REAL-TIME SYSTEM = DISCRETE SYSTEM + CLOCK VARIABLES

by

Rajeev Alur and Thomas A. Henzinger

Memorandum No. UCB/ERL M97/78

20 October 1997

REAL-TIME SYSTEM = DISCRETE SYSTEM + CLOCK VARIABLES

by

Rajeev Alur and Thomas A. Henzinger

Memorandum No. UCB/ERL M97/78

20 October 1997

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Real-time System = Discrete System + Clock Variables[*][†]

Rajeev Alur[‡] Thomas A. Henzinger[§]

Programs such as device drivers and embedded controllers must explicitly refer and react to time. For this purpose, a variety of language constructs—including delays, timeouts, and watchdogs—have been put forward. We advocate an alternative proposal, 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. We illustrate the use of clock variables and guarded wait statements with real-time applications such as round-robin (timeout-driven) and priority (interrupt-driven) scheduling. Clock variables generalize naturally to variables that measure environment parameters other than time. This observation leads to a language for hybrid (mixed digital-analog) applications such as embedded process control.

This paper introduces, gently but rigorously, the clock approach to real-time programming. We present with mathematical precision, assuming no prerequisites other than familiarity with logical and programming notations, the concepts that are necessary for understanding, writing, and executing clock programs. In keeping with an expository style, all references are clustered in bibliographic remarks at the end of each section. The first appendix presents proof rules for verifying temporal properties of clock programs. The second appendix points to selected literature on formal methods and tools for programming with clocks. In particular, the timed automaton, which is a finite-state machine equipped with clocks, has become a standard paradigm for real-time model checking; it underlies the tools HyTech, Kronos, and Uppaal, discussed elsewhere in this volume.

[*]A preliminary version of this paper appeared in the proceedings of the First AMAST Workshop on Real Time, in *Theories and Experiences for Real-time System Development* (T. Rus and C. Rattray, eds.), AMAST Series in Computing 2, World Scientific, 1994, 1–29.

[†]This research was supported in part by the ONR YIP award N00014-95-1-0520, by the NSF CAREER award CCR-9501708, by the NSF grant CCR-9504469, by the AFOSR contract F49620-93-1-0056, by the ARO MURI grant DAAH-04-96-1-0341, by the ARPA grant NAG2-892, and by the SRC contract 95-DC-324.036.

[‡]CIS Department, University of Pennsylvania, and Bell Labs, Lucent Technologies.

[§]EECS Department, University of California at Berkeley.

```
program GateController:
  external train_present: bool;
  loop
    await train_present;
    wait 5;
    CloseGate;
    await ¬train_present;
    OpenGate
    end.
```

Figure 1: Railroad-gate controller

1 Introduction

Consider the program shown in Figure 1, which controls the gate at a railroad crossing. The program continuously watches the value of the external boolean signal *train_present*, which indicates whether a train is near the crossing. One may imagine two sensors on the railroad track, on either side of the crossing, which keep the signal *train_present* true exactly when there is a train between the two sensors. When the signal *train_present* becomes true—i.e., when a train is approaching the crossing—the program *GateController* waits 5 time units before issuing a command to close the gate. Thereafter, as soon as the signal *train_present* becomes false—i.e., when the train is leaving the crossing—a command to open the gate is issued.

The functionality of the program *GateController* is intuitively obvious, and its adequacy in a given environment can be judged by inspection: assuming that the time consumed by the issuing of commands to the gate is negligible, and assuming that the gate requires δ time units for closing, the program is safe if every train needs at least $5 + \delta$ time units to reach the crossing from the point when the signal *train_present* is triggered; otherwise, the gate may be open when a train is in the crossing. Real-life situations tend to be more complicated—often allowing for many different scenarios involving many concurrent activities—than this simple model of a railroad crossing, and the adequacy of embedded controllers usually cannot be asserted confidently by inspection or simulation. Therefore, for mission-critical and other sensitive applications, software tools are being developed for analyzing functional and timing properties of control programs. These tools view a program as denoting a mathematical object, and compute properties of the denoted object. In this paper, we define formally the mathematical objects that are denoted by real-time programs such as *GateController*, and we study some of their properties. Other papers in this volume discuss several tools that are based on the real-time semantics presented

```
program TwoTanks:
   local pipe: {1,2};
   external water_level_1, water_level_2: continuous;
   initially pipe := 1;
   loop
      await water_level_2 = 0;
      pipe := 2;
      await water_level_1 = 0;
      pipe := 1
      end.
```

Figure 2: Water-level controller

here.

Real-time programs are reactive, in that they react to certain external activities, such as signals generated by the environment (**await** *train_present*) or the passage of time (**wait** 5). We fleetingly mentioned the assumption that, in the real-time program *GateController*, the execution of the statements *CloseGate* and *OpenGate* consumes no real time. This, a so-called "synchrony assumption," allows the program to react to external activities immediately (synchronously). Synchrony assumptions are abstractions which are valid when the speed of the external activities is much slower than the speed of program execution. Synchrony assumptions are useful, because, by making all timing explicit (through statements such as **wait** 5), they greatly simplify the mathematical analysis of real-time behavior. Synchrony assumptions, however, also open the way for writing programs that cannot be executed at any speed.

To see this, consider the program shown in Figure 2, which controls the inflow into two water tanks. The program directs a pipe between the two tanks: initially, water is directed to the first tank, until the second tank becomes empty; then, water is redirected to the second tank, until the first tank becomes empty; etc. The position of the pipe is indicated by the boolean-valued variable *pipe*, and the water levels of the two tanks are indicated by the real-valued variables $water\_level_1$ and $water\_level_2$. The water levels are updated continuously, and externally to the program, by the environment. Our synchrony assumption is that the time consumed by the commands for redirecting the pipe is negligible. While the program *TwoTanks* and the corresponding synchrony assumption look innocuous, there are circumstances under which the program cannot be executed. For example, if the incoming pipe carries water at the rate of 3 liters per second, but each tank loses water at the rate of 2 liters per second, the program will instruct the pipe to switch back and forth between the two tanks at shorter and shorter time intervals. Clearly, a point must be reached when these instructions can no longer be followed. It is surprisingly subtle to identify the real-time programs that can be executed no matter how the environment behaves, and this gives us another reason for studying the underlying mathematical objects.

Rather than presenting the formal semantics of a particular programming language with particular real-time constructs, we develop a generic framework for formalizing timed behavior. For this purpose, we begin by reviewing discrete state-transition systems, which are the mathematical objects that underly untimed reactive programs. Then we add clock variables to discrete state-transition systems, and show how the resulting real-time state-transition systems can be used to give a formal meaning to programming constructs that refer to time, such as the **wait** and **await** statements of Figure 1. The framework of clock systems allows us to rigorously study and unambiguously discuss issues such as real-time executability, composition, and verification in an abstract, language-independent setting, which applies, for example, to the concrete languages used by the tools HyTech, Kronos, and Uppaal presented in this volume.

2 Untimed Systems

2.1 States

At any time, the abstract state of a system can be captured by the values of a finite set V of *variables.* The choice of variables, which determines the amount of detail that is captured about a system, depends on the task at hand. For example, for the motion analysis of the solar system, its abstract state can be captured by the masses, positions, and velocities of all bodies; for the strategic analysis of a chess game, its abstract state can be captured by the board positions of all pieces and a bit that indicates whether white or black is to move next; for the functional analysis of a circuit, its abstract state can be captured by the boolean values of all wires and latches; for the functional analysis of a while program, its abstract state can be captured by the position of the program counter and the values of all program variables.[1] 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. We describe states using state predicates. A state predicate is a proposition that is either true or false for each state. For example, the proposition "The black king is being attacked by the white queen" is either true or false for each state of a chess game; the proposition "All output wires are low"

[1] By contrast, for the physical analysis of a circuit, the voltages and lengths of all wires are relevant; for the functional analysis of a C program, the values of all registers and memory locations that contain the run-time environment of the program are relevant.

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 instance, the state predicate $m = n + 2$ is true for the state σ iff $\sigma(m) = \sigma(n) + 2$.

2.2 Transitions

The state of a system changes over time. The values of some variables—like the positions and velocities of the bodies in the solar system—change as continuous functions over time, and the values of other variables—like the positions of chess pieces and the values of program variables—change in discrete steps. We begin by restricting our attention to discrete systems, all of whose variables change their values in discrete steps. The state changes of a discrete system are called transitions. A *transition* 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 transition (σ, σ') takes the system into the target state σ'. We say that a transition is *enabled* in its source state and *disabled* in all other states. Two transitions (σ_1, σ_1') and (σ_2, σ_2') are *consecutive* if the second transition is enabled in the target state of the first transition—i.e., if $\sigma_1' = \sigma_2$.

The transition (σ, σ') is a *stutter transition* if $\sigma = \sigma'$. Stutter transitions arise if a system activity concerns only detail that is not captured by the variables of a system. For example, with our choice of variables for representing chess games, the activity "Turn the chess board clockwise by 90 degrees" appears as a stutter transition. We are interested in systems—such as embedded control systems—that operate within a real-world, real-time environment. However, before we discuss systems that interact with the environment, at first we restrict our attention to closed systems, whose variables capture no information about the environment. To a closed discrete system, every environment activity appears as a stutter transition.

We describe transitions using transition predicates. A transition predicate is a proposition that is either true or false for each transition. For example, the proposition "A black piece is being captured" is either true or false for each move of a chess game; the proposition "The value of m is being incremented" is either true or false for each execution step of a while program. When describing a transition, we use unprimed variables like $v \in V$ to refer to the value of v in the source state of the transition, and primed variables like $v' \notin V$ to refer to the value of v in the target state. The transition predicate "The value of m is being incremented," then, is 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. A *transition predicate* is a formula of predicate logic whose free variables are taken from V and V'. The truth value of the transition predicate ψ for the transition (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 instance, the transition predicate $m' = m + 1$ is true for the transition (σ, σ') iff $\sigma'(m) = \sigma(m) + 1$.

2.3 Behaviors

We make the following three assumptions about a system and its environment. First, within a finite interval of time, there can be only finitely many discrete state changes. Therefore, the behavior of a discrete system results in a countable sequence of transitions. Second, under no circumstance can a system prevent an environment activity. Therefore, in every state of a closed system, a stutter transition is enabled. Third, the environment never terminates. Therefore, the behavior of a closed discrete system results in an infinite sequence of transitions, infinitely many of which are stutter transitions.[2] For example, a legal sequence of moves, interspersed with infinitely many stutter transitions, constitutes a behavior of the game of chess; a sequence of consecutive execution steps, interspersed with infinitely many stutter transitions, constitutes a behavior of a given while program.

Formally, a *behavior* is a countably infinite sequence of states. The behavior $\bar{\sigma} = \sigma_0, \sigma_1, \sigma_2, \sigma_3, \ldots$ can be viewed, alternatively, as the countably infinite sequence of transitions (σ_i, σ_{i+1}), integer $i \geq 0$, with the property that any two neighboring transitions are consecutive. The behavior $\bar{\sigma}$ is called *environment-fair* if $\bar{\sigma}$ contains infinitely many stutter transitions. We describe behaviors using both state predicates and transition predicates. A state predicate is used to restrict 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 $\bar{\sigma}$. A transition predicate is used to restrict the transitions of a behavior, except for the environment transitions: the transition predicate ψ is *invariantly true* for the behavior $\bar{\sigma}$ if ψ is true for all transitions of $\bar{\sigma}$ that are not stutter transitions. Consider, for instance, the behavior $\bar{\sigma}_0$ whose odd transitions, starting from 0, increment the value of m, and whose even transitions are stutter transitions:

$$\bar{\sigma}_0(m) \ = \ 0, 1, 1, 2, 2, 3, 3, 4, 4, \ldots$$

The state predicate $m = 0$ is initially true for $\bar{\sigma}_0$, and the transition predicate $m' = m + 1$ is invariantly true for $\bar{\sigma}_0$.

2.4 Systems

In every state, a transition is either possible (enabled) or impossible (disabled). If several transitions are enabled in a state, the system has several options

[2]The requirement that every behavior must contain infinitely many stutter transitions, while not essential for our treatment of untimed systems, will lead to a pleasant parallelism between untimed and timed systems. This is because for timed systems, a similar requirement is necessary to ensure the divergence of time.

how to proceed. Multiple options may arise because of nondeterminism (as in the game of chess), because several system components proceed individually and independently (as is the case with the environment transitions, which occur independently of the system transitions), or because the variables do not capture sufficient detail to distinguish the sources of distinct activities whose targets can be distinguished. For all of these reasons, a system may have many different behaviors. We describe a system by defining the set of its behaviors. For example, the rules of chess define the set of legal move sequences that start from the initial board configuration; a while program defines the set of possible execution sequences, one sequence for each combination of input values.

Formally, a (*closed discrete*) *system* $\mathcal{S} = (\phi^I, \psi^N)$ is a pair that consists of a state predicate ϕ^I—the *initial condition* of $\mathcal{S}$—and a transition predicate ψ^N—the *next-state condition* of $\mathcal{S}$. The behavior $\bar{\sigma}$ is *possible* for the system $\mathcal{S}$ if (1) the initial condition ϕ^I is initially true for $\bar{\sigma}$, (2) the next-state condition ψ^N is invariantly true for $\bar{\sigma}$, and (3) $\bar{\sigma}$ is environment-fair. 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 next-state condition $m' = m + 1$. Here are some possible behaviors of $\mathcal{S}_0$:

$$\begin{aligned} \bar{\sigma}_0(m) &= 0,1,1,2,2,3,3,4,4,\ldots \\ \bar{\sigma}_1(m) &= 0,0,0,1,2,2,2,3,4,4,4,\ldots \\ \bar{\sigma}_2(m) &= 0,1,2,2,2,2,2,2,2,2,\ldots \end{aligned}$$

Not every set of environment-fair behaviors is the set of possible behaviors for some system. First, a system can neither force nor prevent environment activities: if, between any two transitions, we remove or add a stutter transition to a possible behavior of the system $\mathcal{S}$, then we obtain another possible 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 possible 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 possible behavior of $\mathcal{S}$ (*fusion closure*). Third, the operation of a system is almost entirely characterized by local conditions on individual states and transitions; the only global condition on the possible behaviors of a system is environment-fairness: if all finite prefixes of an environment-fair behavior $\bar{\sigma}$ can be extended to possible behaviors of the system $\mathcal{S}$, then $\bar{\sigma}$ itself is a possible behavior of $\mathcal{S}$ (*limit closure*).[3]

The three closure properties of systems have two important ramifications for the executability of systems. First, the possible behaviors of a system can be generated transition by transition: start with a state for which the initial

[3]Conversely, if we admit infinitary formulas as state and transition predicates, then every set of behaviors that is closed under stuttering, fusion, and limits is the set of possible behaviors for some system.

```
program NdUpDown:
   initially $m := 0$; $n := 1$;
   loop  $true$                     $\rightarrow$  $m := 1$
     or  $m = 0$                    $\rightarrow$  $n := n + 1$
     or  $m = 1 \wedge n > 0$       $\rightarrow$  $n := n - 1$
     end.
```

Figure 3: Guarded-command program

condition is true, and repeatedly choose either a stutter transition or an enabled transition for which the next-state condition is true. Every finite state sequence that is generated in this manner can be extended to a possible behavior of the system (*machine closure*). Second, a system need not progress, ever: when generating a behavior transition by transition, from any point on we may choose stutter transitions only. This is because every finite prefix of a possible behavior followed by infinitely many stutter transitions is also a possible behavior.[4]

2.5 Programs

A convenient notation for describing systems is a pseudo-programming language based on guarded commands. A *guarded-command program* $\mathcal{P}$ consists of a set of initial assignments followed by a set of guarded assignments. The program $\mathcal{P}$ is executed in a stepwise fashion: first, starting from any state, execute the initial assignments to obtain an initial state of $\mathcal{P}$; then continue to select, nondeterministically, and execute either a stutter transition or a guarded assignment whose guard is true. Every countably infinite sequence of states that may result from this stepwise process is called an *execution sequence* of $\mathcal{P}$. Consider, for instance, the guarded-command program *NdUpDown* of Figure 3. 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. Here are two execution sequences of *NdUpDown*:

$$\begin{aligned}\bar{\sigma}_3(m,n) &= (0,1),(0,2),(0,3),(1,3),(1,2),(1,1),(1,0),(1,0),\ldots\\ \bar{\sigma}_4(m,n) &= (0,1),(0,1),(0,2),(0,2),(1,2),(1,1),(1,1),(1,1),\ldots\end{aligned}$$

[4]The reader will have noticed that the playing field between the system and the environment is not level: while the environment always proceeds infinitely often, the system may not be able to proceed infinitely often even if this is permitted by the next-state condition. To ensure that, when given the chance, the system does proceed infinitely often, one typically imposes system-fairness conditions on the possible behaviors of a system. We shall see that this is not necessary for real-time systems.

