Temporal Proof Methodologies for Timed Transition Systems*†

Thomas A. Henzinger
Zohar Manna
Amir Pnueli

TR 93-1330
March 1993

Department of Computer Science
Cornell University
Ithaca, NY 14853-7501

---

*This research was supported in part by an IBM graduate fellowship, by the National Science Foundation under grants CCR-8911512, CCR-8913641, and CCR-9200794, by the Defense Advanced Research Projects Agency under contract N00039-84-C-0211, by the United States Air Force Office of Scientific Research under contracts AFOSR-90-0057 and F49620-93-1-0056, and by the European Community ESPRIT Basic Research Action project 3096 (SPEC).
† A preliminary version of Part I of this paper appeared in the proceedings of the 1991 REX Workshop on *Real Time: Theory in Practice* [HMP92a]; a preliminary version of Part II appeared in the proceediings of the 1991 ACM Symposium on *Principles of Programming Languages* [HMP91].

Temporal Proof Methodologies for Timed Transition Systems[1,2]

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

Zohar Manna
Department of Computer Science
Stanford University
Stanford, CA 94305

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

Abstract. We extend the specification language of temporal logic, the corresponding verification framework, and the underlying computational model to deal with real-time properties of reactive systems. The abstract notion of timed transition systems generalizes traditional transition systems conservatively: qualitative fairness requirements are replaced (and superseded) by quantitative lower-bound and upper-bound timing constraints on transitions. This framework can model real-time systems that communicate either through shared variables or by message passing and real-time issues such as timeouts, process priorities (interrupts), and process scheduling.

We exhibit two styles for the specification of real-time systems. While the first approach uses time-bounded versions of the temporal operators, the second approach allows explicit references to time through a special clock variable. Corresponding to the two styles of specification, we present and compare two different proof methodologies for the verification of timing requirements that are expressed in these styles. For the *bounded-operator* style, we provide a set of proof rules for establishing bounded-invariance and bounded-response properties of timed transition systems. This approach generalizes the standard temporal proof rules for verifying invariance and response properties conservatively. For the *explicit-clock* style, we exploit the observation that every time-bounded property is a safety property and use the standard temporal proof rules for establishing safety properties.

[1]This research was supported in part by an IBM graduate fellowship, by the National Science Foundation under grants CCR-8911512, CCR-8913641, and CCR-9200794, by the Defense Advanced Research Projects Agency under contract N00039-84-C-0211, by the United States Air Force Office of Scientific Research under contracts AFOSR-90-0057 and F49620-93-1-0056, and by the European Community ESPRIT Basic Research Action project 3096 (SPEC).

[2]A preliminary version of Part I of this paper appeared in the proceedings of the 1991 REX Workshop on *Real Time: Theory in Practice* [HMP92a]; a preliminary version of Part II appeared in the proceedings of the 1991 ACM Symposium on *Principles of Programming Languages* [HMP91].

1 Introduction

It is self-evident that the most sensitive and critical among reactive systems, and therefore the ones for which formal methods are needed most direly, are real-time systems. The qualitative requirement of *responsiveness*, that every environment stimulus p must be followed by a system response q, is not adequate for systems with hard real-time deadlines; timeliness has to be ensured through the stronger quantitative requirement of *timed responsiveness*, which imposes a bound on the time interval that is permissible between the stimulus p and the response q. Temporal logic has been used successfully for the specification and verification of qualitative properties of reactive systems (see, for example, [Pnu86] for a survey). Over the past few years, there have been several suggestions for extending the expressive power of temporal logic to handle quantitative timing constraints. These attempts can be roughly classified into two approaches.

The first approach, to which we refer as the *bounded-operator* approach, introduces for each temporal operator, such as $\diamond$ (*eventually*) one or more time-bounded versions. For example, while the formula $\diamond q$ asserts that the event q will happen "eventually" but puts no time bound on when it will happen, the formula $\diamond_{\leq 3}\, q$ predicts an occurrence of q within 3 time units from now. The early proposal [BH81] can be viewed as a precursor of this approach to the specification of timing properties, which has been advocated in [KVdR83, KdR85, Koy90], where the bounded-operator language is called *Metric Temporal Logic*, and in [SPE84]. Bounded-operator temporal logics have been analyzed for their complexity and expressiveness in [EMSS89] and in [AH90, AFH91].

An alternative approach to the specification of timing constraints of reactive systems introduces no new temporal operators but interprets, at each state, one of the nonrigid state variables (we use the variable *now*) as the current time. We refer to this approach as the *explicit-clock* approach, because the only new element is the ability to refer explicitly to the clock. Scattered examples of this method of expressing timing properties have been presented in [PdR82], in [Ron84], and in [Har88, PH88]. A more systematic exposition of this approach and its applications can be found in [Ost90], where the explicit-clock language is called *Real-time Temporal Logic*, and in [LA92]. Explicit-clock temporal logics have been analyzed for their complexity and expressiveness in [AH90] and in [HLP90].

To compare the two approaches, consider the requirement of a timely response q to stimulus p within at most 3 time units. In the bounded-operator approach, this requirement is specified by the formula

$$p \;\rightarrow\; \diamond_{\leq 3}\, q.$$

In the explicit-clock approach, it is expressed by the formula

$$(p \wedge \mathit{now} = t) \;\rightarrow\; \diamond (q \wedge \mathit{now} \leq t + 3),$$

where the rigid variable t is used to record the time of the p-state.

The main contribution of this paper is the elaboration of two proof systems that correspond, respectively, to the two styles for the specification of timing requirements.

It is a well-known observation that with the introduction of an explicit-clock variable, all time-bounded properties can be defined by safety formulas ([Hen92]). For example, the timed response property that, in either style, is expressed above by a liveness-like formula (employing the liveness operator $\diamond$) can alternatively be specified by a formula that uses the clock variable *now* and the safety operator U (*unless*):

$$(p \wedge \mathit{now} = t) \;\rightarrow\; (\mathit{now} \leq t + 3)\, \mathsf{U}\, q.$$

This formula asserts that if p happens at time t, then from this point on the time will not exceed $t+3$ either forever (which is ruled out by an axiom that requires time to progress eventually) or until q happens. Consequently, q must occur within 3 time units from p. It follows from this translation that no new proof rules are necessary for the explicit-clock style of timed specification; all time-bounded properties can, in principle, be verified using a standard, uniform set of timeless rules.

On the other hand, when pursuing the bounded-operator style of timed specification, one discerns a clear dichotomy between *upper-bound* properties such as the bounded-response formula $p \rightarrow \Diamond_{\leq 3}\, q$ considered above, and *lower-bound* properties, such as the bounded-invariance formula

$$p \rightarrow \Box_{<3} \neg q,$$

which states that q cannot happen sooner than 3 time units after any occurrence of p. While upper-bound properties assert that "something good" will happen within a specified amount of time, lower-bound properties assert that "nothing bad" will happen for a certain amount of time. Clearly, while the class of upper-bound properties bears a close resemblance to *liveness* properties, the class of lower-bound properties closely resembles *safety* properties. The proof system we present cultivates both similarities by including separate proof principles for the classes of lower-bound and upper-bound properties. These proof principles can easily be seen to be natural extensions of the standard proof rules for the untimed safety (invariance) and liveness (response) classes, respectively.

In our model, we assume a global, discrete, and asynchronous clock, whose actions (clock ticks) are interleaved with the other system actions ([HMP90]). 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 this 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.

Part I discusses the modeling of real-time systems by transition systems. In Section 2, we introduce the abstract computational model of timed transition systems. The subsequent two sections illustrate how concrete real-time systems can be mapped into this model and how typical real-time phenomena can expressed within the 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.

Part II follows with techniques for the verification of timed transition systems. Section 5 introduces the bounded-operator specification language, and Section 6 presents a proof system for this language. In Section 7, we discuss the alternative, explicit-clock, approach. Section 8 concludes by giving completeness results for both methods.

Part I
Modeling Real-time Systems

We define the formal semantics of a real-time system as a set of timed execution sequences. This is done in two steps. First, we introduce the *abstract* notion 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 to interpret the concrete constructs within the abstract model. We demonstrate that this framework can model a wide variety of real-time phenomena that are encountered in practice, including the timed execution of both multiprocessing and multiprogramming systems.

2 Abstract Model: Timed Transition Systems

The basic computational model we use is that of transition systems ([Kel76, Pnu77]). We generalize this model by imposing timing constraints on the transitions and show that the timing constraints can be interpreted operationally. Similar state-transition formalisms with minimal and maximal delays on the transitions have been defined in [PH88], in [Ost90], and in [LA90, MMT91].

2.1 Transition systems

A transition system is a set of states together with a set of transitions (actions) that transform states. In this paper, we will be concerned with a concrete notion of state that provides values for a certain set of variables. Hence we define a *transition system* $S = \langle V, \Sigma, \Theta, \mathcal{T} \rangle$ as a tuple consisting 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 $\sigma = \sigma_0\sigma_1\sigma_2\ldots$ of states is an *initialized 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 and *completed* at position $i+1$ of σ.

Any execution sequence can, therefore, be thought as consisting of an initial state and an infinite sequence of transitions. Finite (terminating as well as deadlocking) execution sequences are generally represented by infinite extensions that add only idle transitions. The computations of a transition system are obtained by closing the set of initialized computations under suffixes: the state sequence σ is a *computation* of S iff σ is a suffix of an initialized computation of S.

2.2 Adding timing constraints to transition systems

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 the following *operationality restriction* on these transitions: 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 $0 \leq i < n$.

The requirement that all transitions that are initially enabled are not constrained by a finite maximal delay will be consistent with our interpretation that the initial state of an execution sequence is the state in which the system has resided for an infinite amount of past time. The operationality requirement that there is no loop of transitions with a combined maximal delay of 0 will ensure that the system cannot prevent time from advancing.

Timed state sequences

We model the values of a fictitious global clock by the integers Z. The clock ticks are interleaved with the system actions, and thus at any point of an execution sequence of a timed system, either the system state changes or the clock value changes (or neither). A *timed state sequence* $\rho = (\sigma, T)$ consists of an infinite sequence σ of states $\sigma_i \in \Sigma$, where $i \geq 0$, and an infinite sequence T of corresponding times (clock values) $T_i \in \mathsf{Z}$ that satisfy the following conditions:

Bounded monotonicity For all $i \geq 0$,

either $T_{i+1} = T_i$,
or $T_{i+1} = T_i + 1$ and $\sigma_{i+1} = \sigma_i$;

that is, time never decreases. It may increase, by at most 1, 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 *stuttering step*; the case that the time increases by 1 is called a *clock tick*.

Progress For all $i \geq 0$ there is some $j > i$ such that $T_i < T_j$; that is, time never stagnates. Thus there are infinitely many clock ticks in every timed state sequence.

We will also write the timed state sequence ρ as an infinite sequence

$$(\sigma_0, T_0) \longrightarrow (\sigma_1, T_1) \longrightarrow (\sigma_2, T_2) \longrightarrow (\sigma_3, T_3) \longrightarrow \cdots$$

of state-time pairs. By $\rho^i = (\sigma^i, T^i)$ we denote the i-th *suffix* of the timed state sequence ρ; it consists of the infinite sequence $\sigma^i = \sigma_i \sigma_{i+1} \sigma_{i+2} \ldots$ of states and the infinite sequence $T^i = T_i T_{i+1} T_{i+2} \ldots$ of times. Note that ρ^i is, for all $i \geq 0$, again a timed state sequence; that is, the set of timed state sequences is closed under suffixes.

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 $\rho = (\sigma, T)$ is an *initialized computation* of the timed transition system $S = \langle V, \Sigma, \Theta, \mathcal{T}, l, u \rangle$ iff the state sequence σ is an initialized 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 $T_j < T_i + l_\tau$,

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

In other words, once enabled, τ is delayed for at least l_τ clock ticks; 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 $T_j \leq T_i + u_\tau$ such that

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

In other words, once enabled, τ is delayed for at most u_τ clock ticks; 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 an initialized computation may occur at any (integer) time.

We point out that the lower-bound condition does not rule out that a transition τ is taken several times within a time interval of length l_τ; this situation may occur if τ stays enabled after being taken and, if undesirable, can be prevented by refining the state space. Also note that at both stuttering steps and clock ticks of an initialized computation, the idle transition τ_I is taken. Thus an execution sequence of a timed transition system consists of an initial state and a fair merge of (1) a (finite or infinite) sequence of transitions and (2) an infinite sequence of clock ticks.

A timed state sequence ρ is a *computation* of the timed transition system S iff ρ is a suffix of an initialized computation of S. 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 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 [MP92]: τ cannot be continuously enabled without being taken. By S_j^- we denote the weakly-fair transition system that is obtained from the transition system S^- underlying S by adding weak-fairness requirements for all transitions with infinite maximal delays.

Closure properties

Both the computations and the initialized computations of any timed transition system are closed under *stuttering* and under *shifting the origin of time*:

- The addition of finitely many stuttering steps to a timed state sequence does not alter the property of being an (initialized) computation of S.
- The addition of an integer constant to all times of a timed state sequence does not alter the property of being an (initialized) computation of S. In other words, timed transition systems cannot refer to absolute time. Thus we will often assume, without loss of generality, that the time of the first state change of a computation is 0.

2.3 Operationality of timed transition systems

In this subsection, we show that the timed transition system model is operational (machine-closed [AL88]), in the sense that execution sequences can be generated incrementally by a machine that adds one state at a time. Untimed transition systems specify *safety* properties and are, therefore, trivially "executable" in a stepwise fashion:

> *Start with an initial state, and at each point during the stepwise construction of an execution sequence choose, nondeterministically, any one of the enabled transitions.*

To generate an execution sequence of a timed transition system, at each step we have to either choose a transition (without incrementing time) or increment time (while taking the idle transition). In addition, a timed execution sequence has to meet the *liveness* requirements imposed by maximal transition delays of ∞ and by the progress condition on time. While maximal delays of ∞ are standard fairness conditions that can always be met ([AFK88]), we need to show that the stepwise execution of a timed transition system cannot lead to a situation in which time cannot advance ever.

