Timed I/O Automata: A Mathematical Framework for Modeling and Analyzing Real-Time Systems

Dilsun K. Kaynar and Nancy Lynch*
MIT Computer Science and Artificial Intelligence Laboratory

Roberto Segala
Dipartimento di Informatica, Università di Verona

Frits Vaandrager†
Nijmegen Institute for Computing and Information Sciences, University of Nijmegen

Abstract

We describe the Timed Input/Output Automata (TIOA) *framework, a general mathematical framework for modeling and analyzing real-time systems. It is based on* timed I/O automata*, which engage in both discrete transitions and continuous trajectories. The framework includes a notion of external behavior, and notions of composition and abstraction. We define safety and liveness properties for timed I/O automata, and a notion of receptiveness, and prove basic results about all of these notions. The TIOA framework is defined as a special case of the new Hybrid I/O Automata (HIOA) modeling framework for hybrid systems. Specifically, a TIOA is an HIOA with no external variables; thus, TIOAs communicate via shared discrete actions only, and do not interact continuously. This restriction is consistent with previous real-time system models, and gives rise to some simplifications in the theory (compared to HIOA). The resulting model is expressive enough to describe complex timing behavior, and to express the important ideas of previous timed automata frameworks.*

1. Introduction

This paper describes the *Timed Input/Output Automata (TIOA)* framework, a general mathematical framework for modeling and analyzing real-time systems. Designers of real-time systems can use this framework to describe complex systems and decompose them into manageable pieces. In particular, they can use TIOA to describe their systems at multiple levels of abstraction, and to decompose their systems into more primitive, interacting components. Designers can use TIOA to prove safety, liveness, and performance properties of real-time systems. Since TIOA is purely mathematical, such proofs are generally done by hand. However, TIOA is a natural basis for computer support tools, which will be developed in the future. The TIOA framework is general enough to express previous results from other frameworks, such as [24, 23, 5, 22, 21, 29].

This paper summarizes a longer monograph [14], which contains complete definitions and results for the TIOA framework. Here, we simply provide motivation, present the most important definitions and results, and illustrate the ideas with one example—a simple clock synchronization algorithm. More examples, including a simple timeout-based failure detection algorithm and a timing-based mutual exclusion protocol, appear in [14]. We are informal in this paper, relying on [14] to fill in the gaps.

The Framework. The fundamental object in the TIOA framework is a *timed I/O automaton*, a state machine that engages in both *discrete transitions* (which model instantaneous events) and *continuous trajectories* (which model evolution of state over time). The externally visible behavior of each automaton is defined by its set of *traces*—essentially, sequences of actions interspersed with time-passage steps. The framework defines what it means for one automaton to *implement* another, based on inclusion of their sets of traces, and defines various notion of *simulation*, which provide sufficient conditions for demonstrating implementation relationships. TIOAs may be composed to build other TIOAs; automata that are composed interact by means of shared in-

* Corresponding author's email addres: dilsun@theory.lcs.mit.edu. Research supported by DARPA/AFOSR MURI Contract F49620-02-1-0325, DARPA SEC contract F33615-01-C-1850, NSF ITR contract CCR-0121277, and Air Force Aerospace Research-OSR Contract F49620-00-1-0097.

† Supported by EU IST project IST-2001-35304: Advanced Methods for Timed Systems (AMETIST) and PROGRESS project TES4999: Verification of Hard and Softly Timed Systems (HaaST).

put and output actions. The trace set of a composed automaton is determined by the trace sets of its component automata.

We define general notions of safety and liveness properties for TIOAs. We show that standard results about the interplay between safety and liveness properties for untimed systems [2] carry over to TIOAs.