Due to nondeterminism and stuttering, *NdUpDown* has infinitely many execution sequences.[5]

Every guarded-command program $\mathcal{P}$ describes a system whose possible behaviors are the environment-fair execution sequences of $\mathcal{P}$. When, out of convenience, we refer to the initial condition, the next-state condition, and the possible behaviors of the program $\mathcal{P}$, what we mean are the initial condition, the next-state condition, and the possible behaviors of the system described by $\mathcal{P}$. The initial condition of $\mathcal{P}$ is determined by the initial assignments of $\mathcal{P}$, and the next-state condition of $\mathcal{P}$ is determined by the guarded assignments of $\mathcal{P}$. For instance, the initial condition of the guarded-command program *NdUpDown* is given by the state predicate ϕ^I, and the next-state condition of *NdUpDown* is given by the transition predicate ψ^N:

$$\begin{aligned}
\phi^I &= (m = 0 \wedge n = 1) \\
\psi^N &= (m' = 1)_n \vee \\
&\quad\ (m = 0 \wedge n' = n + 1)_m \vee \\
&\quad\ (m = 1 \wedge n > 0 \wedge n' = n - 1)_m
\end{aligned}$$

The expression $(\psi)_{v_0,\ldots,v_k}$, for a transition predicate ψ and a list $v_0, \ldots, v_k$ of variables, is an abbreviation for the transition predicate $\psi \wedge v_0' = v_0 \wedge \ldots \wedge v_k' = v_k$; that is, the variables listed in the subscript remain unchanged. Each guarded assignment contributes a disjunct—called a *transition schema*—to the next-state condition. We leave it to the reader to formally define the initial condition and the next-state condition of an arbitrary guarded-command program.

Bibliographic remarks. The state-transition paradigm originated with state machines and has become a standard model for reactive systems (see, for example, the Kripke-Structure approach of Clarke and Emerson [CE81], the Fair-Transition-System approach of Manna and Pnueli [MP92], and the TLA approach of Lamport [Lam94]). 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 elucidated 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].

[5] While this is irrelevant for our discussion, the nondeterminism of the program *NdUpDown* may be due to two internal, concurrent processes that proceed independently—one repeatedly incrementing and decrementing n, and the other updating m once. If no assumptions are made about the relative speeds of the two processes, then n is incremented arbitrarily often before m is updated.

3 Timed Systems 1: Safety

3.1 Clocks

A timed system is a discrete system whose state changes occur in real-numbered time. Our starting assumption about timed systems is the synchrony hypothesis, which postulates that real time is an environment parameter, or equivalently, the passage of time is an environment activity. Consequently, the value of time changes (increases) only with environment activities, and all state changes of the system occur in zero time. The synchrony hypothesis is a useful mathematical abstraction: any system activity that does consume time can be modeled by two zero-time state changes that mark the beginning and the end, respectively, of the activity. Later, we will see how the system can regain some control about the passage of time, by restricting the amount of time that may pass before the next system activity.

The state of a timed system is determined by two kinds of variables—discrete variables and clock variables. The clock variables range over the nonnegative real numbers. As before, the values of the discrete variables change in discrete steps. By contrast, the values of the clock variables change as continuous functions over time: they increase uniformly, with derivative 1, as time advances. As before, the discrete variables capture no information about the environment: their values are changed explicitly according to a next-state condition. By contrast, the clock variables do capture some information about the environment, namely, the duration of environment activities: their values increase implicitly with every environment activity (in addition, the values of the clock variables may be changed explicitly, in accordance with the next-state condition). Formally, we partition the finite set V of variables into a set of discrete variables and a set of clock variables, called *clocks*. 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 discrete variables m to $\sigma(m)$, and all clocks x to $\sigma(x)+\delta$.

The behavior of a timed system results from two kinds of transitions—jump transitions and delay transitions. A jump transition updates the values of both the discrete variables and the clocks; a delay transition leaves the values of all discrete variables unchanged and advances the values of all clocks. Formally, the transition (σ, σ') is a *delay* if there is a nonnegative real δ—the *duration* of the delay—such that $\sigma' = \sigma + \delta$. In particular, every stutter transition is a delay. If the set V of variables contains at least one clock, then every stutter transition has the duration 0; if V contains no clocks, then all delays are stutter transitions, and we assign to all of them the duration 1. Every transition that is not a delay is called a *jump*, and is assigned the duration 0. Given a behavior $\bar{\sigma}$, let δ_i be the duration of the i-th transition of $\bar{\sigma}$, integer $i \geq 0$. The behavior $\bar{\sigma}$ *diverges* if the infinite sum $\sum_{i\geq 0} \delta_i$ of durations diverges. In particular, if V contains no clocks, then $\bar{\sigma}$ diverges iff $\bar{\sigma}$ is environment-fair—i.e., if it contains infinitely many stutter transitions.

Let ψ be a transition predicate, possibly containing clocks. The predicate ψ is a ***jump-invariant*** for the behavior $\bar{\sigma}$ if ψ is true for all jumps of $\bar{\sigma}$. Consider, for instance, the behavior $\bar{\sigma}_0$ whose even transitions increment the value of the discrete variable m and restart the clock x, and whose odd transitions are delays of duration 2:

$$\bar{\sigma}_5(m,x) \;=\; (0,0),(0,2),(1,0),(1,2),(2,0),(2,2),(3,0),(3,2),(4,0),\ldots$$

In this behavior, m is incremented at times $2, 4, 6, 8, \ldots$ The transition predicate $x \geq 2 \wedge m' = m+1 \wedge x' = 0$ is a jump-invariant for $\bar{\sigma}_5$. If V contains no clocks, then ψ is a jump-invariant for the behavior $\bar{\sigma}$ iff ψ is invariantly true for $\bar{\sigma}$.

3.2 Clock-constrained systems

A (*closed*) *clock-constrained system* $\mathcal{S} = (\phi^I, \psi^N)$ consists, like a discrete system, of an initial condition ϕ^I and a next-state condition ψ^N, except that both the state predicate ϕ^I and the transition predicate ψ^N may contain clocks. The behavior $\bar{\sigma}$ is *possible* for the clock-constrained system $\mathcal{S}$ if (1) the initial condition ϕ^I is initially true for $\bar{\sigma}$, (2) the next-state condition ψ^N is a jump-invariant for $\bar{\sigma}$, and (3) $\bar{\sigma}$ diverges. A discrete system, then, is the special case of a clock-constrained system for which the set V of variables contains no clocks. 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 next-state condition $1 \leq x \leq 2 \wedge m' = m+1 \wedge x' = 0$. Here are some possible behaviors of $\mathcal{S}_1$:

$$\begin{aligned}
\bar{\sigma}_5(m,x) &= (0,0),(0,2),(1,0),(1,2),(2,0),(2,2),(3,0),(3,2),(4,0),\ldots\\
\bar{\sigma}_6(m,x) &= (0,0),(0,1),(1,0),(1,1),(2,0),(2,1),(3,0),(3,1),(4,0),\ldots\\
\bar{\sigma}_7(m,x) &= (0,0),(0,0.5),(0,1.2),(1,0),(1,0),(1,1.5),(2,0),(2,1),\ldots\\
\bar{\sigma}_8(m,x) &= (0,0),(0,1),(0,1.5),(0,2),(0,19),(0,19.1),(0,57),\ldots
\end{aligned}$$

Not every set of divergent behaviors is the set of possible behaviors for some clock-constrained system. As for discrete systems, the possible behaviors of a clock-constrained system are closed under stuttering and under fusion. Third, if all finite prefixes of a divergent behavior $\bar{\sigma}$ can be extended to possible behaviors of the clock-constrained system $\mathcal{S}$, then $\bar{\sigma}$ is a possible behavior of $\mathcal{S}$ (closure under *divergent limits*). Fourth, if we merge two consecutive delays in a possible behavior of $\mathcal{S}$ into a single delay, or if we split a delay into two consecutive delays, then we obtain another possible behavior of $\mathcal{S}$ (closure under *timed stuttering*). Fifth, a clock-constrained system cannot prevent delays: if a finite prefix of a possible behavior of $\mathcal{S}$ is extended by a delay, then we obtain again a prefix of a possible behavior of $\mathcal{S}$ (closure under *waiting*). Notice that if V contains no clocks, then closure under divergent limits simplifies to limit closure, and closure under timed stuttering and closure under waiting are both subsumed by stutter closure.

```
program $CcUpDown$:
  declare $x, y$: clock;
  initially $n := 1$; $x := 0$; $y := 0$;
  loop  $x \leq 10 \wedge 1 \leq y \leq 5$              $\rightarrow$  $n := n + 1$; $y := 0$
    or  $x \geq 12 \wedge 1 \leq y \leq 5 \wedge n > 0$  $\rightarrow$  $n := n - 1$; $y := 0$
    end.
```

Figure 4: Clock-constrained program

Consequently, the possible behaviors of a clock-constrained system can be generated transition by transition: start with a state for which the initial condition is true, and repeatedly choose either a delay of arbitrary duration or an enabled transition for which the next-state condition is true. Every finite state sequence that is generated in this manner can be extended to a possible behavior. Furthermore, a clock-constrained system need not progress: when generating a behavior transition by transition, from any point on we may choose only delays (of any durations whose sum diverges). This is because every finite prefix of a possible behavior followed by infinitely many delays of, say, duration 1 is also a possible behavior.

3.3 Clock-constrained programs

The stepwise executability of clock-constrained systems leads us to a guarded-command notation. The syntax of a *clock-constrained program* $\mathcal{P}$ is the same as for guarded-command programs, except that both the initial assignments and the guarded assignments of $\mathcal{P}$ may contain clocks. The program $\mathcal{P}$ is executed in a stepwise fashion: first, starting from any state, execute the initial assignments to obtain an initial state of $\mathcal{P}$; then continue to select, nondeterministically, and execute either a delay of arbitrary duration, or a guarded command whose guard is true. Every countably infinite sequence of states that may result from this stepwise process is called an *execution sequence* of $\mathcal{P}$. Consider, for instance, the clock-constrained program *CcUpDown* of Figure 4. The program *CcUpDown* contains a discrete 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.

Every clock-constrained program $\mathcal{P}$ describes a clock-constrained system whose possible behaviors are the divergent execution sequences of $\mathcal{P}$. The initial condition of $\mathcal{P}$ is determined by the initial assignments of $\mathcal{P}$, and the next-state condition of $\mathcal{P}$ is determined by the guarded assignments of $\mathcal{P}$. For instance, the initial condition of *CcUpDown* is given by the state predicate ϕ^I, and the next-state condition of *CcUpDown* is given by the transition predicate ψ^N:

$$\begin{aligned} \phi^I &= (n = 1 \land x = 0 \land y = 0) \\ \psi^N &= (x \leq 10 \land 1 \leq y \leq 5 \land n' = n + 1 \land y' = 0)_x \lor \\ &\quad\; (x \geq 12 \land 1 \leq y \leq 5 \land n > 0 \land n' = n - 1 \land y' = 0)_x \end{aligned}$$

Each guarded assignment contributes a disjunct—called a *jump schema*—to the next-state condition. We leave it to the reader to formally define the initial condition and the next-state condition of an arbitrary clock-constrained program.

Bibliographic remarks. The synchrony hypothesis is due to Berry [BG88] (for an introduction to synchronous programming languages, see the monograph by Halbwachs [Hal93]). The resulting dichotomy of jump transitions versus delay transitions has been advocated also by various proponents of the interleaving view of concurrency (see, for example, the Timed-Transition-System approach [HMP94] and the TLA approach [AL94]). Clock variables—as we use them—were first introduced in temporal logic [AH94] (in conjunction with so-called "freeze quantifiers") and in finite automata [AD94]. More details on closure under divergent limits (divergence safety) and closure under timed stuttering can be found in [Hen92, HNSY94].

4 Timed Systems 2: Progress

4.1 Delays

We have imposed progress on the environment of discrete systems, through the requirement of environment-fairness, and on the environment of clock-constrained systems, through the requirement of time-divergence. According to our definitions, however, a system (as opposed to the environment) need not progress. For discrete systems, progress can be ensured by requiring fairness also for transitions that are not stutter transitions; for example, we could require that no transition may be enabled forever (or infinitely often) without being executed. System-fairness requirements are not necessary for clock-constrained systems. Instead, we ensure the progress of a clock-constrained system by permitting the system to prevent certain delays; a jump, then, must be executed before time can advance any further. By constraining delays in this way, the fairness requirement on delays—that time must advance beyond any bound—indirectly imposes fairness on jump transitions also.

We use delay predicates to describe the permissible delays. A delay predicate is a proposition that is true or false for each delay. Then, for any given state σ, the delay predicate χ limits the amount by which the environment can advance time to those delays with source state σ for which χ is true. For instance, the proposition "Time advances only as long as the value of the clock x is less than 5" is either true or false for each delay. We write this proposition as "$x < 5$," implicitly quantifying variables like x to refer to the value of x in all states, including the source state but excluding the target state, that occur during a delay. Hence, in every state σ, the delay predicate $x < 5$ prevents time from advancing for more than $5 - \sigma(x)$ time units (if $\sigma(x) \geq 5$, then the delay predicate $x < 5$ prevents time from advancing at all). This constraint can be used to enforce a jump within $5 - \sigma(x)$ time units.