Let $S = \langle V, \Sigma, \Theta, \mathcal{T}, l, u\rangle$ be a timed transition system. A finite prefix $\rho = (\sigma, T)$ of a timed state sequence is a *partial computation* of S iff it satisfies the initiality, consecution, and lower-bound requirements of initialized computations, and

Finite upper bound Let n be the number of state-time pairs in ρ. For every transition $\tau \in \mathcal{T}$ and position $0 \leq i < n$, either $T_{n-1} \leq T_i + u_\tau$ or there is some position $i < j < n$ with $T_j \leq T_i + u_\tau$ such that either τ is not enabled on σ_j, or τ is taken at position j.

Proposition 1. *Each partial computation of a timed transition system S is a prefix of an initialized computation of S.*

Proof. We use the following strategy to extend a partial computation of the timed transition system S. Let $\mathcal{T}$ be the transition set of S and let $N \geq (|\mathcal{T}| + 1)^2$ be any sufficiently large integer constant. We alternate two phases:

Phase 1 For at least N positions, increment time whenever possible; that is, if all enabled transitions can be delayed for another time unit. On the other hand, if there are any transitions whose (finite) upper-bound requirements prevent the progress of time, take any one of these transitions.

Phase 2 Once every N positions, take every transition with a maximal delay of ∞ that has been enabled long enough. This phase satisfies all infinite upper-bound requirements.

The only way for this strategy not to yield an initialized computation of S is because, from some point on, time does not advance. In this case, there has to be a phase-1 sequence ρ of length $N+1$ of the form

$$(\sigma_i, T_i) \xrightarrow{\tau_i} (\sigma_{i+1}, T_i) \xrightarrow{\tau_{i+1}} \cdots \xrightarrow{\tau_{i+N-1}} (\sigma_{i+N}, T_i).$$

The sequence ρ must contain a subsequence ρ' of length $|\mathcal{T}|+1$ such that every transition in ρ' has been taken at least once before in ρ. It is not hard to see that, contrary to our assumption, the state component of ρ' violates the operationality restriction on maximal delays. ■

Proposition 1 guarantees that a machine that incrementally generates partial computations cannot arrive in a situation in which the progress condition on time cannot be satisfied. It follows that timed transition systems are machine-closed with respect to the progress of time (nonZeno [Hen92, LA92]); they may be "executed" using the following procedure:

> *Start with an initial state, and at each point during the stepwise construction of an execution sequence either choose a transition that has been enabled long enough, or increment time provided that all enabled transitions can be delayed for another time unit.*

3 Concrete Model: 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 timeouts 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

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

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 boolean expression 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 $L_i = \{\ell_0^i, \ldots \ell_{n_i}^i\}$ are called *locations.* The *entry location* — usually ℓ_0^i — is indicated as follows:

$$\longrightarrow \ell_0^i$$

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 over the variables in $U_s \cup U_i$, a *minimal delay* $l \in \mathbb{N}$ and a *maximal delay* $u \in \mathbb{N} \cup \{\infty\}$ such that $u \geq l$:

$$\ell_j^i \xrightarrow[{[l,u]}]{c \;\rightarrow\; \bar{x} := \bar{e}} \ell_k^i$$

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 $c \rightarrow nil := nil$ 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 ensures the process P_i *may* proceed to the location ℓ_k^i only if the control of P_i has resided at the location ℓ_j^i for at least l time units during which the guard c has been continuously true; such a move from ℓ_j^i to ℓ_k^i occurs "instantaneously" and assigns the current values of $\bar{e}$ to the variables $\bar{x}$. The maximal delay u ensures that the control of some process in P *must* proceed before the control of P_i would have resided at the location ℓ_j^i for more than u time units with the guard c continuously true; such a move can be accomplished either by P_i itself or by a competing process that falsifies the value of c. It follows that the control of a process P_i may remain at a location ℓ_j^i forever only in one of two situations: if ℓ_j^i 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_j^i is false infinitely often, we say that P_i has *deadlocked.* The second condition is necessary (although not sufficient) for P_i to deadlock, 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 informal operational view of timed transition diagrams can be captured formally 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* for 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 of 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_0^i 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, $\sigma' \in \tau_0^i(\sigma)$ iff

$\sigma(\pi_i) = \perp$ and $\sigma'(\pi_i) = \ell^i_0$,
$\sigma'(y) = \sigma(y)$ for all $y \in V - \{\pi_i\}$.

If E connects the source location ℓ^i_j to the target location ℓ^i_k and is labeled by the instruction $c \to \bar{x} := \bar{e}$, then $\sigma' \in \tau_E(\sigma)$ 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 $y \in V - \{\pi_i, \bar{x}\}$.

If τ_E is uniquely determined by its source and target locations, we write $\tau^i_{j\to k}$.

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 concrete real-time system P as a set of timed state sequences. The maximal delays of ∞ for all entry transitions guarantee that each component process of P is started at some time. The condition on timed transition diagrams that every cycle contains at least two edges ensures that, except for the idle transition, each transition of the timed transition system S_P is disabled upon completion; the condition that every cycle contains at least one positive (nonzero) maximal delay ensures that S_P is operational.

For instance, the initialized computations of the trivial system P that consists of a single process with the timed transition diagram

$$P: \quad \longrightarrow \ell_0 \xrightarrow[{[0,1]}]{} \ell_1$$

are exactly the timed state sequences that result from closing the two sequences

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

under stuttering and shifting the origin of time (there are no data variables).

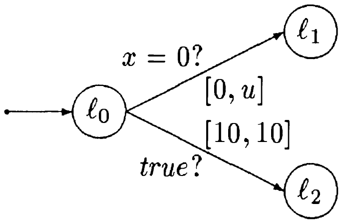
We remark that our semantics of shared-variables multiprocessing systems is conservative over the untimed case. Suppose that the system P contains no delay labels (recall that, in this case, all minimal delays are 0 and all maximal delays are ∞). Then the state components of the initialized computations 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 ([MP92]): 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: Timeout and timely response

To demonstrate the scope of the timed transition diagram language, we model two common real-time phenomena as shared-variables multiprocessing systems. In the first example (*timeout*), a process checks if an external event (i.e., an event caused by another, parallel, process) 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.

Timeout

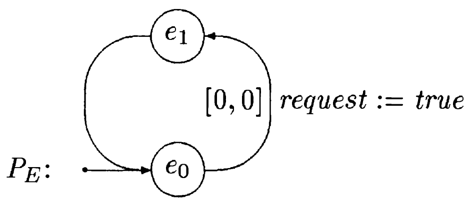
To see how a timeout situation can be programmed, consider the process P with the following timed transition diagram:



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 might 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, this 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 P_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 there are no restrictions on the frequency of requests.

We want to design a traffic light controller P_C 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, in comparison, the time needed for local operations within P_C is negligible. Now let us specify the desired process P_C. The controller P_C should behave in such a way that the combined system

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

satisfies the following three correctness conditions:

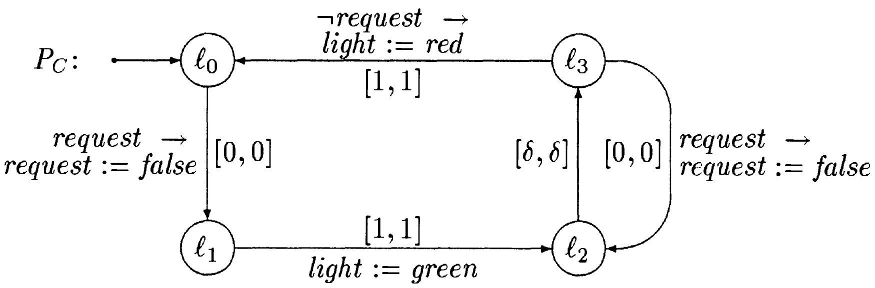
(A) Whenever *request* is *true*, then *light* is *green* within 5 time units for at least 5 time units.

(B) Whenever *request* is *true*, then *request* is *false* within 25 time units.

(C) 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), ensures that the request button is reset and the third condition, (C), prevents the light from being always green.

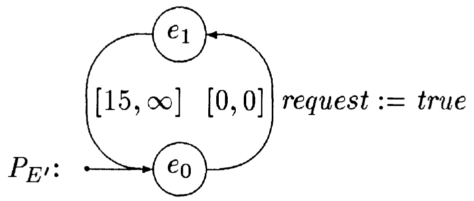
It is not hard to convince ourselves that, once it is started, the following process P_C satisfies the specification:



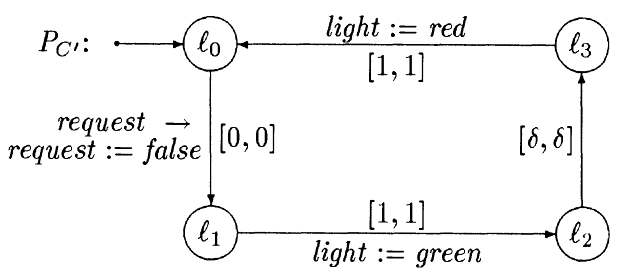
for any delay $4 \leq \delta \leq 23$. This implementation of the traffic light controller turns the light green as soon as possible after a request is received and then waits for δ time units. If the request button has been pushed in the meantime, the light stays green for another δ time units; otherwise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

Multiple traffic lights

We now generalize the *traffic light* example and design a system that reacts to two different types of external events. For this purpose, it will be convenient to accept some additional assumptions about the frequencies of the external events. In particular, we modify our example by adding the assumption that any two requests are at least 15 time units apart; that is,



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

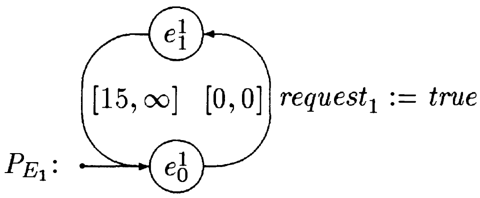


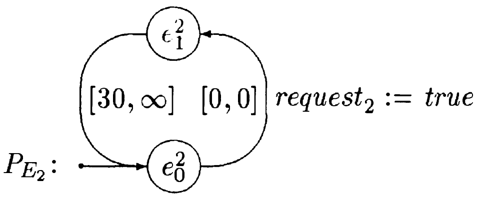
for any delay $4 \leq \delta \leq 17$. The combined system

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

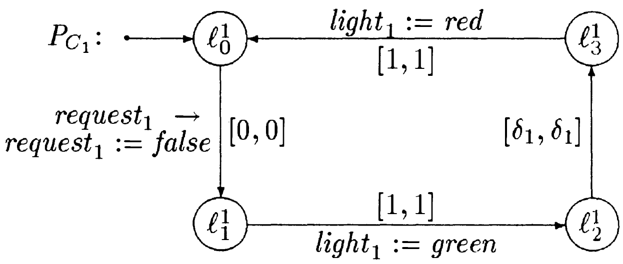
still satisfies the three correctness requirements (A), (B), and (C).

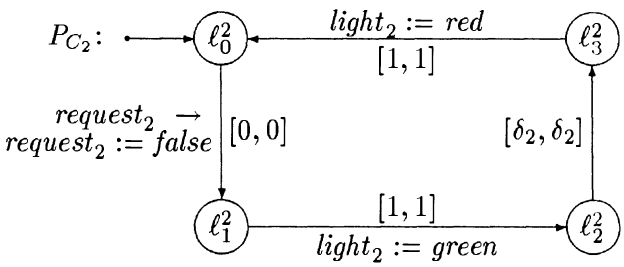
Now consider a more complex traffic light configuration, with two lights and two request buttons on two independent streets. The second light is designed for the special convenience of pedestrians in a hurry: it is required to turn green within 3 time units of a request but, on the other hand, has to 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 — only 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 P_{C_1} and P_{C_2} are executed in a truly concurrent fashion, then the correctness of the entire system

$$P_{\|} : \quad \{request_1 = request_2 = false,\ light_1 = light_2 = red\}\ [P_{E_1} \| P_{E_2} \| P_{C_1} \| P_{C_2}]$$

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 $request_1$ is *true*, then $request_1$ is *false* within 25 time units.

(B_2) Whenever $request_2$ is *true*, then $request_2$ is *false* within 25 time units.

(C_1) Whenever $request_1$ has been *false* for 25 time units, then $light_1$ is *red*.

(C_2) Whenever $request_2$ has been *false* for 25 time units, then $light_2$ is *red*.

A more interesting case arises if only a single processor is available to control both lights and the two processes P_{C_1} and P_{C_2} have to share it. Using the *interleaving* (shuffle) *operator* $|||$ of [Hoa85] (to be defined), we will denote the resulting system by the expression

$$P_{|||}: \quad \{request_1 = request_2 = false,\ light_1 = light_2 = red\}\ [P_{E_1} \| P_{E_2} \| (P_{C_1} ||| P_{C_2})].$$

Note that the behavior of the environment $P_{E_1} \| P_{E_2}$ is still truly concurrent to the behavior of the traffic light controller $P_{C_1} ||| P_{C_2}$, which executes both processes P_{C_1} and P_{C_2} on a single processor in an interleaved fashion.

Let us assume that $\delta_1 = 10$ and $\delta_2 = 2$, in which case $P_{\|}$ is correct. However, if we have no knowledge about the strategy by which the processes P_{C_1} and P_{C_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 $P_{|||}$ does not satisfy the specification consisting of the requirements (A_i), (B_i), and (C_i) for $i = 1, 2$. For suppose that the process P_{C_1} is always given priority over the process P_{C_2}, and the traffic light controller receives a request for the second light only one 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 P_{C_2} that serves the more urgent yet less frequent requests is always given priority over the process P_{C_1}, then $P_{|||}$ 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 formalize the interleaved execution of processes and discuss the modeling of priorities and interrupts in detail, we first introduce message-passing operations for the parallel execution of processes.

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, $U_s = \emptyset$) and a set of message channels. We use 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, and the current value of e is assigned to x. 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 either both processes *must* proceed with the communication, or a third process must falsify one of c and c'.

