Real-time System = Discrete System + Clock Variables*

Rajeev Alur
AT&T Bell Laboratories
600 Mountain Avenue
Murray Hill, NJ 07974
alur@research.att.com

Thomas A. Henzinger†
Computer Science Department
Cornell University
Ithaca, NY 14853
tah@cs.cornell.edu

How can we take a programming language off the shelf and upgrade it into a real-time programming language? Programs such as device drivers and plant controllers must explicitly refer and react to time. For this purpose, a variety of language constructs—including delays, timeouts, and watchdogs—has been put forward. We advocate an alternative answer, namely, to designate certain program variables as clock variables. The value of a clock variable changes as time advances. Timing constraints can be expressed, then, by conditions on clock values. A single new language construct—the guarded wait statement—suffices to enforce the timely progress of a program.

Our presentation proceeds in two steps. First we extend untimed systems (Section 1) with clock variables (Section 2); then we introduce the guarded wait statement (Section 3). The usage of clock variables and the guarded wait statement is illustrated with real-time applications such as round-robin (timeout-driven) scheduling, priority (interrupt-driven) scheduling, and embedded process control (Section 4). Indeed, clock variables generalize naturally to variables that measure environment parameters other than time (Section 5). In keeping with an expository style, all references are clustered in bibliographic remarks at the end of each section. We conclude by pointing to selected literature on formal methods and support tools for our approach to real-time programming (Section 6).

*A preliminary version of this paper appeared in the Proceedings of the First AMAST Workshop on Real-time Systems, Iowa City, IA, 1993.

†Partially supported by the National Science Foundation under grant CCR-9200794 and by the United States Air Force Office of Scientific Research under contract F49620-93-1-0056.

1 Untimed Systems

1.1 States

At any time, the state of a system is determined by the values of a finite set of variables. For example, the state of a chess game is determined by the board positions of all pieces and a bit that indicates whether white or black gets to move next; the state of a circuit is determined by the boolean values of all wires and flip-flops; the state of a while program is determined by the value of the program counter and the values of all program variables (the state of a C program, by contrast, is determined by the values of all memory locations that contain the run-time environment of the program).

We use V as the finite set of *variables*. We assume that all variables in V are typed and use suggestive names like m and n for nonnegative integer variables and x and y for real-valued variables. A *state* is a function that maps every variable in V to a value of the corresponding type.

A state predicate is a proposition that is either true or false for each state. For example, the proposition "The black king is attacked by the white queen" is either true or false for each state of a chess game; the proposition "All output wires are low" is either true or false for each state of a circuit; the proposition "$m = n + 2$" is either true or false for each state of a while program.

Formally, a *state predicate* is a formula of predicate logic whose free variables are taken from V. The truth value of the state predicate ϕ for the state σ is obtained by interpreting each free variable v in ϕ as $\sigma(v)$. For example, the state predicate $m = n + 2$ is true for a state σ iff $\sigma(m) = \sigma(n) + 2$. Every state predicate ϕ defines, then, the set of states, denoted by $[\![\phi]\!]$, for which ϕ is true.

1.2 Actions

The state of a system changes over time. We refer to the state changes of a system as actions. An *action* is a pair (σ, σ') of states that consists of a *source state* σ and a *target state* σ'. Intuitively, if a system is in the source state σ, then the action (σ, σ') takes the system into the target state σ'. We say that an action is *enabled* in its source state and *disabled* in all other states. Two actions (σ_1, σ_1') and (σ_2, σ_2') are *consecutive* if the second action is enabled in the target state of the first action—*i.e.*, if $\sigma_1' = \sigma_2$. The action (σ, σ') is a *null action* if $\sigma = \sigma'$.

Null actions arise in two situations. First, every state provides a system description that is necessarily abstract—*i.e.*, given at a certain level of detail. Consequently, any action that modifies only detail that is not included in a state appears as a null action. Recall, for instance, that we represent the state of a while program by values for the program counter and the program variables. So the action "Load the value of m into the accumulator," which, at a level of greater detail, may be part of the execution of a while program, appears as a null action, because it changes neither the value of the program counter nor the value of any program variable. Second, every system is embedded in an environment. Any environment action that does not modify the system state appears also as a null action.

An action predicate is a proposition that is either true or false for each action. For example, the proposition "A black piece is captured" is either true or false for each move of a chess game; the proposition "Some output wire goes from low to high" is either true or false for each clock cycle of a synchronous circuit; the proposition "The value of m is incremented" is either true or false for each execution step of a while program.

When describing an action, we use variables like $v \in V$ to refer to the value of v in the source state of the action, and primed variables like $v' \notin V$ to refer to the value of v in the target state of the action. The action predicate "The value of m is incremented," for instance, will be written as "$m' = m + 1$"; that is, the new value of m is one greater than the old value of m.

Formally, let V' be the set of primed variables whose unprimed versions occur in V. An *action predicate* is a formula of predicate logic whose free variables are taken from V and V'. The truth value of the action predicate ψ for the action (σ, σ') is obtained by interpreting each free unprimed variable v in ψ as $\sigma(v)$ and each free primed variable v' in ψ as $\sigma'(v)$. For example, the action predicate $m' = m + 1$ is true for an action (σ, σ') iff $\sigma'(m) = \sigma(m) + 1$. Every action predicate ψ defines, then, the set of actions, denoted by $[\![\psi]\!]$, for which ψ is true.

1.3 Behaviors

We restrict ourselves to systems that are *nondeterministic*, *discrete*, and *closed*. While the actions of a probabilistic system occur with certain likelihoods, each action of a nondeterministic system is, in every state, either possible (enabled) or impossible (disabled). While a continuous system—such as the solar system—may perform infinitely many actions in a finite interval of time, the behavior of a discrete system results from a countable sequence of actions. While the behavior of an open system may depend on the environment of the system, every environment action of a closed system appears as a null action. We assume that the environment never terminates; the actions of a system that terminates after a finite number of actions are, therefore, followed by infinitely many null actions.

A *behavior* of a system, then, is a countably infinite sequence of states. Alternatively, a behavior $\bar{\sigma} = \sigma_0, \sigma_1, \sigma_2, \sigma_3, \ldots$ can be viewed as the countably infinite sequence of actions (σ_i, σ_{i+1}), integer $i \geq 0$, with the property that any two neighboring actions are consecutive. For example, a legal sequence of moves, possibly interspersed with null actions, constitutes a behavior of the game of chess; a sequence of consecutive execution steps, possibly interspersed with null actions, constitutes a behavior of a given while program.

We use state and action predicates to describe behaviors. A state predicate restricts the first state of a behavior: the state predicate ϕ is *initially true* for the behavior $\bar{\sigma}$ if ϕ is true for the first state of $\bar{\sigma}$. An action predicate restricts all actions of a behavior except for environment actions: the action predicate ψ is *invariantly true* for the behavior $\bar{\sigma}$ if ψ is true for all actions of $\bar{\sigma}$ that are not null actions. Consider, for instance, the behavior $\bar{\sigma}$ whose odd actions, starting from 0, increment the value of m and whose even actions are null actions:

$$\bar{\sigma}_0(m) \; = \; 0, 1, 1, 2, 2, 3, 3, \ldots$$

The state predicate $m = 0$ is initially true for $\bar{\sigma}$ and the action predicate $m' = m + 1$ is invariantly true for $\bar{\sigma}$.

1.4 Systems

A system defines a set of possible behaviors. For example, the game of chess defines the initial board configuration and the set of all legal move sequences; a while program defines a set of execution-step sequences, one sequence for each combination of input values.

Formally, a *system* $\mathcal{S} = (\phi_0, \psi_\triangleright)$ is a pair that consists of a state predicate ϕ_0—the *initial condition* of $\mathcal{S}$—and an action predicate $\psi_\triangleright$—the *transition condition* of $\mathcal{S}$. The behavior $\bar{\sigma}$ is a behavior of the system $\mathcal{S}$ if (1) the initial condition ϕ_0 is initially true for $\bar{\sigma}$ and (2) the transition condition $\psi_\triangleright$ is invariantly true for $\bar{\sigma}$. Every system $\mathcal{S}$ defines, then, the set of its behaviors, which is denoted by $[\![\mathcal{S}]\!]$.

Consider, for instance, the system $\mathcal{S}_0$ that, starting from 0, increments the value of m arbitrarily often. The system $\mathcal{S}_0$ has the initial condition $m = 0$ and the transition condition $m' = m + 1$. Here are some behaviors of $\mathcal{S}_0$:

$$\begin{aligned}
\bar{\sigma}_1(m) &= 0, 1, 2, 3, 4, 5, 6, \ldots \\
\bar{\sigma}_2(m) &= 0, 1, 1, 2, 2, 3, 3, 4, \ldots \\
\bar{\sigma}_3(m) &= 0, 0, 0, 1, 2, 2, 2, 3, 4, \ldots \\
\bar{\sigma}_4(m) &= 0, 1, 2, 2, 2, 2, 2, 2, 2, 2, \ldots
\end{aligned}$$

Not every set of behaviors can be defined by a system. First, a system cannot prevent environment actions: if, between any two actions, we add a null action to a behavior of the system $\mathcal{S}$, then we obtain another behavior of $\mathcal{S}$ (this property of systems is called *stutter closure*). Second, the future evolution of a system is determined completely by the current state of the system and is independent of the past history of the system: if two behaviors $\bar{\sigma}_1$ and $\bar{\sigma}_2$ of the system $\mathcal{S}$ share a state σ, then by composing the past of σ according to $\bar{\sigma}_1$ and the future of σ according to $\bar{\sigma}_2$ we obtain another behavior of $\mathcal{S}$ (*fusion closure*). Third, the operation of a system is characterized completely by local conditions on individual states and actions and there are no global conditions on the behaviors of a system: if all finite prefixes of a behavior $\bar{\sigma}$ can be extended

$$
\begin{array}{l}
\textbf{program } \mathit{NdUpDown}: \\
\quad \textbf{initially } m := 0;\ n = 1; \\
\quad \textbf{loop} \quad \begin{array}{lll} \mathit{true} & \rightarrow & m := 1 \end{array} \\
\qquad \textbf{or} \quad \begin{array}{lll} m = 0 & \rightarrow & n := n + 1 \end{array} \\
\qquad \textbf{or} \quad \begin{array}{lll} m = 1 \wedge n > 0 & \rightarrow & n := n - 1 \end{array} \\
\qquad \textbf{end}.
\end{array}
$$

Figure 1: Guarded-command program

to behaviors of the system $\mathcal{S}$, then $\bar{\sigma}$ itself is a behavior of $\mathcal{S}$ (*limit closure*).[1]

The three closure properties of systems have two important ramifications on the executability of systems. First, the behaviors of a system can be generated action by action: start with a state for which the initial condition is true and repeatedly choose either a null action or an enabled action for which the transition condition is true. Second, a system need not progress: when generating a behavior action by action, from any point on we may choose null actions only.