Formally, we write (σ, δ) for a delay with the source state σ and the duration δ. A state σ' is *passed* by the delay (σ, δ) if $\sigma' = \sigma + \epsilon$ for some nonnegative real $\epsilon < \delta$. Consequently, if the duration δ is positive, then the source state σ is passed by the delay (σ, δ), but the target state $\sigma + \delta$ is not passed; if (σ, δ) is a stutter transition—i.e., $\delta = 0$—then no state is passed. The syntax of *delay predicates* is identical to the syntax of state predicates. A delay predicate χ is true for the delay (σ, δ) if χ, viewed as a state predicate, is true for all states that are passed by the delay (σ, δ). For instance, the delay predicate $x < 5$ is true for the 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). The truth of delays is preserved by splitting and merging: every delay predicate χ is true for all stutter transitions, and χ is true for two consecutive delays (σ, δ_1) and $(\sigma + \delta_1, \delta_2)$ iff χ is true for the combined delay $(\sigma, \delta_1 + \delta_2)$.

We have used transition predicates to restrict the jumps of a behavior, and we now use delay predicates to restrict the delays of a behavior. The delay predicate χ is a *delay-invariant* for the behavior $\bar{\sigma}$ if χ is true for all delays of $\bar{\sigma}$. Consider again the behavior $\bar{\sigma}_5$ whose even transitions increment the value of the discrete variable m and restart the clock x, and whose odd transitions are delays of duration 2:

$$\bar{\sigma}_5(m, x) \;=\; (0,0), (0,2), (1,0), (1,2), (2,0), (2,2), (3,0), (3,2), (4,0), \ldots$$

The delay predicate $x < 2$ is a delay-invariant for $\bar{\sigma}_5$; the delay predicate $x \neq 1$ is not. While we used, in Section 3.1, the jump-invariant $x \geq 2$ to describe the lower bound of 2 time units between successive increment operations, here we use the delay-invariant $x < 2$ to describe the matching upper bound.

4.2 Real-time systems

A *(closed) real-time system* $\mathcal{S} = (\phi^I, \psi^N, \chi^T)$ is a triple that consists of a clock-constrained system (ϕ^I, ψ^N) and a delay predicate χ^T—the *time-progress condition* of $\mathcal{S}$. The states for which ϕ^I is true are called the *initial states* of $\mathcal{S}$, the jump transitions for which ψ^N is true are called the *possible jumps* of $\mathcal{S}$, and the delays for which χ^T is true are called the *permissible delays* of $\mathcal{S}$. The behavior $\bar{\sigma}$ is *possible* for the real-time system $\mathcal{S}$ if (1) the initial condition ϕ^I is initially true for $\bar{\sigma}$, (2) the next-state condition ψ^N is a jump-invariant for $\bar{\sigma}$, (3) the time-progress condition χ^T is a delay-invariant for $\bar{\sigma}$, and (4) $\bar{\sigma}$ diverges. A clock-constrained system, then, is the special case of a real-time system for which the time-progress condition is the delay predicate *true*, which is true for all delays. If the behavior $\bar{\sigma}$ satisfies clauses (1) to (3), but violates clause (4), then $\bar{\sigma}$ is called a *convergent behavior* of the real-time system $\mathcal{S}$. Convergent behaviors are not possible, but will be useful for identifying the executable real-time systems.

The next-state condition of a real-time system asserts necessary conditions on jumps and the time-progress condition asserts sufficient conditions. For instance, the following real-time system $\mathcal{S}_2 = (\phi^I, \psi^N, \chi^T)$ changes the value of m from 0 to 1 at time 3 at the earliest and at time 5 at the latest:

$$\begin{aligned}
\phi^I &= (m = 0 \wedge x = 0)\\
\psi^N &= (x \geq 3 \wedge m' = 1)\\
\chi^T &= (m = 0 \wedge x < 5) \vee\\
&\ (m = 1)
\end{aligned}$$

While the requirement $x \geq 3$ of the next-state condition ensures the lower time bound of 3 (the jump that changes the value of m may happen at or after time 3), the requirement $x < 5$ of the time-progress condition ensures the upper time bound 5 (the jump 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 behaviors is the set of possible behaviors for some real-time system. While the possible behaviors of a real-time system are closed under stuttering, fusion, divergent limits, and timed stuttering, they are not necessarily closed under waiting. This is because the time-progress condition can enforce the progress of a real-time system.

4.3 Nonzenoness

There is a price to be paid, however, for progress: the possible behaviors of a real-time system may no longer be generated transition by transition. By starting with a state for which the initial condition is true, and repeatedly choosing either a delay that does not violate the time-progress condition, or an enabled transition for which the next-state condition is true, we may end up in a situation from which time cannot diverge. Consider, for instance, the following variant $\mathcal{S}_Z$ of the real-time system $\mathcal{S}_2$:

$$\begin{aligned}
\phi^I &= (m = 0 \wedge x = 0)\\
\psi^N &= (3 \leq x \leq 5 \wedge m' = 1)\\
\chi^T &= (m = 0 \wedge x < 10) \vee\\
&\ (m = 1)
\end{aligned}$$

The real-time systems $\mathcal{S}_2$ and $\mathcal{S}_Z$ have the same possible behaviors, namely, all divergent 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. The real-time system $\mathcal{S}_Z$, however, cannot 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 time-progress 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 convergent behavior of $\mathcal{S}$ is also a finite prefix of a possible behavior of $\mathcal{S}$. Consequently, the possible behaviors of a nonzeno real-time system can be generated transition by transition: start with a state for which the initial condition is true, and repeatedly choose either a delay that does not violate the time-progress condition or an enabled transition for which the next-state condition is true. Every finite state sequence that is generated in this manner can be extended to a possible behavior. In particular, every clock-constrained system is nonzeno. In our example, the real-time system $\mathcal{S}_2$ is nonzeno, but the real-time system $\mathcal{S}_Z$ is not. Nonzenoness, therefore, is a property of a real-time system (syntax) rather than a property of a set of behaviors (semantics). Indeed, every set of possible behaviors for some real-time system is also the set of possible behaviors for some nonzeno real-time system, which can be constructed by strenghtening the time-progress condition appropriately. In our example, the real-time system $\mathcal{S}_Z$ can be transformed into an equivalent nonzeno real-time system by strengthening the time-progress condition to $(m = 0 \wedge x < 5) \vee (m = 1)$.

4.4 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 by an arbitrary amount of time, but at most until the guard, viewed as a state predicate, becomes false. The execution of the guarded wait statement $\chi \rightarrow$ **wait**, therefore, results in a delay that satisfies the delay predicate χ. In particular, the guarded wait statement *false* $\rightarrow$ **wait** results in a stutter transition, 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 waiting 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 consists of a set of initial assignments, followed by a set of guarded assignments, followed by a set of guarded wait statements (all of which may contain clocks). The program $\mathcal{P}$ is executed in a stepwise fashion: first, starting from any state, execute the initial assignments to obtain

```
program RtUpDown1 :
   declare x, y: clock;
   initially n := 1; x := 0; y := 0;
   loop  $x \leq 10 \land y \geq 1$              →  $n := n + 1$; $y := 0$
     or  $x \geq 12 \land y \geq 1 \land n > 0$  →  $n := n - 1$; $y := 0$
     or  $n > 0 \land y < 5$                     →  wait
     or  $n = 0$                                 →  wait
     end.
```

Figure 5: Nonexecutable real-time program

an initial state of $\mathcal{P}$; then continue to select, nondeterministically, and execute either a guarded wait statement, or a guarded command whose guard is true. Every countably infinite sequence of states that may result from this stepwise process is called an *execution sequence* of $\mathcal{P}$. Consider, for instance, the real-time program *RtUpDown1* of Figure 5. 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.

It requires some care to define the possible behaviors of a real-time program. Consider the two guarded wait statements $1 \leq x < 3$ and $2 \leq x < 4$. It may not be possible to merge the consecutive execution of these two guarded wait statements into a single execution step (delay). Consequently, the execution sequences of a real-time program are not necessarily closed under timed stuttering. Hence we define the set of possible behaviors of a real-time program $\mathcal{P}$ to be the smallest set that contains all divergent execution sequences of $\mathcal{P}$ and is closed under timed stuttering. In this way, every real-time program $\mathcal{P}$ describes a real-time system: the initial condition of $\mathcal{P}$ is determined by the initial assignments of $\mathcal{P}$, the next-state condition of $\mathcal{P}$ is determined by the guarded assignments of $\mathcal{P}$, and the time-progress condition of $\mathcal{P}$ is determined by the guarded wait statements of $\mathcal{P}$. For instance, the initial condition of *RtUpDown1* is given by the state predicate ϕ^I, the next-state condition of *RtUpDown1* is given by the transition predicate ψ^N, and the time-progress condition of *RtUpDown1* is given by the delay predicate χ^T:

$$\begin{aligned}
\phi^I &= (n = 1 \land x = 0 \land y = 0)\\
\psi^N &= (x \leq 10 \land y \geq 1 \land n' = n + 1 \land y' = 0)_x \lor\\
&\quad\; (x \geq 12 \land y \geq 1 \land n > 0 \land n' = n - 1 \land y' = 0)_x\\
\chi^T &= (n > 0 \land y < 5) \lor\\
&\quad\; (n = 0)
\end{aligned}$$

```
program RtUpDown2:
   declare $x, y$: clock;
   initially $n := 1$; $x := 0$; $y := 0$;
   loop $x \leq 10 \wedge y \geq 1$                 $\rightarrow$ $n := n+1$; $y := 0$
     or $x \geq 12 \wedge y \geq 1 \wedge n > 0$   $\rightarrow$ $n := n-1$; $y := 0$
     or $x < 10 \wedge y < 5$                    $\rightarrow$ wait
     or $x \geq y+7 \wedge y < 5$                $\rightarrow$ wait
     or $n = 0$                                  $\rightarrow$ wait
     end.
```

Figure 6: Executable real-time program

Each guarded assignment contributes disjunct (a jump schema) to the next-state condition, and each guarded wait statement contributes a disjunct—called a *delay schema*—to the time-progress condition. We leave it to the reader to formally define the initial condition, the next-state condition, and the time-progress condition of an arbitrary real-time program. A clock-constrained program, in particular, is a real-time program with the single guarded wait statement *true* $\rightarrow$ **wait**, which yields the time-progress 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 transitions. A real-time program $\mathcal{P}$, therefore, is *executable* if it describes 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 transitions continuing from σ. Figure 6 shows an executable real-time program that has the same possible behaviors as *RtUpDown1*. The time-progress condition of the executable program *RtUpDown2* is given by the following delay predicate:

$$\begin{aligned} \chi^T \;=\; & (x < 10 \wedge y < 5) \vee \\ & (x \geq y+7 \wedge y < 5) \vee \\ & (n = 0) \end{aligned}$$

The real-time program *RtUpDown2* terminates—i.e., the value of n reaches 0 and execution continues with delays only—within at most 65 time units (to calculate this upper bound on the termination time of *RtUpDown2*, observe that the value of n can be at most 11 when $x = 10$).

4.5 Programming constructs

Our choice of guarded commands as programming language was instructionally motivated. In principle, we may add clocks and guarded wait statements to any programming language. To illustrate this, we now extend a simple language for while programs with clocks and guarded wait statements. Following the synchrony hypothesis, all tests and assignments of a while program are executed instantaneously, consuming zero time; all delays are indicated explicitly by guarded wait statements.

In this context, it is convenient to introduce abbreviations for several common applications of guarded wait statements. First, the instruction **wait** delays a while program 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 for an arbitrary amount of time, but at most until the state predicate ϕ becomes true; it stands for the guarded wait statement $\neg\phi \rightarrow$ **wait**. Third, the instruction **await** ϕ delays the program precisely until the state predicate ϕ becomes true; it stands for the while-program segment

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

Fourth, the instruction **wait** $[c, d]$ delays the program for at least c and at most d time units, for nonnegative constants c and d with $c \leq d$; it stands for the while-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** $[c, c]$.

Consider the real-time while program *RtUpDown3* of Figure 7. To construct the initial condition, next-state condition, and time-progress condition of *RtUpDown3*, we introduce a new discrete variable, the program counter pc, which ranges over the control locations of the program. Figure 8 shows the program *RtUpDown3* without nonatomic abbreviations, and annotated with control locations (including a final control location, ℓ_{19}, which signals the termination of the program). The initial condition of *RtUpDown3* is $pc = \ell_0$, for the initial control location ℓ_0. An execution step of *RtUpDown3* is a move between control locations and corresponds to the execution of either a test, an assignment, or a guarded wait statement. Each such execution step contributes a jump schema to the next-state condition of *RtUpDown3*; each guarded wait statement contributes, in addition, a delay schema to the time-progress condition. For example, the first instruction $\ell_0\colon n := 1$ contributes the jump schema

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

to the next-state condition. The third instruction ℓ_2: **wait** $[1, 5]$ contributes to the next-state condition the jump schemata

$$\begin{array}{l}(pc = \ell_2 \wedge pc' = \ell_3 \wedge z' = 0)_{n,x,y} \vee \\ (pc = \ell_3 \wedge pc' = \ell_4)_{n,x,y,z} \vee \\ (pc = \ell_4 \wedge z < 1 \wedge pc' = \ell_3)_{n,x,y,z} \vee \\ (pc = \ell_4 \wedge z \geq 1 \wedge pc' = \ell_5)_{n,x,y,z}\end{array}$$

```
program RtUpDown3:
  declare $x, y$: clock;
  $n := 1$; $x := 0$;
  wait $[1, 5]$;
  while $x \leq 10$ do
    $n := n + 1$; $y := 0$;
    repeat
      if $x < y + 7$
        then wait watching $x \geq 10 \lor y \geq 5$
        else wait watching $y \geq 5$
        fi
      until $y \geq 1$
    od;
  repeat wait watching $y \geq 5$ until $x \geq 12$;
  while $n > 0$ do
    $n := n - 1$;
    wait $[1, 5]$
    od.
```

Figure 7: Real-time while program

and contributes to the time-progress condition the delay schema

$$(pc = \ell_3 \land z < 5).$$

We leave it to the reader to construct the complete next-state condition of *RtUpDown3* as the disjunction of all jump schemata, and the complete time-progress condition as the disjunction of all delay schemata. The resulting possible behaviors are essentially the possible behaviors of the real-time guarded-command program *RtUpDown2* from Figure 6.[6] It follows, in particular, that the real-time while program *RtUpDown3* is executable and terminates—i.e., it reaches the control location ℓ_{19}—within at most 70 time units.[7]

Bibliographic remarks. Delay-invariants as a means for ensuring the progress of a timed system were first proposed in [HNSY94]; their expressive power

[6]The possible behaviors of *RtUpDown3* would be identical to the possible behaviors of *RtUpDown2* if we allow hiding of the program counter, and merging of consecutive tests and assignments into initial states and into atomic jumps (for instance, the two consecutive assignments $n := n+1$; $y := 0$ of the real-time while program *RtUpDown3* constitute a single jump of the real-time guarded-command program *RtUpDown2*).