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 $c \to \alpha!e$ 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:

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

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

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 $P_1||_sP_2$ for the system P whose component processes P_1 and P_2 start synchronously; that is, the notation $P_1||_sP_2$ 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 $P_1||_sP_2$ 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

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

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 Concrete Model: Multiprogramming Systems

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

$$P_1\colon \quad \longrightarrow \ell_0^1 \xrightarrow[{[1,1]}]{} \ell_1^1$$

$$P_2\colon \quad \longrightarrow \ell_0^2 \xrightarrow[{[1,1]}]{} \ell_1^2$$

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 will be 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 initialized 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). Note that the system $P_1 \| P_2$ has more initialized 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 initialized computations will be defined to be essentially the closure of the two 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, when measuring time, 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 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

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

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. $V = U \cup \{\mu, \pi_1, \dots \pi_m\}$. There are two kinds of control variables: the *processor control variable* μ ranges over the set $\{1, \dots m, \perp\}$; 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 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 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 $P_1, \dots P_m$. If E connects the source location ℓ^i_j to the target location ℓ^i_k and is labeled by the instruction $c \to \bar{x} := \bar{e}$, 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 desire that a multiprogramming system satisfies a real-time requirement under all (fair) scheduling strategies, because it is the scheduling algorithm which usually determines if a system meets its timing constraints. Indeed, fair scheduling strategies admit *thrashing*: by switching control too often between processes, only scheduling transitions might be performed; thus the system might not make make any real progress and will certainly not meet nontrivial 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 of 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 $s \subseteq \{1, \dots m\}$ 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_j with $j \in s$:

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

We write *resume*(j) for *resume*$(\{j\})$ and *suspend* for *resume*$(\{1 \leq j \leq m \mid j \neq i\})$; 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 $P_1, \dots P_m$. If E connects the source location ℓ^i_j to the target location ℓ^i_k and is labeled by the instruction $c \rightarrow$ *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 $y \in V - \{\mu, \pi_i\}$.

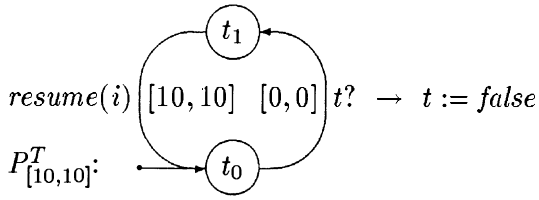
Furthermore, for every scheduling edge E labeled by the minimal delay l and the maximal delay u, let $l_{\tau_E} = l$ and $u_{\tau_E} = u$.

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) *process* $P^T_{[10,10]}$ as a parallel process:



We make sure that the timer process $P^T_{[10,10]}$ is started (i.e., waiting for activation) when the process P_i becomes active. Then the timer is activated by the sequence

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

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

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

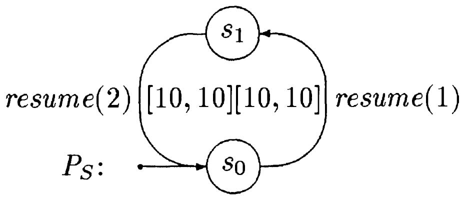
The activation of the timer process $P^T_{[l,u]}$ is abbreviated by the *delay* instruction

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

which delays the currently active process P_i for at least l and at most u time units and activates a process from the set s. 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 process in a timed transition diagram (even though the result is well-defined, it should be avoided that a delayed process is reactivated before the corresponding timer expires).

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 $(P_1|||P_2)||_s P_S$, the *scheduler process*



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 scheduler processes as well.

4.3 Processor allocation

Both the multiprogramming system with a timer process and the multiprogramming system with a central scheduler process are, in fact, combinations of multiprocessing and multiprogramming systems in which several tasks compete for some of the processors. In 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

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

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 simply 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 transitions that have the highest priority among all enabled transitions belong to suspended processes, then the currently active process is interrupted and the execution of one of the suspended processes 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 $p \in \mathbb{N}$ as follows:

$$\ell_j^i \xrightarrow[{[l, u]}]{p:\ c\ \to\ x := e} \ell_k^i$$

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

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

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 $P_1, \ldots P_m$. If E connects the source location ℓ_j^i to the target location ℓ_k^i and is labeled by the instruction p: $c \to \bar{x} := \bar{e}$, then $\sigma \to_E \sigma'$ 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 $y \in V - \{\mu, \pi_i, \bar{x}\}$.

  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.

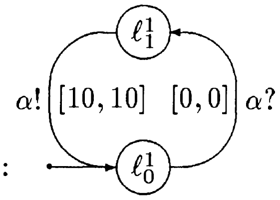
Note that if all transitions have equal priority, then the scheduling strategy is greedy (that is, $\tau_G = \tau_P$). Thus priorities generalize our previous discussion conservatively: all systems can be viewed as priority systems whose instructions have the same default priority, unless they are annotated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

Dynamic priorities

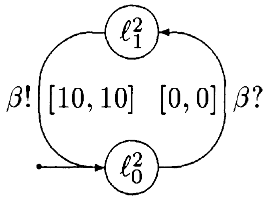
Priorities can be combined with explicit scheduling operations in the obvious way. It is, however, often more convenient to model dynamic scheduling strategies, which change over time, by *dynamic priorities*, which can be modified by any process during execution. Dynamic priorities offer exciting possibilities, 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 as priorities data variables that range over the nonnegative integers. 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.

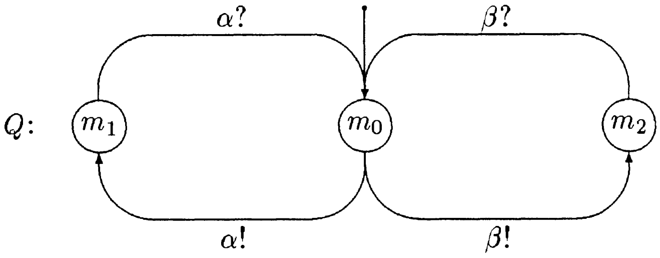
We have not yet pointed out that our interpretation of message passing is not entirely conservative over the untimed case: there the set of legal execution sequences usually is restricted by strong-fairness assumptions for communication transitions ([MP92]). This is convenient for the study of time-independent 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:

P_1:



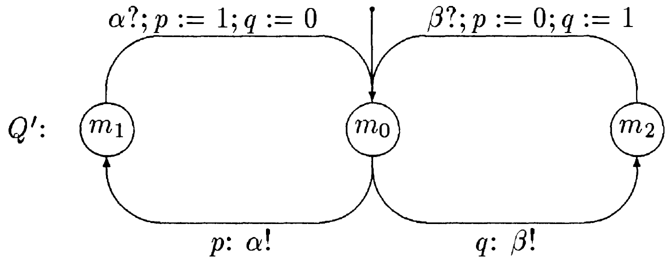
P_2:





(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 concatenate instructions; the default value of priorities is assumed to be 0.) The arbiter Q' modifies the priorities p and q of its nondeterministic alternatives to ensure that the system

$$\{p = q = 0\}[P_1 || P_2 || Q']$$

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 itself 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.

Part II
Verifying Real-time Systems

We define a formal language that is interpreted over timed state sequences. This language is used to specify timed transition systems: a timed transition system S meets the specification ϕ iff all computations of S satisfy ϕ. We present two proof methodologies — bounded-operator reasoning and explicit-clock reasoning — for verifying that a timed transition system meets its specification. Relative-completeness results are given for both proof techniques.

5 Specification Language

As a specification language, we use an extension of linear temporal logic with time-bounded temporal operators. We distinguish between *state* formulas, which assert properties of individual states of a computation, and *temporal* formulas, which assert properties of entire computations.

5.1 State formulas

Let $S = \langle V, \Sigma, \Theta, \mathcal{T}, l, u\rangle$ be a timed transition system. Typically S is associated with a concrete real-time system that belongs to one of the classes we discussed in Part I. Throughout this part, we use the following additional assumptions about the set V of variables:

- We assume that, in addition to data and control variables, V contains sufficiently many *auxiliary variables* that range over the integers and are not changed by any of the transitions of S. We will on occasion need a "new, rigid" variable, and for this purpose we employ one of the auxiliary variables that have not been used previously.

- We assume that, for every variable $x \in V$, there is a corresponding unique primed variable $x' \notin V$ that ranges over the same domain as x.

We are given an assertion language — a first-order language with equality that contains interpreted function and predicate symbols to express operations and relations on the domains of the variables in V. A *state formula* (assertion) is a first-order formula p of the assertion language such that only variables from V occur freely in p. Thus, every state in Σ provides an interpretation for the state formulas. If the state formula p is true in state σ, we say that σ is a p-state.

We use the following abbreviations for state formulas:

- For any transition $\tau \in \mathcal{T}$, the *enabling* condition $enabled(\tau)$ asserts that τ is enabled. In particular, $enabled(\tau_I)$ is equivalent to *true* for the idle transition τ_I.

- For any transition $\tau \in \mathcal{T}$ and state formulas p and q, the *verification* condition $\{p\}\,\tau\,\{q\}$ asserts that if p is true of a state $\sigma \in \Sigma$, then q is true of all τ-successors of σ. In particular,

$\{p\}\,\tau_I\,\{q\}$ stands for the universal closure of the formula $p \rightarrow q$. For any set $\mathcal{T}' \subseteq \mathcal{T}$ of transitions, we write $\{p\}\,\mathcal{T}'\,\{q\}$ for the conjunction

$$\bigwedge_{\tau \in \mathcal{T}'} \{p\}\,\tau\,\{q\}$$

of all individual verification conditions.

- For any transition $\tau \in \mathcal{T}$ and state formulas p and q, the *inverse verification* condition $\{p\}\,\tau^-\,\{q\}$ asserts that if p is true of a state $\sigma \in \Sigma$, then q is true of all τ-predecessors of σ. Observe that all inverse verification conditions are definable by ordinary verification conditions:

  $$\{p\}\,\tau^-\,\{q\} \quad \text{is equivalent to} \quad \{\neg q\}\,\tau\,\{\neg p\}.$$

  In particular, $\{p\}\,\tau_I^-\,\{q\}$ is equivalent to $\{p\}\,\tau_I\,\{q\}$ for the idle transition τ_I. For any set $\mathcal{T}' \subseteq \mathcal{T}$ of transitions, we write $\{p\}\,\mathcal{T}'^-\,\{q\}$ for the conjunction of the inverse verification conditions for all transitions in $\mathcal{T}'$.

Note that while the truth value of an enabling condition depends on the state in which it is interpreted, the verification conditions are state-independent and, thus, equivalent to closed formulas.

In the case that the timed transition system S is associated with a shared-variables multiprocessing system P, it is not hard to see that the enabling and verification conditions of all transitions can indeed be expressed by state formulas. Suppose that P consists of the m processes P_i, for $1 \leq i \leq m$, and the data precondition θ, which is a state formula:

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

Let us assume that each process P_i, for $1 \leq i \leq m$, is given by a timed transition diagram with the locations $\{\ell_0^i, \ldots \ell_{n_i}^i\}$ and the entry location ℓ_0^i. We write $at\_\ell_\perp^i$ for $\pi_i = \perp$, and $at\_\ell_j^i$ for $\pi_i = \ell_j^i$; that is, the control of the process P_i is at the location ℓ_j^i. We abbreviate any disjunction $at\_\ell_j^i \vee at\_\ell_k^i$ further, to $at\_\ell_{j,k}^i$.

1. For each entry transition $\tau_0^i \in \mathcal{T}$ of S_P, the enabling condition $enabled(\tau_0^i)$ is equivalent to the state formula

   $$at\_\ell_\perp^i,$$

   and the verification condition $\{p\}\,\tau_0^i\,\{q\}$ is equivalent to the universal closure of the formula

   $$(p \wedge enabled(\tau_0^i) \wedge (at\_\ell_0^i)' \wedge \bigwedge_{y \in V - \{\pi_i\}} (y' = y)) \rightarrow q',$$

   where the formula q' is obtained from q by replacing every variable with its primed version; for example, $(at\_\ell_0^i)'$ stands for $\pi_i' = \ell_0^i$. The inverse verification condition $\{p\}\,(\tau_0^i)^-\,\{q\}$ is equivalent to the universal closure of the formula

   $$(p' \wedge enabled(\tau_0^i) \wedge (at\_\ell_0^i)' \wedge \bigwedge_{y \in V - \{\pi_i\}} (y' = y)) \rightarrow q.$$

2. All other nonidle transitions of S_P correspond to edges in the timed transition diagrams for the processes P_i. Let $\tau_{j \rightarrow k}^i \in \mathcal{T}$ be such a transition and assume that the corresponding edge

that connects the location ℓ^i_j to the location ℓ^i_k is labeled by the instruction $c \rightarrow \bar{x} := \bar{e}$. Then, the enabling condition $enabled(\tau^i_{j\rightarrow k})$ is equivalent to the state formula

$$at\_\ell^i_j \wedge c,$$

and the verification condition $\{p\}\ \tau^i_{j\rightarrow k}\ \{q\}$ is equivalent to the universal closure of the formula

$$(p \wedge enabled(\tau^i_{j\rightarrow k}) \wedge (at\_\ell^i_k)' \wedge (\bar{x}' = \bar{e}) \wedge \bigwedge_{y \in V - \{\pi_i, x\}} (y' = y)) \rightarrow q'.$$

The inverse verification condition $\{p\}\ (\tau^i_{j\rightarrow k})^-\ \{q\}$ is equivalent to the universal closure of the formula

$$(p' \wedge enabled(\tau^i_{j\rightarrow k}) \wedge (at\_\ell^i_k)' \wedge (\bar{x}' = \bar{e}) \wedge \bigwedge_{y \in V - \{\pi_i, x\}} (y' = y)) \rightarrow q.$$

It is also straightforward to express the enabling and verification conditions as state formulas if the timed transition system S is associated with any of the other concrete real-time systems that we discussed in Part I, such as message-passing, multiprogramming, dynamic, and priority systems.

Synchronous multiprocessing systems

Our examples will be drawn from timed transition systems S that are associated with multiprocessing systems P of the form

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

all of whose component processes start synchronously (i.e., at the exact same time). The corresponding timed transition system has a single entry transition that sets all control variables, simultaneously, to the entry locations of the individual processes. For multiprocessing systems P, it is convenient to define the following two additional abbreviations for state formulas:

- The *ready* condition *ready* holds precisely in the initial states Θ of S_P; it indicates that none of the processes of P has started yet. Consequently, *ready* stands for the state formula

$$\theta \wedge (\bigwedge_{1 \leq i \leq m} at\_\ell^i_\perp).$$

- The *synchronous starting* condition *start* indicates that all processes of P have entered their entry locations, but none has proceeded any further; that is, *start* abbreviates the state formula

$$\theta \wedge (\bigwedge_{1 \leq i \leq m} at\_\ell^i_0).$$

Note that if P is synchronous, then the two verification conditions

$$\{ready\}\ \mathcal{T} - \tau_I\ \{start\},$$

$$\{start\}\ (\mathcal{T} - \tau_I)^-\ \{ready\}$$

are valid (by $\mathcal{T} - \tau$ we denote the set difference $\mathcal{T} - \{\tau\}$).