1.5 Programs

We use a guarded-command programming language to define systems. A *guarded-command program* $\mathcal{P}$ consists of a set of assignments, which defines the initial condition of $\mathcal{P}$, followed by a set of guarded commands, which defines the transition condition of $\mathcal{P}$. The program $\mathcal{P}$ is executed in a stepwise fashion: first, starting from any state, execute the inital assignments to obtain an initial state of $\mathcal{P}$; then continue to select, nondeterministically, either a null action or a guarded command whose guard is true. We write $[\![\mathcal{P}]\!]$ for the set of execution sequences of the guarded-command program $\mathcal{P}$.

Consider, for instance, the guarded-command program *NdUpDown* of Figure 1. The program *NdUpDown* starts from an initial state σ with $\sigma(m) = 0$ and $\sigma(n) = 1$. The value of n is incremented arbitrarily often before the program may change the value of m to 1. Thereafter, the value of n may be decremented repeatedly as long as it remains nonnegative.

The guarded-command program *NdUpDown* defines a system whose behaviors are the execution sequences of *NdUpDown*. The initial condition of *NdUpDown* is given by the state predicate ϕ_0 and the transition condition of

[1] Conversely, if we admit infinitary formulas as state and action predicates, then every set of behaviors that is closed under stuttering, fusion, and limits is definable by a system.

$NdUpDown$ is given by the action predicate $\psi_{\triangleright}$:

$$
\begin{aligned}
\phi_0 &= (m = 0 \wedge n = 1) \\
\psi_{\triangleright} &= (m' = 1)_n \vee \\
&\quad\; (m = 0 \wedge n' = n + 1)_m \vee \\
&\quad\; (m = 1 \wedge n > 0 \wedge n' = n - 1)_m
\end{aligned}
$$

The expression $(\psi)_{\overline{v}}$, for an action predicate ψ and a vector $\overline{v}$ of variables, is an abbreviation for the action predicate $\psi \wedge \overline{v}' = \overline{v}$; that is, the variables listed as a subscript remain unchanged. Each guarded command of the program $NdUpDown$ contributes a disjunct—called an *action schema*—to the transition condition. We leave it to the reader to translate an arbitrary guarded-command program $\mathcal{P}$ into an initial condition and a transition condition that generate the set $[\![\mathcal{P}]\!]$ of behaviors.

Bibliographic remarks. Our system model is fairly standard, provided one accepts the interleaving view of concurrent activities (see, for example, the TLA approach of Lamport [Lam91] and the transition system approach of Manna and Pnueli [MP92]). As a word of caution, we point out that our actions are pairs of states rather than, as in TLA, sets of pairs of states. Our choice leads to a pleasant parallelism between state predicates and action predicates: a state predicate defines a set of states; an action predicate defines a set of actions. Stutter closure has been advocated by Lamport [Lam83]; limit closure (safety) was formally defined by Alpern, Demers, and Schneider [ADS86]; the relationship between systems and closure properties was studied by Emerson [Eme83]; the executability (machine closure) of systems, by Apt, Francez, and Katz [AFK88]. Guarded commands were introduced by Dijkstra [Dij75]; our guarded-command language is inspired by the UNITY language of Chandy and Misra [CM88].

2 Timed Systems 1: Safety

2.1 Clocks and delays

The state of a timed system is determined by two kinds of variables—*system variables* and *clock variables*. The value of each system variable is changed explicitly by the transition condition of the system; the value of each clock variable increases implicitly as time advances. Formally, we partition the finite set V of variables into system variables and clock variables. The clock variables—or *clocks*—range over the nonnegative real numbers. We reserve the letters x, y, and z for clocks. Given a state σ and a nonnegative real number δ, we write $\sigma + \delta$ for the state that maps all system variables m to $\sigma(m)$ and all clocks x to $\sigma(x) + \delta$.

The behavior of a timed system results from two kinds of actions—*system actions* and *time actions*. System actions update the values of the system variables and reset some of the clocks to 0; time actions do not modify the value of any system variables and uniformly advance the values of all clocks. It is perhaps best to think of time actions as environment actions: the environment can advance time and, in doing so, increases the values of the clocks by the same amount; the system, on the other hand, has only one way of changing the value of a clock, namely, to restart it. In other words, timed systems are open systems whose environment actions appear as both time actions and null actions.

Formally, the action (σ, σ') is a system action if for all clock variables x, either $\sigma'(x) = \sigma(x)$ or $\sigma'(x) = 0$; the action (σ, σ') is a time action—or *delay*—if there is a nonnegative real δ—the *duration* of the delay—such that $\sigma' = \sigma + \delta$. System actions have duration 0. Every null action is, by definition, both a system action and a delay of duration 0.

We henceforth write (σ, δ) for a delay with source state σ and duration δ. A state σ' is *passed by* the delay (σ, δ) if $\sigma' = \sigma + \epsilon$ for some nonnegative real $\epsilon < \delta$. In particular, no state is passed by a null action, and while the source state σ is passed by a delay (σ, δ) with positive duration $\delta > 0$, the target state $\sigma + \delta$ is not passed by the delay (σ, δ).

A *timed behavior* of a system, then, is a countably infinite sequence of system actions and delays such that any two neighboring actions are consecutive. Given a timed behavior $\bar{\sigma}$, let δ_i be the duration of the i-th action of $\bar{\sigma}$. The timed behavior $\bar{\sigma}$ *diverges* if the infinite sum $\sum_{i \geq 0} \delta_i$ of durations diverges. A state σ *occurs* in the timed behavior $\bar{\sigma}$ if either σ is the source state of a system action of $\bar{\sigma}$, or σ is passed by a delay of $\bar{\sigma}$.

2.2 Clock constraints

State sets and action sets of timed systems are, as before, defined by formulas of predicate logic with free variables from V and V'. We restrict, however, the way in which clocks may appear in such formulas to rule out timed systems that within a finite interval of time make infinitely many decisions on how to proceed. For example, we do not wish to consider system actions that are enabled iff the value of a certain clock is rational.

Let (σ, δ) be a delay, let ϕ be a state predicate, and let ψ be an action predicate. The characteristic function of ϕ maps each nonnegative real $\epsilon < \delta$ to 1 if ϕ is true for $\sigma + \epsilon$, and otherwise to 0; the characteristic function of ψ maps ϵ to 1 iff ψ is enabled in $\sigma + \epsilon$. A state or action predicate *varies finitely* over the delay (σ, δ) if its characteristic function has finitely many discontinuities in the interval $(0, \delta)$. Abstractly, we restrict ourselves to state predicates and action predicates that vary finitely over all delays.

Concretely, we offer a syntax for the timed systems that base their decisions on comparisons of clock values with each other and with integer constants: a *clock constraint* is a formula of the form $x \leq d$, $c \leq y$, or $x + c \leq y + d$, where x and y are clocks, and c and d are nonnegative integers. Moreover, every system action leaves the value of a clock either unchanged or reset to 0: a *reset condition* is a formula of the form $x' = x$ or $x' = 0$, for a clock x.

A *timed state predicate*, then, is a formula of predicate logic whose free variables are taken from V such that all clocks occur in clock constraints only; a *timed action predicate* is a formula of predicate logic whose free variables are taken from V and V' such that all clocks and primed clocks occur in clock constraints and reset conditions only (in particular, all primed clocks of a timed action predicate occur in reset conditions only). While, for example, the formula $y \leq x < y+3 \wedge z = x+1$ is a timed state predicate, the formulas $x+y = 3$ and $2x \leq y$ are not. It is not difficult to check that every timed state predicate and every timed action predicate varies finitely over all delays.[2]

The timed action predicate ψ is *invariantly true* for the timed behavior $\bar{\sigma}$ if ψ is true for all actions of $\bar{\sigma}$ that are not delays. Consider, for instance, the timed behavior $\bar{\sigma}$ whose even actions increment the value of the system variable m and restart the clock x, and whose odd actions are delays of duration 2:

$$\bar{\sigma}(m, x) \;=\; (0,0),(0,2),(1,0),(1,2),(2,0),(2,2),(3,0),(3,2),(4,0),\ldots$$

The timed action predicate $x = 2 \wedge m' = m+1 \wedge x' = 0$ are invariantly true for $\bar{\sigma}$.

2.3 Clock-constrained systems

A *clock-constrained system* $\mathcal{S} = (\phi_0, \psi_\triangleright)$ is a pair that consists of a timed state predicate ϕ_0—the *initial condition* of $\mathcal{S}$—and a timed action predicate $\psi_\triangleright$—the *transition condition* of $\mathcal{S}$. The timed behavior $\bar{\sigma}$ is a behavior of the clock-constrained system $\mathcal{S}$ if (1) the initial condition of $\mathcal{S}$ is initially true for $\bar{\sigma}$ and (2) the transition condition of $\mathcal{S}$ is invariantly true for $\bar{\sigma}$. Every clock-constrained system $\mathcal{S}$ defines, then, the set of its divergent behaviors, which is denoted by $[\![\mathcal{S}]\!]$.

