Modularity for Timed and Hybrid Systems[⋆,⋆⋆]

Rajeev Alur Thomas A. Henzinger

EECS Department, University of California, Berkeley, CA 94720-1770, U.S.A.
Email: {alur,tah}@eecs.berkeley.edu

Abstract. In a trace-based world, the modular specification, verification, and control of live systems require each module to be *receptive*; that is, each module must be able to meet its liveness assumptions no matter how the other modules behave. In a real-time world, liveness is automatically present in the form of *diverging time*. The receptiveness condition, then, translates to the requirement that a module must be able to let time diverge no matter how the environment behaves. We study the receptiveness condition for real-time systems by extending the model of *reactive modules* to timed and hybrid modules. We define the receptiveness of such a module as the existence of a winning strategy in a game of the module against its environment. By solving the game on region graphs, we present an (optimal) EXPTIME algorithm for checking the receptiveness of propositional timed modules. By giving a fixpoint characterization of the game, we present a symbolic procedure for checking the receptiveness of linear hybrid modules. Finally, we present an assume-guarantee principle for reasoning about timed and hybrid modules, and a method for synthesizing receptive controllers of timed and hybrid modules.

1 Introduction

Over the past decade, much research has focused on the modeling and verification of *timed* systems [12], which have hard real-time constraints, and *hybrid* systems [16, 11, 7], which contain both discrete and continuous components. Most of this research[3] has emphasized *closed* systems, which can be considered in isolation, and neglected *open* systems, whose behavior is influenced by the behavior of an external environment.[4] For example, the ubiquitous train-gate system from the real-time literature is usually studied as a compound closed system, and no properties are proved about how the train behaves relative to *any* gate, or about how the gate behaves relative to any train. This is because if the components of a system interact with each other, then each component by itself must be treated as an open system.

One reason for the lack of emphasis on open real-time systems may be that a proper formalization is far from "obvious." When time is of the essence, a liveness assumption enters automatically, namely, the assumption that no system should be able to prevent time from diverging. A physical discrete system may "stop" time finitely often within any finite time span, to perform its actions, but it may not perform infinitely many actions within a finite amount of time. Consider, for example, the classical zeno paradox: a discrete observer that looks at a runner at times $1/2$, $3/4$, $7/8$,

⋆ To appear in the *Proceedings of the 8th International Conference on Concurrency Theory* (CONCUR 97), Lecture Notes in Computer Science, Springer-Verlag, 1997.

⋆⋆ 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.

3 Including most previous work by the authors.

4 A notable exception is the work on timed and hybrid I/O automata [17, 23].

etc., will never observe the runner cross the finishing line. Indeed, applied to a timed system that represents the runner and to a control objective that the state *finished* is never entered, classical control methods yield a controller that "prevents" the runner from finishing by issuing infinitely many control actions [22, 24]. Such a controller, of course, cannot be realized physically.[5]

For *live closed* systems, the appropriate condition is *machine closure* [3]: every finite run can be extended to an infinite live run. In the case of real-time systems, the machine-closure condition is usually called *nonzenoness* [9, 18]: every finite trajectory can be extended to a divergent trajectory; that is, no matter what the system does, there is always a possibility for time to diverge. Since machine closure is not closed under parallel composition, for *live open* systems, the appropriate condition becomes trickier. To see this, consider a module P that issues an output at time 0, then waits for an input; if the input arrives at time δ_1, the module issues the next output at time $\delta_1 + 1/2$ and waits for another input; if the second input arrives at time δ_2, the next output is issued at time $\delta_2 + 1/4$; etc. Second, consider a module Q that waits for an input and once the input arrives at time ϵ_1, the module issues an output at time $\epsilon_1 + 1/2$; then the module waits for another input, and if the second input arrives at time ϵ_2, the next output is issued at time $\epsilon_2 + 1/4$; etc. While each module, P and Q, by itself is nonzeno (there is always a possibility for time to diverge), the composition $P||Q$ generates events at times 0, 1/2, 1, 5/4, 3/2, 13/8, etc., and thus prevents time from progressing past 2.

A proper condition for live open systems, therefore, must take into account adversarial, rather than cooperative, environments. Such a condition, called *receptiveness*, is best formulated as a game [13, 8, 17]: a module is receptive iff in a two-player game against the environment, the module has a strategy to generate an infinite live run. Then, the composition of two receptive modules is again receptive (and machine-closed). In the case of real-time systems, a receptiveness game was first defined for I/O automata [17]. Here, we define and algorithmically analyze a receptiveness game for timed systems that are modeled as discrete systems with clock variables, à la *timed automata* [2], and for hybrid systems that are modeled as discrete systems with continuous variables, à la *hybrid automata* [1].

In each move of our receptiveness game for timed systems, the module proposes to let time $\delta \geq 0$ pass, and the environment proposes to let time $\delta' \geq 0$ pass (if the module wants to update the discrete state, then $\delta = 0$, and the same is true for the environment). If either $\delta = 0$ or $\delta' = 0$, then the module and the environment update the discrete state, according to a protocol for discrete systems, and no time elapses. If both $\delta > 0$ and $\delta' > 0$, then time $min(\delta, \delta')$ elapses, and the discrete state stays unchanged. If $\delta \leq \delta'$, the move is charged to the module; otherwise, the move is charged to the environment. The module is called *receptive* iff in this game, it has a strategy that will never generate a convergent trajectory unless all but finitely many moves are charged to the environment.

Since timed and hybrid automata are models for closed systems, we extend the open-system model of *reactive modules* [5] with clock variables, to obtain *timed modules*, and with continuous variables, to obtain *hybrid modules*. The formalism of reactive modules was developed for specifying highly heterogeneous systems, with mixed hardware and software components, and mixed synchronous and asynchronous interactions between components. We continue this theme of heterogeneity by providing reactive modules with mechanisms for specifying mixed timing, as well as mixed discrete-continuous behavior.

[5] To circumvent such absurdities, [10] introduce an "anti-Zeno" constant δ for controllers, which ensures that a controller cannot act more than once every δ time units. This requirement, of course, may cause us to fail finding a controller for a perfectly controllable situation.

By defining and studying the receptiveness game for timed and hybrid modules, we accomplish four results. First, we extend the assume-guarantee principle for modular reasoning from reactive modules to timed and hybrid modules; the soundness of the principle depends on the receptiveness of all participating modules.[6] Second, by giving a fixpoint characterization of the receptiveness game, we develop a symbolic procedure for checking the receptiveness of timed and linear hybrid modules (which are closely related to linear hybrid automata [6]); this procedure can be easily implemented in existing tools such as KRONOS [14] and HYTECH [19]. Third, by reducing the receptiveness game to a coBüchi game on finite region graphs, we give an exponential algorithm for checking the receptiveness of propositional timed modules (which are timed modules with finitely many discrete states); the algorithm is optimal, as we show the problem to be complete for EXPTIME. Fourth, we address the controller-synthesis problem for timed and hybrid modules, and show how classical methods (both symbolic and enumerative) can be used for synthesizing controllers that are guaranteed to be receptive (provided such a controller exists).