5.2 Temporal formulas

Temporal formulas are constructed from state formulas by boolean connectives and time-bounded temporal operators. They are interpreted over timed state sequences. In this paper, we are interested in proving two important classes of real-time properties — bounded-invariance properties and bounded-response properties. Thus we restrict ourselves to the following temporal formulas:

- Every state formula p is a temporal formula; it is true over the timed state sequence $\rho = (\sigma, T)$ iff the initial state σ_0 is a p-state.

- Every boolean combination of temporal formulas is a temporal formula, whose truth over a timed state sequence is determined from the truth of its components in the standard way.

- *Bounded-unless formulas*: If p is a state formula, ϕ a temporal formula, and $l \in \mathsf{N}$, then $p\,\mathsf{U}_{\geq l}\,\phi$ is a temporal formula; it is true over the timed state sequence $\rho = (\sigma, T)$ iff either all σ_i, for $i \geq 0$, are p-states, or there is some position $i > 0$ such that $T_i \geq T_0 + l$, and ϕ is true over the i-th suffix ρ^i of ρ, and all σ_j, for $0 \leq j < i$, are p-states. We use the abbreviations $p\,\mathsf{U}\,\phi$ and $\Box_{<l}\,p$ for the formulas $p\,\mathsf{U}_{\geq 0}\,\phi$ and $p\,\mathsf{U}_{\geq l}\,\mathit{true}$, respectively.

- *Bounded-eventually formulas*: If ϕ is a temporal formula and $u \in \mathsf{N}$, then $\Diamond_{\leq u}\,\phi$ is a temporal formula; it is true over the timed state sequence $\rho = (\sigma, T)$ iff there is some position $i \geq 0$ such that $T_i \leq T_0 + u$ and ϕ is true over the i-th suffix ρ^i of ρ.

Temporal-logic aficionados will readily recognize the operators $\Box_{<l}$, $\Diamond_{\leq u}$, and $\mathsf{U}_{\geq l}$ as time-bounded versions of the standard (untimed) *always*, *eventually*, and strict *unless* operators. In particular, the bounded-unless formula $p\,\mathsf{U}_{\geq 0}\,q$ is true over a timed state sequence $\rho = (\sigma, T)$ iff the untimed (strict) *unless formula* $p\,\mathsf{U}\,q$ is true over the state component σ of ρ:

> either all σ_i, for $i \geq 0$, are p-states,
> or there is some position $i > 0$ such that q is true over the i-th suffix σ^i of σ and all σ_j,
> for $0 \leq j < i$, are p-states.

For a general addition of time-bounded operators to linear temporal logic, see [AH90]. From now on, we use the convention that the letters p, q, r as well as φ (and primed versions) denote state formulas, while the letters ϕ, ψ, and χ stand for temporal formulas.

S-validity and S-soundness

Let S be a timed transition system. A temporal formula ϕ is *S-valid* iff it is true over all computations of S. While (general) validity — truth over all timed state sequences — implies S-validity for the given system S, the converse does not necessarily hold. In fact, even a *state* formula p that is S-valid may not be true in some states of S that do not occur along any run of S and, hence, p may not be generally valid. To show that the given system S meets the specification ϕ, it suffices to show that ϕ is S-valid.

A proof rule is *S-sound* iff the S-validity of all premises implies the S-validity of the conclusion. Note that a (generally) sound proof rule — whose conclusion is valid if all premises are valid — might not be S-sound for the given system S, and vice versa. For verifying properties of a timed transition system S, we are interested in S-sound rules.

Bounded invariance and bounded response

Two important classes of timing requirements for real-time systems can be defined by temporal formulas:

- A *bounded-invariance* property asserts that a condition holds continuously for a certain amount of time; it is often used to specify that something does not happen for a certain amount of time. Formally, we express bounded-invariance properties by temporal formulas of the form

  $$p \rightarrow \Box_{<l}\, q,$$

  for state formulas p and q and a nonnegative integer $l \in \mathbb{N}$. The formula $p \rightarrow \Box_{<l}\, q$ is S-valid, for a timed transition system S, iff for all (initialized) computations $\rho = (\sigma, T)$ of S and all $i \geq 0$ and $j > i$,

  if σ_i is a p-state and $T_j < T_i + l$,
  then σ_i and σ_j are q-states;

  that is, every p-state is a q-state and no p-state is followed by a $\neg q$-state within time less than l. A typical application of bounded invariance states a lower bound l on the termination of a multiprocessing system P with the termination condition r: the temporal formula

  $$\mathit{ready} \rightarrow \Box_{<l}\, \neg r$$

  asserts that, if not started before time t, then P will not reach a final state before time $t + l$.

- A *bounded-response* property asserts that something happens within a certain amount of time. Formally, we express bounded-response properties by temporal formulas of the form

  $$p \rightarrow \Diamond_{\leq u}\, q,$$

  for state formulas p and q and a nonnegative integer $u \in \mathbb{N}$. The formula $p \rightarrow \Diamond_{\leq u}\, q$ is S-valid, for a timed transition system S, iff for all (initialized) computations $\rho = (\sigma, T)$ of S and all $i \geq 0$,

  if σ_i is a p-state,
  then there is some q-state σ_j, with $j \geq i$, such that $T_j \leq T_i + u$;

  that is, every p-state is followed by a q-state within time u. A typical application of bounded response states an upper bound u on the termination of a multiprocessing system P with the termination condition r: the temporal formula

  $$\mathit{start} \rightarrow \Diamond_{\leq u}\, r$$

  asserts that if all component processes of P are started synchronously at time t, then P is guaranteed to reach a final state no later than at time $t + u$.

As the computations of timed transition systems are closed under shifting the origin of time, we shall, without loss of generality, henceforth assume that the starting time t of a synchronous multiprocessing system is 0.

Monotonicity rules

We now introduce two important proof rules that are S-sound for every timed transition system S. The monotonicity rule U-MON allows us to weaken any of the three arguments of the *bounded-unless* operator:

$$\text{U-MON}\quad \frac{p \rightarrow p' \qquad \phi \rightarrow \phi' \qquad l' \leq l}{(p\,\mathsf{U}_{\geq l}\,\phi) \rightarrow (p'\,\mathsf{U}_{\geq l'}\,\phi')}$$

A second monotonicity rule, $\diamond$-MON, weakens either argument of the *bounded-eventually* operator:

$$\diamond\text{-MON}\quad \frac{\phi \rightarrow \phi' \qquad u' \geq u}{(\diamond_{\leq u}\,\phi) \rightarrow (\diamond_{\leq u'}\,\phi')}$$

It is not hard to see that both monotonicity rules are S-sound for every timed transition system S. Since propositional reasoning, too, is S-sound for every system S, we will refer to applications of the two weakening rules and propositional reasoning in derivations through the simple annotation "by monotonicity." For example, from the bounded-unless formula

$$p \rightarrow q\,\mathsf{U}_{\geq l}\,r,$$

we can establish, by monotonicity, both the bounded-invariance formula

$$p \rightarrow \Box_{<l}\,q$$

and the (unbounded) unless formula

$$p \rightarrow q\,\mathsf{U}\,r.$$

6 Bounded-operator Reasoning

Let $S = \langle V, \Sigma, \Theta, \mathcal{T}, l, u\rangle$ be a timed transition system. We present a deductive system for establishing the S-validity of bounded-invariance and bounded-response properties. The proof rules fall into four categories:

1. The *single-step* rules derive real-time properties that follow from the lower-bound or upper-bound requirement for a single transition.
2. The *transitivity* rules combine two real-time properties of the same type — that is, either two bounded-invariance properties or two bounded-response properties — into a composite timing property.
3. The *induction* rules combine arbitrarily many real-time properties of the same type into a composite timing property.
4. The *crossover* rules combine two real-time properties of opposite types into a composite timing property.

We begin by presenting this "bounded-operator" methodology for the restricted case of deterministic systems that can be verified without crossover reasoning: the timed transition system S is *deterministic* iff it is associated with a multiprocessing system P and any two guards on outgoing edges of the same vertex in the timed transition diagram representation of P are disjoint. It follows that for deterministic systems, on each state the idle transition and at most one nonidle transition per process are enabled. Nondeterministic systems require more complex (conditional) single-step reasoning, which is treated in the subsequent subsection. The need for crossover reasoning arises in connection with the completeness argument of Section 8 and will be discussed there.

6.1 Deterministic rules

In a first attempt at proving time-constrained response formulas, we decorate the proof rules for establishing unconstrained response formulas ([MP84]) with time bounds. Furthermore, for each of these *upper-bound* rules for deriving bounded-response properties, there is corresponding *lower-bound* rule for deriving bounded-invariance properties. In keeping with our standard order, we will introduce each lower-bound rule before the corresponding upper-bound rule.

Single-step rules

The *single-step lower-bound rule* uses the minimal delay $l_\tau \in \mathsf{N}$ of a transition $\tau \in \mathcal{T}$ to infer a bounded-unless formula:

$$\begin{array}{ll|}
\textsf{U-SS} & p \;\rightarrow\; \neg enabled(\tau) \\
 & p \;\rightarrow\; \varphi \\
 & \{\varphi\}\,\mathcal{T}-\tau\,\{\varphi\} \\
 & \varphi \;\rightarrow\; q \\
 & (\varphi \wedge enabled(\tau)) \;\rightarrow\; r \\
\hline
 & p \;\rightarrow\; q\,\mathsf{U}_{\geq l_\tau}\, r
\end{array}$$

The rule U-SS derives a *temporal* (bounded-unless) formula from premises all of which are *state* formulas, whose S-validity typically is shown by proving them generally valid. The state formula φ is called the *invariant* of the rule. Choosing r to be *true*, the rule infers a bounded-invariance property,

$$p \;\rightarrow\; \Box_{<l_\tau}\, q$$

(note that the last premise holds trivially in this case). To see why the rule U-SS is S-sound, observe that whenever the transition τ is not enabled, it cannot be taken for at least l_τ time units. During this time, the invariant φ is true.

The *single-step upper-bound rule* uses the maximal delay $u_\tau \in \mathsf{N}$ of a transition $\tau \in \mathcal{T}$ to infer a bounded-response formula:

$$\begin{array}{ll}
\Diamond\textsf{-SS} & p \;\rightarrow\; (\varphi \vee q) \\
 & \varphi \;\rightarrow\; enabled(\tau) \\
 & \{\varphi\}\,\mathcal{T}-\tau\,\{\varphi \vee q\} \\
 & \{\varphi\}\,\tau\,\{q\} \\
\hline
 & p \;\rightarrow\; \Diamond_{\leq u_\tau}\, q
\end{array}$$

This rule derives a temporal bounded-response formula from premises all of which are state formulas. The state formula φ is again called the *invariant* of the rule. To see why the rule $\Diamond$-SS is S-sound, recall that the transition τ has to be taken before it would be continuously enabled for more than u_τ time units.

To demonstrate a typical application of the single-step rules, we consider the single-process system P with the data precondition $x = 0$ and the following timed transition diagram:

$$\{x=0\} \longrightarrow (\ell_0) \xrightarrow[{[2,3]}]{x=0?} (\ell_1)$$

The process P confirms that $x = 0$ and proceeds to the location ℓ_1. Because of the delay interval $[2,3]$ of the transition $\tau_{0\to 1}$, the final location ℓ_1 cannot be reached before time 2 and must be reached by time 3 (recall that P is taken to start at time 0). Using single-step reasoning, we can carry out a formal proof of this analysis. The bounded-invariance property that P does not terminate before time 2,

$$\mathit{ready} \;\rightarrow\; \Box_{<2}\, \neg at\_\ell_1,$$

is established by an application of the single-step lower-bound rule U-SS with respect to the transition $\tau_{0\to 1}$ (let the invariant φ be $at\_\ell_{\perp,0}$). The bounded-response property that P terminates by time 3,

$$\mathit{start} \;\rightarrow\; \Diamond_{\leq 3}\, at\_\ell_1,$$

follows from the single-step upper-bound rule $\Diamond$-SS with respect to the transition $\tau_{0\to 1}$ (use the invariant $at\_\ell_0 \wedge x = 0$).

Transitivity rules

To combine a finite number of successive timing constraints into a more complicated real-time property, we introduce transitivity rules. The *transitive lower-bound rule* combines two bounded-unless formulas:

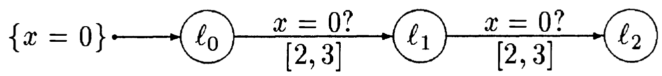
$$\boxed{\begin{array}{ll} \textbf{U-TRANS} & \phi \;\rightarrow\; p\,\mathsf{U}_{\geq l_1}\,\chi \\ & \underline{\chi \;\rightarrow\; q\,\mathsf{U}_{\geq l_2}\,\psi} \\ & \phi \;\rightarrow\; (p \vee q)\,\mathsf{U}_{\geq l_1+l_2}\,\psi \end{array}}$$

We refer to the formula χ as the *link* of the rule. The *transitive upper-bound rule* combines two bounded-response formulas:

$$\boxed{\begin{array}{ll} \Diamond\textbf{-TRANS} & \phi \;\rightarrow\; \Diamond_{\leq u_1}\,\chi \\ & \underline{\chi \;\rightarrow\; \Diamond_{\leq u_2}\,\psi} \\ & \phi \;\rightarrow\; \Diamond_{\leq u_1+u_2}\,\psi \end{array}}$$

The formula χ is again called the *link* of the rule. Both transitivity rules are easily seen to be S-sound for every timed transition system S.

We demonstrate the application of the transitivity rules by examining the single-process system P with the following timed transition diagram:



We want to show that P terminates not before time 4 and not after time 6. First, we prove the lower bound on the termination of P:

$$ready \;\rightarrow\; \Box_{<4}\, \neg at\_\ell_2.$$

By the transitive lower-bound rule **U-TRANS**, it suffices to show the two premises

$$ready \;\rightarrow\; (\neg at\_\ell_2)\, \mathsf{U}_{\geq 2}\, at\_\ell_0, \qquad (1)$$

$$at\_\ell_0 \;\rightarrow\; (\neg at\_\ell_2)\, \mathsf{U}_{\geq 2}\, true. \qquad (2)$$

Both premises can be established by single-step lower-bound reasoning. To show the premise (1), we apply the rule **U-SS** with respect to the transition $\tau_{0\rightarrow 1}$, using the invariant $at\_\ell_{\perp,0}$; the premise (2) follows from the rule **U-SS** with respect to the transition $\tau_{1\rightarrow 2}$ and the invariant $at\_\ell_{0,1}$.