Consider, for instance, the clock-constrained system $\mathcal{S}_1$ that, starting from 0, increments the value of m arbitrarily often such that the time difference between consecutive increment operations is at least 1 time unit and at most 2 time units. The system $\mathcal{S}_1$ has the initial condition $m = 0 \wedge x = 0$ and the transition condition $1 \leq x \leq 2 \wedge m' = m+1 \wedge x' = 0$. Here are some divergent behaviors of $\mathcal{S}_1$:

$$\begin{aligned}
\bar{\sigma}_1(m,x) &= (0,0),(0,1),(1,0),(1,1),(2,0),(2,1),(3,0),\ldots \\
\bar{\sigma}_2(m,x) &= (0,0),(0,2),(1,0),(1,2),(2,0),(2,2),(3,0),\ldots \\
\bar{\sigma}_3(m,x) &= (0,0),(0,0.5),(0,1.2),(1,0),(1,0),(1,1.5),(2,0),\ldots \\
\bar{\sigma}_4(m,x) &= (0,0),(0,1),(0,1.5),(0,2),(0,19),\ldots
\end{aligned}$$

[2] This is the case also for timed state and action predicates that are infinitary formulas, because the set V of variables is finite. Conversely, there is an infinitary formula with addition of clocks that does not vary finitely over all delays ($x = 1 \vee 2x = 1 \vee 3x = 1 \vee \ldots$). Decidability considerations, too, suggest the restriction of clock constraints to successor and comparison operations on clocks.

Not every set of divergent timed behaviors can be defined by a clock-constrained system. First, the options of a clock-constrained system change only finitely often within a finite interval of time: the transition condition $\psi_{\triangleright}$ of $\mathcal{S}$ varies finitely over all delays of behaviors of $\mathcal{S}$ (*finitely varying branching*). Second, if we split a delay of a behavior of $\mathcal{S}$ into two consecutive delays, or if we merge two consecutive delays of a behavior of $\mathcal{S}$ into a single delay, then we obtain another behavior of $\mathcal{S}$ (closure under *timed stuttering*; as a delay may be split into itself followed by a null action, closure under timed stuttering implies stutter closure). Third, the behaviors of $\mathcal{S}$ are closed under *fusion* and, fourth, under *divergent limits*: if all finite prefixes of a divergent timed behavior $\bar{\sigma}$ can be extended to behaviors of $\mathcal{S}$, then $\bar{\sigma}$ itself is a behavior of $\mathcal{S}$ (as the infinite sequence of null actions is not a behavior of $\mathcal{S}$, no clock-constrained system is closed under limits). Fifth, a clock-constrained system cannot prevent delays: if a finite prefix of a behavior of $\mathcal{S}$ is extended by a delay, then we obtain again a prefix of a behavior of $\mathcal{S}$ (closure under *waiting*).

Consequently, the divergent behaviors of a clock-constrained system can be generated action by action: (1) start with a state for which the initial condition is true and repeatedly choose either a delay or an enabled action for which the transition condition is true; (2) to generate divergent timed behaviors, make sure to choose appropriate delays—*i.e.*, choose a sequence of delays whose sum diverges. Furthermore, a clock-constrained system need not progress: when generating a divergent behavior action by action, from any point on we may choose delays only.

2.4 Clock-constrained programs

A *clock-constrained program* $\mathcal{P}$ consists of a set of assignments followed by a set of guarded commands such that (1) all assignments to clocks are of the form $x := 0$ and (2) all guards are timed state predicates. The program $\mathcal{P}$ is executed in a stepwise fashion: first, starting from any state, execute the inital assignments to obtain an initial state of $\mathcal{P}$; then continue to select, nondeterministically, either a delay or a guarded command whose guard is true. We write $[\![\mathcal{P}]\!]$ for the set of divergent execution sequences of the clock-constrained program $\mathcal{P}$.

Consider, for instance, the clock-constrained program *CcUpDown* of Figure 2. The program *CcUpDown* contains a system variable, n, and two clocks, x and y. The program starts from an initial state in which n has the value 1 and both x and y have the value 0. Within the first 10 time units, the value of n is incremented arbitrarily often as long as the time difference between consecutive increment operations is at least 1 time unit and at most 5 time units. Beginning at time 12, the value of n may be decremented repeatedly, under the same timing constraints, as long as n remains nonnegative.

The clock-constrained program *CcUpDown* defines a clock-constrained system whose behaviors are the divergent execution sequences of *CcUpDown*. The initial condition of *CcUpDown* is given by the timed state predicate ϕ_0 and the

```
program CcUpDown:
   declare x, y: clock;
   initially n := 1; x := 0; y := 0;
   loop x ≤ 10 ∧ 1 ≤ y ≤ 5              →  n := n + 1; y := 0
     or x ≥ 12 ∧ 1 ≤ y ≤ 5 ∧ n > 0      →  n := n − 1; y := 0
     end.
```

Figure 2: Clock-constrained program

transition condition of *CcUpDown* is given by the timed action predicate $\psi_{\triangleright}$:

$$
\begin{array}{rcl}
\phi_0 & = & (n = 1 \ \wedge\ x = 0 \ \wedge\ y = 0) \\
\psi_{\triangleright} & = & (x \leq 10 \ \wedge\ 1 \leq y \leq 5 \ \wedge\ n' = n + 1 \ \wedge\ y' = 0)_x \ \vee \\
 & & (x \geq 12 \ \wedge\ 1 \leq y \leq 5 \ \wedge\ n > 0 \ \wedge\ n' = n - 1 \ \wedge\ y' = 0)_x
\end{array}
$$

Each guarded command of the program *CcUpDown* contributes a disjunct—called a *system action schema*—to the transition condition. We leave it to the reader to translate an arbitrary clock-constrained program $\mathcal{P}$ into an initial condition and a transition condition that generate the set $[\![\mathcal{P}]\!]$ of timed behaviors.

Bibliographic remarks. Clock variables—as we use them—were first introduced in temporal logic [AH89] (originally as so-called "freeze" quantifiers) and in finite automata [AD90]. Both papers realized the importance of the choice of clock constraints that are admitted by a system; this choice has immediate ramifications on semantic issues and verification problems. The dichotomy of system actions versus time actions, which had been advocated repeatedly by various proponents of the interleaving view of concurrency, was perhaps standardized by the real-time system model of timed transition systems [HMP91]. In particular, the timed transition system model has led to the study of closure under timed stuttering, divergent limits (divergence safety), and finitely varying branching [Hen91, Hen92, HNSY92].

3 Timed Systems 2: Progress

3.1 Delay predicates

Neither untimed systems nor clock-constrained systems need to progress—*i.e.*, at any point a system may fail to perform another action while the environment keeps performing actions. To ensure the progress of an untimed system, one typically assumes fairness requirements on the system actions: a system action must not be enabled forever (or infinitely often) without being taken. To ensure the progress of a timed system, it will suffice to assume a fairness requirement on the time actions: time must advance beyond any bound. While we have seen that a restriction to divergent timed behaviors does not ensure progress *per se*, we will use the divergence of time to enforce system actions by constraining time actions. For this purpose, we introduce delay predicates. A delay predicate typically enforces a system action by limiting the amount by which the environment may advance time in a given state.

A delay predicate is a proposition that is either true or false for each delay. For example, the proposition "Throughout the delay, the clock x shows at most 5" is either true or false for a delay; we will write this proposition as "$x < 5$," using variables like x to refer to the value of x in all states that are passed by the delay. The delay predicate $x < 5$, then, prevents time from advancing past a state in which the value of the clock x is 5.

Formally, the syntax of *delay predicates* is identical to the syntax of timed state predicates. The delay predicate χ is true for a delay if χ, viewed as a timed state predicate, is true for all states that are passed by the delay. For example, the delay predicate $x < 5$ is true for a delay (σ, δ) iff $\sigma(x) + \delta \leq 5$ (since, by definition, the target state of a delay is not passed by the delay, the delay predicates $x < 5$ and $x \leq 5$ are true for the same delays). Every delay predicate χ defines, then, the set $[\![\chi]\!]$ of delays for which χ is true. In particular, the set $[\![\chi]\!]$ contains all null actions and is closed under both the splitting and the merging of delays; that is, a delay predicate χ is true for two consecutive delays (σ, δ_1) and $(\sigma + \delta_1, \delta_2)$ iff χ is true for the combined delay $(\sigma, \delta_1 + \delta_2)$.

While a timed action predicate restricts the system actions of a timed behavior, a delay predicate restricts the delays of a timed behavior. The delay predicate χ is *invariantly true* for the timed behavior $\bar{\sigma}$ if χ is true for all delays of $\bar{\sigma}$. Consider again the timed behavior $\bar{\sigma}$ whose even actions increment the value of the system variable m and restart the clock x, and whose odd actions are delays of duration 2:

$$\bar{\sigma}(m, x) \ = \ (0,0),(0,2),(1,0),(1,2),(2,0),(2,2),(3,0),(3,2),(4,0),\ldots$$

The delay predicate $x < 2$ is invariantly true for $\bar{\sigma}$; the delay predicate $x \neq 1$ is not.

3.2 Real-time systems

A *real-time system* $\mathcal{S} = (\phi_0, \psi_{\triangleright}, \chi_{\Box})$ is a triple that consists of a clock-constrained system $(\phi_0, \psi_{\triangleright})$ and a delay predicate $\chi_{\Box}$—the *environment condition* of $\mathcal{S}$. The timed behavior $\bar{\sigma}$ is a behavior of the real-time system $\mathcal{S}$ if (1) $\bar{\sigma}$ is a behavior of the clock-constrained system that underlies $\mathcal{S}$ and (2) the environment condition of $\mathcal{S}$ is invariantly true for $\bar{\sigma}$. Every real-time system $\mathcal{S}$ defines, then, the set of its divergent behaviors, which is denoted by $[\![\mathcal{S}]\!]$.