As is to be expected, "open" problems about timed systems (receptiveness checking, control) are theoretically harder than the corresponding "closed" problems (nonzenoness checking, verification); while the latter are generally complete for PSPACE, the former are complete for EXPTIME. In practice, both open and closed problems can be solved using symbolic fixpoint computations. It should also be noted that while throughout, we consider the dense time domain of the nonnegative reals, our motivation and our conclusions apply equally to the digital-clock model, where all time-stamps are truncated to integer values, and a system may prevent time from diverging by insisting on infinitely many moves of delay 0.

2 Timed Modules

We extend the model of *reactive modules* [5] to allow for the specification of real-time behavior.

Discrete variables vs. clock variables. A timed module P has a finite set of typed variables, denoted X_P. Some of the variables are updated in a discrete fashion, and the other variables change continuously when time elapses. Accordingly, the set X_P of module variables is partitioned into two sets, the set $discX_P$ of *discrete variables*, and the set $clkX_P$ of *clock variables*. The type of all clock variables is $\mathbb{R}$. A *state* of P is a valuation for the variables in X_P. Events, such as the sending of messages, are modeled by toggling discrete variables of type $\mathbb{B}$.

System vs. environment. The module P represents a system that interacts with an environment. Some of the variables in X_P are updated by the module, and the other variables in X_P are updated by the environment. Accordingly, the set X_P is partitioned into two sets, the set $ctrX_P$ of *controlled variables*, and the set $extlX_P$ of *external variables*. A controlled variable may be either discrete or a clock, and an external variable may be either discrete or a clock.

States vs. observations. Not all controlled variables of the module P are visible to the environment. Accordingly, the set $ctrX_P$ is partitioned further into two sets, the set $privX_P$ of *private variables*, and the set $intfX_P$ of *interface variables*. The interface variables and the external variables are *observable*, denoted $obsX_P$. An *observation* of P is a valuation for the variables in $obsX_P$. The observation of a state s, then, is the projection $s[obsX_P]$ of s to the observable variables.

Update rounds vs. time rounds. The module P proceeds in a sequence of rounds. The first round is an *initialization round*, during which the variables in X_P are initialized. Each subsequent round

[6] A more specialized assume-guarantee principle for timed systems was presented in [27]. Its soundness follows from syntactic restrictions that ensure receptiveness.

is either an update round or a time round. During each *update round*, the variables in X_P are updated, by the module and the environment, in zero time. During each *time round*, the values of all discrete variables in $discX_P$ remain unchanged, and the values of all clock variables in $clkX_P$ increase continuously and uniformly, at the rate 1, as time advances. For a variable x, we use the unprimed symbol x to refer to the value of x at the beginning of a round, and the primed symbol x' to refer to the value of x at the end of a round. If s is a valuation for a set X of variables, by X' we denote the corresponding set $\{x' \mid x \in X\}$ of primed variables, and by s' we denote the valuation for X' that assigns the value $s[x]$ to each variable x'.

Update rounds. As in synchronous languages such as ESTEREL, each update round consists of several subrounds. Unlike in ESTEREL, however, every variable is updated in exactly one subround of each update round. The controlled variables are partitioned into groups called atoms, and the variables within a group are updated simultaneously, in the same subround of each round. The atoms are partially ordered. If atom A precedes atom B in the partial ordering, then in each round, the A-subround must precede the B-subround, and the updated values of the variables controlled by B may depend on the updated values of the variables controlled by A. The updates that are permitted by an atom are specified using executable actions. For two sets X and Y of variables, an *action* from X to Y is a binary relation between the valuations for X and the valuations for Y. The action α from X to Y is *executable* if for every valuation s for X, the number of valuations t for Y with $(s, t) \in \alpha$ is nonzero and finite. Executable actions are nonblocking (always enabled) and ensure finite control branching.

Time rounds. Each time round has a positive real-valued *duration* (update rounds are defined to have duration 0). For a state s, and a real $\delta \in \mathbb{R}_{\geq 0}$, we write $s + \delta$ for the state that assigns the value $x[s]$ to each discrete variable x and the value $x[s] + \delta$ to each clock variable x. Thus, if the state at the beginning of a time round with duration δ is s, then the state at the end of the round is $s + \delta$. The durations that are permitted by an atom are specified using executable delays. A *delay* over a set X of variables is a binary relation between the valuations for X and the valuations for X'. The delay β over X is *executable* if the following two conditions are met. First, $(s, s') \in \beta$ for every valuation s for X. Second, for every valuation s for X and every real $\delta \in \mathbb{R}_{\geq 0}$, if $(s, s' + \delta) \in \beta$, then for all nonnegative reals $\epsilon < \delta$, both $(s, s' + \epsilon) \in \beta$ and $(s + \epsilon, s' + \delta) \in \beta$. An executable delay specifies for every state a maximal (possibly 0 or infinite) duration that may elapse (possibly in several steps) without violating the delay.

Definition 1. *[***Timed atom***]* Let X be a finite set of typed variables. A (*timed*) X*-atom* A consists of a declaration and a body. The atom declaration consists of a set $ctrX_A \subseteq X$ of *controlled variables* and a set $waitX_A \subseteq X \backslash ctrX_A$ of *awaited variables*. The atom body consists of an executable *initial action* $Init_A$ from $waitX'_A$ to $ctrX'_A$, an executable *update action* $Update_A$ from $X \cup waitX'_A$ to $ctrX'_A$, and an executable delay $Delay_A$ over $ctrX_A \cup waitX_A$. We require the update action $Update_A$ to be *round-insensitive*; that is, $(s \cup s'[waitX'_A], s'[ctrX'_A]) \in Update_A$ for every valuation s for X. ■

During the initialization round, the atom A waits for the initial values of the variables in $waitX_A$ before initializing the variables in $ctrX_A$. During each subsequent round, the X-atom A reads the values of the variables in X at the beginning of the round and decides which duration (possibly 0) it is prepared to let elapse. We say that in state s, the atom A *permits* the duration $\delta \in \mathbb{R}_{\geq 0}$ if $(s, s' + \delta) \in Delay_A$. If the duration of the round is decided to be 0, then the round is an update round, and the atom waits for the updated values of the variables in $waitX_A$ before updating the variables in $ctrX_A$. We say that the variable x *awaits* the variable y, and write $x \succ_A y$, if

$x \in ctrX_A$ and $y \in waitX_A$. Round-insensitivity ensures that whenever the values of the awaited variables do not change, then the values of the controlled variables may remain unchanged.[7]

Definition 2. *[***Timed module***]* A (*timed*) *module* P consists of a declaration and a body. The module declaration is a finite set X_P of typed variables that is partitioned into private variables $privX_P$, interface variables $intfX_P$, and external variables $extlX_P$. The module body is a set $\mathcal{A}_P$ of X_P-atoms such that (1) $(\cup_{A \in \mathcal{A}_P} ctrX_A) = ctrX_P$, (2) $ctrX_A \cap ctrX_B = \emptyset$ for all atoms A and B in $\mathcal{A}_P$, and (3) the transitive closure $\succ_P = (\cup_{A \in \mathcal{A}_P} \succ_A)^+$ is asymmetric. ■

The first two conditions ensure that the atoms of P control precisely the variables in $ctrX_P$, and that each variable in $ctrX_P$ is controlled by precisely one atom. The third condition ensures that the await dependencies among the variables in X_P are acyclic. A linear ordering $A_0, \ldots, A_k$ of the atoms in $\mathcal{A}_P$ is *consistent* if for all $0 \leq i < j \leq k$, the awaited variables of A_i are disjoint from the controlled variables of A_j. The asymmetry of $\succ_P$ ensures that there exists a consistent ordering.