[7]The difference in termination time between *RtUpDown3* and *RtUpDown2* is due to the instruction ℓ_{16} : wait $[1, 5]$, which is executed one last time just before the termination of the while program.

```
program $RtUpDown3$:
  declare $x, y, z$: clock;
  $\ell_0: n := 1$; $\ell_1: x := 0$;
  $\ell_2: z := 0$; repeat $\ell_3$: wait watching $z \geq 5$  $\ell_4$: until $z \geq 1$;
  while $\ell_5: x \leq 10$ do
    $\ell_6: n := n + 1$; $\ell_7: y := 0$;
    repeat
      if $\ell_8: x < y + 7$
        then $\ell_9$: wait watching $x \geq 10 \vee y \geq 5$
        else $\ell_{10}$: wait watching $y \geq 5$
        fi
    $\ell_{11}$: until $y \geq 1$
    od;
  repeat $\ell_{12}$: wait watching $y \geq 5$  $\ell_{13}$: until $x \geq 12$;
  while $\ell_{14}: n > 0$ do
    $\ell_{15}: n := n - 1$;
    $\ell_{16}: z := 0$; repeat $\ell_{17}$: wait watching $z \geq 5$  $\ell_{18}$: until $z \geq 1$
    od;
  loop $\ell_{19}$: wait end.
```

Figure 8: Annotated real-time while program

was studied in [HKWT95]. Nonzenoness (timed machine closure) was defined independently by several researchers [Hen92, AL94]; Abadi and Lamport coined the phrase. An algorithm that turns a finitary (cf. Appendix B) real-time system into a nonzeno real-time system with the same set of possible behaviors can be found in [HNSY94]. The real-time program *RtUpDown1* is derived from an example, due to Pnueli [HMP94], 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 the maximal possible value of n is 6, and *RtUpDown2* will terminate within 40 time units). The **wait watching** macro is inspired by ESTEREL [BG88]. The use of control-location labels to define the transitions of a while program is standard practice in state-transition semantics (see, for example, [MP92]).

5 Timed Systems 3: Reactivity

5.1 Untimed processes

The parallel composition of while programs is called a concurrent program. Whereas a while program describes a sequential discrete system with a single

```
program ParUp:
  initially $m := 0$; $n := 0$;
  cobegin
    loop if $\ell_0$: $even(n)$ then $\ell_1$: $m := m + 1$ fi end
    ||
    loop $k_0$: $n := n + m$ end
    coend.
```

Figure 9: Concurrent program

control flow, a concurrent program describes a discrete system that consists of multiple concurrent processes, each with its own control flow. The transitions of the concurrent processes of a discrete system are interleaved nondeterministically, and interspersed with environment transitions; that is, two transitions of a process may be separated by any number of transitions of other processes and by any number of stutter transitions. The interleaving of concurrent transitions reflects the underlying assumption that the execution speeds of the processes are unknown, independent, possibly very different and even varying. By ignoring the possibility of simultaneous concurrent transitions, the interleaving view provides a useful abstraction: in every state, the number of enabled transitions is equal to the sum (rather than the product) of transitions that are enabled for each process.

Consider, for example, the concurrent program *ParUp* of Figure 9. The program *ParUp* consists of two interacting processes. The "left" process repeatedly increments the value of m dependent on the value of n; the "right" process repeatedly increases the value of n by the value of m. The stepwise execution of *ParUp* interleaves the transitions of both processes with stutter transitions: any number of transitions of the left process may occur before a transition of the right process, and vice versa. The nondeterministic interleaving of concurrent transitions results in one of many possible behaviors. To track the control flows of all processes, we introduce a program counter for each process—in this example, the two program counters *left* and *right*.[8] Here are some possible behaviors

[8]Since the right process has a single control location, there is really no need for the program counter *right* in this example.

```
program RtParUp:
  initially m := 0; n := 0;
  cobegin
    loop wait 3; if even(n) then wait 1; m := m + 1 fi end
    ||
    loop wait 7; n := n + m end
    coend.
```

Figure 10: Concurrent real-time program

of *ParUp* (the value of *right* is always k_0):

$$\begin{aligned}
\bar{\sigma}_9(\mathit{left}, m, n) &= (\ell_0,0,0),(\ell_1,0,0),(\ell_1,0,0),(\ell_1,0,0),(\ell_0,1,0),(\ell_0,1,1),\\
&\quad (\ell_0,1,1),(\ell_0,1,1),(\ell_0,1,2),(\ell_0,1,2),(\ell_1,1,2),(\ell_1,1,3),\ldots\\
\bar{\sigma}_{10}(\mathit{left}, m, n) &= (\ell_0,0,0),(\ell_1,0,0),(\ell_0,1,0),(\ell_1,1,0),(\ell_0,2,0),(\ell_0,2,0),\\
&\quad (\ell_0,2,2),(\ell_1,2,2),(\ell_0,3,2),(\ell_0,3,5),(\ell_0,3,8),(\ell_0,3,8),\ldots\\
\bar{\sigma}_{11}(\mathit{left}, m, n) &= (\ell_0,0,0),(\ell_1,0,0),(\ell_0,1,0),(\ell_0,1,1),(\ell_0,1,1),(\ell_0,1,2),\\
&\quad (\ell_0,1,2),(\ell_0,1,3),(\ell_0,1,3),(\ell_0,1,4),(\ell_0,1,4),\ldots\\
\bar{\sigma}_{12}(\mathit{left}, m, n) &= (\ell_0,0,0),(\ell_1,0,0),(\ell_0,1,0),(\ell_0,1,0),(\ell_1,1,0),(\ell_0,2,0),\\
&\quad (\ell_0,2,0),(\ell_1,2,0),(\ell_0,3,0),(\ell_0,3,0),(\ell_1,3,0),\ldots
\end{aligned}$$

Disregarding stutter transitions, the behavior $\bar{\sigma}_9$ alternates transitions from both processes. A possible behavior of *ParUp* need not be fair to either or both processes; for instance, the behavior $\bar{\sigma}_{12}$ does not contain any transitions of the right process.

The concurrent program *ParUp* describes a discrete system over the four variables m, n, *left*, and *right*. Each execution step of either process contributes a jump schema to the next-state condition of *ParUp*. The initial condition of *ParUp* is given by the state predicate ϕ^I, and the next-state condition of *ParUp* is given by the transition predicate ψ^N:

$$\begin{aligned}
\phi^I &= (\mathit{left} = \ell_0 \wedge \mathit{right} = k_0 \wedge m = 0 \wedge n = 0)\\
\psi^N &= (\mathit{left} = \ell_0 \wedge \mathit{even}(n) \wedge \mathit{left}' = \ell_1)_{\mathit{right},m,n} \vee\\
&\quad (\mathit{left} = \ell_0 \wedge \neg\mathit{even}(n) \wedge \mathit{left}' = \ell_0)_{\mathit{right},m,n} \vee\\
&\quad (\mathit{left} = \ell_1 \wedge \mathit{left}' = \ell_0 \wedge m' = m + 1)_{\mathit{right},n} \vee\\
&\quad (n' = n + m)_{\mathit{left},\mathit{right},m}
\end{aligned}$$

We leave it to the reader to formally define the initial condition and the next-state condition of an arbitrary concurrent program.

```
program RtParUp:
   declare $x, y$: clock;
   initially $m := 0$; $n := 0$;
   cobegin
     loop
       $\ell_0$: $x := 0$; repeat $\ell_1$: wait watching $x \geq 3$  $\ell_2$: until $x \geq 3$;
       if $\ell_3$: $even(n)$ then
         $\ell_4$: $x := 0$; repeat $\ell_5$: wait watching $x \geq 1$  $\ell_6$: until $x \geq 1$;
         $\ell_7$: $m := m + 1$
         fi
       end
     ||
     loop
       $k_0$: $y := 0$; repeat $k_1$: wait watching $y \geq 7$  $k_2$: until $y \geq 7$;
       $k_3$: $n := n + m$
       end
     coend.
```

Figure 11: Annotated concurrent real-time program

5.2 Timed processes

The parallel composition of real-time while programs is called a concurrent real-time program. Whereas a real-time while program describes a sequential real-time system with a single control flow, a concurrent real-time program describes a real-time system that consists of multiple processes that proceed in parallel. If the processes are constrained in time, then the assumption of speed independence is no longer valid, and some interleaved sequences of concurrent transitions may no longer be possible. While the delays of concurrent processes must overlap, it is still convenient to interleave concurrent jumps that occur simultaneously.[9] This is made possible by the synchrony assumption: since, under this assumption, finite sequences of discrete state changes consume zero time, the interleaving view can be maintained for concurrent jumps within a real-time system.

Consider, for example, the concurrent real-time program *RtParUp* of Figure 10. The timed program *RtParUp* is obtained from the untimed program *ParUp* by adding delays before tests and 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 jump of the left process may interrupt a

[9]As in the untimed case, the alternative (combining simultaneous jumps into tuples of jumps) may lead, in any state, to an exponential proliferation of enabled transitions.

delay of the right process, and vice versa. Figure 11 shows the program *RtParUp* without nonatomic abbreviations, and annotated with control locations. Here are two possible behaviors of *RtParUp* (the values of both program counters are omitted, and all repetitions of quadruples are suppressed):

$$\begin{array}{rcl}\bar{\sigma}_{13}(m,n,x,y) & = & (0,0,0,0),(0,0,3,3),(0,0,0,3),(0,0,1,4),(1,0,1,4),\\ & & (1,0,0,4),(1,0,3,7),(1,1,3,7),(1,1,3,0),(1,1,0,0),\\ & & (1,1,3,3),(1,1,0,3),(1,1,3,6),(1,1,0,6),(1,1,1,7),\ldots\\ \bar{\sigma}_{14}(m,n,x,y) & = & (0,0,0,0),(0,0,3,3),(0,0,0,3),(0,0,1,4),(1,0,1,4),\\ & & (1,0,0,4),(1,0,3,7),(1,0,0,7),(1,1,0,7),(1,1,0,0),\\ & & (1,1,1,1),(2,1,1,1),(2,1,0,1),(2,1,3,4),(2,1,0,4),\ldots\end{array}$$

In particular, the timing constraints of *RtParUp* rule out the strict alternation of jumps from the left process and the right process; that is, no possible behavior of the timed program *RtParUp* corresponds to the possible behavior $\bar{\sigma}_9$ of the untimed program *ParUp*. Also, all possible behaviors of *RtParUp* contain both infinitely many jumps of the left process and infinitely many jumps of the right process; that is, the timing constraints ensure the progress of both processes. On the other hand, if a concurrent real-time program contains no clocks, then it has the same possible behaviors as the syntactically identical untimed program.

The concurrent real-time program *RtParUp* describes a real-time system over the two clocks x and y and the four discrete variables m, n, *left*, and *right*, where *left* and *right* are program counters for the left and right process, respectively. Each execution step of either process contributes a jump schema to the next-state 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 time-progress condition of *RtParUp*. The initial, next-state, and time-progress conditions of *RtParUp* are shown in Figure 12. As usual, we leave it to the reader to formally define the real-time system that is described by an arbitrary concurrent real-time program.

5.3 Open systems and open programs

When we consider a single process of a concurrent program in isolation, the other processes of the program appear as environment to the process under consideration. We assume that for each variable there is one process that has exclusive write access to the variable (we say that the process controls the variable), and we assume that every process has read access to every variable.[10] In particular, a process can read variables that are controlled by other processes. For example, in the concurrent program *ParUp*, the left process controls m, and

[10]These assumptions are chosen solely to keep the exposition simple. A write-shared variable can be modeled, using our definitions, as a separate process (alternatively, the definition of open systems can be generalized to permit shared write access). Read access can be restricted by introducing an operator for the hiding of variables.

$$
\begin{array}{rcl}
\phi^I & = & (\mathit{left} = \ell_0 \wedge \mathit{right} = k_0 \wedge m = 0 \wedge n = 0) \\[1ex]
\psi^N & = & (\mathit{left} = \ell_0 \wedge \mathit{left}' = \ell_1 \wedge x' = 0)_{\mathit{right},m,n,y} \vee \\
& & (\mathit{left} = \ell_1 \wedge \mathit{left}' = \ell_2)_{\mathit{right},m,n,x,y} \vee \\
& & (\mathit{left} = \ell_2 \wedge x < 3 \wedge \mathit{left}' = \ell_1)_{\mathit{right},m,n,x,y} \vee \\
& & (\mathit{left} = \ell_2 \wedge x \geq 3 \wedge \mathit{left}' = \ell_3)_{\mathit{right},m,n,x,y} \vee \\
& & (\mathit{left} = \ell_3 \wedge \mathit{even}(n) \wedge \mathit{left}' = \ell_4)_{\mathit{right},m,n,x,y} \vee \\
& & (\mathit{left} = \ell_3 \wedge \neg\mathit{even}(n) \wedge \mathit{left}' = \ell_0)_{\mathit{right},m,n,x,y} \vee \\
& & (\mathit{left} = \ell_4 \wedge \mathit{left}' = \ell_5 \wedge x' = 0)_{\mathit{right},m,n,y} \vee \\
& & (\mathit{left} = \ell_5 \wedge \mathit{left}' = \ell_6)_{\mathit{right},m,n,x,y} \vee \\
& & (\mathit{left} = \ell_6 \wedge x < 1 \wedge \mathit{left}' = \ell_5)_{\mathit{right},m,n,x,y} \vee \\
& & (\mathit{left} = \ell_6 \wedge x \geq 1 \wedge \mathit{left}' = \ell_7)_{\mathit{right},m,n,x,y} \vee \\
& & (\mathit{left} = \ell_7 \wedge \mathit{left}' = \ell_0 \wedge m' = m + 1)_{\mathit{right},n,x,y} \vee \\
& & (\mathit{right} = k_0 \wedge \mathit{right}' = k_1 \wedge y' = 0)_{\mathit{left},m,n,x} \vee \\
& & (\mathit{right} = k_1 \wedge \mathit{right}' = k_2)_{\mathit{left},m,n,x,y} \vee \\
& & (\mathit{right} = k_2 \wedge y < 7 \wedge \mathit{right}' = k_1)_{\mathit{left},m,n,x,y} \vee \\
& & (\mathit{right} = k_2 \wedge y \geq 7 \wedge \mathit{right}' = k_3)_{\mathit{left},m,n,x,y} \vee \\
& & (\mathit{right} = k_3 \wedge \mathit{right}' = k_0 \wedge n' = n + m)_{\mathit{left},m,x,y} \\[1ex]
\chi^T & = & (\mathit{left} = \ell_1 \wedge \mathit{right} = k_1 \wedge x < 3 \wedge y < 7) \vee \\
& & (\mathit{left} = \ell_5 \wedge \mathit{right} = k_1 \wedge x < 1 \wedge y < 7)
\end{array}
$$

Figure 12: Real-time system (ϕ^I, ψ^N, χ^T) described by the program *RtParUp*

the right process controls n. The left process, however, updates m dependent on the value of n, and the right process updates n dependent on the value of m. In this way, the left process takes into account information about its environment, which includes the right process, and the right process takes into account information about its environment, which includes the left process. Hence our formalisms of closed discrete systems (where the variables capture no environment information) and closed real-time systems (where the variables capture no environment information except for the duration of environment activities) are inadequate for representing individual processes. We now extend these formalisms to so-called open systems, whose variables may represent environment parameters.

An open system has two kinds of variables—controlled variables, which capture parameters that are updated by system activities, and uncontrolled variables, which capture parameters that are updated by environment activities. Accordingly, the behavior of an open system has two kinds of jump transitions—controlled jumps and uncontrolled jumps. A controlled jump represents a system activity, which updates the values of the controlled variables according to a next-

```
program ParUpLeft:
    local $m$: nat;
    external $n$: nat;
    initially $m := 0$;
    loop if $\ell_0$: even($n$) then $\ell_1$: $m := m + 1$ fi end.
```

Figure 13: Open program

state condition, and leaves the values of the uncontrolled variables unchanged. An uncontrolled jump represents an environment activity, which arbitrarily, and nondeterministically, updates the values of the uncontrolled variables, and leaves the values of the controlled variables unchanged.