The upper bound on the termination of P,

$$start \;\rightarrow\; \Diamond_{\leq 6}\, at\_\ell_2,$$

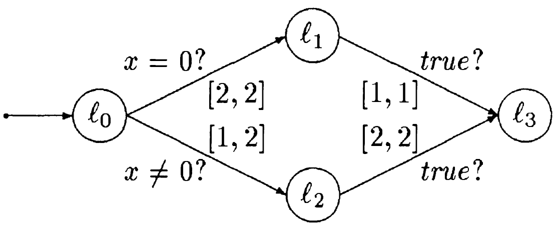
is concluded by the transitive upper-bound rule **◇-TRANS**. It suffices to show the premises

$$start \;\rightarrow\; \Diamond_{\leq 3}\, (at\_\ell_1 \wedge x = 0),$$

$$(at\_\ell_1 \wedge x = 0) \;\rightarrow\; \Diamond_{\leq 3}\, at\_\ell_2,$$

both of which can be established by single-step upper-bound reasoning (we use the two invariants $at\_\ell_0 \wedge x = 0$ and $at\_\ell_1 \wedge x = 0$, respectively). Note that for lower-bound reasoning the link at_ℓ_0 identifies the last state *before* the transition $\tau_{0\rightarrow 1}$ is taken, while for upper-bound reasoning the link $at\_\ell_1 \wedge x = 0$ refers to the first state *after* $\tau_{0\rightarrow 1}$ is taken.

For an example with a (deterministic) branching structure, consider the process P' with the following timed transition diagram:



We show that P' terminates either at time 3 or at time 4. The proof requires a case analysis on the initial value of x, which determines which path of the transition diagram is taken. The lower bound

$$ready \;\rightarrow\; \Box_{<3}\, \neg at\_\ell_3$$

is implied by the two bounded-invariance formulas

$$(ready \wedge x = 0) \;\rightarrow\; \Box_{<3}\, \neg at\_\ell_3,$$

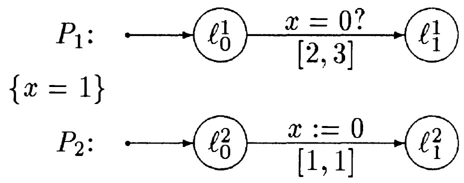
$$(ready \wedge x \neq 0) \rightarrow \Box_{<3} \neg at\_\ell_3,$$

both of which can be derived by transitive lower-bound reasoning (as links use the two state formulas $at\_\ell_{\perp,0} \wedge x = 0$ and $at\_\ell_{\perp,0} \wedge x \neq 0$, respectively). The upper bound

$$start \rightarrow \Diamond_{\leq 4}\, at\_\ell_3$$

follows by a similar case analysis and transitive upper-bound reasoning.

So far we have examined only single-process examples. In general, several processes that communicate through shared variables interfere with each other. Consider the synchronous two-process shared-variables multiprocessing system with the data precondition $x = 1$ and the following timed transition diagrams:



The first process, P_1, is identical to a previous example; with a minimal delay of 2 time units and a maximal delay of 3 time units, it confirms that $x = 0$ and proceeds to the location ℓ_1^1. However, this time the value of x is not 0 from the very beginning, but set to 0 by the second process, P_2, only at time 1. Thus, P_1 can reach its final location ℓ_1^1 no earlier than at time 3 and no later than at time 4.

For a formal proof we need the transitivity rules. The bounded-invariance property

$$ready \rightarrow \Box_{<3} \neg at\_\ell_1^1$$

is established by an application of the transitive lower-bound rule U-**TRANS**. It suffices to show the premises

$$ready \rightarrow (\neg at\_\ell_1^1)\, \mathsf{U}_{\geq 1}\, (at\_\ell_{\perp,0}^1 \wedge x = 1),$$
$$(at\_\ell_{\perp,0}^1 \wedge x = 1) \rightarrow (\neg at\_\ell_1^1)\, \mathsf{U}_{\geq 2}\, true,$$

both of which follow from single-step lower-bound reasoning. Similarly, the transitive upper-bound rule $\Diamond$-**TRANS** is used to show the bounded-response property

$$start \rightarrow \Diamond_{\leq 4}\, at\_\ell_1^1$$

from the link $at\_\ell_0^1 \wedge x = 0$.

Induction rules

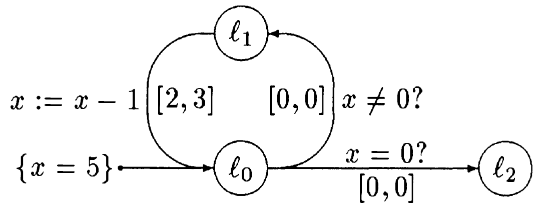
To prove lower and upper bounds on the execution time of program loops, we need to combine a state-dependent number of bounded-invariance or bounded-response properties. For this purpose it is economical to have induction schemes.

The *inductive lower-bound rule* U-**IND** generalizes the transitive lower-bound rule U-**TRANS**; it combines a potentially large number of similar bounded-unless formulas in a single proof step. Assume that the new, rigid variable $i \in V$ ranges over the integers $\mathbb{Z}$; for every positive integer $n > 0$:

$$\begin{array}{|ll|}\hline \textbf{U-IND} & \dfrac{(\varphi(i) \wedge i > 0) \;\rightarrow\; p\,\mathsf{U}_{\geq l}\,\varphi(i-1)}{\varphi(n) \;\rightarrow\; p\,\mathsf{U}_{\geq n\cdot l}\,\varphi(0)} \\ \hline \end{array}$$

By $\varphi(i-1)$ we denote the state formula that results from the inductive invariant $\varphi(i)$ by replacing all occurrences of the variable i with the expression $i-1$; the formulas $\varphi(n)$ and $\varphi(0)$ are obtained analogously. Note that every instance of the rule **U-IND**, for any constant $n > 0$, is derivable from the transitive lower-bound rule **U-TRANS**.

For a demonstration of inductive lower-bound reasoning, we consider the following single-process system P:



The process P decrements the value of x until it is 0, at which point P proceeds to the location ℓ_2. Since x starts out with the value 5 and each decrement operation takes at least 2 time units, while the tests are instantaneous, the final location ℓ_2 cannot be reached before time 10. This lower bound,

$$\mathit{ready} \;\rightarrow\; \Box_{<10}\,\neg \mathit{at\_}\ell_2,$$

follows by transitivity and monotonicity from the two bounded-unless formulas

$$\mathit{ready} \;\rightarrow\; (\neg \mathit{at\_}\ell_2)\,\mathsf{U}_{\geq 2}\,(\mathit{at\_}\ell_1 \wedge x = 5), \tag{1}$$

$$(\mathit{at\_}\ell_1 \wedge x = 5) \;\rightarrow\; (\neg \mathit{at\_}\ell_2)\,\mathsf{U}_{\geq 8}\,(\mathit{at\_}\ell_1 \wedge x = 1). \tag{2}$$

The first property, (1), is enforced by two single-step lower bounds; the second property, (2), can be derived by the inductive lower-bound rule **U-IND** from the premise

$$(\mathit{at\_}\ell_1 \wedge x = i + 1 \wedge i > 0) \;\rightarrow\; (\neg \mathit{at\_}\ell_2)\,\mathsf{U}_{\geq 2}\,(\mathit{at\_}\ell_1 \wedge x = i),$$

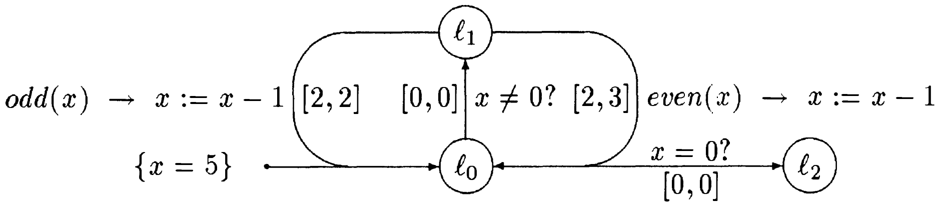
which is concluded by transitive reasoning.

The inductive lower-bound rule has a twin that combines several similar bounded-response formulas by adding up their upper bounds u. In fact, both induction rules can be generalized, by letting the bounds l and u vary as functions of i. In its more general form, we state only the *inductive upper-bound rule*. It uses again a new, rigid variable $i \in V$ that ranges over the integers $\mathbb{Z}$; for every nonnegative integer $n \geq 0$:

$$\begin{array}{|ll|}\hline \textbf{◇-IND} & \dfrac{(\varphi(i) \wedge i > 0) \;\rightarrow\; \Diamond_{\leq u_i}\,\varphi(i-1)}{\varphi(n) \;\rightarrow\; \Diamond_{\leq \Sigma_{0<i\leq n} u_i}\,\varphi(0)} \\ \hline \end{array}$$

Every instance of this rule, for any constant $n \geq 0$, is derivable from the transitive upper-bound rule **◇-TRANS**.

The general form of the inductive upper-bound rule is useful to prove upper bounds for programs with loops whose execution time is not uniform. An example for such a system is the following *odd-even* variant of the process P:



The upper bound

$$start \;\rightarrow\; \Diamond_{\leq 12}\, at\_\ell_2$$

follows by transitivity from the bounded-response property

$$start \;\rightarrow\; \Diamond_{\leq 12}(at\_\ell_0 \;\wedge\; x = 0),$$

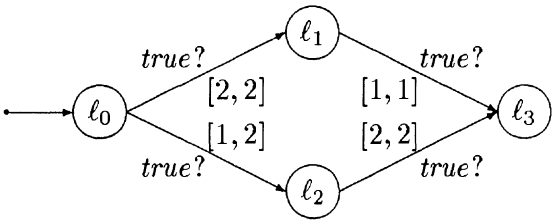
which can be concluded by the inductive upper-bound rule $\Diamond$-**IND** from the premise

$$(at\_\ell_0 \;\wedge\; x = i \;\wedge\; i > 0) \;\rightarrow\; \Diamond_{\leq 2+even(i)}\,(at\_\ell_0 \;\wedge\; x = i - 1)$$

(the expression $even(i)$ evaluates to either 1 or 0 depending on whether the value of i is even). This bounded-response formula follows from transitive reasoning.

6.2 Conditional rules

Unfortunately, the proof rules we have designed are not strong enough to show tight bounds on *nondeterministic* systems. To see this, consider the following nondeterministic variant P of a process encountered previously:



As before, P terminates either at time 3 or at time 4. However, during an execution of P, one of the two transitions $\tau_{0\rightarrow 1}$ and $\tau_{0\rightarrow 2}$ is chosen nondeterministically. Thus we cannot carry out a case analysis with respect to a state formula that selects a unique guard. Instead, we proceed in two steps. First we establish an untimed safety formula that enumerates all possible nondeterministic choices. Then we decorate the unbounded temporal formula with time bounds.

Step 1 To establish the S-validity of a temporal formula ϕ that contains only unbounded *unless* operators (i.e., $\mathsf{U}_{\geq 0}$), it suffices to show that ϕ is true over all run fragments of the untimed transition system S^- that underlies S. This can be achieved with the help of any conventional timeless proof system (for instance, the proof system given in [MP83]).

For example, to derive the lower bound 3 on the termination of our example P, we show the untimed formula

$$ready \;\rightarrow\; ((at\_\ell_\perp \;\mathsf{U}\; at\_\ell_0 \;\mathsf{U}\; at\_\ell_1) \;\vee\; (at\_\ell_\perp \;\mathsf{U}\; at\_\ell_0 \;\mathsf{U}\; at\_\ell_2)) \qquad (\dagger)$$

(nested *unless* operators associate to the right).

Step 2 To add time bounds to this disjunction of nested unless formulas, we need *conditional* single-step rules. They establish single-step real-time bounds under the assumption that a particular disjunct has been chosen. The time bounds can, then, be combined by the transitivity rules.

Nondeterministic lower bounds

The *conditional single-step lower-bound rule* uses the minimal delay $l_\tau \in \mathsf{N}$ of a transition $\tau \in \mathcal{T}$:

$$\boxed{\begin{array}{ll} \textsf{U-CSS} & \begin{array}{c} p \rightarrow \neg enabled(\tau) \\ \{q\}\, \mathcal{T} - \tau \,\{\neg r\} \\ \hline (p \,\mathsf{U}_{\geq l}\, q \,\mathsf{U}\, (r \wedge \phi)) \;\rightarrow\; (p \,\mathsf{U}_{\geq l}\, q \,\mathsf{U}_{\geq l_\tau}\, (r \wedge \phi)) \end{array} \end{array}}$$

The rule U-**CSS** is S-sound for any temporal formula ϕ. The second premise ensures that only the transition τ can establish $r \wedge \phi$, but to do so, it must be enabled for at least l_τ time units, during which q is true.