Module execution. During the initialization round, first the external variables are assigned arbitrary values of the appropriate types, and then the atoms in $\mathcal{A}_P$ are executed in a consistent order. Each subsequent round is either an update round or a time round whose duration is permitted by all atoms. During an update round, first the external variables are assigned arbitrary values of the appropriate types, and then the update actions of the atoms are executed in a consistent order.

Module syntax. Variable declarations are indicated by keywords such as **awaits**, for the awaited variables of an atom, and **private**, for the private variables of a module. Clock variables have the type $\mathbb{C}$. Initial and update actions are specified by the keywords **init** and **update**, followed by guarded assignments. Delays are specified by the keyword **delay** followed by guarded invariants, where the guard constrains unprimed variables and the invariant constrains primed clock variables. An invariant permits all durations that do not invalidate the invariant. If several guards are true, then one of the corresponding assignments or invariants is chosen nondeterministically. If none of the guards are true, then all controlled variables stay unchanged and no positive duration is permitted.

Example: timed circuits. The module *Delay* of Figure 1 specifies a delay element that copies the boolean input *in* to the boolean output *out* after a delay between 1 to 2 time units. Initially, the module is stable with the output equal to the input. In each update round, if the input changes, the module becomes unstable. Once it becomes unstable, within 1 to 2 time units, it toggles the output and returns to a stable state. To enforce the time bounds, a clock x is started whenever the module becomes unstable. Then, the guard $x \geq 1$ enforces the lower time bound, and the invariant $x' \leq 2$ enforces the upper time bound. If the input changes while the module is unstable, a hazard occurs, and the output may change arbitrarily, independently of the input. Note that if the module is stable or hazardous, any amount of time may elapse. It is easy to describe synchronous gates and latches as modules [5]. Asynchronous circuits, then, can be described by combining gates, latches, and delay elements [13]. ■

Propositional timed modules. A timed module P is *propositional* if all discrete variables of P have finite types (boolean or enumerated), and if the initial actions, the update actions, and the delays of P constrain the clock variables in very restricted ways: in guards and invariants, clocks are compared to rational constants, and in assignments, clocks are either left unchanged or assigned rational constants. For example, the delay element *Delay* is a propositional timed module. In a technical sense, propositional timed modules are open versions of timed automata [2].

[7] Round-insensitivity is not required for discrete modules [5], but allows the treatment of degenerate, zero-duration time rounds as update rounds.

module *Delay*
  interface *out* : $\mathbb{B}$
  external *in* : $\mathbb{B}$
  private *state* : $\{$*stable*, *unstable*, *hazard*$\}$; $x : \mathbb{C}$
  atom *state*, *out*, x **awaits** *in*$'$
    init $state' := stable;\ out' := in'$
    update
    $[\!]\ state = stable \wedge in' \neq in \ \rightarrow\ state' := unstable;\ x' := 0$
    $[\!]\ state = unstable \wedge x \geq 1 \ \rightarrow\ state' := stable;\ out' := in'$
    $[\!]\ state = unstable \wedge x < 1 \wedge in' \neq in \ \rightarrow\ state' := hazard$
    $[\!]\ state = hazard \ \rightarrow\ out' := 0$
    $[\!]\ state = hazard \ \rightarrow\ out' := 1$
    delay
    $[\!]\ state = stable \ \rightarrow\ true$
    $[\!]\ state = unstable \wedge x \leq 2 \ \rightarrow\ x' \leq 2$
    $[\!]\ state = hazard \ \rightarrow\ true$

Fig. 1. Delay element

Transition graph of a module. Every module P defines an edge-labeled transition graph whose vertices are the states of P, and whose labels are nonnegative reals that represent the durations of transitions. A state s is *initial* if $(s'[waitX'_A], s'[ctrX'_A]) \in Init_A$ for each atom A of P. For two states s and t, define $s \xrightarrow{0} t$ if $(s \cup t'[waitX'_A], t'[ctrX'_A]) \in Update_A$ for each atom A of P, and define $s \xrightarrow{\delta} t$, for a positive duration $\delta \in \mathbb{R}_{>0}$, if $t = s + \delta$ and $(s, t') \in Delay_A$ for each atom A of P.

Trajectories of a module. A *trajectory* of the module P is a finite sequence of states and durations of the form $s_0 \xrightarrow{\delta_1} s_1 \xrightarrow{\delta_2} \cdots \xrightarrow{\delta_n} s_n$. We denote this trajectory by the pair $(\overline{s}_{0..n}, \overline{\delta}_{1..n})$. The *length* of the trajectory $(\overline{s}_{0..n}, \overline{\delta}_{1..n})$ is n, and its *accumulated duration* is the sum $\Sigma_{1 \leq i \leq n} \delta_i$ of all transition durations. The trajectory $(\overline{s}_{0..n}, \overline{\delta}_{1..n})$ is a *source-s* trajectory if $s_0 = s$, and an *initialized* trajectory if s_0 is an initial state of P. If $(\overline{s}_{0..n}, \overline{\delta}_{1..n})$ is an initialized trajectory, then s_n is a *reachable* state of P. If the module P contains a single atom A only, we sometimes refer to the trajectories and reachable states of P as trajectories and reachable states of A.

Traces of a module. If $(\overline{s}_{0..n}, \overline{\delta}_{1..n})$ is an (initialized) trajectory of the module P, then the corresponding sequence $s_0[obsX_P] \xrightarrow{\delta_1} \cdots \xrightarrow{\delta_n} s_n[obsX_P]$ of observations and durations is an (initialized) *trace* of P. Thus, a trace retains information about changes to observations and the corresponding times. We write $(\overline{a}_{0..n}, \overline{\delta}_{1..n})$ for the trace $a_0 \xrightarrow{\delta_1} \cdots \xrightarrow{\delta_n} a_n$. To obtain the trace language of a module, we close the set of initialized traces under stuttering. Closure under stuttering combines consecutive transitions that do not change the values of observable discrete variables and produce no discontinuity in the evolution of observable clock variables.