Formally, an *open real-time system* $\mathcal{S} = (V^C, \phi^I, \psi^N, \chi^T)$ is a quadruple that consists of a set $V^C \subseteq V$ of variables—the variables *controlled* by $\mathcal{S}$—and a real-time system (ϕ^I, ψ^N, χ^T) such that (1) the initial condition ϕ^I contains no free (uncontrolled) variables from $U = V - V^C$, and (2) the next-state condition ψ^N contains no free variables from U'. The *closure* of $\mathcal{S}$ is the closed real-time system $\mathcal{S}^C = (\phi^I, \psi^C, \chi^T)$ with the modified next-state condition

$$\psi^C \;=\; (\psi^N \wedge \bigwedge_{v \notin V^C} v = v') \vee (\bigwedge_{v \in V^C} v = v').$$

The left disjunct of ψ^C ensures that the controlled jumps do not update uncontrolled variables; the right disjunct is a jump schema for uncontrolled jumps: it permits all possible changes to the values of the uncontrolled variables, while ensuring that no controlled variables are updated. The behavior $\bar{\sigma}$ is *possible* (or *convergent*) for the open real-time system $\mathcal{S}$ if $\bar{\sigma}$ is a possible (convergent) behavior of the closure $\mathcal{S}^C$. Both the set V^C of controlled variables and the set $V - V^C$ of uncontrolled variables may or may not contain clocks. The open real-time system $\mathcal{S}$ is an *open clock-constrained system* if the the time-progress condition χ^T is *true*; $\mathcal{S}$ is an *open discrete system* if, in addition, the set V of variables contains no clocks.

In programs, we declare controlled variables as **local**, and uncontrolled variables as **external**. External variables cannot occur on the left-hand side of assignments. A program is *open* if some variables are declared to be external. Recall, for example, the concurrent program *ParUp*. The left process by itself is given by the open program *ParUpLeft* of Figure 13. The open program *ParUpLeft* describes an open discrete system with the controlled variables m and *left*, and the uncontrolled variables n and *right*. The initial condition is given by the state predicate ϕ_l^I, and the next-state condition is given by the transition predicate ψ_l^N:

$$\begin{aligned}
\phi_l^I &= (\mathit{left} = \ell_0 \wedge m = 0) \\
\psi_l^N &= (\mathit{left} = \ell_0 \wedge \mathit{even}(n) \wedge \mathit{left}' = \ell_1)_m \vee \\
&\quad\ (\mathit{left} = \ell_0 \wedge \neg \mathit{even}(n) \wedge \mathit{left}' = \ell_0)_m \vee \\
&\quad\ (\mathit{left} = \ell_1 \wedge \mathit{left}' = \ell_0 \wedge m' = m + 1)
\end{aligned}$$

Here are two possible behaviors of *ParUpLeft* (the value of *right* is always k_0):

$$\begin{aligned}
\bar{\sigma}_9(\mathit{left}, m, n) &= (\ell_0, 0, 0), (\ell_1, 0, 0), (\ell_1, 0, 0), (\ell_1, 0, 0), (\ell_0, 1, 0), (\ell_0, 1, 1), \\
&\quad\ (\ell_0, 1, 1), (\ell_0, 1, 1), (\ell_0, 1, 2), (\ell_0, 1, 2), (\ell_1, 1, 2), (\ell_1, 1, 3), \ldots \\
\bar{\sigma}_{15}(\mathit{left}, m, n) &= (\ell_0, 0, 57), (\ell_0, 0, 18), (\ell_1, 0, 18), (\ell_0, 1, 18), (\ell_0, 1, 0), \\
&\quad\ (\ell_1, 1, 0), (\ell_1, 1, 1), (\ell_0, 2, 1), (\ell_0, 2, 1), (\ell_0, 2, 18), \ldots
\end{aligned}$$

In particular, every possible behavior of *ParUp* is also a possible behavior of *ParUpLeft*.

Symmetrically, the right process of the concurrent program *ParUp* describes an open discrete system with the controlled variables n and *right*, and the uncontrolled variables m and *left*. The initial condition is given by the state predicate ϕ_r^I, and the next-state condition is given by the transition predicate ψ_r^N:

$$\begin{aligned}
\phi_r^I &= (\mathit{right} = k_0 \wedge n = 0) \\
\psi_r^N &= (n' = n + m)_{\mathit{right}}
\end{aligned}$$

The right process of the concurrent real-time program *RtParUp* describes an open real-time system with the controlled variables n, y, and *right*, and the following initial, next-state, and time-progress conditions:

$$\begin{aligned}
\phi_r^I &= (\mathit{right} = k_0 \wedge n = 0) \\
\psi_r^N &= (\mathit{right} = k_0 \wedge \mathit{right}' = k_1 \wedge y' = 0)_n \vee \\
&\quad\ (\mathit{right} = k_1 \wedge \mathit{right}' = k_2)_{n,y} \vee \\
&\quad\ (\mathit{right} = k_2 \wedge y < 7 \wedge \mathit{right}' = k_1)_{n,y} \vee \\
&\quad\ (\mathit{right} = k_2 \wedge y \geq 7 \wedge \mathit{right}' = k_3)_{n,y} \vee \\
&\quad\ (\mathit{right} = k_3 \wedge \mathit{right}' = k_0 \wedge n' = n + m)_y \\
\chi_r^T &= (\mathit{right} = k_1 \wedge y' < 7)
\end{aligned}$$

Notice that every possible behavior of *RtParUp* is also a possible behavior of the right process.

Open systems can be composed. Let $\mathcal{S}_1 = (V_1^C, \phi_1^I, \psi_1^N, \chi_1^T)$ and $\mathcal{S}_2 = (V_2^C, \phi_2^I, \psi_2^N, \chi_2^T)$ be two open real-time systems with disjoint sets V_1^C and V_2^C of controlled variables. The ***parallel composition*** of $\mathcal{S}_1$ and $\mathcal{S}_2$ is the open real-time system $\mathcal{S}_1 || \mathcal{S}_2$ with the set $V_1^C \cup V_2^C$ of controlled variables, the initial condition $\phi_1^I \wedge \phi_2^I$, the next-state condition

$$(\psi_1^N \wedge \bigwedge_{v \in V_2^C} v = v') \vee (\psi_2^N \wedge \bigwedge_{v \in V_1^C} v = v'),$$

```
program Nonreceptive:
   local m: nat;
   external n: nat;
   initially m := 0;
   loop
     wait $\frac{1}{2^m}$;
     m := m + 1;
     if even(n) then await ¬even(n) else await even(n) fi
     end.
```

Figure 14: Nonexecutable open real-time program

and the time-progress condition $\chi_1^T \wedge \chi_2^T$. Consequently, the initial states of $\mathcal{S}_1||\mathcal{S}_2$ are those states that are initial for both $\mathcal{S}_1$ and $\mathcal{S}_2$; the possible jumps of $\mathcal{S}_1||\mathcal{S}_2$ are those jumps that are possible for the closures of both $\mathcal{S}_1$ and $\mathcal{S}_2$; the permissible delays of $\mathcal{S}_1||\mathcal{S}_2$ are those delays that are permissible for both $\mathcal{S}_1$ and $\mathcal{S}_2$. If $\mathcal{S}_1^C = (\phi_1^I, \psi_1^C, \chi_1^T)$ is the closure of $\mathcal{S}_1$, and $\mathcal{S}_2^C = (\phi_2^I, \psi_2^C, \chi_2^T)$ is the closure of $\mathcal{S}_2$, then the closure of $\mathcal{S}_1||\mathcal{S}_2$ is $(\phi_1^I \wedge \phi_2^I, \psi_1^C \wedge \psi_2^C, \chi_1^T \wedge \chi_2^T)$. It follows that the possible behaviors of the compound system $\mathcal{S}_1||\mathcal{S}_2$ are precisely those behaviors that are possible for both component systems $\mathcal{S}_1$ and $\mathcal{S}_2$. For example, the possible behaviors of *ParUp* are those behaviors that are possible for both the left and the right process, and the same is true for *RtParUp*.

5.4 Receptiveness

While the appropriate condition for the executability of real-time systems is nonzenoness, the appropriate condition for the executability of open real-time systems is more complicated. This is because nonzenoness is not preserved by parallel composition. To see this, consider the open real-time program *Nonreceptive* of Figure 14. The local variable m is initialized to 0 and updated by the program. The external variable n is updated, arbitrarily and nondeterministically, by the environment. The program *Nonreceptive* increments m at time 1, and whenever the environment changes the value of n from even to odd or from odd to even, exactly $\frac{1}{2^m}$ time units after the change of n. For example, if the environment increments n at times 1, 2, 3, ..., then the program may increment m at times 1, 1.5, 2.25, 3.125, ...

However, if the environment of *Nonreceptive* behaves like the program itself, only with the roles of m and n interchanged, then m may be incremented at times 1, 1.5, 2.25, 2.625, 2.7875, ..., and n may be incremented at times 1, 2, 2.5, 2.75, 2.825, ...The resulting behavior converges. Indeed, there is no possible behavior in this case: every stepwise execution of the two symmetric concurrent processes results necessarily in a convergent behavior. Since the program and the environment are symmetric, both are equally to blame for preventing time from diverging. This happens despite the fact that the closure of *Nonreceptive* is nonzeno: every finite sequence of controlled jumps, uncontrolled jumps, and delays can be extended to an infinite, divergent behavior. To obtain such a divergent extension, however, the environment must collaborate with the program (specifically, the environment must wait long enough between consecutive updates of n). We define a sufficient condition—called receptiveness—for the executability of open real-time systems in all "reasonable" environments, where reasonableness is a much weaker assumption about environments than collaboration: roughly speaking, an environment is reasonable if it does not unilaterally prevent time from diverging. In particular, many adversarial environments are reasonable. This is why the quantification over all reasonable environments is best understood as a game of the system against the environment.

The game between an open real-time system and the environment may start at any state that can be reached from an initial state by a finite sequence of jumps (controlled and uncontrolled) and delays. With each move, the system proposes either new values for the controlled variables which satisfy the next-state condition, or a duration for a delay which satisfies the time-progress condition. The environment, independently and simultaneously, proposes either new values for the uncontrolled variables or a duration for a delay. The move leads to a new state in one of three possible ways: (1) the system has proposed new values for the controlled variables, and the new state results from the corresponding controlled jump; (2) the environment has proposed new values for the uncontrolled variables, and the new state results from the corresponding uncontrolled jump; (3) both the system and the environment have proposed durations for delays, and the new state results from the delay whose duration is the minimum of the two proposed durations. If both the system and the environment have proposed new values for variables, then the result of the move is, nondeterministically, either (1) or (2).

The game is played for a countably infinite sequence of moves, which generate a countably infinite sequence of states. The outcome of the game is either a possible behavior of the system, or a convergent behavior. The system wins the game if either the outcome diverges, or the environment is to blame for convergence. The latter is the case if, after finitely many moves, either the system always proposes new values for the controlled variables but its proposal is never chosen (because (2) is always chosen over (1)), or the system always proposes durations that are strictly larger than the durations that are chosen (because the environment always proposes either a smaller duration than the system, or new values for some uncontrolled variables when the system proposes a positive duration). An open real-time system is called receptive if it has a winning strategy in this game. The existence of a winning strategy ensures that if time is prevented from diverging, it must be the fault of the environment.

We now formalize the receptiveness game. Let $\mathcal{S} = (V^C, \phi^I, \psi^N, \chi^T)$ be an

```
program RoundRobin:
   local i: task;
   initially i := FirstTask;
   loop
      ResumeTask(i);
      wait 10;
      SuspendTask(i);
      i := NextTask(i)
      end.
```

Figure 15: Round-robin scheduler

open real-time system. A *strategy* for $\mathcal{S}$ is a function s that maps every finite state sequence $\sigma_0, \ldots, \sigma_i$ either to a state σ' such that the next-state condition ψ^N is true for the transition (σ_i, σ'), or to a nonnegative real δ such that the time-progress condition χ^T is true for the delay (σ_i, δ). A behavior $\bar{\sigma}$ is an *outcome* of the strategy s if there is a position $j \geq 0$ such that for all positions $i \geq j$, either (1) $s(\sigma_0, \ldots, \sigma_i)$ is a state and the transition (σ_i, σ_{i+1}) is a controlled jump with $\sigma_{i+1} = s(\sigma_0, \ldots, \sigma_i)$, or (2) the transition (σ_i, σ_{i+1}) is an uncontrolled jump—i.e., $\sigma_i(v) = \sigma_{i+1}(v)$ for all variables $v \in V^C$—or (3) $s(\sigma_0, \ldots, \sigma_i)$ is a real number and the transition (σ_i, σ_{i+1}) is a delay of duration at most $s(\sigma_0, \ldots, \sigma_i)$. Notice that strategies are applied only eventually, from some position j onwards. The outcome $\bar{\sigma}$ is *fair* to the strategy s if there are infinitely many positions $i \geq 0$ such that the transition (σ_i, σ_{i+1}) is either a controlled jump or a delay of duration $s(\sigma_0, \ldots, \sigma_i)$. By contrast, in an unfair outcome, from some position onwards the environment determines all moves. The open real-time system $\mathcal{S}$ is *receptive* if there exists a strategy s for $\mathcal{S}$—the *receptiveness strategy*—such that no fair outcome of s is a convergent behavior of $\mathcal{S}$. In particular, every open clock-constrained system is receptive (consider the strategy that always proposes a delay of duration 1).

For every open real-time system $\mathcal{S}$, if $\mathcal{S}$ is receptive, then the closure $\mathcal{S}^C$ is nonzeno. This is because every finite prefix of a convergent behavior of $\mathcal{S}^C$ can be extended to a possible behavior of $\mathcal{S}^C$ by repeated application of the receptiveness strategy for $\mathcal{S}$. It can be shown that receptiveness is preserved by parallel composition: for open real-time systems $\mathcal{S}_1$ and $\mathcal{S}_2$, if both $\mathcal{S}_1$ and $\mathcal{S}_2$ are receptive, then $\mathcal{S}_1 || \mathcal{S}_2$ is also receptive. An open real-time program $\mathcal{P}$, therefore, is *executable* if it describes a receptive real-time system: in composition with any executable environment, an executable program can be executed step by step. For example, both processes of the real-time program *RtParUp* are executable, but the real-time program *Nonreceptive* is not.

```
program NrRoundRobin:
  local i: task; x: clock;
  external done: array of bool;
  initially i := FirstTask;
  loop
    ResumeTask(i);
    x := 0; await x ≥ 10 ∨ done[i];
    if ¬done[i] then SuspendTask(i) fi;
    repeat i := NextTask(i) until ¬done[i]
    end.
```

Figure 16: Nonexecutable round-robin scheduler for terminating tasks

5.5 Embedded programming

At last, we are ready to illustrate our approach to real-time programming by designing typical application programs such as embedded controllers and schedulers. Both embedded controllers and schedulers are examples of open real-time systems: an embedded controller is a real-time process that runs concurrently with the plant processes that are being controlled; a scheduler is a real-time process that runs concurrently with the task processes that are being scheduled. For example, the open real-time program *GateController* of Figure 1 controls the gate at a railroad crossing. The gate is closed 5 time units after a train signals its approach by setting the external bit *train_present*, and the gate is opened again as soon as the train signals its exit from the crossing by resetting *train_present*.