In our example, we use the conditional single-step lower-bound rule U-**CSS** with respect to the transitions $\tau_{0\rightarrow 1}$ and $\tau_{0\rightarrow 2}$ to derive the conditional single-step bounds

$$(at\_\ell_\perp \,\mathsf{U}\, at\_\ell_0 \,\mathsf{U}\, at\_\ell_1) \;\rightarrow\; (at\_\ell_\perp \,\mathsf{U}\, at\_\ell_0 \,\mathsf{U}_{\geq 2}\, at\_\ell_1),$$

$$(at\_\ell_\perp \,\mathsf{U}\, at\_\ell_0 \,\mathsf{U}\, at\_\ell_2) \;\rightarrow\; (at\_\ell_\perp \,\mathsf{U}\, at\_\ell_0 \,\mathsf{U}_{\geq 1}\, at\_\ell_2).$$

They allow us to conclude, from (†),

$$ready \;\rightarrow\; ((at\_\ell_\perp \,\mathsf{U}\, at\_\ell_0 \,\mathsf{U}_{\geq 2}\, at\_\ell_1) \vee (at\_\ell_\perp \,\mathsf{U}\, at\_\ell_0 \,\mathsf{U}_{\geq 1}\, at\_\ell_2)). \qquad (\ddagger)$$

To collapse nested *bounded-unless* operators, we use the temporal formula U-**COLL**:

$$\boxed{\textsf{U-COLL} \quad (p \,\mathsf{U}_{\geq l_1}\, q \,\mathsf{U}_{\geq l_2}\, \phi) \;\rightarrow\; ((p \vee q) \,\mathsf{U}_{\geq l_1 + l_2}\, \phi)}$$

Note that this temporal formula, which is generally valid, can be derived by from the transitive lower-bound rule U-**TRANS** by using the two tautologies

$$(p \,\mathsf{U}_{\geq l_1}\, q \,\mathsf{U}_{\geq l_2}\, \phi) \;\rightarrow\; (p \,\mathsf{U}_{\geq l_1}\, q \,\mathsf{U}_{\geq l_2}\, \phi),$$

$$(q \,\mathsf{U}_{\geq l_2}\, \phi) \;\rightarrow\; (q \,\mathsf{U}_{\geq l_2}\, \phi).$$

From (‡) we obtain by collapsing and monotonicity

$$ready \;\rightarrow\; ((at\_\ell_{\perp,0} \,\mathsf{U}_{\geq 2}\, at\_\ell_1) \vee (at\_\ell_{\perp,0} \,\mathsf{U}_{\geq 1}\, at\_\ell_2));$$

that is, using the (untimed) validity $p \,\mathsf{U}\, true$ and monotonicity,

$$ready \;\rightarrow\; ((at\_\ell_{\perp,0} \,\mathsf{U}_{\geq 2}\, at\_\ell_1 \,\mathsf{U}\, true) \vee (at\_\ell_{\perp,0} \,\mathsf{U}_{\geq 1}\, at\_\ell_2 \,\mathsf{U}\, true)).$$

Adding conditional single-step lower bounds for the transitions $\tau_{1\rightarrow 3}$ and $\tau_{2\rightarrow 3}$ gives

$$ready \;\rightarrow\; ((at\_\ell_{\perp,0} \,\mathsf{U}_{\geq 2}\, at\_\ell_1 \,\mathsf{U}_{\geq 1}\, true) \vee (at\_\ell_{\perp,0} \,\mathsf{U}_{\geq 1}\, at\_\ell_2 \,\mathsf{U}_{\geq 2}\, true)),$$

and by collapsing and monotonicity we finally arrive at the desired bounded-invariance property

$$ready \;\rightarrow\; \Box_{<3}\, \neg at\_\ell_3.$$

Nondeterministic upper bounds

Conditional upper-bound reasoning does not require the nesting of *unless* operators. The *conditional single-step upper-bound rule* uses the maximal delay $u_\tau \in \mathsf{N}$ of a transition $\tau \in \mathcal{T}$:

$$\boxed{\begin{array}{ll} \Diamond\textbf{-CSS} & p \rightarrow enabled(\tau) \\ & \{p\}\,\tau\,\{\neg p\} \\ \cline{2-2} & (p \,\mathsf{U}\, \phi) \rightarrow \Diamond_{\leq u_\tau} \phi \end{array}}$$

Clearly, the rule $\Diamond$-**CSS** is S-sound for any temporal formula ϕ. Note that the second premise of this rule is trivially valid if τ becomes disabled by being taken, as is the case for all transitions of a timed transition system that is given by timed transition diagrams (recall that we have ruled out self-loops in transition diagrams). It is also worth pointing out that both conditional lower-bound and conditional upper-bound reasoning rely only on assumptions that are built only from state formulas by positive boolean connectives and unbounded *unless* operators and, therefore, define untimed safety properties. Accordingly, the first step of conditional reasoning can be carried out by any untimed method for deriving safety properties.

To derive the upper bound 4 on the termination of our example P, we show first the untimed formula

$$start \rightarrow ((at\_\ell_0 \,\mathsf{U}\, at\_\ell_1) \vee (at\_\ell_0 \,\mathsf{U}\, at\_\ell_2)).$$

By the conditional single-step upper-bound rule $\Diamond$-**CSS** with respect to the transitions $\tau_{0\rightarrow 1}$ and $\tau_{0\rightarrow 2}$, we derive the conditional single-step bounds

$$(at\_\ell_0 \,\mathsf{U}\, at\_\ell_1) \rightarrow \Diamond_{\leq 2}\, at\_\ell_1,$$

$$(at\_\ell_0 \,\mathsf{U}\, at\_\ell_2) \rightarrow \Diamond_{\leq 2}\, at\_\ell_2.$$

They allow us to conclude

$$start \rightarrow (\Diamond_{\leq 2}\, at\_\ell_1 \vee \Diamond_{\leq 2}\, at\_\ell_2).$$

Now we can proceed by *unconditional* upper-bound reasoning to arrive at the desired bounded-response property

$$start \rightarrow \Diamond_{\leq 4}\, at\_\ell_3.$$

7 Explicit-clock Reasoning

None of our state formulas is able to refer to the value of the time, because the only real-time references that are admitted in temporal formulas are time bounds on temporal operators. In this section, we investigate the consequences of extending the notion of state, by adding a variable *now* that represents, in every state, the current time. This extension is interesting, because once we are given explicit access to the global clock through the clock variable *now*, both bounded-invariance and bounded-response properties can alternatively be formulated as unbounded unless formulas and, consequently, be verified by conventional timeless techniques for establishing safety properties of transition systems.

7.1 Explicit-clock transition systems

Let $S = \langle V, \Sigma, \Theta, \mathcal{T}, l, u \rangle$ be a timed transition system. We introduce the following new variables:

- A *clock variable* *now* that ranges over the integers; it records, in every state σ_i of a computation $\rho = (\sigma, T)$, the corresponding time T_i.
- For every transition $\tau \in \mathcal{T}$, a *delay counter* δ_τ that ranges over the set $\{0, 1, \ldots u_\tau\}$ of nonnegative integers; it records, in every state of a computation, for how many clock ticks the transition τ has been continuously enabled without being taken. We write $\delta^i_{j\to k}$ short for $\delta_{\tau^i_{j\to k}}$.

The *explicit-clock transition system* $S^* = \langle V^*, \Sigma^*, \Theta^*, \mathcal{T}^* \rangle$ associated with S is defined to be the following untimed transition system:

1. $V^* \ = \ V \cup \{now\} \cup \{\delta_\tau \mid \tau \in \mathcal{T}\}$.
2. Σ^* contains all interpretations of V^*. Thus, every *explicit-clock state* $\sigma \in \Sigma^*$ of S^* is a tuple that contains a state $\sigma^- \in \Sigma$ of S, a value $\sigma(now) \in \mathsf{Z}$ for the clock variable *now*, and a value $\sigma(\delta_\tau) \in \mathsf{N}$ for each delay counter δ_τ.
3. An explicit-clock state $\sigma \in \Sigma^*$ is initial iff it extends an initial state of S:
$$\sigma \in \Theta^* \quad \text{iff} \quad \sigma^- \in \Theta.$$
4. Every transition of S is extended: $\mathcal{T}^*$ contains, for every $\tau \in \mathcal{T}$, a transition τ^* such that $(\sigma_1, \sigma_2) \in \tau^*$ iff for all $\tau' \in \mathcal{T}$,
$$\begin{aligned}
&(\sigma_1^-, \sigma_2^-) \in \tau,\\
&\sigma_1(\delta_\tau) \geq l_\tau,\\
&\sigma_2(now) = \sigma_1(now),\\
&\sigma_2(\delta_{\tau'}) = \begin{cases} \sigma_1(\delta_{\tau'}) & \text{if } \tau' \neq \tau \text{ and } \tau' \text{ is enabled on } \sigma_2^-,\\ 0 & \text{otherwise.}\end{cases}
\end{aligned}$$
The second clause, $\sigma_1(\delta_\tau) \geq l_\tau$, enforces all lower-bound requirements on computations.
In addition, $\mathcal{T}^*$ contains the idle transition τ_I^* and the *tick transition* τ_T^*, which advances time: $(\sigma_1, \sigma_2) \in \tau_T^*$ iff for all $\tau' \in \mathcal{T}$,
$$\begin{aligned}
&\sigma_1^- = \sigma_2^-,\\
&\sigma_2(now) = \sigma_1(now) + 1,\\
&\sigma_2(\delta_{\tau'}) = \begin{cases} \sigma_1(\delta_{\tau'}) + 1 & \text{if } \tau' \neq \tau \text{ and } \tau' \text{ is enabled on } \sigma_2^-,\\ 0 & \text{otherwise,}\end{cases}\\
&\sigma_2(\delta_{\tau'}) \leq u_{\tau'}.
\end{aligned}$$
The last clause enforces all finite upper-bound requirements on computations.

From S^* we obtain a *fair transition system* ([MP92]) by adding the following fairness requirements:

5. A *weak-fairness* (justice) assumption stipulates that a transition cannot be continuously enabled without being taken. Let the *weakly-fair extension* S^j of S^* be the fair transition system that is obtained from S^* by adding a weak-fairness assumption for every transition τ^* iff τ has a maximal delay of ∞.

6. A *strong-fairness* assumption stipulates that a transition cannot be enabled infinitely often without being taken. Let the *strongly-fair extension* S^f of S^* be the fair transition system that is obtained from S^j by adding a strong-fairness assumption for the tick transition τ_T.

The weak-fairness requirements enforce all infinite upper-bound requirements on computations. The strong-fairness requirement for the tick transition ensures the progress of time: from the operationality restriction of S it follows that every initialized computation of S^f contains infinitely many tick transitions.

If $\sigma = \sigma_0\sigma_1\sigma_2\ldots$ is an infinite sequence of explicit-clock states from Σ^*, we write σ^- for the corresponding sequence $\sigma_0^-\sigma_1^-\sigma_2^- \ldots$ of states from Σ. It is not hard to see that the timed transition system S and the fair explicit-clock transition system S^f are related in the following way:

- For every initialized computation (σ, T) of S, there is an infinite sequence σ^* of explicit-clock states with $(\sigma^*)^- = \sigma$ and $\sigma^*(now) = T$ such that σ^* is an initialized computation of S^f (in the first state of σ^*, choose the delay counters of all enabled transitions larger than all minimal delays of S; otherwise, let all delay counters record the times that the corresponding transitions have been enabled).

- For every initialized computation σ of S^f, the timed state sequence $(\sigma^-, \sigma(now))$ is an initialized computation of S.

7.2 Explicit-clock formulas

We now translate every bounded-invariance and bounded-response formula ϕ over V into an untimed unless formula ϕ^* that contains the clock variable *now*. The *explicit-clock formula* ϕ^* is constructed such that it is S^*-valid iff ϕ is S-valid:

- The explicit-clock translation of the bounded-invariance formula $p \to \Box_{<l}\, q$ is

$$(p \wedge now = t) \;\to\; q \,\mathsf{U}\, (now \geq t + l),$$

  for a new, rigid variable $t \in V$ that ranges over the integers Z (recall that V supplies suitable variables that occur neither in the description of S nor in p or q).

- The explicit-clock translation of the bounded-response formula $p \to \Diamond_{\leq u}\, q$ is

$$(p \wedge \neg q \wedge now = t) \;\to\; (now \leq t + u) \,\mathsf{U}\, q.$$

  for a new, rigid variable $t \in V$ that ranges over Z.

Both unless formulas use the rigid variable t to record the time of the p-state. In the case of bounded-response properties, the explicit-clock translation exploits the fact that the time is guaranteed to reach and surpass $t + u$, for any value of t. We emphasize that neither of the state formulas p and q may contain free occurrences of the clock variable or any of the delay counters.

From the correspondence between the computations of the timed transition system S and the fair explicit-clock transition system S^f it follows that the explicit-clock formula ϕ^* is S^f-valid iff ϕ is S-valid. Indeed, since the explicit-clock translations of bounded-invariance and bounded-response properties are safety formulas, there is no need to add fairness assumptions to the explicit-clock transition system:

Proposition 2. *If S is a timed transition system and ϕ is a bounded-invariance or bounded-response formula, then the explicit-clock translation ϕ^* is S^*-valid iff ϕ is S-valid.*

7.3 Untimed temporal reasoning about real time

Proposition 2 leads to an alternative and quite different approach to the verification of real-time properties: to prove the S-validity of a real-time property ϕ (over V), we establish instead the S^*-validity of the untimed safety formula ϕ^* (over V^*). To show the unbounded unless formulas that result from translating bounded-invariance and bounded-response properties, a single timeless *unless rule* suffices ([MP83]):

$$\boxed{\begin{array}{ll} \textbf{UNLESS} & p \rightarrow \varphi \\ & \{\varphi\}\, \mathcal{T}^*\, \{\varphi \vee r\} \\ & \varphi \rightarrow q \\ \hline & p \rightarrow q \,\mathsf{U}\, r \end{array}}$$

We point out that all three premises of the unless rule are state formulas over the augmented set V^* of variables; their S^*-validity typically is shown by proving them generally valid. The state formula φ is called the *invariant* of the rule, because the main (i.e., second) premise asserts that φ is preserved by all transitions of the system S^* (unless the desired state condition r is established).

To demonstrate this kind of "explicit-clock" real-time reasoning, consider again the single-process system P with the data precondition $x = 0$ and the following timed transition diagram:

$$\{x = 0\} \longrightarrow \textcircled{\ell_0} \xrightarrow[{[2,3]}]{x = 0?} \textcircled{\ell_1}$$

The lower bound on the termination of P,

$$\mathit{ready} \;\rightarrow\; \Box_{<2} \neg \mathit{at\_}\ell_1,$$

is translated into the explicit-clock formula

$$(\mathit{ready} \wedge \mathit{now} = t) \;\rightarrow\; (\neg \mathit{at\_}\ell_1) \,\mathsf{U}\, (\mathit{now} \geq t + 2),$$

which can be derived by the unless rule from the invariant

$$(\mathit{at\_}\ell_\perp \wedge \mathit{now} \geq t) \vee (\mathit{at\_}\ell_0 \wedge \mathit{now} \geq t + \delta_{0\rightarrow 1})$$

(recall that the delay counter $\delta_{0\rightarrow 1}$ of the transition $\tau_{0\rightarrow 1}$ ranges over the set $\{0, 1, 2, 3\}$ only). The upper bound on the termination of P,

$$\mathit{start} \;\rightarrow\; \Diamond_{\leq 3}\, \mathit{at\_}\ell_1,$$

is translated into the untimed unless formula

$$(\mathit{start} \wedge \neg \mathit{at\_}\ell_1 \wedge \mathit{now} = t) \;\rightarrow\; (\mathit{now} \leq t + 3) \,\mathsf{U}\, \mathit{at\_}\ell_1,$$

which can be concluded by the unless rule from the invariant

$$\mathit{at\_}\ell_0 \wedge x = 0 \wedge t \leq \mathit{now} \leq t + 3 \wedge \mathit{now} \leq t + \delta_{0\rightarrow 1}.$$

8 Completeness

Throughout this section, we consider completeness *relative to state reasoning* (relative to assertional validity); that is, we assume to be given an oracle that decides the validity of closed state formulas. The unless rule is known to be complete relative to state reasoning for establishing unless formulas of transition systems, provided the underlying assertion language is sufficiently expressive to encode computations ([MP83]). From Proposition 2 it follows immediately that explicit-clock reasoning is relative complete for showing bounded-invariance as well as bounded-response properties of timed transition systems. As for bounded-operator reasoning, we first show relative completeness in the case that all real-time constraints are either lower-bound constraints or upper-bound constraints. Then we motivate the need for crossover reasoning in the general case and present crossover rules to combine lower-bound and upper-bound constraints.