Definition 3. *[***Trace language***]* The *(timed) trace language* L_P of a module P is the least set such that (1) every initialized trace of P belongs to L_P, and (2) if $(\overline{a}_{0..n}, \overline{\delta}_{1..n})$ belongs to L_P, and for some $0 < i \leq n$, both $a_i[discX_P] = a_{i-1}[discX_P]$ and $a_{i+1}[clkX_P] = a_{i-1}[clkX_P] + \delta_i + \delta_{i+1}$, then $a_0 \xrightarrow{\delta_1} \cdots \xrightarrow{\delta_{i-1}} a_{i-1} \xrightarrow{\delta_i + \delta_{i+1}} a_{i+1} \xrightarrow{\delta_{i+2}} \cdots \xrightarrow{\delta_n} s_n$ also belongs to L_P. ■

For every module P, the trace language L_P is prefix-closed. Since all update actions are nonblocking, every trace in L_P is a proper prefix of some other trace in L_P.

Implementation preorder. The trace semantics of a module P consists of the trace language L_P, together with all information that is necessary for describing the possible interactions of P with the environment: the set $intfX_P$ of interface variables, the set $extlX_P$ of external variables, and the await dependencies $\succ_P \cap (intfX_P \times obsX_P)$ between interface variables and observable variables.

Definition 4. *[**Implementation**]* The module P *implements* the module Q, written $P \preceq Q$, if (1) every interface variable of Q is an interface variable of P, (2) every external variable of Q is an observable variable of P, (3) for all variables x in $obsX_Q$ and all variables y in $intfX_Q$, if $y \succ_Q x$ then $y \succ_P x$, and (4) if $(\overline{a}_{0..n}, \overline{\delta}_{1..n})$ belongs to L_P, then the projection $(\overline{a}[obsX_Q]_{0...n}, \overline{\delta}_{1...n})$ belongs to L_Q. ■

The first three conditions ensure that the compatibility constraints imposed by P on its environment are stronger than those imposed by Q. The fourth condition is conventional trace inclusion. It is easy to check that the implementation relation $\preceq$ is a preorder (i.e., reflexive and transitive).

Parallel composition. Modules can be combined using the three operations of variable renaming, variable hiding, and parallel composition [5]. Here, we focus on parallel composition only. The two modules P and Q are *compatible* if (1) the interface variables of P and Q are disjoint, and (2) the await dependencies among the observable variables of P and Q are acyclic —that is, the transitive closure $(\succ_P \cup \succ_Q)^+$ is asymmetric. It follows that if P and R are compatible modules, and $P \preceq Q$, then Q and R are also compatible.

Definition 5. *[**Composition**]* If P and Q are two compatible modules, then the *composition* $P||Q$ is the module with the set $privX_{P||Q} = privX_P \cup privX_Q$ of private variables, the set $intfX_{P||Q} = intfX_P \cup intfX_Q$ of interface variables, the set $extlX_{P||Q} = (extlX_P \cup extlX_Q) \backslash intfX_{P||Q}$ of external variables, and the set $\mathcal{A}_{P||Q} = \mathcal{A}_P \cup \mathcal{A}_Q$ of atoms. ■

It is easy to check that for two compatible modules P and Q, the composition $P||Q$ is again a module. Henceforth, whenever we write $P||Q$, we assume that P and Q are compatible.

Proposition 6. *The composition operator has the following properties.*

(1) A trace $(\overline{a}_{0..n}, \overline{\delta}_{1..n})$ belongs to $L_{P||Q}$ iff the projection $(\overline{a}[obsX_P]_{0..n}, \overline{\delta}_{1..n})$ belongs to L_P and the projection $(\overline{a}[obsX_Q]_{0..n}, \overline{\delta}_{1..n})$ belongs to L_Q.
(2) $P||Q \preceq P$.
(3) If $P \preceq Q$, then $P||R \preceq Q||R$.

Thus, parallel composition behaves like logical conjunction. Property (3) asserts that the implementation preorder is a congruence with respect to composition.

3 Nonzenoness and Receptiveness

It is easy to specify modules that prevent time from diverging. We now rule out such modules, which cannot be realized physically.

Nonzeno modules. Consider the module *Zeno* of Figure 2. While every trajectory of *Zeno* can be extended to a trajectory of arbitrary length, not every trajectory can be extended to a trajectory of arbitrary accumulated duration: by choosing the update $a' := 1$, the module *Zeno* can prevent the divergence of time.

module *Zeno*
  private $a : \mathbb{B};\ x : \mathbb{C}$
  atom a, x
    init $a' := 0;\ x' := 0$
    update
      $[\!] \ a = 0 \ \rightarrow \ x' := 0$
      $[\!] \ a = 0 \ \rightarrow \ a' := 1$
    delay
      $[\!] \ x < 2 \ \rightarrow \ x' < 2$

module *Nonreceptive*
  external $a : \mathbb{B}$
  private $x : \mathbb{C}$
  atom x
    init $x' := 0$
    update
      $[\!] \ a = 0 \ \rightarrow \ x' := 0$
    delay
      $[\!] \ x < 2 \ \rightarrow \ x' < 2$

Fig. 2. Zeno and nonreceptive modules

Definition 7. *[***Nonzenoness***]* A module P is *nonzeno* if for every reachable state s of P, there exists a source-s trajectory of accumulated duration 1. ■

Since trajectories of accumulated duration 1 can be concatenated to a trajectory of arbitrary accumulated duration, nonzeno modules cannot prevent time from diverging.

Proposition 8. *For every nonzeno module P, every trace in L_P can be extended to a trace of arbitrary accumulated duration.*

The symbolic fixpoint-computation procedure of [21] for checking if a given timed automaton is nonzeno can be used for timed modules also. Furthermore, it follows that for propositional timed modules, the problem of checking nonzenoness is complete for PSPACE.

Receptive modules. Nonzenoness is an existential property of a module, and hence, it is not preserved under composition [9, 17]. A simple case in point is the module *Nonreceptive* of Figure 2. While the module *Nonreceptive* is nonzeno, its ability to let time diverge depends on the cooperation of the environment. In particular, if the environment keeps the value of the external variable a always 1, then time cannot progress beyond 2. Consequently, we want to consider only those modules that cannot prevent time from diverging no matter what a "reasonable" environment does. In particular, a nonzeno environment would not be considered reasonable, because it may be the source of the trouble. For compound modules that prevent time from diverging, we must therefore assign the "blame" to one or both components. This assignment of blame is best formalized as a game.

We formalize the receptiveness game for atoms. To define receptiveness we need to consider infinite trajectories: an *ω-trajectory* of the atom A is an infinite sequence $(\overline{s}_{0..}, \overline{\delta}_{1..}) = s_0 \xrightarrow{\delta_1} s_1 \xrightarrow{\delta_2} \cdots$ of states and durations such that every finite prefix of $(\overline{s}_{0..}, \overline{\delta}_{1..})$ is a trajectory of A. The accumulated duration $\Sigma_{i \geq 1} \delta_i$ of the ω-trajectory $(\overline{s}_{0..}, \overline{\delta}_{1..})$ may be finite or infinite.