Schedulers can operate according to various protocols. The round-robin scheduler *RoundRobin* of Figure 15 schedules a task for exactly 10 time units, and then moves on to the next task. The local scheduler variable i indicates which task is currently running. If tasks may terminate, a more sophisticated scheduler is needed. We assume that when task i terminates, the external bit $done[i]$ is set. The round-robin scheduler *NrRoundRobin* of Figure 16 schedules a task either for 10 time units or until the task terminates, whatever happens first. This scheduler, however, is not executable: given a sequence of tasks that terminate within 1, 0.5, 0.25, ... time units, the stepwise execution of *NrRoundRobin* produces a convergent behavior. Such a scheduler is not physically realizable, because it would have to detect the termination of a task, suspend the task, and resume a new task within an arbitrarily short period of time. An executable version of a round-robin scheduler for terminating tasks is shown in Figure 17. This scheduler describes a receptive system, because it needs 1 time unit for suspending a task and resuming another task.

```
program $TtRoundRobin$:
  local $i$: task; $x$: clock;
  external $done$: array of bool;
  initially $i := FirstTask$;
  loop
    $ResumeTask(i)$;
    $x := 0$; await $x \geq 10 \vee done[i]$;
    if $\neg done[i]$ then $SuspendTask(i)$ fi;
    wait 1;
    repeat $i := NextTask(i)$ until $\neg done[i]$
    end.
```

Figure 17: Executable round-robin scheduler for terminating tasks

Finally, the priority scheduler *Priority* of Figure 18 schedules a task until it terminates or until an interrupt arrives from a task with higher priority, whatever happens first. An interrupt is indicated by the uncontrolled bit *interrupt* being toggled. The unit delay of the scheduler ensures executability. The bit *prev_interrupt* guarantees that a single interrupt is not lost if it occurs after the program has computed the task with the highest priority but before the program goes on to wait for an interrupt. (Of course, an even number of simultaneous interrupts may be lost; so if interrupts from several tasks can arrive simultaneously at precisely the same instant in real-numbered time, then a separate bit *interrupt*$[i]$ is needed for each task i.)

Bibliographic remarks. The interleaving view of concurrency was first advocated by Dijkstra [Dij65]. Parallel composition as intersection (or conjunction) is a salient feature if systems are identified with sets of possible behaviors (see, for example, [AL93]). For untimed systems, receptiveness was first defined by Dill [Dil89b]; for timed systems, by Gawlick, Segala, Sogaard-Andersen, and Lynch [GSSAL94]. There, it was also shown that receptiveness is closed under parallel composition. An "industrial-strength" version of our model for open discrete systems and open real-time systems, complete with capabilities for synchronizing concurrent transitions and for restricting read access to variables, can be found in [AH96, AH97]. The latter reference also contains an algorithm that checks whether a finitary (cf. Appendix B) open real-time system is receptive. Schedulers and railroad-gate controllers pervade the real-time literature as examples [HJL93, HMP94].

```
program $Priority$:
   local $i$: task; $prev\_interrupt$: bool;
   external $interrupt$: bool; $done$: array of bool;
   loop
     $prev\_interrupt := interrupt$;
     $i := MaxPriorityTask$;
     $ResumeTask(i)$;
     if $prev\_interrupt$
       then await $\neg interrupt \vee done[i]$
       else await $interrupt \vee done[i]$
       fi;
     if $\neg done[i]$ then $SuspendTask(i)$ fi;
     wait 1
     end.
```

Figure 18: Priority scheduler

6 Timed Systems 4: Continuity

6.1 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 may deviate from the actual time by a factor of p, for a positive real number $p > 0$. If the clock is started at 0, after t time units it may show as little as t/p and as much as $t \cdot p$. We indicate that the drift of the clock x is bounded by p with the clock declaration x: **clock drift** p (the condition **drift** 1 is suppressed as usual), and p is called the ***drift bound*** of x. For example, Figure 19 shows the railroad-gate controller from Figure 1, assuming that the drift of the controller clock x is bounded by $p = 1.1$. This version of the controller may close the gate as early as (approx.) 4.55 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 2, all states that are passed by delays satisfy the differential constraint $0.5 \leq \dot{x} \leq 2$. Referring to its mixed discrete-continuous nature, a system with drifting clocks is called "hybrid." Hybrid systems have both discrete variables, which are updated in discrete steps that consume no time (resulting in jumps), and continuous real-valued variables—

```
program DcGateController:
   local x: clock drift 1.1;
   external train_present: bool;
   loop
      await train_present;
      x := 0; await x = 5;
      CloseGate;
      await ¬train_present;
      OpenGate
      end.
```

Figure 19: Railroad-gate controller with a drifting clock

such as clocks and drifting clocks—which are updated as continuous functions while time elapses (during delays, resulting in so-called "flows"). When more general differential constraints are permitted to describe the evolution of continuous variables, they can be used to measure not only time, or the values of drifting physical clocks, but arbitrary continuous parameters such as temperature, pressure, or position.

6.2 Hybrid systems

The behavior of a hybrid system consists of jumps and flows. The flows are generalizations of delays. Formally, a *flow* (f, δ) consists of a nonnegative real δ—the *duration* of the flow—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;[11] in particular, for every variable $v \in V$ that does not range over the reals, the projection $f|_v$ must be a constant function). The state $f(0)$ is the *source state* of the flow (f, δ), and the state $f(\delta)$ is the *target state* of the flow. A state σ' is *passed* by the flow (f, δ) if $\sigma' = f(\epsilon)$ for some nonnegative real $\epsilon < \delta$. A *hybrid behavior* is a countably infinite sequence $\bar{\tau} = \tau_0, \tau_1, \tau_2, \ldots$ of jumps and flows such that any two neighboring elements of $\bar{\tau}$ are consecutive—i.e., for all $i \geq 0$, the target state of τ_i is equal to the source state of τ_{i+1}. The definitions of divergence, initial truth of state predicates, and jump-invariants carry over directly from behaviors to hybrid behaviors.

We describe flows using flow predicates. Let $\dot{V}$ be the set of dotted variables $\dot{v}$ for which the undotted version v occurs in V and ranges over the reals. A *flow predicate* is a formula of predicate logic whose free variables are taken from V and $\dot{V}$. The flow predicate χ is true for the flow (f, δ) if χ is true for all states

[11]For all reals ϵ, define $f|_v(\epsilon) = f(\epsilon)(v)$.

```
program Thermostat:
  local heat: {on, off}; θ: continuous;
  initially heat := on; θ := 68;
  loop  heat = off ∧ θ ≤ 65  →  heat := on
    or  heat = on ∧ θ ≥ 70   →  heat := off
    or  heat = on ∧ θ < 70   →  wait[θ̇ = c₀(c₁ − θ)]
    or  heat = off ∧ θ > 65  →  wait[θ̇ = −c₀θ]
    end.
```

Figure 20: Thermostat program

σ' that are passed by the flow (f, δ), where the truth value of χ for $\sigma' = f(\epsilon)$ is obtained as follows: each free variable $v \in V$ in χ is interpreted as the value $\sigma'(v)$ of $f|_v$ at ϵ, and each free variable $v \in \dot{V}$ in χ is interpreted as the first derivative of $f|_v$ at ϵ. The flow predicate χ is a *flow-invariant* for the hybrid behavior $\bar{\tau}$ if χ is true for all flows of $\bar{\tau}$.

A (*closed*) *hybrid system* is a real-time system whose time-progress condition is a flow predicate. The hybrid behavior $\bar{\tau}$ is *possible* for the hybrid system $\mathcal{S}$ if (1) the initial condition of $\mathcal{S}$ is initially true for $\bar{\tau}$, (2) the next-state condition of $\mathcal{S}$ is a jump-invariant for $\bar{\tau}$, (3) the time-progress condition of $\mathcal{S}$ is a flow-invariant for $\bar{\tau}$, and (4) $\bar{\tau}$ diverges. Suppose that $\mathcal{S}$ is a hybrid system with the time-progress condition χ^T. The variable v is a *discrete variable* of $\mathcal{S}$ if either v is not a real-valued variable, or the time-progress condition χ^T implies $\dot{v} = 0$; that is, the variable v is not modified during flows. All other variables in V are *continuous variables* of $\mathcal{S}$. In particular, the continuous variable x is a *clock* if the time-progress condition χ^T implies $x \geq 0 \wedge \dot{x} = 1$. A real-time system, then, is the special case of a hybrid system for which all variables in V are discrete variables or clocks. Other examples of continuous variables are drifting clocks and stopwatches: the continuous variable y is a *drifting clock*, with drift bound $p > 0$, if the time-progress condition χ^T implies $y \geq 0 \wedge 1/p \leq \dot{y} \leq p$; the continuous variable z is a *stopwatch* if the time-progress condition χ^T implies $(z \geq 0 \wedge \dot{z} = 1) \vee (\dot{z} = 0)$. Unlike clocks, a stopwatch can be halted and later restarted from the value at which it was halted.

Continuous variables can be used to measure continuous parameters other than time. Consider, for example, a thermostat that regulates the room temperature. Suppose that 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 $\theta(t) = 65e^{-c_0 t} + c_1(1 - e^{-c_0 t})$,

```
program $Train$:
   local $d$: continuous; $train\_present$: bool;
   loop
      $\ell_0: d := 5000$;
      repeat
         $\ell_1$: wait$[-55 \leq \dot{d} \leq -45]$ watching $d \leq 1000$
         $\ell_2$: until $d \leq 1000$;
      $\ell_3: train\_present := true$;
      repeat
         $\ell_4$: wait$[-50 \leq \dot{d} \leq -35]$ watching $d \leq -100$
         $\ell_5$: until $d \leq -100$;
      $\ell_6: train\_present := false$;
      $\ell_7$: wait$[-55 \leq \dot{d} \leq -45]$
      end.
```

Figure 21: Train program

where c_1 is a constant determined by the power of the heater. The resulting hybrid system has the initial condition ϕ^I, the next-state condition ψ^N, and the time-progress condition χ^T:

$$\begin{aligned}
\phi^I &= (heat = on \wedge \theta = 68) \\
\psi^N &= (heat = off \wedge \theta \leq 65 \wedge heat' = on)_\theta \vee \\
&\quad\; (heat = on \wedge \theta \geq 70 \wedge heat' = off)_\theta \\
\chi^T &= (heat = on \wedge \dot{\theta} = c_0(c_1 - \theta) \wedge \theta < 70) \vee \\
&\quad\; (heat = off \wedge \dot{\theta} = -c_0\theta \wedge \theta > 65)
\end{aligned}$$

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

The definitions of nonzenoness, open systems, and receptiveness carry over directly from real-time systems to hybrid systems, provided that all uncontrolled variables are discrete. For a discussion of open hybrid systems with uncontrolled continuous variables, such as the water-level controller from Figure 2, we refer the interested reader to the references given in the bibliographic remarks.

6.3 Hybrid programs

We describe hybrid systems by guarded-command programs such as the *Thermostat* program of Figure 20, which describes the example from Section 6.2, or by while programs such as the *Train* program of Figure 21. Clock variables (possibly drifting) are declared, as before, as **clock**; all other continuous variables,

$$
\begin{array}{rcl}
\phi^I & = & (pc = \ell_0) \\[1em]
\psi^N & = & (pc = \ell_0 \ \wedge \ pc' = \ell_1 \ \wedge \ d' = 5000)_{train\_present} \ \vee \\
& & (pc = \ell_1 \ \wedge \ pc' = \ell_2)_{train\_present,d} \ \vee \\
& & (pc = \ell_2 \ \wedge \ d > 1000 \ \wedge \ pc' = \ell_1)_{train\_present,d} \ \vee \\
& & (pc = \ell_2 \ \wedge \ d \leq 1000 \ \wedge \ pc' = \ell_3)_{train\_present,d} \ \vee \\
& & (pc = \ell_3 \ \wedge \ pc' = \ell_4 \ \wedge \ train\_present' = true)_d \ \vee \\
& & (pc = \ell_4 \ \wedge \ pc' = \ell_5)_{train\_present,d} \ \vee \\
& & (pc = \ell_5 \ \wedge \ d > -100 \ \wedge \ pc' = \ell_4)_{train\_present,d} \ \vee \\
& & (pc = \ell_5 \ \wedge \ d \leq -100 \ \wedge \ pc' = \ell_6)_{train\_present,d} \ \vee \\
& & (pc = \ell_6 \ \wedge \ pc' = \ell_7 \ \wedge \ train\_present' = false)_d \ \vee \\
& & (pc = \ell_7 \ \wedge \ pc' = \ell_0)_{train\_present,d} \\[1em]
\chi^T & = & (pc = \ell_1 \ \wedge \ -55 \leq \dot{d} \leq -45 \ \wedge \ d > 1000) \ \vee \\
& & (pc = \ell_4 \ \wedge \ -50 \leq \dot{d} \leq -35 \ \wedge \ d > -100) \ \vee \\
& & (pc = \ell_7 \ \wedge \ -55 \leq \dot{d} \leq -45)
\end{array}
$$

Figure 22: Hybrid system (ϕ^I, ψ^N, χ^T) described by the program *Train*

as **continuous**. Given a state predicate ϕ and a differential constraint χ,[12] the guarded wait statement $\phi \rightarrow$ **wait**$[\chi]$ delays the program by an arbitrary amount of time, but at most until the guard becomes false. Throughout the delay, the evolution of the continuous variables is governed by the diffential constraint χ. The execution of the guarded wait statement $\phi \rightarrow$ **wait**$[\chi]$, therefore, results in a flow that satisfies the flow predicate $\phi \wedge \chi$. As expected, the instruction **wait**$[\chi]$ **watching** ϕ stands for the guarded wait statement $\neg\phi \rightarrow$ **wait**$[\chi]$.

In the *Train* program, the continuous variable d represents the distance, in meters, of a train from the railroad crossing. The dotted variable $\dot{d}$, then, represents the speed of the train in meters per time unit. 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 *train_present*, and slows down to 35 to 50 meters per time unit. When the train is 100 meters past the crossing, it leaves and may return along a cyclic track whose length is at least 5,100 meters. The hybrid program *Train* describes, therefore, the hybrid system of Figure 22. We may wish to compose the *Train* program with the gate controller *DcGateController*. It is left to the ambitious reader to determine the hybrid system that is described by the concurrent hybrid program *Train*||*DcGateController*. The resulting system satisfies the safety

[12]Differential constraints are flow predicates with certain restrictions on executability, such as solvability of differential equations. For details, we refer the interested reader to the references given in the bibliographic remarks.

requirement that whenever the train is within 10 meters of the crossing ($-10 \leq d \leq 10$), the gate is closed.