The transition condition of a real-time system asserts necessary conditions on system actions and the environment condition asserts sufficient conditions on system actions. For example, the following real-time system $\mathcal{S}_2 = (\phi_0, \psi_{\triangleright}, \chi_{\Box})$ changes the value of m from 0 to 1 at time 3 at the earliest and at time 5 at the latest:

$$\begin{array}{rcl} \phi_0 & = & (m = 0 \wedge x = 0) \\ \psi_{\triangleright} & = & (x \geq 3 \wedge m' = 1) \\ \chi_{\Box} & = & (m = 0 \wedge x < 5) \vee \\ & & (m = 1) \end{array}$$

While the requirement $x \geq 3$ of the transition condition ensures the lower time bound of 3—the system transition that changes the value of m may happen at or after time 3—the requirement $x < 5$ of the environment condition ensures the upper time bound 5—the transition must happen at or before time 5, because time cannot advance past 5 before the value of m has changed to 1.

Not every set of divergent timed behaviors can be defined by a real-time system. While all real-time systems exhibit finitely varying branching and are closed under timed stuttering, fusion, and divergent limits, they are not necessarily closed under waiting. This is because the environment condition can enforce the progress of a real-time system.

3.3 Real-time executability

There is a price to be paid, however, for progress: the divergent behaviors of a real-time system may no longer be generated action by action. By starting with a state for which the initial condition is true and repeatedly choosing either a delay that does not violate the environment condition or an enabled action for which the transition condition is true, we may be led into a situation from which time cannot diverge.

Consider, for example, the following variant $\mathcal{S}_Z$ of the real-time system $\mathcal{S}_2$:

$$\begin{array}{rcl} \phi_0 & = & (m = 0 \wedge x = 0) \\ \psi_{\triangleright} & = & (3 \leq x \leq 5 \wedge m' = 1) \\ \chi_{\Box} & = & (m = 0 \wedge x < 10) \vee \\ & & (m = 1) \end{array}$$

The real-time systems $\mathcal{S}_2$ and $\mathcal{S}_Z$ have the same divergent behaviors—formally, $[\![\mathcal{S}_2]\!] = [\![\mathcal{S}_Z]\!]$—namely, all divergent timed behaviors in which the value of m changes from 0 to 1 at time 3 at the earliest and at time 5 at the latest. Only the real-time system $\mathcal{S}_2$, however, can be executed in a stepwise fashion. Executing the real-time system $\mathcal{S}_Z$ we might start from the initial state by choosing a delay of duration, say, 8, which does not violate the environment condition. But there is no divergent continuation from the resulting state σ with $\sigma(m) = 0$ and $\sigma(x) = 8$: we have painted ourselves into a corner by having chosen an initial delay whose duration was greater than 5.

Real-time systems that do not exhibit this problem are called nonzeno: a real-timed system $\mathcal{S}$ is *nonzeno* if every finite prefix of a behavior of $\mathcal{S}$ is a finite prefix of a divergent behavior of $\mathcal{S}$. In particular, while the real-time system $\mathcal{S}_2$ is nonzeno, the real-time system $\mathcal{S}_Z$ is not. Nonzenoness, then, is a property of a real-time system (syntax) rather than a property of a set of timed behaviors (semantics). Indeed, every set of timed behaviors that can be defined by a real-time system can also be defined by a nonzeno real-time system, whose environment condition is suitably strengthened. For instance, the real-time system $\mathcal{S}_Z$ can be transformed into an equivalent nonzeno real-time system by strengthening the environment condition to $(x < 5) \vee (m = 1)$.

3.4 Real-time programs

A *guarded wait statement* is an instruction of the form $\chi \rightarrow$ **wait**, where the guard χ is a delay predicate. A guarded wait statement delays the program execution by an arbitrary amount of time, but at most until the guard, viewed as a timed state predicate, becomes false. The guarded wait statement $\chi \rightarrow$ **wait** results, then, in a delay that satisfies the delay predicate χ. In particular, the guarded wait statement *false* $\rightarrow$ **wait** results in a null action, and the guarded wait statement *true* $\rightarrow$ **wait** results in a delay of arbitrary duration. We generally prefer the guard $x < 5$ over the equivalent delay predicate $x \leq 5$, perhaps because we like to think of the guarded wait statement $x < 5 \rightarrow$ **wait** as delaying execution as long as the value of x is less than 5, and terminating as soon as the value of x becomes 5.

A *real-time program* $\mathcal{P}$ is a clock-constrained program that contains guarded wait statements in addition to guarded commands. The program $\mathcal{P}$ is executed in a stepwise fashion: first, starting from any state, execute the inital assignments to obtain an initial state of $\mathcal{P}$; then, continue to select, nondeterministically, either a guarded wait statement or a guarded command whose guard is true. Since it may not be possible to merge the consecutive execution of two different guarded wait statements into a single execution step, the resulting set $[\mathcal{P}]$ of divergent execution sequences is not necessarily closed under timed stuttering. The set $[\![\mathcal{P}]\!]$ of behaviors of the real-time program $\mathcal{P}$, therefore, is defined as the timed stutter closure of $[\mathcal{P}]$—*i.e.*, the smallest superset of $[\mathcal{P}]$ that is closed under timed stuttering.

Consider, for instance, the real-time program *RtUpDown1* of Figure 3. The program *RtUpDown1* behaves like the clock-constrained program *CcUpDown* except that the value of n must be incremented or decremented (depending on how much time has elapsed since the program was started) at least once every 5 time units, until n reaches the value 0.

The real-time program *RtUpDown1* defines a real-time system that has the same behaviors. The initial condition of *RtUpDown1* is given by the timed state predicate ϕ_0, the transition condition of *RtUpDown1* is given by the timed action predicate $\psi_{\triangleright}$, and the environment condition of *RtUpDown1* is given by

$$
\begin{array}{l}
\textbf{program } \textit{RtUpDown1}: \\
\quad \textbf{declare } x, y \text{: } \textbf{clock}; \\
\quad \textbf{initially } n := 1;\ x := 0;\ y := 0; \\
\quad \begin{array}{lllll}
\textbf{loop} & x \le 10 \wedge y \ge 1 & \rightarrow & n := n + 1;\ y := 0 \\
\quad \textbf{or} & n > 0 \wedge x \ge 12 \wedge y \ge 1 & \rightarrow & n := n - 1;\ y := 0 \\
\quad \textbf{or} & n > 0 \wedge y < 5 & \rightarrow & \textbf{wait} \\
\quad \textbf{or} & n = 0 & \rightarrow & \textbf{wait} \\
\quad \textbf{end}. & & &
\end{array}
\end{array}
$$

Figure 3: Real-time program

the delay predicate $\chi_{\Box}$:

$$
\begin{array}{lcl}
\phi_0 & = & (n = 1 \wedge x = 0 \wedge y = 0) \\
\psi_{\triangleright} & = & (x \le 10 \wedge y \ge 1 \wedge n' = n + 1 \wedge y' = 0)_x \vee \\
 & & (x \ge 12 \wedge y \ge 1 \wedge n > 0 \wedge n' = n - 1 \wedge y' = 0)_x \\
\chi_{\Box} & = & (n > 0 \wedge y < 5) \vee \\
 & & (n = 0)
\end{array}
$$

Each guarded command of the program *RtUpDown1* contributes disjunct (a system action schema) to the transition condition, and each guarded wait statement of *RtUpDown1* contributes a disjunct—called a *delay schema*—to the environment condition. We leave it to the reader to translate an arbitrary real-time program $\mathcal{P}$ into an initial condition, a transition condition, and an environment condition that define the set $[\![\mathcal{P}]\!]$ of timed behaviors. A clock-constrained program, in particular, is a real-time program with the guarded wait statement *true* $\rightarrow$ **wait**, which yields the environment condition *true*.

The execution strategy for real-time programs succeeds only if it cannot lead to a state from which there is no divergent continuation of actions. A real-time program $\mathcal{P}$, then, is *executable* if it defines a nonzeno real-time system. The real-time program *RtUpDown1*, for example, is not executable. To see this, observe that the stepwise execution of *RtUpDown1* may lead to a state σ with $\sigma(n) = 5$, $\sigma(x) = 11$, and $\sigma(y) = 5$, and there is no divergent sequence of actions continuing from σ. Figure 4 shows an executable real-time program that has the same behaviors as *RtUpDown1*. The environment condition of the executable program *RtUpDown2* is

$$\chi_{\Box}: \quad (x < 10 \wedge y < 5) \vee (x \ge y + 7 \wedge y < 5) \vee (n = 0).$$

The real-time program *RtUpDown2* terminates—*i.e.*, the value of n reaches 0 and execution continues with delays only—within at most 75 time units (to calculate this time bound on the termination of *RtUpDown2*, observe that the value of n can be at most 12 when $x = 11$).

```
program RtUpDown2 :
  declare x, y: clock;
  initially n := 1;  x := 0;  y := 0;
  loop  x ≤ 10 ∧ y ≥ 1                →  n := n + 1;  y := 0
    or  n > 0 ∧ x ≥ 12 ∧ y ≥ 1        →  n := n − 1;  y := 0
    or  x < 10 ∧ y < 5                →  wait
    or  x ≥ y + 7 ∧ y < 5             →  wait
    or  n = 0                         →  wait
    end.
```

Figure 4: Executable real-time program

Bibliographic remarks. Nonzenoness (timed executability) was defined independently by several researchers [AL92, Hen92]; Abadi and Lamport coined the phrase. The first proposal for environment conditions as a means of ensuring the progress of a timed system advocated the use of program invariants [HNSY92]; that paper also presents an algorithm for transforming a real-time system into an equivalent nonzeno system. The real-time program *RtUpDown1* is derived from an example, due to Pnueli [HMP91], that illustrates that an increase in the lower bound on the duration of a delay may lead to a decrease in the running time of a program (to see this, replace the lower bounds of 1 on the clock y by 2; then *RtUpDown2* will terminate within 50 time units).