Consider an X-atom A, and for simplicity, assume that A has no awaited variables. The receptiveness game starts in a reachable state of A. The two players, the protagonist representing the atom, and the adversary representing the environment, take turns to incrementally produce an ω-trajectory of A. Following [17], each round is charged to one of the two players depending on which player blocks the passage of time. Suppose that the current state of the game is s. In each round, first, the protagonist either chooses, in conformance with the update action of A, a new valuation t for the controlled variables in $ctrX_A$, or it proposes a positive duration $\delta \in \mathbb{R}_{>0}$ that is permitted by the delay of A.

– If the protagonist chooses new values for the controlled variables, then the adversary chooses a new valuation u for the remaining variables in $X \backslash ctrX_A$, and the state of the game changes to $t \cup u$. In this case, the round is charged to the protagonist.
– If the protagonist proposes a duration $\delta > 0$, then the adversary either chooses a new valuation u for the variables in $X \backslash ctrX_A$, or it also proposes a positive duration $\delta' \in \mathbb{R}_{>0}$. In the former case, no time elapses, the state of the game changes to $s[ctrX_A] \cup u$, and the round is charged to the adversary. In the latter case, a time round of duration equal to the minimum of δ and δ' is executed. If $\delta < \delta'$, then the state of the game changes to $s + \delta$ and the round is charged to the adversary. If $\delta \geq \delta'$, then the state of the game changes to $s + \delta'$ and the round is charged to the protagonist.

At any round, if the accumulated duration of the trajectory produced so far reaches (or exceeds) 1, the protagonist wins. If the game continues at infinitum, an ω-trajectory with duration less than 1 is produced. In this case, the protagonist wins the game iff only finitely many rounds are charged to the protagonist. Observe that the charging of rounds is asymmetric: when both the players propose an update round, or both propose a time round of the same duration, the round is charged to the protagonist. This, together with the requirement that only finitely many rounds can be charged to a winning protagonist, ensures that the atom does not collaborate with the environment to block the passage of time. If the protagonist has a strategy to win the game, the atom A is called receptive.

In general, when the atom A has awaited variables, each round consists of three steps: first, the adversary either chooses to update the awaited variables or it proposes a positive duration, followed by the two steps described above. We formalize the moves of the protagonist by defining a pair of strategies: a delay strategy to propose the duration of the next round, and an update strategy to choose the new values of the controlled variables. For a set Y of variables, let Σ_Y be the set of valuations for Y. A *delay strategy* for the X-atom A is a function $F_d : \Sigma_X \to \mathbb{R}_{\geq 0}$ such that if $t = s + F_d(s)$, then $(s, t') \in Delay_A$. An *update strategy* for A is a function $F_u : \Sigma_X \times \Sigma_{waitX_A} \to \Sigma_{ctrX_A}$ such that if $F_u(s, t) = u$, then $(s \cup t', u') \in Update_A$. The update strategy F_u *matches* the delay strategy F_d if for every valuation s for X, if $F_d(s) > 0$ then $F_u(s, s[waitX_A]) = s[ctrX_A]$. For a delay strategy F_d and a matching update strategy F_u, the ω-trajectory $(\overline{s}_{0..}, \overline{\delta}_{1..})$ of A is a (F_d, F_u)-*outcome* if for all $i \geq 0$, (1) the duration of each round is bounded by the duration selected by the delay strategy: $\delta_{i+1} \leq F_d(s_i)$; and (2) in each round of duration 0, the controlled part of the new state is selected by the update strategy: if $\delta_{i+1} = 0$, then $s_{i+1}[ctrX_A] = F_u(s_i, s_{i+1}[waitX_A])$.

Definition 9. *[***Receptiveness***]* An atom A is *receptive* if there exist a delay strategy F_d and a matching update strategy F_u for A such that there is no ω-trajectory $(\overline{s}_{0..}, \overline{\delta}_{1..})$ of A satisfying (1) s_0 is a reachable state of A, (2) $(\overline{s}_{0..}, \overline{\delta}_{1..})$ is an (F_d, F_u)-outcome, (3) the accumulated duration of $(\overline{s}_{0..}, \overline{\delta}_{1..})$ is less than 1, and (4) there are infinitely many positions i with $F_d(s_i) = \delta_{i+1}$. A module P is *receptive* if all atoms of P are receptive. ■

Update and delay strategies have been defined history-free (i.e., they depend only on the current state of the game). It is easy to check that the availability of more powerful, history-dependent strategies would not alter the definition of receptiveness. Since the atoms of a compound module are the atoms of the component modules, receptiveness is closed under parallel composition.

Proposition 10. *If two modules P and Q are receptive, then the module $P||Q$ is also receptive.*

The following theorem identifies receptiveness is a sufficient condition for nonzenoness that is closed under parallel composition.

Theorem 11. *Every receptive module is nonzeno.*

3.1 Assume-Guarantee Reasoning for Timed Modules

Consider the problem of verifying that the (complex) module $P_1||P_2$ implements the (simpler) module $Q_1||Q_2$. Since the implementation preorder is a congruence with respect to parallel composition, it suffices to prove the desired refinement for each component separately: (a) $P_1 \preceq Q_1$, and (b) $P_2 \preceq Q_2$. These proof obligations, however, are rarely satisfied if the components interact. An assume-guarantee principle allows us to replace (a) and (b) by two weaker obligations, namely, (a') $P_1||Q_2 \preceq Q_1$ and (b') $Q_1||P_2 \preceq Q_2$ [25, 8, 4]. Obligation (a') asserts that P_1 implements Q_1, under the hypothesis that its environment behaves like Q_2, and obligation (b') asserts that P_2 implements Q_2, under the hypothesis that its environment behaves like Q_1. Despite the apparent circularity, the assume-guarantee principle is valid if all involved modules are receptive.

Theorem 12. *For receptive modules P_1, P_2, Q_1, and Q_2, if $P_1||Q_2 \preceq Q_1$ and $Q_1||P_2 \preceq Q_2$, then $P_1||P_2 \preceq Q_1||Q_2$.*

3.2 Deciding the Receptiveness of Propositional Timed Atoms

We now address the problem of checking automatically if a given atom A is receptive. For this purpose, we view the receptiveness game as a two-player infinite game on an extended state space with a coBüchi winning condition. First, we introduce a new clock variable now, which is 0 at the beginning of the game, and changes value only by advancing with time. Second, we introduce a binary variable $blame$, whose type is the set $\{atom, env\}$. The value of $blame$ is set to $atom$ during each round that is charged to the atom, and it is set to env during each round that is charged to the environment. Then, the *positions* of the game are the triples (s, ϵ, b), where s is a state of A, the nonnegative real $\epsilon \in \mathbb{R}_{\geq 0}$ gives a value to the clock variable now, and the bit b gives a value to the variable $blame$. The *winning condition* for the protagonist is the temporal-logic formula

$$\varphi_{win}: \ \Diamond(now = 1) \vee \Diamond\Box(blame = env)$$

("eventually $now = 1$, or only finitely often $blame = atom$"). It follows that the atom A is receptive iff for all reachable states s of A, the protagonist has a winning strategy in the position $(s, 0, atom)$.