Bibliographic remarks. The bounded-drift assumption underlies the clock synchronization problem for distributed systems (see, for example, the survey by Schneider [Sch87]). The dichotomy of discrete transitions (jumps) versus continuous transitions (flows) was introduced by Manna, Maler, and Pnueli [MMP92]. The use of flow-invariants to model hybrid systems in general [ACH+95], and drifting clocks in particular [AHH96], was developed in the framework of hybrid automata (see, for example, the survey [Hen96]). Open hybrid systems were first discussed by Lynch, Segala, Vaandrager, and Weinberg [LSVW96]. A detailed presentation of our model for open hybrid systems, complete with uncontrolled continuous variables and a discussion of executable differential constraints, can be found in [AH97]. Recent workshop proceedings provide an excellent overview of applications for hybrid systems [GNRR93, ANKS95, AHS96]. The thermostat example is due to Nicollin, Sifakis, and Yovine [NSY93].

7 Conclusion

We illustrated the clock paradigm for specifying timed systems. The clock paradigm permits a clean and natural extension of the state-transition semantics from discrete reactive systems to real-time systems: lower time bounds on transitions, which specify when transitions may happen, are included in the next-state condition of a system; upper time bounds, which specify when transitions must happen, are described by the time-progress condition, which replaces the fairness condition of a system. The dichotomy between instantaneous transitions (jumps) and time transitions (delays) permits the extension of the interleaving view of concurrency from untimed to timed system: the parallel composition of clock systems interleaves jumps while overlaying delays. Furthermore, the clock paradigm can be readily generalized for specifying hybrid systems, with variables that change continuously over time.

For dense-time systems, which may exhibit Zeno phenomena, executability becomes a subtle issue. Hence we developed a formal semantics for clock systems in order to understand their operational behavior. This semantics can be used also for reasoning about functional and timing properties of clock systems. The two appendices below discuss, very briefly, the deductive and the algorithmic approach to real-time reasoning using the clock paradigm.

Appendix A: Proving Properties of Real-time Systems

(by Rajeev Alur, Thomas A. Henzinger, and Peter W. Kopke)

We formalize system requirements as sets of behaviors. Intuitively, a requirement is the set of those behaviors that satisfy the requirement. For instance, for a discrete integer variable m, the requirement "m is always even" is the set of behaviors that contain only states that map m to even numbers; the requirement "m is sometimes odd" is the set of behaviors that contain some state that maps m to an odd number. Requirements that involve continuous variables are more subtle. For instance, for a clock x, the requirement "x is sometimes equal to 1" is not simply the set of behaviors that contain some state that maps x to 1, but the set of behaviors that contain or pass some state that maps x to 1 (where a state σ is *passed* by the behavior $\bar{\sigma}$ if σ is passed by some delay of $\bar{\sigma}$). For the railroad crossing from Section 6.3, then, the safety requirement "Whenever the train is within 10 meters of the crossing, the gate is closed" is the set of behaviors that do not contain or pass a state σ for which the gate is open and $-10 \leq \sigma(d) \leq 10$.

A set of behaviors is called a *property*. Let $\mathcal{S}$ be a closed real-time system.[13] We write $[\![\mathcal{S}]\!]$ for the set of possible behaviors of $\mathcal{S}$, and $[\mathcal{S}]$ for the set of possible and convergent behaviors of $\mathcal{S}$. Then $\mathcal{S}$ is nonzeno iff $[\![\mathcal{S}]\!]$ is dense in $[\mathcal{S}]$ with respect to the Cantor metric on infinite sequences [Hen92].[14] The real-time system $\mathcal{S}$ *satisfies* the property Π if $[\![\mathcal{S}]\!] \subseteq \Pi$. The real-time system $\mathcal{S}$ *pre-satisfies* the property Π if $[\mathcal{S}] \subseteq \Pi$. Clearly, pre-satisfaction is a sufficient but not a necessary condition for satisfaction. Any sound and complete proof calculus for discrete systems gives rise to a sound and complete calculus for proving if a real-time system pre-satisfies a property from a given class. Such a calculus, therefore, is sound but may not be complete for proving if a real-time system satisfies a property from that class.

Properties can be described, for example, by formulas of linear temporal logic [MP92]. For a state predicate ϕ, the temporal formula $\Box\phi$ describes the set of behaviors that contain (and pass) only states for which ϕ is true, and the temporal formula $\Diamond\phi$ describes the set of behaviors that contain (or pass) some state for which ϕ is true. Since state predicates may contain clocks, temporal formulas can express timing requirements. For example, the requirement "The real-time while program *RtUpDown3* terminates within 15 to 70 time units" is specified by the temporal formula

$$\Diamond(pc = \ell_{19} \wedge 15 \leq x \leq 70),$$

because the clock x of the program *RtUpDown3* measures the total elapsed

[13] For the verification of an open system, consider its closure.

[14] In the Cantor metric, two infinite sequences are close if they have long prefixes in common: the longer the longest common prefix, the closer the two sequences.

time. We write $\mathcal{S} \models \vartheta$ if the real-time system $\mathcal{S}$ satisfies the property that is described by the temporal formula ϑ. Based on [HK94], we present a deductive method for proving assertions of the form $\mathcal{S} \models \vartheta$. The method is interesting, because it relies on considerable technology transfer from the discrete case. The two extensions that are necessary for proving properties of real-time systems concern (1) the density of the time domain, which causes passed states to be of interest,[15] and (2) the divergence of time.[16]

Safety properties

A *safety property* is a closed set of behaviors [ADS86].[17] For instance, the property "m is always even" is a safety property; the property "m is sometimes odd" is not. Every temporal formula of the form $\Box\phi$, for a state predicate ϕ, describes a safety property. By elementary topology, for nonzeno real-time systems, the pre-satisfaction of safety properties coincides with satisfaction [Hen92]. Hence, for proving safety properties of nonzeno real-time systems, we can use any calculus for proving safety properties of discrete systems.

Consider, for instance, the invariance rule SAFE of Manna and Pnueli [MP95], whose translation into our framework is shown in Figure 23. The rule SAFE is sound and complete for proving assertions of the form $\mathcal{S} \models \Box\phi$, for a nonzeno real-time system $\mathcal{S} = (\phi^I, \psi^N, \chi^T)$ and a state predicate ϕ. In the figure, we use the following notation. We write $\models \phi$ if the state predicate ϕ is valid—i.e., if ϕ is true for all states (this can be checked by predicate logic). The formula $V' = V + \delta$, for a nonnegative real δ, stands for the conjunction

$$(\bigwedge_{v \in D} v' = v) \;\wedge\; (\bigwedge_{x \in C} x' = x + \delta),$$

where D is the set of discrete variables and C is the set of clocks. For a state predicate ϕ, the state predicate $\phi + \delta$ is obtained by replacing every free occurrence of each clock x in ϕ by $x + \delta$. For a state predicate ϕ, the state predicate ϕ' is obtained by replacing every free occurrence of each variable v in ϕ by the primed version v'. The auxiliary state predicate φ of the rule SAFE is called an *inductive invariant*. The challenging part of a safety proof is the construction of a suitable inductive invariant.

We use the rule SAFE to show that the executable real-time while program *RtUpDown3* terminates within 15 to 70 time units. For this purpose, we need to prove the assertion

$$\mathit{RtUpDown3} \models \Diamond(pc = \ell_{19} \wedge 15 \leq x \leq 70).$$

[15]It should be noted that the density of the time domain also renders unappealing the next-time operator of temporal logic.

[16]As discussed below, the divergence of time becomes an issue only for liveness properties.

[17]Closure in the Cantor metric is exactly limit closure, as defined in Section 2.4.

$$\begin{array}{l}
\models \phi^I \Rightarrow \varphi \\
\models (\varphi \wedge \psi^N) \Rightarrow \varphi' \\
\models (\varphi \wedge (\exists \delta \mid \delta \geq 0 : V' = V + \delta \wedge (\forall \epsilon \mid 0 \leq \epsilon < \delta : \chi^T + \epsilon))) \Rightarrow \varphi' \\
\models \varphi \Rightarrow \phi \\
\hline
\mathcal{S} \models \Box \phi
\end{array}$$

Figure 23: Rule SAFE for proving invariants of nonzeno real-time systems

Every such time-bounded $\Diamond$ property is a safety property (this folk theorem is formalized in [Hen92]). In particular, over the possible behaviors of *RtUpDown3*, our proof obligation is equivalent to the conjunction of the two safety assertions

$$\mathit{RtUpDown3} \models \Box(pc = \ell_{19} \Rightarrow x \geq 15)$$

and

$$\mathit{RtUpDown3} \models \Box(x > 70 \Rightarrow pc \in \{\ell_0, \ell_1, \ell_{19}\}).$$

The first $\Box$ property specifies the lower bound on termination; it states that the program will not terminate before 15 time units. The second $\Box$ property specifies the upper bound; it states that the program will terminate within 70 time units. This is because, as time diverges, eventually the value of x will be greater than 70. When this happens, according to the second $\Box$ property, the program will have terminated (assuming the technical proviso that the initial value of x is not greater than 70). Both $\Box$ properties can be concluded from the rule SAFE with the inductive invariant shown in Figure 24. All premises follow by predicate logic.

Liveness properties

While nonzenoness enables the direct application of discrete techniques to the verification of safety properties for real-time systems, liveness properties require a different approach. Our solution uses an auxiliary clock *tick* to identify the divergent behaviors. Let $\mathcal{S} = (\phi^I, \psi^N, \chi^T)$ be a closed real-time system. We write $\mathcal{S} \vdash \vartheta$ if $\mathcal{S}$ pre-satisfies the property that is described by the temporal formula ϑ. We define a new real-time system $\mathcal{S}_{tick}$ by adding to the clocks in V a new clock variable, called *tick*, adding the conjunct $tick = 0$ to ϕ^I, and adding the jump schema $(tick = 1 \wedge tick' = 0)_V$ as a disjunct to ψ^N. Since the possible behaviors of $\mathcal{S}$ are closed under timed stuttering, it follows that for every temporal formula ϑ,

$$\mathcal{S} \models \vartheta \quad \text{iff} \quad \mathcal{S}_{tick} \vdash (\Box\Diamond(tick = 0) \wedge \Box\Diamond(tick = 1)) \Rightarrow \vartheta.$$

$$
\begin{array}{lll}
& \bigvee_{0\le i\le 19} pc = \ell_i & \\
\wedge & pc = \ell_1 & \Rightarrow\ n = 1 \\
\wedge & pc = \ell_2 & \Rightarrow\ n = 1 \wedge x = 0 \\
\wedge & pc \in \{\ell_3, \ell_4\} & \Rightarrow\ n = 1 \wedge x = z \wedge z \le 5 \\
\wedge & pc = \ell_5 & \Rightarrow\ 1 \le n \le 11 \wedge \\
& & \quad (n = 1 \Rightarrow 1 \le x \le 5) \wedge \\
& & \quad (n = 2 \Rightarrow x \le 5 + y) \wedge \\
& & \quad (n \ge 2 \Rightarrow n - 1 + y \le x \le 10 + y \wedge 1 \le y \le 5) \\
\wedge & pc = \ell_6 & \Rightarrow\ n \ge 1 \wedge n \le x \le 10 \wedge \\
& & \quad (n = 1 \Rightarrow x \le 5) \\
\wedge & pc = \ell_7 & \Rightarrow\ n \ge 2 \wedge n - 1 \le x \le 10 \wedge \\
& & \quad (n = 2 \Rightarrow x \le 5) \\
\wedge & pc \in \{\ell_8, \ell_9, \ell_{10}, \ell_{11}\} & \Rightarrow\ n \ge 2 \wedge n - 1 + y \le x \le 10 + y \wedge y \le 5 \wedge \\
& & \quad (n = 2 \Rightarrow x \le 5 + y) \\
\wedge & pc \in \{\ell_{12}, \ell_{13}\} & \Rightarrow\ 3 \le n \le 11 \wedge x \le 10 + y \wedge y \le 5 \\
\wedge & pc = \ell_{14} & \Rightarrow\ n \ge 0 \wedge 15 - n \le x \le 70 - 5n \\
\wedge & pc = \ell_{15} & \Rightarrow\ n > 0 \wedge 15 - n \le x \le 70 - 5n \\
\wedge & pc = \ell_{16} & \Rightarrow\ n \ge 0 \wedge 14 - n \le x \le 65 - 5n \\
\wedge & pc \in \{\ell_{17}, \ell_{18}\} & \Rightarrow\ n \ge 0 \wedge 14 - n + z \le x \le 65 - n + z \wedge z \le 5 \\
\wedge & pc = \ell_{19} & \Rightarrow\ x \ge 15
\end{array}
$$

Figure 24: Inductive invariant φ for the real-time while program *RtUpDown3*

We may use discrete temporal reasoning to prove the latter. In this way, any sound and complete calculus for proving temporal properties of discrete systems provides a sound and complete calculus for proving properties of real-time systems. The main drawback of this approach is its complexity: to prove a $\Diamond$ property of a real-time system, we must use a rule for proving $\Box\Diamond$ properties of discrete systems. Such rules can be found, for example, in [MP84].

Appendix B: Automatic Analysis of Real-time Systems

In Appendix A, we outlined a methodology for establishing properties of real-time systems using mathematical proof. Similar proof methods can be found in [dBHdRR92, MKP96] and are supported by deductive verification tools such as STeP [BBC$^+$96]. An alternative, more limited but more automated, approach employs algorithms for establishing properties of real-time systems. The algorithmic approach, often referred to as *model checking*, has proved successful for the analysis of large-scale untimed systems [CGL94], and has recently been extended to real-time systems. Here we only attempt to direct the reader to some of the relevant literature.

System specification: timed automata and rectangular automata

Our definition of real-time system is too general to permit automatic analysis, and two restrictions are necessary. A real-time system is *finitary* if (1) all discrete variables are boolean, and (2) all clock variables occur, in initial, next-state, and time-progress conditions, only within atomic constraints of the form $x \sim c$ or $x' = x$ or $x' = c$, where x is a clock, c is a rational constant, and $\sim \in \{<, =, >\}$ is a comparison operator. Thus, in a finitary real-time system, clock values can be compared with constant values and reset to constant values. The discrete aspect of a closed finitary real-time system can be represented by a graph whose vertices encode values for the boolean variables. The representation of a closed finitary real-time system as a graph annotated with clock constraints is called a *timed automaton* [AD94].[18] Since clocks range over the nonnegative reals, every nontrivial timed automaton has infinitely many states. However, in [AD94] it is shown that for every timed automaton we can construct a bisimilar finite-state system (where bisimilarity is defined with respect to the event alphabet that consists of the edges of the timed automaton). This observation forms the basis of all verification algorithms for timed automata.

If the clocks of a finitary real-time system are permitted to drift with constant, rational drift bounds, we obtain a *finitary drifting-clock system*. The representation of a closed finitary drifting-clock system as a graph annotated with constraints on drifting clocks is called an *initialized rectangular automaton* [HKPV95].[19] There are initialized rectangular automata that are not bisimilar to any finite-state system [Hen95]. However, in [HKPV95] it is shown that for every initialized rectangular automaton we can construct a finite-state system with the same language (over the alphabet of edges of the rectangular automaton). This observation is central to the algorithmic verification of rectangular automata.

Requirement specification: timed automata and real-time logics

Two popular specification languages for the algorithmic verification of untimed systems are finite automata and propositional temporal logics. In order to specify timing constraints, these languages can be extended by adding clock variables.[20] If we judiciously add clocks to finite automata, we obtain the timed automata (TA) discussed above [AD94]; from propositional linear temporal logic, we obtain the real-time logic TPTL [AH94]; from the propositional

[18]Timed automata with a time-progress (delay-invariant) condition instead of a fairness (acceptance) condition are sometimes referred to as *timed safety automata* [HNSY94].