4 Real-time Programming

4.1 Sequential real-time processes

Our choice of guarded commands as programming language has largely been instructionally motivated. In general, we may add clocks and guarded wait statements to any programming language. To illustrate this thesis, we now extend a language of while programs with clocks and guarded wait statements. As in synchronous programming languages, all instructions of a while program are executed immediately and instantaneously; all delays are indicated explicitly by guarded wait statements.

It is convenient to introduce abbreviations for several common applications of guarded wait statements. First, the instruction **wait** delays the program execution for an arbitrary amount of time; it stands for the guarded wait statement *true* $\rightarrow$ **wait**. Second, the instruction **wait watching** ϕ delays the program execution for an arbitrary amount of time, but at most until the timed state predicate ϕ becomes true; it stands for the guarded wait statement $\neg\phi \rightarrow$ **wait**. Third, the instruction **await** ϕ delays the program execu-

```
program RtUpDown3:
  declare x, y: clock;
  n := 1; x := 0;
  wait [1, 5];
  while x ≤ 10 do
    n := n + 1; y := 0;
    repeat
      if x < y + 7
        then wait watching x ≥ 10 ∨ y ≥ 5
        else wait watching y ≥ 5
        fi
      until y ≥ 1
    od;
  repeat wait watching y ≥ 5 until x ≥ 12;
  while n > 0 do
    n := n − 1;
    wait [1, 5]
    od.
```

Figure 5: Real-time while program

tion precisely until the timed state predicate ϕ becomes true; it stands for the program segment

$$\textbf{repeat wait watching } \phi \textbf{ until } \phi.$$

Fourth, the instruction **wait** $[c, d]$ delays the program execution at least c and at most d time units, for nonnegative integers c and d with $c \leq d$; it stands for the program segment

$$z := 0;\ \textbf{repeat wait watching } z \geq d \textbf{ until } z \geq c,$$

where z is a new clock. We write **wait** c short for **wait** $[c, c]$,

Now consider the real-time while program *RtUpDown3* of Figure 5. We present a formal semantics for real-time while programs such that the program *RtUpDown3* defines essentially the same real-time system as the guarded-command program *RtUpDown2* of Section 3.4.

We construct the real-time system that is defined by a while program $\mathcal{P}$ by encoding the control structure of $\mathcal{P}$ as a formula. For this purpose, we introduce a new system variable—the *program counter pc*—that ranges over the control locations of $\mathcal{P}$. If ℓ_0 is the initial control location, then the initial condition of $\mathcal{P}$ is $pc = \ell_0$. An execution step of $\mathcal{P}$ is a move between control locations and corresponds to the execution of a guarded command (typically, an assignment or a test) or the execution of a guarded wait statement. Each such execution step contributes a system action schema to the transition condition of $\mathcal{P}$; each guarded wait statement contributes, in addition, a delay schema to the environment condition of $\mathcal{P}$.

Consider, for example, the initial program segment

$$\ell_0 : n := 1; \quad \ell_1 : x := 0$$

of the while program *RtUpDown3*; it contributes the system action schema

$$(pc = \ell_0 \ \wedge\ pc' = \ell_1 \ \wedge\ n' = 1)_{x,y}$$

to the transition condition of *RtUpDown3*. The program segment

$$\ell_2 : \textbf{wait watching}\ x \geq 10 \vee y \geq 5 \quad \ell_3 : \textbf{until}\ y \geq 1$$

contributes to the transition condition of *RtUpDown3* the system action schema

$$(pc = \ell_2 \ \wedge\ pc' = \ell_3)_{n,x,y}$$

and contributes to the environment condition of *RtUpDown3* the delay schema

$$(pc = \ell_2 \ \wedge\ x < 10 \ \wedge\ y < 5).$$

We leave it to the reader to define the transition condition of *RtUpDown3* as the disjunction of all system action schemata and the environment condition of *RtUpDown3* as the disjunction of all delay schemata. If we adjust the resulting timed behaviors by (1) projecting out the program counter and (2) merging certain consecutive states (for example, the consecutive assignments $n := n + 1;\ y := 0$ of the while program *RtUpDown3* constitute a single action of the guarded-command program *RtUpDown2*), then we obtain precisely the real-time system that is defined by the guarded-command program *RtUpDown2*. It follows, in particular, that the real-time while program *RtUpDown3* is executable.

4.2 Concurrent real-time processes

So far, we have discussed sequential processes and sequential real-time processes. We now illustrate that concurrent processes, too, accommodate naturally the introduction of timing constraints through clocks and guarded wait statements: while every concurrent while program defines a system, every concurrent real-time program defines a real-time system.

The actions of concurrent untimed processes are interleaved. Consider, for instance, the concurrent while program *ParUp* of Figure 6. The program *ParUp* consists of two independent processes. The "left" process repeatedly increases the value of the variable m; the "right" process repeatedly increments the value

```
program ParUp:
  initially m := 0; n := 0;
  cobegin
    loop ℓ0: m := m + 1;  ℓ1: m := m + 2 end
    ||
    loop k0: n := n + 1 end
    coend.
```

Figure 6: Concurrent while program

of n. A behavior of *ParUp* interleaves, without regard to any fairness requirements, null actions with actions of either process: an action of the left process may be followed by any number of actions of the right process, and vice versa. Here are some behaviors of *ParUp*:

$$\begin{array}{rcl}
\bar{\sigma}_1(m,n) & = & (0,0),(1,0),(1,1),(3,1),(3,2),(4,2),(4,3),\ldots \\
\bar{\sigma}_2(m,n) & = & (0,0),(1,0),(3,0),(4,0),(6,0),(6,1),(7,1),\ldots \\
\bar{\sigma}_3(m,n) & = & (0,0),(0,1),(0,2),(0,3),(1,3),(1,4),(1,5),\ldots \\
\bar{\sigma}_4(m,n) & = & (0,0),(0,1),(0,1),(1,1),(3,1),(4,1),(4,2),\ldots
\end{array}$$

The behavior $\bar{\sigma}_1$, for example, alternates the actions of both processes.

To define the semantics of the program *ParUp* formally, we use two program counters, *left* and *right*—one for each process (since the right process has a single control location, there is actually no need for the program counter *right* in this example). Each execution step of either process contributes an action schema to the transition condition of *ParUp*. The concurrent program *ParUp* defines, then, the system with the initial condition ϕ_0 and the transition condition $\psi_\triangleright$:

$$\begin{array}{rcl}
\phi_0 & = & (\mathit{left} = \ell_0 \wedge \mathit{right} = k_0 \wedge m = 0 \wedge n = 0) \\
\psi_\triangleright & = & (\mathit{left} = \ell_0 \wedge \mathit{left}' = \ell_1 \wedge m' = m+1)_{\mathit{right},n} \;\vee \\
 & & (\mathit{left} = \ell_1 \wedge \mathit{left}' = \ell_0 \wedge m' = m+2)_{\mathit{right},n} \;\vee \\
 & & (n' = n+1)_{\mathit{left},\mathit{right},m}
\end{array}$$

The interleaving of the actions of concurrent untimed processes reflects an underlying assumption that the execution speeds of the processes are unknown, independent, possibly very different and even varying. Timing constraints relativize the assumption of speed independence by ruling out certain interleavings of the system actions of concurrent timed processes. This is because, while the system actions of concurrent timed processes are interleaved, their delays overlap.

Consider, for instance, the concurrent real-time program *RtParUp* of Figure 7. The real-time program *RtParUp* is obtained from the untimed program

```
program RtParUp:
  initially m := 0; n := 0;
  cobegin
    loop wait 3; m := m + 1; wait 1; m := m + 2 end
    ||
    loop wait 7; n := n + 1 end
    coend.
```

Figure 7: Concurrent real-time program

ParUp by adding delays between consecutive assignments. During the execution of *RtParUp*, a delay of the left process will overlap with one or more delays of the right process, and vice versa. Also, a system action of the left process may interrupt a delay of the right process, and vice versa.

Figure 8 shows the program *RtParUp* without abbreviations and annotated with control locations. Here are two behaviors of *RtParUp*:

$$\begin{aligned}\bar{\sigma}_5(m,n,x,y) &= (0,0,0,0),(0,0,3,3),(1,0,3,3),(1,0,0,3),(1,0,1,4),\\ &\quad (3,0,1,4),(3,0,0,4),(3,0,3,7),(4,0,3,7),(4,1,3,7),\ldots\\ \bar{\sigma}_6(m,n,x,y) &= (0,0,0,0),(0,0,3,3),(1,0,3,3),(1,0,0,3),(1,0,1,4),\\ &\quad (3,0,1,4),(3,0,0,4),(3,0,3,7),(3,1,3,7),(4,1,3,7),\ldots\end{aligned}$$

In particular, the timing constraints of *RtParUp* rule out the strict alternation of system actions from the left process and the right process; that is, no behavior of the real-time program *RtParUp* corresponds to the behavior $\bar{\sigma}_1$ of the untimed program *ParUp*.

When defining the semantics of *RtParUp*, each execution step of either process contributes a system action schema to the transition condition of *RtParUp*. In addition, each pair of guarded wait statements—one from the left process and the other one from the right process—contributes a delay schema to the environment condition of *RtParUp*. The concurrent real-time program *RtParUp* defines, then, the real-time system of Figure 9. This real-time system is the cartesian product of the two real-time systems that are defined by the component processes.