Symbolic strategy checking. We give a symbolic algorithm for computing the set of positions in which the protagonist has a winning strategy. To simplify the presentation, we assume that the atom A has no awaited variables. A *region* σ of A is a set of game positions. We define an operator Pre on regions such that a position belongs to $Pre(\sigma)$ iff the atom A has a way of choosing new values for the controlled variables or of proposing a positive duration so that, irrespective of what the environment does, the position after the next move will be in σ.

Definition 13. *[***Pre***]* For a region σ of the atom A, the region $Pre(\sigma)$ is the set of positions (s, ϵ, b) that satisfy one of the following two conditions: (1) there exists a valuation t for the controlled variables $ctrX_A$ such that $(s, t) \in Update_A$ and for every valuation u for $X \backslash ctrX_A$, the position $(t \cup u, \epsilon, atom)$ is in σ; or (2) there exists a positive duration $\delta \in \mathbb{R}_{>0}$ such that (i) for every valuation u for $X \backslash ctrX_A$, the position $(s[ctrX_A] \cup u, \epsilon, env)$ is in σ, (ii) the position $(s + \delta, \epsilon + \delta, atom)$ is in σ, and (iii) for every positive duration $\delta' < \delta$, the position $(s + \delta', \epsilon + \delta', env)$ is in σ. ■

Observe that Pre is a monotonic operator on regions. The positions in which the protagonist has a winning strategy can be characterized using Pre, boolean operators on regions, and fixpoints.

Theorem 14. *The atom A is receptive iff for every reachable state s of A, the position $(s, 0, atom)$ belongs to the set $\nu X. \mu Y. Pre((now \geq 1 \vee blame = env \vee Y) \wedge X)$.*

This theorem immediately suggests a symbolic fixpoint-computation procedure for checking receptiveness. The procedure is effective as long as we know how to compute the operator Pre. This is the case, for example, for propositional timed modules, where the procedure can be implemented using existing symbolic model checkers such as KRONOS [14].

Enumerative strategy checking. For every timed automaton, there exists a finite partitioning of the state space called *region equivalence* [2]. The definition of region equivalence carries over straightforwardly to the states and the positions of the atoms of propositional timed modules. For an atom A of a propositional timed module, we write $\cong^R_A$ for the region equivalence of A. The number of equivalence classes of $\cong^R_A$ is finite, exponential in the size of the description of A. A *block* of $\cong^R_A$ is a union of $\cong^R_A$-equivalence classes.

Proposition 15. *If A is an atom of a propositional timed module, and the region σ is a block of the region equivalence $\cong^R_A$, then the region $Pre(\sigma)$ is also a block of $\cong^R_A$.*

This implies that all regions generated during the symbolic fixpoint-computation procedure are blocks of $\cong^R_A$, and therefore, for propositional timed modules, the termination of the procedure is guaranteed. Alternatively, one can construct the $\cong^R_A$-quotient graph of the infinite graph for the receptiveness game, and solve the coBüchi game on the resulting finite graph. Since the number of positions of the $\cong^R_A$-quotient graph is exponential in the description of A, and the complexity of solving a coBüchi game on a finite graph is quadratic in the number of vertices, we have an exponential decision procedure for checking receptiveness. The procedure is optimal, because already solving finite reachability games on timed automata is EXPTIME-hard [20].

Theorem 16. *Given an atom A of a propositional timed module, the problem of checking if A is receptive is complete for* EXPTIME.

It should be noted that, since receptiveness is closed under parallel composition, it suffices to check atoms for receptiveness in isolation.

3.3 Controller Synthesis for Propositional Timed Modules

The supervisory-control problem for modules asks, given a module P (the "plant") and a set $safe$ of observations of P, construct a module Q (the "supervisor" or "controller") such that the observations of all reachable states of $P||Q$ are contained in $safe$ [26]. This problem, and more general control problems, can be solved using fixpoint computation [24]: first, compute the set $controllable$ of states of the module P in which the environment has a winning strategy for the winning condition $\Box safe$; then, construct a module Q that, when composed with P, prevents P from leaving the set $controllable$. In this formulation of the problem, however, the synthesized controller Q may achieve its goal by stopping time (an example of this is given at the end of Section 4). Such a controller cannot be realized physically. Hence, we reformulate the problem, and solve the reformulated problem.

Definition 17. *[***Control***]* The *supervisory-control problem* for modules asks, given a receptive module P and a set *safe* of observations of P, construct a receptive module Q such that the observations of all reachable states of $P||Q$ are contained in *safe* (or indicate that no such module Q exists). ■

If the module P contains k atoms, then the ω-trajectories of P can be considered possible outcomes of a $(k+1)$-player infinite game between the atoms and the environment. Each round of the game is charged to one of the atoms or to the environment, depending which player proposes the minimal duration (in the case of ties, the environment —that is, the controller— is charged). We extend the state space of P by introducing a new clock variable T and a binary variable $blame$, whose type is $\{module, env\}$. The initial value of T is 0, and T is reset to 0 whenever its value reaches 1. The value of $blame$ is set to $module$ during each round that is charged to some atom of P, and to env during each round that is charged to the environment. Then, the initial state s of the module P is controllable iff the environment has a winning strategy in the position $(s, 0, module)$ for the winning condition

$$\psi_{win}\colon\ \Box safe \wedge (\Box\Diamond(blame = env) \Rightarrow \Box\Diamond(T = 0) \wedge \Box\Diamond(T = 1))$$

("always $safe$, and if infinitely often $blame = env$, then infinitely often T is reset"). Since the winning condition ψ_{win} is a single-pair Streett condition, it suffices to consider memory-free strategies, and the winning positions can be characterized using a symbolic fixpoint expression. This gives a symbolic procedure for solving the supervisory-control problem. Since the complexity of solving a single-pair Streett game on a finite graph is cubic in the number of vertices [15], we have an exponential decision procedure for the supervisory control of propositional timed modules.

Theorem 18. *Given a receptive propositional timed module P and a set* safe *of observations of P, the supervisory-control problem $(P$, safe$)$ is complete for* EXPTIME.

It should be noted that, since receptiveness enters into ψ_{win} as a Streett condition, the supervisory-control problem can be solved at no extra cost for more complex control requirements than $\Box safe$.

4 Hybrid Modules

We generalize timed modules to hybrid modules. All continuous variables of a timed module are clocks. Hybrid modules admit more general continuous variables such as temperature or pressure. Hence, for a hybrid module P, the set X_P of module variables is partitioned into two disjoint sets, the set $discX_P$ of discrete variables and the set $contX_P$ of continuous variables. The type of all continuous variables is $\mathbb{R}$. During each update round, the variables in X_P are updated, as before, in zero time. During each time round, the values of all discrete variables in $discX_P$ remain unchanged, and the values of all continuous variables in $contX_P$ evolve continuously for the duration of the round (which is positive).