[19]In a noninitialized rectangular automaton, the drift bounds of a clock may vary.

[20]In addition to finite automata and temporal logics, also process algebras have been enriched with clock variables [NS91, LV92, DB96].

branching-time logic CTL, we obtain the real-time logic TCTL [ACD93]. While we have seen, in Appendix A, that certain timing requirements of a system can be specified using references to the clocks of the system, for other requirements it may be necessary (or convenient) to introduce new clocks in the specification. Within temporal formulas, the scope of these *specification clocks* is determined by quantifiers that initialize the clocks. For example, the TPTL formula

$$\Box z := 0.\,(\phi_1 \Rightarrow \Diamond(\phi_2 \land z \leq 5))$$

contains a specification clock z that is bound by the quantifier "$z := 0$," which initializes the clock to 0. The formula asserts that along every possible behavior of a system, each state for which ϕ_1 is true must be followed within 5 time units by a state for which ϕ_2 is true. A summary of automata-based and logic-based real-time specification languages can be found in [AH92].

Verification algorithms

We have two fundamental decidability results for the verification of timed and hybrid systems: for finitary real-time systems, TCTL specifications can be checked [ACD93]; for finitary drifting-clock systems, TA specifications without specification clocks and TPTL specifications without specification clocks can be checked [HKPV95]. These results depend on the finitary bisimilarity relations of finitary real-time systems and on the finitary language-equivalence relations of finitary drifting-clock systems, respectively. Based on the decidability results, several verification algorithms for finitary real-time systems have been implemented [DW95, AK96, BLL$^+$96, DOTY96].

The efficiency of these tools has been improved along many dimensions, of which we mention two. First, an incremental approach is useful for coping with the high cost of analyzing a large number of clocks [AIKY95]: initially all clock constraints are ignored; then the clock constraints are added one by one, as needed to prove a given specification. Second, a symbolic approach avoids the expensive construction of the bisimilarity quotient (the so-called "region graph") of a finitary real-time system [HNSY94]. Suppose, for instance, that we wish to prove a system invariant. For this purpose, we need to compute the set of all states that appear along possible behaviors of the system. The state predicate that characterizes this set of reachable states can be computed by symbolic execution of the system. The efficiency of the computation depends on the representation of state predicates. In the case of finite-state systems, the involved state predicates are boolean expressions, and binary decision diagrams have turned out to provide a cost-effective representation [McM93]. For finitary real-time systems, the involved state predicates contain clock constraints (of the form $x - y \sim c$), and difference-bounds matrices have been the data structure of choice [Dil89a] (alternative representations are being investigated, for example, in [ABK$^+$97]).

Both decidability results for real-time verification are sharp: for TA and TPTL specifications with specification clocks, the validity problem (which is equal to the verification problem over the system with all possible behaviors) is undecidable [AD94, AH94]; for several generalizations of finitary real-time and drifting-clock systems, such as timed automata with a single stopwatch, simple reachability questions are undecidable [HKPV95]. These undecidability results have led researchers to consider several special cases: in [AFH94, RS97] the use of specification clocks is restricted to refer only to immediately preceding or succeeding occurrences of events; in [AFH96] the use of specification clocks is restricted to refer only to the approximate times of events; in [AH94, HK97] the durations of all delays are restricted to be (observed as) integers. The verification problem for finitary drifting-clock systems can be solved under any of these restrictions.

Perhaps most encouraging is the observation that even the verification problem for general hybrid systems, while undecidable, can be solved in many instances of practical interest using the symbolic approach (see, for example, [HW95, HWT96]). In particular, symbolic model checkers have been implemented for hybrid systems whose continuous dynamics are approximated by piecewise-linear envelopes [HHWT97].

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

References

[ABK+97] E. Asarin, M. Bozga, A. Kerbrat, O. Maler, A. Pnueli, and A. Rasse. Data structures for the verification of timed automata. In O. Maler, editor, *HART 97: Hybrid and Real-time Systems*, Lecture Notes in Computer Science 1201, pages 346–360. Springer-Verlag, 1997.

[ACD93] R. Alur, C. Courcoubetis, and D.L. Dill. Model checking in dense real time. *Information and Computation*, 104(1):2–34, 1993.

[ACH+95] R. Alur, C. Courcoubetis, N. Halbwachs, T.A. Henzinger, P.-H. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. The algorithmic analysis of hybrid systems. *Theoretical Computer Science*, 138:3–34, 1995.

[AD94] R. Alur and D.L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.

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

[AFH94] R. Alur, L. Fix, and T.A. Henzinger. A determinizable class of timed automata. In D.L. Dill, editor, *CAV 94: Computer-aided Verification*, Lecture Notes in Computer Science 818, pages 1–13. Springer-Verlag, 1994.

[AFH96] R. Alur, T. Feder, and T.A. Henzinger. The benefits of relaxing punctuality. *Journal of the ACM*, 43(1):116–146, 1996.

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

[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.

[AH94] R. Alur and T.A. Henzinger. A really temporal logic. *Journal of the ACM*, 41(1):181–204, 1994.

[AH96] R. Alur and T.A. Henzinger. Reactive modules. In *Proceedings of the 11th Annual Symposium on Logic in Computer Science*, pages 207–218. IEEE Computer Society Press, 1996.

[AH97] R. Alur and T.A. Henzinger. Modularity for timed and hybrid systems. In A. Mazurkiewicz and J. Winkowski, editors, *CONCUR 97: Concurrency Theory*, Lecture Notes in Computer Science 1243, pages 74–88. Springer-Verlag, 1997.

[AHH96] R. Alur, T.A. Henzinger, and P.-H. Ho. Automatic symbolic verification of embedded systems. *IEEE Transactions on Software Engineering*, 22(3):181–201, 1996.

[AHS96] R. Alur, T.A. Henzinger, and E.D. Sontag, editors. *Hybrid Systems III: Verification and Control.* Lecture Notes in Computer Science 1066. Springer-Verlag, 1996.

[AIKY95] R. Alur, A. Itai, R.P. Kurshan, and M. Yannakakis. Timing verification by successive approximation. *Information and Computation*, 118(1):142–157, 1995.

[AK96] R. Alur and R.P. Kurshan. Timing analysis in COSPAN. In R. Alur, T.A. Henzinger, and E.D. Sontag, editors, *Hybrid Systems III*, Lecture Notes in Computer Science 1066, pages 220–231. Springer-Verlag, 1996.

[AL93] M. Abadi and L. Lamport. Composing specifications. *ACM Transactions on Programming Languages and Systems*, 15(1):73–132, 1993.

[AL94] M. Abadi and L. Lamport. An old-fashioned recipe for real time. *ACM Transactions on Programming Languages and Systems*, 16(5):1543–1571, 1994.

[ANKS95] P. Antsaklis, A. Nerode, W. Kohn, and S. Sastry, editors. *Hybrid Systems II.* Lecture Notes in Computer Science 999. Springer-Verlag, 1995.

[BBC$^+$96] N. Bjørner, A. Browne, E. Chang, M. Colón, A. Kapur, Z. Manna, H.B. Sipma, and T.E. Uribe. STeP: deductive-algorithmic verification of reactive and real-time systems. In R. Alur and T.A. Henzinger, editors, *CAV 96: Computer-aided Verification*, Lecture Notes in Computer Science 1102, pages 415–418. Springer-Verlag, 1996.

[BG88] G. Berry and G. Gonthier. The synchronous programming language Esterel: Design, semantics, implementation. Technical Report 842, INRIA, 1988.

[BLL$^+$96] J. Bengtsson, K.G. Larsen, F. Larsson, P. Pettersson, and W. Yi. Uppaal: a tool-suite for automatic verification of real-time systems. In R. Alur, T.A. Henzinger, and E.D. Sontag, editors, *Hybrid Systems III*, Lecture Notes in Computer Science 1066, pages 232–243. Springer-Verlag, 1996.

[CE81] E.M. Clarke and E.A. Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Workshop on Logic of Programs*, Lecture Notes in Computer Science 131, pages 52–71. Springer-Verlag, 1981.

[CGL94] E.M. Clarke, O. Grumberg, and D.E. Long. Verification tools for finite-state concurrent systems. In *A Decade of Concurrency: Reflections and Perspectives*, Lecture Notes in Computer Science 803. Springer-Verlag, 1994.

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

[DB96] P.A. D'Argenio and E. Brinksma. A calculus for timed automata. In B. Jonsson and J. Parrow, editors, *FTRTFT 96: Formal Techniques in Real-time and Fault-tolerant Systems*, Lecture Notes in Computer Science 1135, pages 110–129. Springer-Verlag, 1996.

[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.

[Dij65] E.W. Dijkstra. Solution of a problem in concurrent programming control. *Communications of the ACM*, 8(9):569, 1965.

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

[Dil89a] D.L. Dill. Timing assumptions and verification of finite-state concurrent systems. In J. Sifakis, editor, *CAV 89: Automatic Verification Methods for Finite-state Systems*, Lecture Notes in Computer Science 407, pages 197–212. Springer-Verlag, 1989.

[Dil89b] D.L. Dill. *Trace Theory for Automatic Hierarchical Verification of Speed-independent Circuits.* The MIT Press, 1989.

[DOTY96] C. Daws, A. Olivero, S. Tripakis, and S. Yovine. The tool KRONOS. In R. Alur, T.A. Henzinger, and E.D. Sontag, editors, *Hybrid Systems III*, Lecture Notes in Computer Science 1066, pages 208–219. Springer-Verlag, 1996.

[DW95] D.L. Dill and H. Wong-Toi. Verification of real-time systems by successive over- and underapproximation. In P. Wolper, editor, *CAV 95: Computer-aided Verification*, Lecture Notes in Computer Science 939, pages 409–422. Springer-Verlag, 1995.

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

[GNRR93] R.L. Grossman, A. Nerode, A.P. Ravn, and H. Rischel, editors. *Hybrid Systems I.* Lecture Notes in Computer Science 736. Springer-Verlag, 1993.

[GSSAL94] R. Gawlick, R. Segala, J.F. Sogaard-Andersen, and N.A. Lynch. Liveness in timed and untimed systems. In S. Abiteboul and E. Shamir, editors, *ICALP 94: Automata, Languages, and Programming*, Lecture Notes in Computer Science 820, pages 166–177. Springer-Verlag, 1994.

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

[Hen92] T.A. Henzinger. Sooner is safer than later. *Information Processing Letters*, 43(3):135–141, 1992.

[Hen95] T.A. Henzinger. Hybrid automata with finite bisimulations. In Z. Fülöp and F. Gécseg, editors, *ICALP 95: Automata, Languages, and Programming*, Lecture Notes in Computer Science 944, pages 324–335. Springer-Verlag, 1995.

[Hen96] T.A. Henzinger. The theory of hybrid automata. In *Proceedings of the 11th Annual Symposium on Logic in Computer Science*, pages 278–292. IEEE Computer Society Press, 1996.

[HHWT97] T.A. Henzinger, P.-H. Ho, and H. Wong-Toi. HYTECH: a model checker for hybrid systems. In O. Grumberg, editor, *CAV 97: Computer-aided Verification*, Lecture Notes in Computer Science 1254, pages 460–463. Springer-Verlag, 1997.

[HJL93] C.L. Heitmeyer, R.D. Jeffords, and B.G. Labaw. A benchmark for comparing different approaches for specifying and verifying real-time systems. In *Proceedings of the Tenth International Workshop on Real-time Operating Systems and Software*, 1993.

[HK94] T.A. Henzinger and P.W. Kopke. Verification methods for the divergent runs of clock systems. In H. Langmaack, W.-P. de Roever, and J. Vytopil, editors, *FTRTFT 94: Formal Techniques in Real-time and Fault-tolerant Systems*, Lecture Notes in Computer Science 863, pages 351–372. Springer-Verlag, 1994.

[HK97] T.A. Henzinger and P.W. Kopke. Discrete-time control for rectangular hybrid automata. In P. Degano, R. Gorrieri, and A. Marchetti-Spaccamela, editors, *ICALP 97: Automata, Languages, and Programming*, Lecture Notes in Computer Science 1256, pages 582–593. Springer-Verlag, 1997.

[HKPV95] T.A. Henzinger, P.W. Kopke, A. Puri, and P. Varaiya. What's decidable about hybrid automata? In *Proceedings of the 27th Annual Symposium on Theory of Computing*, pages 373–382. ACM Press, 1995.

[HKWT95] T.A. Henzinger, P.W. Kopke, and H. Wong-Toi. The expressive power of clocks. In Z. Fülöp and F. Gécseg, editors, *ICALP 95: Automata, Languages, and Programming*, Lecture Notes in Computer Science 944, pages 417–428. Springer-Verlag, 1995.

[HMP94] T.A. Henzinger, Z. Manna, and A. Pnueli. Temporal proof methodologies for timed transition systems. *Information and Computation*, 112(2):273–337, 1994.

[HNSY94] T.A. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. Symbolic model checking for real-time systems. *Information and Computation*, 111(2):193–244, 1994.

[HW95] P.-H. Ho and H. Wong-Toi. Automated analysis of an audio control protocol. In P. Wolper, editor, *CAV 95: Computer-aided Verification*, Lecture Notes in Computer Science 939, pages 381–394. Springer-Verlag, 1995.

[HWT96] T.A. Henzinger and H. Wong-Toi. Using HyTech to synthesize control parameters for a steam boiler. In J.-R. Abrial, E. Börger, and H. Langmaack, editors, *Formal Methods for Industrial Applications: Specifying and Programming the Steam Boiler Control*, Lecture Notes in Computer Science 1165, pages 265–282. Springer-Verlag, 1996.

[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, 1983.

[Lam94] L. Lamport. The Temporal Logic of Actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, 1994.

[LSVW96] N.A. Lynch, R. Segala, F. Vaandrager, and H.B. Weinberg. Hybrid I/O Automata. In R. Alur, T.A. Henzinger, and E.D. Sontag, editors, *Hybrid Systems III*, Lecture Notes in Computer Science 1066, pages 496–510. Springer-Verlag, 1996.

[LV92] N.A. Lynch and F. Vaandrager. Action transducers and timed automata. In R.J. Cleaveland, editor, *CONCUR 92: Theories of Concurrency*, Lecture Notes in Computer Science 630, pages 436–455. Springer-Verlag, 1992.

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

[MKP96] Z. Manna, Y. Kesten, and A. Pnueli. Verifying clocked transition systems. In R. Alur, T.A. Henzinger, and E.D. Sontag, editors, *Hybrid Systems III*, Lecture Notes in Computer Science 1066, pages 13–40. Springer-Verlag, 1996.

[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.

[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.

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

[MP95] Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems: Safety.* Springer-Verlag, 1995.

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

[NSY93] X. Nicollin, J. Sifakis, and S. Yovine. From ATP to timed graphs and hybrid systems. *Acta Informatica*, 30:181–202, 1993.

[RS97] J.-F. Raskin and P.-Y. Schobbens. State clock logic: a decidable real-time logic. In O. Maler, editor, *HART 97: Hybrid and Real-time Systems*, Lecture Notes in Computer Science 1201, pages 33–47. Springer-Verlag, 1997.

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