4.3 Embedded real-time processes

At last, we illustrate our approach to real-time programming by designing typical application programs such as schedulers and embedded controllers. A scheduler is a real-time process that runs concurrently to the tasks that are being scheduled; an embedded controller runs concurrently to continuous environment processes.

program $RtParUp$:
  declare x, y: **clock**;
  initially $m := 0$; $n := 0$;
  cobegin
    loop
      $\ell_0 : x := 0$; **repeat** ℓ_1: **wait watching** $x \geq 3$ ℓ_2: **until** $x \geq 3$;
      $\ell_3 : m := m + 1$;
      $\ell_4 : x := 0$; **repeat** ℓ_5: **wait watching** $x \geq 1$ ℓ_6: **until** $x \geq 1$;
      $\ell_7 : m := m + 2$
      end
    ||
    loop
      $k_0 : y := 0$; **repeat** k_1: **wait watching** $y \geq 7$ k_2: **until** $y \geq 7$;
      $k_3 : n := n + 1$
      end
    coend.

Figure 8: Annotated concurrent real-time program

The simple round-robin scheduler *RoundRobin* of Figure 10 schedules a task for exactly 10 time units, and then moves on to the next task. The more sophisticated round-robin scheduler *TtRoundRobin* of Figure 11 schedules a task either for 10 time units or until the task terminates, whatever happens first. The variable i indicates which task is currently running; the bit $done[i]$ indicates if task i has terminated.

The priority scheduler *Priority* of Figure 12 schedules a task until it terminates or until an interrupt arrives from a task with higher priority, whatever happens first. The scheduler *Priority* is a real-time process even though it does not use a clock: the guarded wait statement ensures that *Priority* waits until a high-priority task sets the bit *interrupt*. If we wish to guarantee that no interrupt is lost, we must assume that no two high-priority tasks arrive simultaneously at precisely the same instant in real-numbered time; alternatively, we could provide a separate bit $interrupt[i]$ for each task i.

The real-time process *GateController* of Figure 13 controls the gate at a railroad crossing. The gate is closed 5 time units after a train signals its approach by setting the bit *arrival*, and the gate is opened again as soon as the train signals its exit from the crossing by setting the bit *exit*. The more sophisticated controller *MtGateController* of Figure 14 accommodates multiple trains, perhaps on parallel tracks, in the crossing. This controller uses the variable n to count the number of trains that are currently in the crossing under the assumption that no two trains arrive simultaneously. The gate is opened only when all

$$
\begin{array}{rcl}
\phi_0 & = & (\mathit{left} = \ell_0 \wedge \mathit{right} = k_0 \wedge m = 0 \wedge n = 0) \\
\\
\psi_{\triangleright} & = & (\mathit{left} = \ell_0 \wedge \mathit{left}' = \ell_1 \wedge x' = 0)_{right,m,n,y} \vee \\
& & (\mathit{left} = \ell_1 \wedge \mathit{left}' = \ell_2)_{right,m,n,x,y} \vee \\
& & (\mathit{left} = \ell_2 \wedge x < 3 \wedge \mathit{left}' = \ell_1)_{right,m,n,x,y} \vee \\
& & (\mathit{left} = \ell_2 \wedge x \geq 3 \wedge \mathit{left}' = \ell_3)_{right,m,n,x,y} \vee \\
& & (\mathit{left} = \ell_3 \wedge \mathit{left}' = \ell_4 \wedge m' = m + 1)_{right,n,x,y} \vee \\
& & (\mathit{left} = \ell_4 \wedge \mathit{left}' = \ell_5 \wedge x' = 0)_{right,m,n,y} \vee \\
& & (\mathit{left} = \ell_5 \wedge \mathit{left}' = \ell_6)_{right,m,n,x,y} \vee \\
& & (\mathit{left} = \ell_6 \wedge x < 1 \wedge \mathit{left}' = \ell_5)_{right,m,n,x,y} \vee \\
& & (\mathit{left} = \ell_6 \wedge x \geq 1 \wedge \mathit{left}' = \ell_7)_{right,m,n,x,y} \vee \\
& & (\mathit{left} = \ell_7 \wedge \mathit{left}' = \ell_0 \wedge m' = m + 2)_{right,n,x,y} \vee \\
& & (\mathit{right} = k_0 \wedge \mathit{right}' = k_1 \wedge y' = 0)_{left,m,n,x} \vee \\
& & (\mathit{right} = k_1 \wedge \mathit{right}' = k_2)_{left,m,n,x,y} \vee \\
& & (\mathit{right} = k_2 \wedge y < 7 \wedge \mathit{right}' = k_1)_{left,m,n,x,y} \vee \\
& & (\mathit{right} = k_2 \wedge y \geq 7 \wedge \mathit{right}' = k_3)_{left,m,n,x,y} \vee \\
& & (\mathit{right} = k_3 \wedge \mathit{right}' = k_0 \wedge n' = n + 1)_{left,m,x,y} \\
\\
\chi_{\Box} & = & (\mathit{left} = \ell_1 \wedge \mathit{right} = k_1 \wedge x' < 3 \wedge y' < 7) \vee \\
& & (\mathit{left} = \ell_5 \wedge \mathit{right} = k_1 \wedge x' < 1 \wedge y' < 7)
\end{array}
$$

Figure 9: Real-time system $(\phi_0, \psi_{\triangleright}, \chi_{\Box})$ defined by the program *RtParUp*

trains have left the crossing—*i.e.*, when the value of n is 0.

Bibliographic remarks. The synchronicity assumption—that program instructions are instantaneous and time is introduced through the environment—is due to Berry, whose ESTEREL language inspired also the **wait watching** macro (for an introduction to synchronous programming languages, see the monograph by Halbwachs [Hal93]). The use of control-location labels to define the system actions of a while program is standard practice of the interleaving school for concurrency [Lam91, MP92]. Schedulers and railroad gate controllers pervade the real-time literature as examples.

5 Timed Systems 3: Continuous Environment

5.1 Drifting clocks

So far, we have assumed that all clocks are mathematically precise. In distributed systems, however, one is often confronted with physical clocks, which are necessarily imprecise. Typically all that is known about a process clock x is that the drift of x is bounded; that is, in any time interval of length 1 the clock

program *RoundRobin* :
  initially $i := FirstTask$;
  loop
    $ResumeTask(i)$;
    wait 10;
    $SuspendTask(i)$;
    $i := NextTask(i)$
    end.

Figure 10: Round-robin scheduler

program *TtRoundRobin* :
  declare x : **clock**;
  initially $i := FirstTask$;
  loop
    $ResumeTask(i)$;
    $x := 0$; **await** $x \geq 10 \vee done[i]$;
    if $\neg done[i]$ **then** $SuspendTask(i)$ **fi**;
    $i := NextTask(i)$
    end.

Figure 11: Round-robin scheduler for terminating tasks

may deviate from the actual time by $\pm\epsilon$, for a real number $\epsilon > 0$. We indicate that the drift of a clock x is bounded by ϵ with the clock declaration **declare** x: **clock drift** ϵ (the condition **drift** 0 is suppressed as usual). For example, Figure 15 shows the railroad gate controller of the previous section (Figure 13), assuming that the drift of the controller clock x is bounded by $\epsilon = 0.1$. This version of the controller may close the gate as early as 4.5 time units and as late as 5.5 time units after the train signals its approach.

The behavior of drifting clocks during delays is best described by differential constraints. For this purpose, let $\dot{v}$ be the first derivative of the variable v. Then, if the drift of the clock x is bounded by 0.1, all states that are passed by delays satisfy the differential constraint $0.9 \leq \dot{x} \leq 1.1$. The resulting system is called "hybrid," because it consists of both discrete system variables that are updated by system actions and continuous environment variables—namely, drifting clocks—that are updated by the environment during delays.

```
program Priority:
  initially interrupt := false;
  loop
    i := MaxPriorityTask;
    ResumeTask(i);
    await interrupt ∨ done[i];
    if interrupt then interrupt := false fi;
    if ¬done[i] then SuspendTask(i) fi
    end.
```

Figure 12: Priority scheduler

```
program GateController:
  initially arrival := false; exit := false;
  loop
    await arrival; arrival := false;
    wait 5;
    CloseGate;
    await exit; exit := false;
    OpenGate
    end.
```

Figure 13: Railroad gate controller

5.2 Hybrid systems

We now illustrate that clock variables can be generalized to model not only time, or drifting physical clocks, but arbitrary continuous environment parameters such as temperature or position.

The behavior of a hybrid system results from system actions and hybrid delays. A *hybrid delay* (f, δ) consists of a nonnegative real number δ—the duration of the delay—and a differentiable function f from the closed interval $[0, \delta]$ of the real line to states (a function f from the reals to states is differentiable if for every variable v, the projection $f(v)$ is differentiable). The state $f(0)$ is the source state of the hybrid delay (f, δ); the state $f(\delta)$ is its target state. A state σ' is passed by the hybrid delay (f, δ) if $\sigma' = f(\epsilon)$ for some nonnegative real $\epsilon < \delta$. A *hybrid behavior* is a countably infinite sequence of system actions and hybrid delays such that any two neighboring sequence elements are consecutive.

Hybrid delays are described by hybrid delay predicates. Let $\dot{V}$ be the set of

```
program MtGateController:
  declare x: clock;
  initially arrival := false; exit := false;
  loop
    await arrival; arrival := false; n := 1;
    x := 0;
    repeat
      wait watching arrival ∨ exit ∨ x ≥ 5;
      if arrival then arrival := false; n := n + 1 fi;
      if exit then exit := false; n := n − 1 fi
      until x ≥ 5;
    CloseGate;
    repeat
      wait watching arrival ∨ exit;
      if arrival then arrival := false; n := n + 1 fi;
      if exit then exit := false; n := n − 1 fi
      until n = 0;
    OpenGate
    end.
```

Figure 14: Railroad gate controller for multiple trains

dotted variables whose undotted versions occur in V. A *hybrid delay predicate* is a formula of predicate logic whose free variables are taken from V and $\dot{V}$. The hybrid delay predicate χ is true for a hybrid delay if χ is true for all states that are passed by the delay. The hybrid delay predicate χ is invariantly true for a hybrid behavior $\bar{\sigma}$ if χ is true for all hybrid delays of $\bar{\sigma}$.