Formally, during each time round, the trajectory of a hybrid module follows a flow. A *flow* for X is a continuous and piecewise-smooth function f from the nonnegative reals $\mathbb{R}_{\geq 0}$ to the set Σ_X of valuations for X such that $f(\delta)[x] = f(0)[x]$ for every discrete variable x and every nonnegative real $\delta \in \mathbb{R}_{\geq 0}$. By *piecewise-smooth* we mean that the nonnegative real line $\mathbb{R}_{\geq 0}$ can be partitioned into finitely many intervals such that the function f is in C^∞ on each interval. The flow f is a *source-s* flow if $f(0) = s$. We write $\dot{f}$ for the first derivative of the flow f. Notice that the function $\dot{f}$ is again piecewise-smooth but not necessarily continuous.

The flows that are permitted by a hybrid atom are specified using executable activities. Consider three sets X, Y, and Z of variables with $Y \subseteq X$ and $Z \subseteq X \backslash Y$. An *activity* γ from X to Z, given Y, is a ternary relation between the valuations for X, the flows for Y, and the flows for Z such that $(s, g, h) \in \gamma$ implies that g is a source-$s[Y]$ flow and h is a source-$s[Z]$ flow. The activity γ is *executable* if the following two conditions are met. First, for every valuation s for X and every flow g for Y, there is a flow h for Z such that $(s, g, h) \in \gamma$. Second, for every real $\delta \in \mathbb{R}_{\geq 0}$, if

$(s, g, h) \in \gamma$ and $g'(\epsilon) = g(\epsilon)$ for all $\epsilon \leq \delta$, then there is a flow h' such that $(s, g', h') \in \gamma$ and $h'(\epsilon) = h(\epsilon)$ for all $\epsilon \leq \delta$. The first condition is a nonblocking condition; the second condition ensures that the value $h(\delta)$ of the chosen flow h at time δ depends only on the values $g(\epsilon)$ of the given flow g at times $\epsilon \leq \delta$.

We use differential equations and differential inequalities to specify activities. As examples, consider the following activities from $\{y, z\}$ to $\{z\}$, given $\{y\}$. For an initial value z_0 of z and a flow g for y, the differential constraint $\dot{z} := 1$ specifies the flow $f(\delta)[z] = z_0 + \delta$; that is, z is a clock variable. The differential constraint $\dot{z} := z$ specifies the flow $f(\delta)[z] = e^{\ln z_0 + \delta}$; that is, z increases exponentially independent of y. The differential constraint $\dot{z} := \dot{y}$ specifies the flow $f(\delta)[z] = z_0 + (g(\delta)[y] - g(0)[y])$; that is, z copies the rate of y. The differential constraint $0.9 \leq \dot{z} \leq 1.1$ specifies the infinite set of flows f with $f(0)[z] = z_0$ and $0.9 \leq \dot{f}(\delta) \leq 1.1$ for all $\delta \in \mathbb{R}_{\geq 0}$; that is, z behaves like a clock with a drift of at most 10%.

The durations that are permitted by a hybrid atom are specified using executable delays. For an activity γ from X to Z, given Y, a delay β over $Y \cup Z$ is *γ-executable* if for every valuation s for $Y \cup Z$ and every real $\delta \in \mathbb{R}_{\geq 0}$, (1) $(s, s') \in \beta$ and (2) if $(s, g, h) \in \gamma$ and $(s, g(\delta)' \cup h(\delta)') \in \beta$, then $(s, g(\epsilon)' \cup h(\epsilon)') \in \beta$ and $(g(\epsilon) \cup h(\epsilon), g(\delta)' \cup h(\delta)') \in \beta$ for all nonnegative reals $\epsilon < \delta$. A γ-executable delay specifies for every state and every flow that is possible according to γ a maximal (possibly 0 or infinite) permissible duration.

Definition 19. *[***Hybrid atoms and modules***]* Let X be a finite set of typed variables. A *hybrid X-atom A* consists of a declaration and a body. The atom declaration is the same as for a timed atom. The atom body consists of an executable *initial action $Init_A$* from $waitX'_A$ to $ctrX'_A$, an executable *update action $Update_A$* from $X \cup waitX'_A$ to $ctrX'_A$, an executable activity $Flow_A$ from X to $ctrX_A$, given $waitX_A$, and a $Flow_A$-executable delay $Delay_A$ over $ctrX_A \cup waitX_A$. A *hybrid module* is the same as a timed module except that its atoms are hybrid. The *composition* of hybrid modules is defined as for timed modules. ■

Linear hybrid modules. Particularly suitable for analysis is a subclass of hybrid modules called linear hybrid modules. Essentially, a hybrid module P is *linear* if all discrete variables of P have finite types, and the continuous variables X of P occur in the guards, assignments, invariants, and differential constraints of P only within linear expressions over X or over $\dot{X} = \{\dot{x} \mid x \in X\}$. In a technical sense, linear hybrid modules are open versions of linear hybrid automata [6].

Trace semantics. An X-atom A *permits* the flow f for X if $(f(0), f[waitX_A], f[ctrX_A]) \in Flow_A$. The transition graph of a hybrid module P is best defined as an edge-labeled graph whose vertices are the flows over X_P that are permitted by all atoms of P, and whose labels are nonnegative reals that represent the duration of flows: if a flow f has an outgoing edge labeled δ, then on all trajectories through that edge the flow f contributes duration δ. Formally, a flow f of P is *initial* if $(f(0)'[waitX'_A], f(0)'[ctrX'_A]) \in Init_A$ for each atom A of P. For two flows f and g of P and a nonnegative duration $\delta \in \mathbb{R}_{\geq 0}$, we define $f \xrightarrow{\delta} g$ if $(f(0), f(\delta)') \in Delay_A$ and $(f(\delta) \cup g(0)'[waitX'_A], g(0)'[ctrX'_A]) \in Update_A$ for each atom A of P. Trajectories, then, are sequences of flows and durations, and traces are projections of trajectories to the observable variables. The trace-language of a hybrid module is obtained by closing its traces under stuttering. Nonzenoness for hybrid modules is defined as for timed modules.