8.1 Crossover-free reasoning

Given a timed transition system S, we assume that all untimed safety properties of S can be derived; that is, we assume an untimed proof system that is complete for untimed safety reasoning. Although such a proof system cannot exist for most data domains, there are temporal proof systems that are complete relative to state reasoning ([MP89]). In addition, we suppose that the nontrivial timing constraints of S are either all minimal delays or all maximal delays. The following theorem shows that under these assumptions, our bounded-operator rules can derive every bounded-invariance and bounded-response property of S.

Theorem. *Let $S = \langle V, \Sigma, \Theta, \mathcal{T}, l, u\rangle$ be a timed transition system such that either $l_\tau = 0$ for all $\tau \in \mathcal{T}$ or $u_\tau = \infty$ for all $\tau \in \mathcal{T}$. Let ϕ be a bounded-invariance or a bounded-response formula. If ϕ is S-valid, then it can be derived by the monotonicity, transitivity, and conditional single-step rules relative to untimed safety reasoning.*

Proof. (I) Suppose that all maximal delays of S are ∞. First we observe that, under the given restrictions, untimed reasoning is complete for untimed properties of S. This is because, in the absence of finite maximal delays, there is a time sequence T for every computation σ of the untimed weakly-fair transition system S_j^- that underlies S such that (σ, T) is a computation of S (choose all time steps large enough). It follows that any untimed temporal formula that is S-valid is also S_j^--valid and, thus, can be established by untimed reasoning.

Any bounded-response property is either trivially not S-valid or can be established by untimed safety reasoning. Now suppose that the bounded-invariance property

$$p \;\rightarrow\; \Box_{<l} \neg q \tag{1}$$

is S-valid; we show that it can be derived within our proof system. The main idea is to see that in order for (1) to be valid, for any p-state in an initialized computation of S there has to be a sequence of nonoverlapping single-step lower bounds that add up to at least l before a q-state can be reached. We show that there are only finitely many such ways in which a q-state can be delayed for l time units; hence they can be enumerated by a single untimed formula.

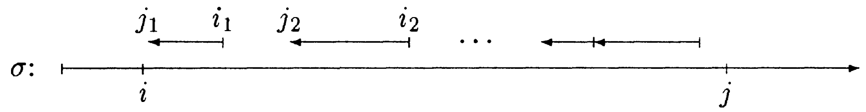
Consider an arbitrary computation $\rho = (\sigma, T)$ of S such that σ_i, for $i \geq 0$, is a p-state. Let σ_j be the first q-state with $j \geq i$; if no such state exists, let $j = \infty$. We write τ_k for the transition that is completed at position $k \geq 0$ of ρ. A lower-bound l-constraint pattern for $\sigma_{i..j}$ is a finite sequence of nonoverlapping single-step lower bounds between i and j that add up to at least l. Formally, a constraint pattern C is a sequence of transitions $\tau_{i_1}, \ldots \tau_{i_n}$. The pattern C is a lower-bound l-constraint pattern iff

$$\sum_{1 \leq k \leq n} l_{\tau_{i_k}} \geq l;$$

it is a lower-bound constraint pattern for $\sigma_{i..j}$ iff

(a) $i = i_0 < i_1 < \cdots < i_n \leq j$ and
(b) for all $1 \leq k \leq n$, the transition τ_{i_k} is not enabled on some state σ_{j_k} such that $i_{k-1} \leq j_k < i_k$.

A lower-bound constraint pattern for $\sigma_{i..j}$ can be visualized by annotating the computation ρ with backward arrows that represent single-step lower bounds:



Two constraint patterns are equivalent iff one is a subpattern of the other (i.e., can be obtained by omitting transitions). It is not hard to show the following two properties of lower-bound constraint patterns:

Property A There is a lower-bound l-constraint pattern for $\sigma_{i..j}$ (use the truth of (1) over the i-th suffix of ρ).

Property B There are only finitely many different equivalence classes of lower-bound l-constraint patterns.

We add, for every transition $\tau \in \mathcal{T}$, the boolean variable $completed_\tau$ to our assertion language; $completed_\tau$ is intended to be true in a state σ_i, for $i \geq 0$, of a computation $\rho = (\sigma, \tau)$ iff the transition τ is completed at the position i of ρ. For our purpose, it will suffice that $completed_\tau$ satisfies the two axioms (which are state formulas)

$$\{true\}\, \tau\, \{completed_\tau\},$$

$$\{true\}\, \mathcal{T} - \tau\, \{\neg completed_\tau\}. \qquad (\dagger)$$

By Property A, there is an untimed formula of the form

$$(\neg q)\, \mathsf{U}^0\, (\neg q \wedge \neg enabled(\tau_{i_1}))\, \mathsf{U}\, (\neg q)\, \mathsf{U}\, (\neg q \wedge completed_{\tau_{i_1}})\, \mathsf{U} \ldots$$
$$\mathsf{U}\, (\neg q \wedge completed_{\tau_{i_n}})$$

that is true over the i-th suffix of ρ (where $p\mathsf{U}^0\phi$ stands for $\phi \vee (p\mathsf{U}^0\phi)$). Since there are, by Property B, only finitely many formulas of this form, $p \rightarrow \psi$ for some finite disjunction ψ of nested unless formulas is S-valid and, thus, given by untimed safety reasoning. From ($\dagger$) we infer by the conditional single-step lower-bound rule **U-CSS** with respect to any transition $\tau \in \mathcal{T}$ that

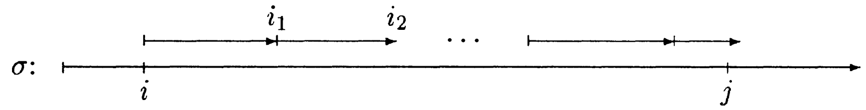
$$(\neg enabled(\tau))\, \mathsf{U}_{\geq l}\, \varphi\, \mathsf{U}\, (completed_\tau \wedge \phi) \quad \rightarrow$$
$$(\neg enabled(\tau))\, \mathsf{U}_{\geq l}\, \varphi\, \mathsf{U}_{\geq l_\tau}\, (completed_\tau \wedge \phi)$$

for any state formula φ and temporal formula ϕ. Hence we can decorate the untimed nested unless formula with time bounds. By repeated collapsing and monotonicity similar to the sample lower-bound derivation of Subsection 6.2, we arrive at the desired bounded-invariance property (1).

(II) Now suppose that all minimal delays of S are 0; the proof proceeds similarly to the previous case. Untimed reasoning is complete for untimed properties of S, because S is operational. Any bounded-invariance property is either trivially not S-valid or can be established by untimed safety reasoning. So let us assume that the bounded-response property

$$p \rightarrow \Diamond_{\leq u} q \tag{2}$$

is S-valid. Consequently, every p-state in an initialized computation of S has to be followed by a q-state that can be reached by a sequence of overlapping single-step upper bounds that add up to at most u. We visualize single-step upper bounds by forward arrows:



Formally, let $\rho = (\sigma, T)$ be a computation of S such that σ_i, for $i \geq 0$, is a p-state, and let σ_j be the first q-state with $j \geq i$. For the sake of simplicity, we assume that the transition τ_k, which is completed at the position $k \geq 0$ of ρ, is not enabled on σ_k (otherwise split τ_k into two identical transitions with different names). A constraint pattern $\tau_{i_1}, \ldots \tau_{i_n}$ is an upper-bound u-constraint pattern iff

$$\sum_{1 \leq k \leq n} u_{\tau_{i_k}} \leq u;$$

it is an upper-bound constraint pattern for $\sigma_{i..j}$ iff

(a) $i = i_0 < i_1 < \cdots < i_{n-1} < j \leq i_n$ and
(b) for all $1 \leq k \leq n$, the transition τ_{i_k} is enabled but not completed at all states σ_{j_k} such that $i_{k-1} \leq j_k < i_k$.

It is not hard to see that upper-bound constraint patterns, too, satisfy two crucial properties:

Property A There is an upper-bound u-constraint pattern for $\sigma_{i..j}$ (use the truth of (2) over the i-th suffix of ρ).

Property B There are only finitely many different equivalence classes of upper-bound u-constraint patterns (use the operationality of S).

By Property A, there is an untimed formula of the form

$$\begin{array}{l}(\mathit{enabled}(\tau_{i_1}) \wedge \neg \mathit{completed}_{\tau_{i_1}}) \,\mathsf{U}\, (\mathit{enabled}(\tau_{i_2}) \wedge \neg \mathit{completed}_{\tau_{i_2}}) \,\mathsf{U}\, \ldots \\ \quad \mathsf{U}\, (\mathit{enabled}(\tau_{i_n}) \wedge \neg \mathit{completed}_{\tau_{i_n}}) \,\mathsf{U}\, q\end{array}$$

that is true over the i-th suffix of ρ. By Property B, there is again a finite disjunction ψ of nested unless formulas such that the implication $p \rightarrow \psi$ is S-valid and, therefore, given by untimed safety reasoning. By repeated application of the conditional single-step upper-bound rule $\Diamond$-**CSS**, transitivity, and monotonicity, we arrive at the desired bounded-response property (2). More specifically, to collapse nested *bounded-eventually* operators, we can use the valid temporal formula $\Diamond$-**COLL**, which is derivable from the transitive upper-bound rule $\Diamond$-**TRANS**:

$$\boxed{\Diamond\text{-}\mathbf{COLL} \quad (\Diamond_{\leq u_1} \Diamond_{\leq u_2} \phi) \rightarrow \Diamond_{\leq u_1 + u_2} \phi}$$

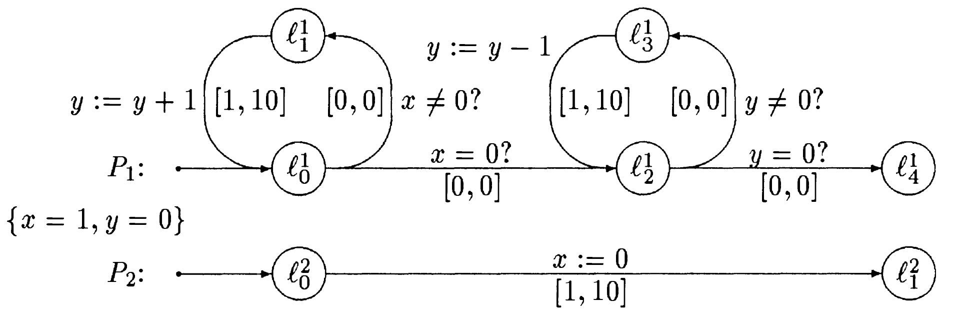
■

8.2 Crossover reasoning

So far, we have used lower-bound rules to derive bounded-invariance properties and we have used upper-bound to derive bounded-response properties. In general, the situation is more complicated: *both* the lower-bound and the upper-bound rules may be necessary to derive a bounded-invariance (or bounded-response) property. Indeed, we may need additional *crossover* rules, which combine lower-bound and upper-bound requirements.

Example: Race condition

The need for crossover rules can be illustrated by a multiprocessing system that looks innocent at first glance but turns out to be rather intricate, because its execution time depends on an interesting interplay of the minimal delays and the maximal delays for transitions that belong to different processes. This *increment-decrement* system is defined by the following timed transition diagram:



We wish to analyze the worst-case (maximal) running time of the synchronous two-process shared-variables multiprocessing system

$$\{x = 1, y = 0\}[P_1 \|_s P_2].$$

Note that the first process, P_1, consumes the maximal amount of time if its first loop, in which the value of y is incremented, is executed as often (fast) as possible — 11 times: the control of P_1 may enter the first loop 11 times before and at time 10, the latest time at which the second process closes the loop, and it may spend another 10 time units in the first loop after the guard has been reversed. In this worst (slowest) case, the first loop is left at time 20 with $y = 11$ and, thus, the second loop may use up no more than 110 time units. It follows that P_1 terminates by time 130.

Assuming that assignments cost at least 2 time units (instead of 1), tests still being free, the maximal value of y would be only 6, implying termination by time 80: the increase of individual *lower* bounds decreases the composite *upper* bound! This phenomenon vividly demonstrates that real-time reasoning amounts to more than simply adding up minimal delays or maximal delays of individual transitions; it shows that lower-bound and upper-bound requirements are not independent, but may jointly affect the global time bounds for the execution time of a system.

Let us now formally prove the upper bound 130 on the termination of P_1 by explicit-clock reasoning. To simplify the derivation, we may assume that both processes start simultaneously at time 0. Then we can infer the explicit-clock formula

$$(\mathit{start} \wedge \mathit{now} = \delta^1_{0\to 1} = \delta^2_{0\to 1} = 0) \;\rightarrow\; (\mathit{now} \leq 130)\, \mathsf{U}\, \mathit{at\_}\ell^1_4$$

by the unless rule from the following global invariant:

$$\begin{array}{l}
(at_\ell_0^1 \wedge at_\ell_0^2 \wedge (y = now = \delta^2_{0\to 1} = 0 \vee 1 \leq y \leq now = \delta^2_{0\to 1})) \vee \\
(at_\ell_1^1 \wedge at_\ell_0^2 \wedge y + \delta^1_{1\to 0} \leq now = \delta^2_{0\to 1}) \vee \\
(at_\ell_0^1 \wedge at_\ell_1^2 \wedge 1 \leq y \leq 11 \wedge now \leq 20) \vee \\
(at_\ell_1^1 \wedge at_\ell_1^2 \wedge y \leq 10 \wedge now \leq 10 + \delta^1_{1\to 0}) \vee \\
(at_\ell_2^1 \wedge at_\ell_1^2 \wedge y \leq 11 \wedge now + 10y \leq 130) \vee \\
(at_\ell_3^1 \wedge at_\ell_1^2 \wedge 1 \leq y \leq 11 \wedge now + 10y \leq 130 + \delta^1_{3\to 2}).
\end{array}$$

This proof of timely termination resembles a mechanical, exhaustive case analysis of all possible state-time combinations that can occur during an execution of the two processes of the *increment-decrement* system. The *bounded-operator* proof of the desired bound on termination, on the other hand, closely follows the intuitive argument we outlined above.