A *hybrid system*, then, is a real-time system whose environment condition is a hybrid delay predicate. The hybrid behavior $\bar{\sigma}$ is a behavior of the hybrid system $\mathcal{S}$ if (1) the initial condition of $\mathcal{S}$ is initially true for $\bar{\sigma}$ and (2) the transition condition and the environment condition of $\mathcal{S}$ are invariantly true for $\bar{\sigma}$. Suppose that $\mathcal{S}$ is a hybrid system with the environment condition $\chi_\Box$. A variable m is a (discrete) system variable of $\mathcal{S}$ if the environment condition $\chi_\Box$ implies $\dot{m} = 0$; that is, the variable m is not modified during delays. All other variables in V are (continuous) environment variables. In particular, a variable x is a clock if the environment condition $\chi_\Box$ implies $\dot{x} = 1$; the variable y is a clock with bounded drift $\epsilon > 0$ if the environment condition $\chi_\Box$ implies $1 - \epsilon \leq \dot{y} \leq 1 + \epsilon$; the variable z is a stopwatch if the environment condition $\chi_\Box$ implies $(\dot{z} = 0) \vee (\dot{z} = 1)$.

Environment variables, however, are not limited to measuring time. Consider the example of a thermostat that regulates the room temperature. Suppose that

```
program DcGateController:
   declare x: clock drift 0.1;
   initially arrival := false; exit := false;
   loop
      await arrival; arrival := false;
      x := 0; await x = 5;
      CloseGate;
      await exit; exit := false;
      OpenGate
      end.
```

Figure 15: Railroad gate controller with a drifting clock

the initial room temperature is 68 degrees and the heat is turned on. When the temperature reaches 70 degrees, the heat is turned off, and the temperature θ decreases over time t according to the exponential function $\theta(t) = 70e^{-c_0 t}$, where c_0 is a constant determined by the room; when the temperature falls to 65 degrees, the heat is turned on, and the temperature θ increases according to the exponential function $\theta(t) = 65e^{-c_0 t} + c_1(1 - e^{-c_0 t})$, where c_1 is a constant determined by the power of the heater. The resulting hybrid system is defined by the initial condition ϕ_0, the transition condition $\psi_{\triangleright}$, and the environment condition $\chi_{\square}$:

$$\begin{aligned}
\phi_0 &= (heat = on \wedge \theta = 68) \\
\psi_{\triangleright} &= (heat = off \wedge \theta \leq 65 \wedge heat' = on)_\theta \vee \\
&\quad\; (heat = on \wedge \theta \geq 70 \wedge heat' = off)_\theta \\
\chi_{\square} &= (heat = on \wedge \dot{\theta} = c_0(c_1 - \theta) \wedge \theta < 70) \vee \\
&\quad\; (heat = off \wedge \dot{\theta} = -c_0\theta \wedge \theta > 65)
\end{aligned}$$

Here, the status of the heater is modeled by the discrete (boolean-valued) system variable *heat*; the room temperature, by the continuous (real-valued) environment variable θ.

5.3 Hybrid programs

We define hybrid systems by while programs such as the *Train* process of Figure 16. Clock variables (possibly drifting) are declared, as before, as **clock**; all other environment variables, as **continuous**. In particular, the environment variable d represents the distance, in meters, of the train from a railroad crossing. The dotted variable $\dot{d}$ represents, then, the speed of the train in meters per time unit. All undeclared variables, such as the bits *arrival* and *exit*, are discrete by default.

$$
\begin{array}{l}
\textbf{program } \mathit{Train}: \\
\quad \textbf{declare } d\colon \textbf{continuous}; \\
\quad \textbf{loop} \\
\quad\quad \ell_0: \textbf{wait}; \\
\quad\quad \ell_1: d := 5000; \\
\quad\quad \textbf{repeat} \\
\quad\quad\quad \ell_2: \textbf{wait}[-55 \leq \dot{d} \leq -45] \textbf{ watching } d \leq 1000 \\
\quad\quad\quad \ell_3: \textbf{until } d \leq 1000; \\
\quad\quad \ell_4: \mathit{arrival} := \mathit{true}; \\
\quad\quad \textbf{repeat} \\
\quad\quad\quad \ell_5: \textbf{wait}[-50 \leq \dot{d} \leq -35] \textbf{ watching } d \leq -100 \\
\quad\quad\quad \ell_6: \textbf{until } d \leq -100; \\
\quad\quad \ell_7: \mathit{exit} := \mathit{true} \\
\quad\quad \textbf{end}.
\end{array}
$$

Figure 16: Hybrid program

Given a hybrid delay predicate χ and a vector $\overline{v}$ of variables, we write $(\chi)_{\overline{v}}$ for the hybrid delay predicate $\chi \wedge \dot{\overline{v}} = 0$; that is, the variables listed as a subscript remain unchanged. Given a hybrid program, let $\overline{m}$ be the vector of discrete variables, let $\overline{x}$ be the vector of clocks, and let $\overline{\epsilon}$ be the corresponding vector of drift bounds. The instruction **wait**$[\chi]$ **watching** ϕ maintains the hybrid delay predicate χ while delaying the program for an arbitrary amount of time, but at most until the timed state predicate ϕ becomes true; it stands for the guarded wait statement

$$(1 - \overline{\epsilon} \leq \dot{\overline{x}} \leq 1 + \overline{\epsilon} \wedge \chi \wedge \neg\phi)_{\overline{m}} \rightarrow \textbf{wait}.$$

In our example, the speed of the train stays between 45 and 55 meters per time unit until the train is 1,000 meters from the crossing. At this point, the train signals its approach, by setting the bit *arrival*, and slows down to 35 to 50 meters per time unit. After the train is 100 meters past the crossing, it leaves and may return along a cyclic track at any time, provided the length of the loop is at least 5,000 meters. The hybrid program *Train* defines, therefore, the hybrid system of Figure 17.

At last, we may wish to combine, using the parallel-composition operator $||$, the *Train* process with the gate controller *DcGateController* into a concurrent hybrid program. The corresponding hybrid system is defined, as usual, as the cartesian product of the component systems: the initial condition is obtained by taking the conjunction of the initial conditions of the components; the transition condition, by taking the disjunction of the transition conditions of the components; and the hybrid delay schemata of the environment condition result from

$$
\begin{aligned}
\phi_0 &= (pc = \ell_0) \\[1ex]
\psi_{\triangleright} &= (pc = \ell_0 \wedge pc' = \ell_1)_{arrival, exit, d} \vee \\
&\quad (pc = \ell_1 \wedge pc' = \ell_2 \wedge d' = 5000)_{arrival, exit} \vee \\
&\quad (pc = \ell_2 \wedge pc' = \ell_3)_{arrival, exit, d} \vee \\
&\quad (pc = \ell_3 \wedge d > 1000 \wedge pc' = \ell_2)_{arrival, exit, d} \vee \\
&\quad (pc = \ell_3 \wedge d \leq 1000 \wedge pc' = \ell_4)_{arrival, exit, d} \vee \\
&\quad (pc = \ell_4 \wedge pc' = \ell_5 \wedge arrival' = true)_{exit, d} \vee \\
&\quad (pc = \ell_5 \wedge pc' = \ell_6)_{arrival, exit, d} \vee \\
&\quad (pc = \ell_6 \wedge d > -100 \wedge pc' = \ell_5)_{arrival, exit, d} \vee \\
&\quad (pc = \ell_6 \wedge d \leq -100 \wedge pc' = \ell_7)_{arrival, exit, d} \vee \\
&\quad (pc = \ell_7 \wedge pc' = \ell_0 \wedge exit' = true)_{arrival, d} \\[1ex]
\chi_{\Box} &= (pc = \ell_0)_{arrival, exit} \vee \\
&\quad (pc = \ell_2 \wedge -55 \leq \dot{d} \leq -45 \wedge d \leq 1000)_{arrival, exit} \vee \\
&\quad (pc = \ell_5 \wedge -50 \leq \dot{d} \leq -35 \wedge d \leq -100)_{arrival, exit}
\end{aligned}
$$

Figure 17: Hybrid system $(\phi_0, \psi_{\triangleright}, \chi_{\Box})$ defined by the program *Train*

taking all pairwise conjunctions of the component delay schemata. We leave the details to the ambitious reader.

Bibliographic remarks. The bounded-drift assumption underlies the clock synchronization problem for distributed systems (see, for example, the survey by Schneider [Sch87]). A recent workshop proceedings provides an excellent current overview of computer-science models for hybrid systems [Ner93]. The dichotomy of discrete system actions versus continuous environment activities (hybrid delays) was introduced by Manna, Maler, and Pnueli [MMP92]. The use of environment conditions (invariants) to model hybrid systems in general [ACHH93], and drifting clocks in particular [AHH93], was first worked out in an automata-theoretic framework. The thermostat example is due to Nicollin, Sifakis, and Yovine [NSY92].

6 Analysis of Timed Systems

Now that we have defined the formal semantics of real-time systems, we wish to develop both methods and tools for proving the correctness of real-time programs. The verification problem has been addressed by several papers in recent years; here we attempt only to direct the reader to some of this literature.

6.1 Specification

First, we need a specification language for stating the correctness requirements of a given real-time program. Process algebras, temporal logics, and automata are popular specification languages for untimed systems. Hence a variety of real-time extensions of these languages has been proposed and investigated, including the process algebras TCSP [RR88], ATP [NRSV90], and TCCS [Wan90]; the temporal logics RTL [JM86], TPTL [AH89], TCTL [ACD90], MTL [AH90, Koy90], and RTTL [Ost90]; as well as timed automata [AD90] and timed I/O automata [MMT91, LV92a] (see [NS91] for a survey of real-time process algebras; [AH92] for a survey of logic-based and automata-based real-time languages; and [dBHdRR92] for a variety of applications).