Receptiveness. Consider a hybrid atom without awaited variables. A round of the receptiveness game proceeds as follows: first, the atom proposes a duration δ and chooses either a new valuation (if $\delta = 0$) or a flow (if $\delta > 0$) for the controlled variables; then, the environment proposes a duration $\delta' \leq \delta$ and chooses either a new valuation (if $\delta' = 0$) or a flow (if $\delta' > 0$) for the

module *TwoTanks*
  interface $w_L, w_R : \mathbb{R}$
  external *pipe*: $\{$*left*, *right*$\}$
  atom w_L, w_R
    init $w'_L := 5;\ w'_R := 10$
    flow
    $[\!]$ *pipe* = *left* $\rightarrow$ $\dot{w}_L := 1;\ \dot{w}_R := -2 \cdot \mathrm{sign}(w_R)$
    $[\!]$ *pipe* = *right* $\rightarrow$ $\dot{w}_L := -2 \cdot \mathrm{sign}(w_L);\ \dot{w}_R := 1$
    delay
    $[\!]$ *true* $\rightarrow$ *true*

Fig. 3. Two tanks running out of water

uncontrolled variables. The round is resolved as before: if $\delta = \delta' = 0$, then both the controlled and the uncontrolled variables are updated according to the chosen valuations, and the round is charged to the atom; if $\delta > \delta' = 0$, then the controlled variables stay unchanged, the uncontrolled variables are updated, and the round is charged to the environment; if $\delta = \delta' > 0$, then both the controlled and the uncontrolled variables evolve according to the chosen flows for duration δ, and the round is charged to the atom; if $\delta > \delta' > 0$, then both the controlled and the uncontrolled variables evolve according to the chosen flows for duration δ', and the round is charged to the environment. The atom is *receptive* if it has a strategy such that starting from any reachable state, the strategy either produces a trajectory of accumulated duration 1, or it results in an infinite sequence of rounds of which only finitely many are charged to the atom. As before, a hybrid module is receptive if all its atoms are receptive.

Similar to Section 3, it can be shown that the receptive hybrid modules are closed under parallel composition, that every receptive hybrid module is nonzeno, and that the assume-guarantee principle holds for receptive hybrid modules.[8] Furthermore, the fixpoint expression from Theorem 16 provides a symbolic procedure for checking the receptiveness of linear hybrid modules, for which the Pre operator can be computed. Similarly, the fixpoint characterization of the winning condition for the game from Section 3.3 suggests a symbolic procedure for synthesizing receptive controllers for linear hybrid automata. Both procedures, while not guaranteed to terminate, can be implemented using the primitives supplied by the symbolic model checker HYTECH [19] for linear hybrid automata.

Example: water tanks. The hybrid module $TwoTanks$ of Figure 3 models two water tanks and a common water source that provides water at the rate of 3 units per second. Through a pipe, the water source can be directed either to the left tank or to the right tank. Both tanks have openings at the bottom, and from each tank, water drains at the rate of 2 units per second. Initially, the left tank contains $w_L = 5$ units of water, and the right tank contains $w_R = 10$ units of water. This system is of interest, because by ignoring the receptiveness condition, one can devise a controller that keeps both water levels at no less than 2 units (i.e., the condition $safe$ is $w_L \geq 2 \wedge w_R \geq 2$): whenever w_L falls to 2, direct the pipe to the left tank, and whenever w_R falls to 2, direct the pipe to the right tank. Such a controller, of course, cannot be realized physically, because it would cause the pipe to switch back and forth infinitely often within a finite amount of time. Indeed, according to our definitions, there is no receptive controller that keeps both water levels positive forever. ■

[8] If the implementation preorder considers ω-trace inclusion, rather than the inclusion of finite traces, then finite interface branching is required for the soundness of the assume-guarantee principle.

References

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

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

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

[4] R. Alur, T.A. Henzinger. Local liveness for compositional modeling of fair reactive systems. In *Computer-aided Verification*, Springer LNCS 939, pp. 166-179, 1995.

[5] R. Alur, T.A. Henzinger. Reactive modules. In *Proc. IEEE Symp. Logic in Computer Science*, pp. 207–218, 1996.

[6] R. Alur, T.A. Henzinger, P.-H. Ho. Automatic symbolic verification of embedded systems. *IEEE Trans. Software Engineering*, 22:181–201, 1996.

[7] R. Alur, T.A. Henzinger, E.D. Sontag, eds. *Hybrid Systems III: Verification and Control*. Springer LNCS 1066, 1996.

[8] M. Abadi, L. Lamport. Composing specifications. *ACM Trans. Programming Languages and Systems*, 15:73–132, 1993.

[9] M. Abadi, L. Lamport. An old-fashioned recipe for real time. In *Real Time: Theory in Practice*, Springer LNCS 600, pp. 1–27, 1992.

[10] E. Asarin, O. Maler, A. Pnueli. Symbolic controller synthesis for discrete and timed systems. In *Hybrid Systems II*, Springer LNCS 999, pp. 1–20, 1995.

[11] P. Antsaklis, A. Nerode, W. Kohn, S. Sastry, eds. *Hybrid Systems II*. Springer LNCS 999, 1995.

[12] J.W. de Bakker, K. Huizing, W.-P. de Roever, G. Rozenberg, eds. *Real Time: Theory in Practice*. Springer LNCS 600, 1992.

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

[14] C. Daws, A. Olivero, S. Tripakis, S. Yovine. The tool KRONOS. In *Hybrid Systems III*, Springer LNCS 1066, pp. 208–219, 1996.

[15] E.A. Emerson, C. Jutla. The complexity of tree automata and logics of programs. In *Proc. IEEE Symp. Foundations of Computer Science*, pp. 328–337, 1988.

[16] R.L. Grossman, A. Nerode, A.P. Ravn, H. Rischel, eds. *Hybrid Systems*. Springer LNCS 736, 1993.

[17] R. Gawlick, R. Segala, J.F. Sogaard-Andersen, N.A. Lynch. Liveness in timed and untimed systems. In *Automata, Languages, and Programming*, Springer LNCS 820, pp. 166-177, 1994.

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

[19] T.A. Henzinger, P.-H. Ho, H. Wong-Toi. HYTECH: the next generation. In *Proc. IEEE Real-time Systems Symp.*, pp. 56–65, 1995.

[20] T.A. Henzinger, P.W. Kopke. Discrete-time control for rectangular hybrid automata. In *Automata, Languages, and Programming*, Springer LNCS, 1997.

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

[22] G. Hoffmann, H. Wong-Toi. The input-output control of real-time discrete-event systems. In *Proc. IEEE Real-time Systems Symp.*, pp. 256–265, 1992.

[23] N.A. Lynch, R. Segala, F. Vaandrager, H.B. Weinberg. Hybrid I/O Automata. In *Hybrid Systems III*, Springer LNCS 1066, pp. 496–510, 1996.

[24] O. Maler, A. Pnueli, J. Sifakis. On the synthesis of discrete controllers for timed systems. In *Theoretical Aspects of Computer Science*, Springer LNCS 900, pp. 229–242, 1995.

[25] A. Pnueli. In transition from global to modular temporal reasoning about programs. In *Logics and Models of Concurrent Systems*, Springer LNCS, pp. 123–144, 1984.

[26] P.J. Ramadge, W.M. Wonham. Supervisory control of a class of discrete-event processes. *SIAM J. Control and Optimization*, 25:206–230, 1987.

[27] S. Tasiran, R. Alur, R.P. Kurshan, R.K. Brayton. Verifying abstractions of timed systems. In *Concurrency Theory*, Springer LNCS 1119, pp. 546–562, 1996.