Crossover rules

To mimic the informal argument for the timely termination of the *increment-decrement* system by a bounded-operator proof, we use the *crossover upper-bound rule*:

$$\boxed{\begin{array}{ll}
\textbf{$\Diamond$-MIX} & u < l \\
& p \;\rightarrow\; \Diamond_{\leq u}\, \hat{p} \\
& q \;\rightarrow\; \Box_{<l}\, \hat{q} \\
& \{p\}\,(\mathcal{T} - \tau_I)^-\,\{q\} \\
& p \;\rightarrow\; \neg ready \\
\hline
& p \;\rightarrow\; \Diamond_{\leq u}\,(\hat{p} \wedge \hat{q})
\end{array}}$$

This rule is a modification of the temporal formula

$$(\Diamond_{\leq u}\, \hat{p} \wedge \Box_{<l}\, \hat{q}) \;\rightarrow\; \Diamond_{\leq u}\,(\hat{p} \wedge \hat{q}),$$

which is valid if $u < l$. The more complicated form of the rule is needed, because reasoning about lower bounds and reasoning about upper bounds are asymmetric: while bounded-invariance formulas refer, intuitively, to the last state *before* a transition is taken, bounded-response formulas refer to the first state *after* a transition is taken. This dichotomy is captured by the inverse verification condition

$$\{p\}\,(\mathcal{T} - \tau_I)^-\,\{q\},$$

which requires that in any computation $\rho = (\sigma, T)$ of S, if σ_{i+1} is a p-state and $\sigma_{i+1} \neq \sigma_i$, then σ_i is a q-state. Also observe that every computation of S whose first state falsifies the ready condition is the suffix of another computation of S whose first state satisfies *ready*. The S-soundness of the rule **$\Diamond$-MIX** follows.

We give here only a brief sketch of the bounded-operator proof for the bounded-response property

$$start \;\rightarrow\; \Diamond_{\leq 130}\, at\_\ell_4^1$$

of the *increment-decrement* system. The derivation relies, as expected, on an interplay of lower-bound and upper-bound rules. First we show that within 10 time units P_1 can increase the value of y at most to 10:

$$ready \;\rightarrow\; \Box_{<11}\,(y \leq 10);$$

this is done by inductive lower-bound reasoning. Then we apply the crossover upper-bound rule $\diamond$-**MIX** to the single-step upper bound

$$start \;\rightarrow\; \diamond_{\leq 10}(at\_\ell_1^2 \,\wedge\, x = 0),$$

thus obtaining the bounded-response property

$$start \;\rightarrow\; \diamond_{\leq 10}(y \leq 10 \,\wedge\, at\_\ell_1^2 \,\wedge\, x = 0).$$

From here we proceed by pure upper-bound reasoning, performing a case analysis on the locations of P_1.

While the crossover upper-bound rule combines a bounded-invariance property and a bounded-response property into a bounded-response property, its counterpart, the *crossover lower-bound rule*, yields a bounded-unless formula:

$$\begin{array}{|ll|}
\hline
\textsf{U}\text{-}\mathbf{MIX} & u < l \\
& p \;\rightarrow\; \Box_{<l}\,\hat{p} \\
& q \;\rightarrow\; \diamond_{\leq u}\,\hat{q} \\
& \{p\}\,\mathcal{T} - \tau_I\,\{q\} \\
& \hat{q} \;\rightarrow\; (\hat{q}\,\mathsf{U}\,(\hat{q} \,\wedge\, \neg\hat{p})) \\
\cline{2-2}
& p \;\rightarrow\; (\hat{p}\,\mathsf{U}_{\geq l}\,\hat{q}) \\
\hline
\end{array}$$

This rule is S-sound, because it originates with the valid temporal formula (for $u < l$)

$$(\Box_{<l}\,\hat{p} \,\wedge\, \diamond_{\leq u}(\hat{q}\,\mathsf{U}\,(\hat{q} \,\wedge\, \neg\hat{p}))) \;\rightarrow\; (\hat{p}\,\mathsf{U}_{\geq l}\,\hat{q}).$$

Note that the last premise, which contains only an unbounded *unless* operator, can be established by untimed reasoning.

The crossover lower-bound rule $\textsf{U}$-**MIX** can be used to derive the lower bound

$$ready \;\rightarrow\; \Box_{<2}\,\neg at\_\ell_4^1$$

of the *increment-decrement* system.

9 Discussion

The *increment-decrement* example of the previous section illustrates the trade-off between bounded-operator reasoning and explicit-clock reasoning. Compare the two proofs of the upper bound on termination: while the bounded-operator (or "hidden-clock") style of real-time verification refers to time only through the relative offsets of time-bounded temporal operators, the explicit-clock style uses ordinary untimed temporal operators and refers to the absolute time in state formulas. Both styles trade off the complexity of the temporal proof structure against the complexity of the state invariants:

- The *hidden-clock* approach combines the conclusions of single-step rules within complex proof structures similar to the proof lattices for establishing ordinary (untimed) *liveness* properties ([OL82],[MP84]). Each application of a single-step rule, however, uses a simple *local* invariant that refers to a few system locations only.

- The *explicit-clock* method employs only the plain unless rule — an (untimed) *safety* rule — but requires a powerful *global* invariant that relates all system locations.

An alternative proof of the lower and upper bounds on the termination of the *increment-decrement* system can be found in [LA90]; the proof presented there uses global mappings between time-constrained automata with explicit references to time and is similar to our explicit-clock approach.

Open problems and recent developments

There are several problems that have been left open in this paper and some progress has been made since its submission. First, we presented bounded-operator proof rules and explicit-clock translations for the verification of bounded-invariance and bounded-response properties only. While these two classes seem to cover the most important timing constraints, naturally we wish to classify more complex real-time properties of systems and provide the corresponding bounded-operator proof rules and explicit-clock translations. Recall, for example, the *traffic light* system from Subsection 3.3. While the two correctness conditions (A) and (B) can be specified by temporal formulas,

$$\mathit{request} \;\rightarrow\; \Diamond_{\leq 5}\,\Box_{<5}\,(\mathit{light} = \mathit{green}), \tag{A}$$

$$\mathit{request} \;\rightarrow\; \Diamond_{\leq 25}\,\neg\mathit{request}, \tag{B}$$

the third part of the specification, condition (C), is most naturally expressed using the time-constrained *past* temporal operator $\boxminus_{<25}$ ("throughout the last 25 time units"):

$$(\,\boxminus_{<25}\,\neg\mathit{request}) \;\rightarrow\; \mathit{light} = \mathit{red}. \tag{C}$$

Only the formula (B) defines a simple bounded-response property. The formula (A) follows by monotonicity from the bounded-response property

$$\mathit{request} \;\rightarrow\; \Diamond_{\leq 5}\,(\mathit{completed}_{1\rightarrow 2} \,\vee\, \mathit{completed}_{3\rightarrow 2})$$

and the bounded-invariance property

$$(\mathit{completed}_{1\rightarrow 2} \,\vee\, \mathit{completed}_{3\rightarrow 2}) \;\rightarrow\; \Box_{<5}\,(\mathit{light} = \mathit{green})$$

(we write $\mathit{completed}_{i\rightarrow j}$ short for the boolean condition that indicates the completion of the transition from ℓ_i to ℓ_j; also assume $\delta = 4$).

Second, we proved the relative completeness of bounded-operator reasoning only in the case that the lower bounds and the upper bounds do not interfere with each other. In [Hen91] it has been shown that some history information, say, in form of time-constrained past temporal operators is necessary to achieve relative completeness in the general case. Note that some information about the past of a state in a computation is available in explicit-clock reasoning, namely, in form of the delay counters. An alternative method of incorporating history information relies on an *age* operator that indicates, in each state of a computation, for how long a formula has been continuously true ([MMP92]).

Third, we used the discrete time domain of the integers. This does not necessarily mean that all events occur at integer points in time, only that the time of events is recorded by a fictitious digital clock with finite precision ([AH92]). While the general verification problem is more difficult for a dense model of time ([AH90]), it has been shown that bounded-invariance and bounded-response properties of timed transition systems can be established over the real numbers as time domain using the proof methods presented in this paper ([HMP92b]).

Acknowledgment. We thank Rajeev Alur for many helpful discussions.

References

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

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

[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.

[AH92] R. Alur and T.A. Henzinger. Logics and models of real time: a survey. In J.W. de Bakker, K. Huizing, W.-P. de Roever, and G. Rozenberg, editors, *Real Time: Theory in Practice*, Lecture Notes in Computer Science 600, pages 74–106. Springer-Verlag, 1992.

[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.

[BH81] A. Bernstein and P.K. Harter, Jr. Proving real-time properties of programs with temporal logic. In *Proceedings of the Eighth Annual Symposium on Operating System Principles*, pages 1–11. ACM Press, 1981.

[EMSS89] E.A. Emerson, A.K. Mok, A.P. Sistla, and J. Srinivasan. Quantitative temporal reasoning. Presented at the First Annual Workshop on Computer-aided Verification, Grenoble, France, 1989.

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

[Hen91] T.A. Henzinger. *The Temporal Specification and Verification of Real-time Systems.* PhD thesis, Stanford University, 1991.

[Hen92] T.A. Henzinger. Sooner is safer than later. *Information Processing Letters*, 43:135–141, 1992.

[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.

[HMP92a] T.A. Henzinger, Z. Manna, and A. Pnueli. Timed transition systems. In J.W. de Bakker, K. Huizing, W.-P. de Roever, and G. Rozenberg, editors, *Real Time: Theory in Practice*, Lecture Notes in Computer Science 600, pages 226–251. Springer-Verlag, 1992.

[HMP92b] T.A. Henzinger, Z. Manna, and A. Pnueli. What good are digital clocks? In W. Kuich, editor, *ICALP 92: Automata, Languages, and Programming*, Lecture Notes in Computer Science 623, pages 545–558. Springer-Verlag, 1992.

[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.

[KdR85] R. Koymans and W.-P. de Roever. Examples of a real-time temporal specification. In B.D. Denvir, W.T. Harwood, M.I. Jackson, and M.J. Wray, editors, *The Analysis of Concurrent Systems*, Lecture Notes in Computer Science 207, pages 231–252. Springer-Verlag, 1985.

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

[Koy90] R. Koymans. Specifying real-time properties with metric temporal logic. *Real-time Systems*, 2(4):255–299, 1990.

[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.

[KVdR83] R. Koymans, J. Vytopil, and W.-P. de Roever. Real-time programming and asynchronous message passing. In *Proceedings of the Second Annual Symposium on Principles of Distributed Computing*, pages 187–197. ACM Press, 1983.

[LA90] N.A. Lynch and H. Attiya. Using mappings to prove timing properties. In *Proceedings of the Ninth Annual Symposium on Principles of Distributed Computing*, pages 265–280. ACM Press, 1990.

[LA92] L. Lamport and M. Abadi. An old-fashioned recipe for real time. In J.W. de Bakker, K. Huizing, W.-P. de Roever, and G. Rozenberg, editors, *Real Time: Theory in Practice*, Lecture Notes in Computer Science 600, pages 1–27. Springer-Verlag, 1992.

[MMP92] O. Maler, Z. Manna, and A. Pnueli. From timed to hybrid systems. In J.W. de Bakker, K. Huizing, W.-P. de Roever, and G. Rozenberg, editors, *Real Time: Theory in Practice*, Lecture Notes in Computer Science 600, pages 447–484. Springer-Verlag, 1992.

[MMT91] M. Merritt, F. Modugno, and M.R. Tuttle. Time-constrained automata. In J.C.M. Baeten and J.F. Groote, editors, *CONCUR 91: Theories of Concurrency*, Lecture Notes in Computer Science 527, pages 408–423. Springer-Verlag, 1991.

[MP83] Z. Manna and A. Pnueli. Proving precedence properties: the temporal way. In J. Diaz, editor, *ICALP 83: Automata, Languages, and Programming*, Lecture Notes in Computer Science 154, pages 491–512. Springer-Verlag, 1983.

[MP84] Z. Manna and A. Pnueli. Adequate proof principles for invariance and liveness properties of concurrent programs. *Science of Computer Programming*, 4(3):257–289, 1984.

[MP89] Z. Manna and A. Pnueli. Completing the temporal picture. In G. Ausiello, M. Dezani-Ciancaglini, and S. Ronchi Della Rocca, editors, *ICALP 89: Automata, Languages, and Programming*, Lecture Notes in Computer Science 372, pages 534–558. Springer-Verlag, 1989.

[MP92] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer-Verlag, 1992.

[OL82] S. Owicki and L. Lamport. Proving liveness properties of concurrent programs. *ACM Transactions on Programming Languages and Systems*, 4(3):455–495, 1982.

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

[PdR82] A. Pnueli and W.-P. de Roever. Rendez-vous with Ada: a proof-theoretical view. In *Proceedings of the SIGPLAN AdaTEC Conference on Ada*, pages 129–137. ACM Press, 1982.

[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.

[Pnu86] A. Pnueli. Applications of temporal logic to the specification and verification of reactive systems: a survey of current trends. In J.W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *Current Trends in Concurrency*, Lecture Notes in Computer Science 224, pages 510–584. Springer-Verlag, 1986.

[Ron84] D. Ron. Temporal verification of communication protocols. Master's thesis, The Weizmann Institute of Science, Rehovot, Israel, 1984.

[SPE84] D.E. Shasha, A. Pnueli, and W. Ewald. Temporal verification of carrier-sense local area network protocols. In *Proceedings of the 11th Annual Symposium on Principles of Programming Languages*, pages 54–65. ACM Press, 1984.