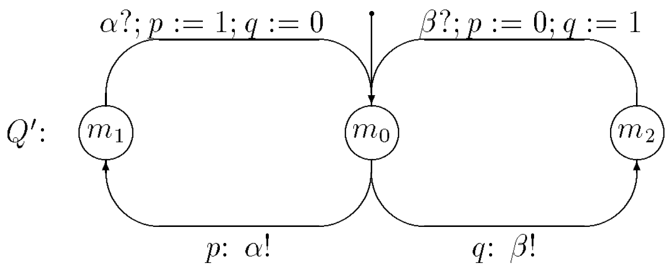
We have not yet pointed out that our interpretation of message passing is not entirely conservative over the untimed case: there the set of legal execution sequences usually is restricted by strong-fairness assumptions for communication transitions [MP89]. This is convenient for the study of time-independent properties of a system, where simple fairness assumptions about "nondeterministic" branching points abstract complex implementation details. Consider, for example, the multiprocessing system $P_1 \| P_2 \| Q$ that consists of the following three processes P_1, P_2, and Q:





(Recall that we may omit the data components of message-passing operations, if they are immaterial.) The arbiter Q mediates between the two processes P_1 and P_2 and uses synchronous communication on the two channels α and β to ensure mutual exclusion: P_1 and P_2 can never be simultaneously in their critical sections ℓ_1^1 and ℓ_1^2, respectively.

Strong-fairness assumptions on the communication transitions are used to guarantee that, in addition to mutual exclusion, neither of the two processes P_1 and P_2 is shut out from its critical section forever: the arbiter cannot always prefer one process over the other. Any such infinitary fairness assumption, however, is clearly without bearing on the satisfaction of a real-time requirement such as the demand that a process has to wait at most 10 time units before being able to enter its critical section. As has been the case with scheduling, we encounter again a situation in which the infinitary notion of "fairness" is adequate for proving untimed properties, yet entirely inadequate for proving timing constraints. To verify compliance with real-time requirements, we can no longer forgo an explicit description of how the arbiter Q decides between the two processes P_1 and P_2 when both are waiting to enter their critical sections. For instance, the following refinement Q' of Q never makes the same "nondeterministic" choice twice in a row:



(We use semicolons to group several instructions to an atomic transition, which is defined in the obvious fashion. The default value of priorities is assumed to be 1.) The arbiter Q' modifies the priorities p and q of its nondeterministic alternatives to ensure that the system

$$\{p = q = 1\}[P_1 \| P_2 \| Q']$$

satisfies the requirement that each process has to wait at most 10 time units before being able to enter its critical section. Note that none of the two nondeterministic alternatives is ever disabled, but, at any time, one of them is "preferred."

Finitary branching fairness

Since infinitary fairness assumptions, such as weak fairness for scheduling and strong fairness for synchronization, are insufficient to guarantee the satisfaction of real-time deadlines, one may choose to add finitary branching conditions to timed transition systems. Such a finitary notion of fairness would restrict the nondeterminism of a system. We may want to require, for example, that no competitor of a transition τ can be taken more than n times without τ itıself being taken (a similar concept has been called *bounded fairness* in [Jay88]). We prefer, both for scheduling and synchronization, an explicit description of the selection process to such implicit assumptions. Since all selection processes that we have found useful can be described within our language, we see no need to introduce additional concepts that would only complicate any verification methodology.

5 Conclusion

With timed transition systems, we presented an abstract model for real-time systems. We then explored its scope and demonstrated its practicality by modeling many important constructs of real-time computing. We conclude by pointing out that any abstract description of a real-time system ought to meet certain theoretical requirements as well. In [Hen91b], we suggested that system descriptions be (1) refinable, (2) digitizable, and (3) operational, and we showed that timed transition systems satisfy all three criteria:

1. We associated a fixed set of global states with every real-time system. This static view prohibits the study of large systems for managerial reasons. With each step in the hierarchical specification, design, and verification of a complex system, the state space must be refinable through expansion of its visible portion [Lam83]. The vertical decomposition of systems can be formalized by refinement mappings between increasingly detailed system descriptions [AL88]. Since expanding the visible portion of the state space may increase the frequency at which state changes occur, the timed state sequence semantics of a system description needs to be closed under our notion of stuttering to guarantee the existence of refinement mappings. The *refinability* of timed transition systems is further discussed in [Hen91b].

2. For verification, it is often convenient, and sometimes necessary, to assume a fictitious digital clock, which records the times of state changes with finitary precision only [AH89]. The integers suffice as time domain for such a digital-clock model. For the direct use of a digital-clock model, a system description must be independent of the time domain, in the sense that digitizing all execution sequences of a system can be achieved by simply changing the time domain from the reals to the integers. Sets of timed state sequences that enjoy this property are called digitizable. The *digitizability* of timed transition systems and the ensuing amenability to discrete-time verification methods is discussed in [Hen91b].