Particularly compatible with our approach are the temporal logics TPTL (linear time) and TCTL (branching time), as well as timed automata: TPTL and TCTL are obtained by adding clock variables, reset quantifiers, and clock constraints to temporal logic; timed automata are obtained by adding clocks, reset conditions, and clock constraints to finite automata. If the acceptance condition of a timed automaton is replaced by an environment condition, the resulting timed safety automaton [HNSY92] corresponds precisely to our notion of real-time system.

To illustrate the use of clocks for specification, let us specify a few properties of the real-time while program *RtUpDown3* (Figure 5) using the temporal logic TPTL. First, consider the invariance formula

$$\varphi_1: \quad \Box\,(n \leq 12).$$

A (timed) behavior $\bar{\sigma}$ satisfies the formula φ_1 iff the (timed) state predicate $n \leq 12$ is true for all states that occur in $\bar{\sigma}$; a (real-time) program $\mathcal{P}$ satisfies φ_1 iff every behavior of $\mathcal{P}$ satisfies φ_1.

Second, we wish to specify the timing behavior of the program *RtUpDown3*. Let x be a clock. The time-bounded response formula

$$\varphi_2: \quad x := 0.\,\Diamond\,(x \leq 75\ \wedge\ n = 0)$$

asserts that the timed state predicate $n = 0$ becomes true, in all behaviors of a real-time program, within at most 75 time units from the initial state. Dually, the time-bounded invariance formula

$$\varphi_3: \quad x := 0.\,\Box\,(x < 15\ \rightarrow\ n \neq 0)$$

asserts that the value of n does not become 0, in all behaviors of a real-time program, before time 15. The reader may check that, indeed, the program *RtUpDown3* satisfies all three TPTL-formulas φ_1, φ_2, and φ_3. We conclude that the real-time program *RtUpDown3* takes at least 15 time units and at most 75 time units to terminate.

6.2 Verification

Second, we need to verify that a given real-time program satisfies its specification. For this purpose, most methods for proving the correctness of untimed programs have been extended to timed programs, including assertional methods [Hoo92, SBM92], proof systems for temporal logics [HMP91, AL92], and simulation methods [LA90, LV92b]. None of these approaches, however, use clocks, and we would like to see a proof system for real-time programs with clocks.

On the other hand, clocks have been central to the finite-state approach to verification. A (real-time) system is *finite-state* if all (system) variables range over finite domains. The verification problem for untimed finite-state systems is decidable, and a variety of software tools have been developed and successfully applied—particularly to the verification of protocols and circuits. Real-time systems are more complex: even in the case of real-time finite-state systems, the clocks range over the infinite domain of the nonnegative real numbers. It is possible, however, to construct a finite quotient of the infinite state space that is adequate for solving the reachability problem for real-time finite-state systems. This observation led to verification algorithms for finite-state real-time systems [ACD90, AD90, AFH91].

The efficiency and scope of the finite-state approach to the verification of real-time systems has been improved along several directions. First, algorithms for constructing the minimal adequate quotient of an infinite state space have been developed [ACH+92, LY92]. Second, to cope with the high computational complexity of analyzing clock constraints, an incremental approach has been implemented [AIKY92]: initially all clock constraints are ignored; then the clock constraints are added one by one, as needed to prove a given specification. For example, the real-time program *RtUpDown3* continues to satisfy the properties φ_1 and φ_2 without the clock constraint $x \geq 12$.

A third technique for improving both the efficiency and the scope of verification algorithms is based on symbolic computation [HNSY92]. Suppose, for instance, we wish to prove that a certain property is an invariant of a real-time system. For this purpose, we need to compute the set of states that appear along all behaviors of the system. The timed state predicate that characterizes this set of reachable states is the least fixpoint of an equation on timed state predicates. The fixpoint can be computed by symbolic execution of the system, which is guaranteed to terminate if the system is finite-state. The efficiency of the fixpoint computation depends on the representation of state predicates; in the case of untimed systems, binary decision diagrams have turned out to provide a cost effective data structure for representing state predicates [McM93].

A hybrid system is *piecewise linear* if all continuous variables are bounded by piecewise-linear functions. In particular, clocks and stopwatches are piecewise linear; drifting clocks are bounded by piecewise-linear functions; all other variables can be approximated by piecewise-linear envelopes. The symbolic verification method applies also to the class of piecewise-linear hybrid systems, albeit it may not terminate [ACHH93]. Prototype implementations are under way both at Cornell [AHH93] and in Grenoble [NOSY93].

Acknowledgments. The authors thank Limor Fix, David Karr, Peter Kopke, and Fred Schneider for helpful comments.

References

[ACD90] R. Alur, C. Courcoubetis, and D.L. Dill. Model checking for real-time systems. In *Proceedings of the Fifth Annual Symposium on Logic in Computer Science*, pages 414–425. IEEE Computer Society Press, 1990.

[ACH+92] R. Alur, C. Courcoubetis, N. Halbwachs, D.L. Dill, and H. Wong-Toi. Minimization of timed transition systems. In S.A. Smolka, editor, *CONCUR 92: Theories of Concurrency*, Lecture Notes in Computer Science 630, pages 340–354. Springer-Verlag, 1992.

[ACHH93] R. Alur, C. Courcoubetis, T.A. Henzinger, and P.-H. Ho. Hybrid automata: an algorithmic approach to the specification and verification of hybrid systems. In A. Nerode, editor, *Theory of Hybrid Systems*, Lecture Notes in Computer Science. Springer-Verlag, 1993.

[AD90] R. Alur and D.L. Dill. Automata for modeling real-time systems. In M.S. Paterson, editor, *ICALP 90: Automata, Languages, and Programming*, Lecture Notes in Computer Science 443, pages 322–335. Springer-Verlag, 1990.

[ADS86] B. Alpern, A.J. Demers, and F.B. Schneider. Safety without stuttering. *Information Processing Letters*, 23(4):177–180, 1986.

[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.

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

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

[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.

[AHH93] R. Alur, T.A. Henzinger, and P.-H. Ho. Automatic symbolic verification of embedded systems. In *Proceedings of the 14th Annual Real-time Systems Symposium*, pages 2–11. IEEE Computer Society Press, 1993.

[AIKY92] R. Alur, A. Itai, R.P. Kurshan, and M. Yannakakis. Timing verification by successive approximation. In G. Bochmann, editor, *CAV 92: Computer-aided Verification*, Lecture Notes in Computer Science. Springer-Verlag, 1992.

[AL92] M. Abadi and L. Lamport. 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.

[CM88] K.M. Chandy and J. Misra. *Parallel Program Design: A Foundation.* Addison-Wesley Publishing Company, 1988.

[dBHdRR92] 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. Springer-Verlag, 1992.

[Dij75] E.W. Dijkstra. Guarded commands, nondeterminacy, and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, 1975.

[Eme83] E.A. Emerson. Alternative semantics for temporal logics. *Theoretical Computer Science*, 26(1):121–130, 1983.

[Hal93] N. Halbwachs. *Synchronous Programming of Reactive Systems.* Kluwer Academic Publishers, 1993.

[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(3):135–141, 1992.

[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.

[HNSY92] T.A. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. Symbolic model checking for real-time systems. In *Proceedings of the Seventh Annual Symposium on Logic in Computer Science*, pages 394–406. IEEE Computer Society Press, 1992.

[Hoo92] J. Hooman. Compositional verification of real-time systems using extended Hoare trples. 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 252–290. Springer-Verlag, 1992.

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

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

[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.

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

[Lam91] L. Lamport. The temporal logic of actions. Technical Report 79, DEC Systems Research Center, Palo Alto, California, 1991.

[LV92a] N. Lynch and F. Vaandrager. Action transducers and timed automata. In S.A. Smolka, editor, *CONCUR 92: Theories of Concurrency*, Lecture Notes in Computer Science 630, pages 436–455. Springer-Verlag, 1992.

[LV92b] N. Lynch and F. Vaandrager. Forward and backward simulations for timing-based 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 397–446. Springer-Verlag, 1992.

[LY92] D. Lee and M. Yannakakis. Online minimization of transition systems. In *Proceedings of the 24th Annual Symposium on Theory of Computing*, pages 264–274. ACM Press, 1992.

[McM93] K.L. McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem.* Kluwer Academic Publishers, 1993.

[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.

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

[Ner93] A. Nerode, editor. *Theory of Hybrid Systems.* Lecture Notes in Computer Science. Springer-Verlag, 1993.

[NOSY93] X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. An approach to the description and analysis of hybrid systems. In A. Nerode, editor, *Theory of Hybrid Systems*, Lecture Notes in Computer Science. Springer-Verlag, 1993.

[NRSV90] X. Nicollin, J.-L. Richier, J. Sifakis, and J. Voiron. ATP: an algebra for timed processes. In M. Broy and C.B. Jones, editors, *Proceedings of the IFIP WG2.2/2.3 Working Conference on Programming Concepts and Methods*, pages 415–442. Elsevier Science Publishers (North-Holland), 1990.

[NS91] X. Nicollin and J. Sifakis. An overview and synthesis on timed process algebras. In K. Larsen and A. Skou, editors, *CAV 91: Computer-aided Verification*, Lecture Notes in Computer Science 575, pages 376–398. Springer-Verlag, 1991.

[NSY92] X. Nicollin, J. Sifakis, and S. Yovine. From ATP to timed graphs and 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 549–572. Springer-Verlag, 1992.

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

[RR88] G.M. Reed and A.W. Roscoe. A timed model for communicating sequential processes. *Theoretical Computer Science*, 58(1/2/3):249–261, 1988.

[SBM92] F.B. Schneider, B. Bloom, and K. Marzullo. Putting time into proof outlines. 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 618–639. Springer-Verlag, 1992.

[Sch87] F.B. Schneider. Understanding protocols for byzantine clock synchronization. Technical Report 87-859, Cornell University, 1987.

[Wan90] Y. Wang. Real-time behavior of asynchronous agents. In J.C.M. Baeten and J.W. Klop, editors, *CONCUR 90: Theories of Concurrency*, Lecture Notes in Computer Science 458, pages 502–520. Springer-Verlag, 1990.