3. We want a system description to be executable. For untimed systems, this is the case if the liveness component of a system description does not preclude any safe prefixes of execution sequences; that is, a stepwise interpreter cannot "paint itself into a corner" from which no continuation is possible [AFK88]. The issue is more subtle in the timed case because of the implicit liveness requirement that time must progress eventually. In other words, we wish to rule out system descriptions that prevent time from progressing. Such system descriptions are called operational [Hen91a] (or machine-closed [LA]). The *operationality* of timed transition systems is discussed in [Hen91b].

No study of an abstract model for real-time systems is complete without progress on the verification front. Let P be a real-time system whose set of possible timed executions is the set $[\![P]\!]$ of timed state sequences and let ϕ be a specification that is satisfied by the timed state sequences in the set $[\![\phi]\!]$. Verification of P with respect to ϕ amounts, then, to checking the containment

$$[\![P]\!] \stackrel{?}{\subseteq} [\![\phi]\!]$$

of sets of timed state sequences. Systems are given as expressions of an implementation language; specifications as expressions of a specification language. We presented the implementation language of timed transition diagrams and defined the set $[\![P]\!]$ of timed executions for systems that are given in the timed transition diagram language as the set of computations for the timed transition system S_P. Verification methods of timed transition systems have been developed for various logical specification languages. The methods include both algorithmic techniques for finite-state systems [AH89, AH90, HLP90, Ost90, Hen91b] and deductive techniques based on proof systems [Hen90, Ost90, HMP91, Hen91b]. For an overview of the verification methods for timed transition systems, we refer to the article *Logics and Models of Real Time* in this volume.

Acknowledgment. The authors thank Rajeev Alur and Eddie Chang for many helpful comments.

References

[AFK88] K.R. Apt, N. Francez, and S. Katz. Appraising fairness in languages for distributed programming. *Distributed Computing*, 2(4):226–241, 1988.

[AH89] R. Alur and T.A. Henzinger. A really temporal logic. In *Proceedings of the 30th Annual Symposium on Foundations of Computer Science*, pages 164–169. IEEE Computer Society Press, 1989.

[AH90] R. Alur and T.A. Henzinger. Real-time logics: complexity and expressiveness. In *Proceedings of the Fifth Annual Symposium on Logic in Computer Science*, pages 390–401. IEEE Computer Society Press, 1990.

[AL88] M. Abadi and L. Lamport. The existence of refinement mappings. In *Proceedings of the Third Annual Symposium on Logic in Computer Science*, pages 165–175. IEEE Computer Society Press, 1988.

[BKP86] H. Barringer, R. Kuiper, and A. Pnueli. A really abstract concurrent model and its temporal logic. In *Proceedings of the 13th Annual Symposium on Principles of Programming Languages*, pages 173–183. ACM Press, 1986.

[Har88] E. Harel. Temporal analysis of real-time systems. Master's thesis, The Weizmann Institute of Science, Rehovot, Israel, 1988.

[Hen90] T.A. Henzinger. Half-order modal logic: how to prove real-time properties. In *Proceedings of the Ninth Annual Symposium on Principles of Distributed Computing*, pages 281–296. ACM Press, 1990.

[Hen91a] T.A. Henzinger. Sooner is safer than later. Technical report, Stanford University, 1991.

[Hen91b] T.A. Henzinger. *The Temporal Specification and Verification of Real-time Systems*. PhD thesis, Stanford University, 1991.

[HLP90] E. Harel, O. Lichtenstein, and A. Pnueli. Explicit-clock temporal logic. In *Proceedings of the Fifth Annual Symposium on Logic in Computer Science*, pages 402–413. IEEE Computer Society Press, 1990.

[HMP90] T.A. Henzinger, Z. Manna, and A. Pnueli. An interleaving model for real time. In *Proceedings of the Fifth Jerusalem Conference on Information Technology*, pages 717–730. IEEE Computer Society Press, 1990.

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

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, 1985.

[Jay88] D.N. Jayasimha. *Communication and Synchronization in Parallel Computation.* PhD thesis, University of Illinois at Urbana-Champaign, 1988.

[Kel76] R.M. Keller. Formal verification of parallel programs. *Communications of the ACM*, 19(7):371–384, 1976.

[KSdR+88] R. Koymans, R.K. Shyamasundar, W.-P. de Roever, R. Gerth, and S. Arun-Kumar. Compositional semantics for real-time distributed computing. *Information and Computation*, 79(3):210–256, 1988.

[LA] L. Lamport and M. Abadi. Refining and composing real-time specifications. This volume.

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

[MP89] Z. Manna and A. Pnueli. The anchored version of the temporal framework. In J.W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *Linear Time, Branching Time, and Partial Order in Logics and Models for Concurrency*, Lecture Notes in Computer Science 354, pages 201–284. Springer-Verlag, 1989.

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

[PH88] A. Pnueli and E. Harel. Applications of temporal logic to the specification of real-time systems. In M. Joseph, editor, *Formal Techniques in Real-time and Fault-tolerant Systems*, Lecture Notes in Computer Science 331, pages 84–98. Springer-Verlag, 1988.

[Pnu77] A. Pnueli. The temporal logic of programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, pages 46–57. IEEE Computer Society Press, 1977.