A TIOA may exhibit *Zeno behavior*, by performing infinitely many discrete actions in a finite amount of time, or by simply preventing a certain time from being reached. We define a notion of *receptiveness* for TIOAs, to capture the notion that a TIOA does not contribute to producing Zeno behavior. This notion is defined in terms of the existence of a "strategy" for scheduling the TIOA's transitions; such a strategy is itself formalized as a TIOA. We prove a result showing that the composition of two receptive TIOAs is also receptive. This result provides a compositional technique for verifying the absence of Zeno behavior. We also generalize the notion of receptiveness to one that says that a TIOA guarantees a particular liveness property, and prove a result saying that the composition of two TIOAs that are receptive for particular liveness properties is receptive for the "composition" of the two liveness properties. This provides a compositional technique for verifying certain liveness properties.

The monograph [14] contains other material, for example, a description of the timed automata of Alur and Dill [5, 3] and those of Merritt, Modugno, and Tuttle [24] as specializations of our TIOA model.

Evolution of the framework. The TIOA framework has evolved from several previous modeling frameworks for real-time systems [24, 22, 21, 29]. The framework of Merritt, Modugno, and Tuttle [24] is a modification of the earlier I/O automata model for asynchronous, untimed discrete systems [20], in which upper and lower time bounds are associated with certain "tasks". The framework of Lynch and Vaandrager [22, 21] simplifies and generalizes that of Merritt et al. by introducing explicit time-passage transitions and allowing rather arbitrary restrictions on their occurrence. Segala et al. [29] added a treatment of liveness to the framework of [22, 21]. These models have been used fairly extensively in analyzing the correctness and timing behavior of protocols; see, for example, [7, 19, 11].

Recently, Lynch, Segala, and Vaandrager presented the *Hybrid Input/Output Automaton (HIOA)* modeling framework for hybrid (continuous/discrete) systems [17], which evolved from the earlier hybrid system model [18]. An HIOA is a kind of nondeterministic, possibly infinite-state, state machine. The state of an HIOA is determined by a valuation of state variables that are internal to the automaton. An HIOA may also have external variables, which model information flowing continuously into and out of the system. The state of an HIOA can change in two ways: by *discrete transitions*, which change the state instantaneously, or according to *trajectories*, which describe the evolution of the internal and external variables over intervals of time. The HIOA framework includes composition and abstraction, and a treatment of receptiveness in terms of strategies that are formalized as HIOAs. HIOAs have been used in analyzing the behavior of automated transportation systems [30, 16, 26], robotics systems [10], and other hybrid systems.

Having developed the very general HIOA framework, we decided to revisit the old timed automata frameworks and develop a new one that is "upward compatible" with HIOA. HIOA has several features that seem useful for modeling real-time systems. For example, trajectories lead to simpler mathematical definitions and proofs than time-passage transitions. Also, structured states are useful in writing real-time system specifications, and this structure does not complicate mathematical results. Furthermore, the mathematical notion of a *trace*, which is used to represent the external behavior of an HIOA, can be specialized to a neat representation of the external behavior of a timed automaton. (Earlier definitions, such as the one in [22], blurred certain technical distinctions involving right-open and right-closed time intervals.) Moreover, the HIOA approach to handling receptiveness, by modeling strategies as automata, is simpler than the approach taken earlier, e.g., in [29].

For these reasons, we have defined a new TIOA framework as a special case of HIOA. In particular, we define a *Timed I/O Automaton* to be an HIOA without any external variables. Thus, TIOAs may communicate via shared actions only, not shared variables, that is, they do not interact continuously, but only via discrete actions. Interesting continuous evolution involves only internal automaton state components. The TIOA model does not impose any other restrictions on the expressive power of HIOA.

Limiting communication to discrete interactions is an apt choice since the previous timed I/O automata frameworks also use this type of communication. On the other hand, by avoiding any further restrictions on the general HIOA model, we obtain an expressive model suitable for specifying complex timing behavior. For example, we do not require variables to be either discrete or to evolve at the same rate as real time as is the case in some other models [5, 27]. Consequently, algorithms such as clock synchronization algorithms that use local clocks evolving at different and varying rates can be formalized naturally in our framework. Our framework is expressive enough to express the important ideas of previous timed automata frameworks, including [24, 23, 5, 22, 21, 29].

Having no external variables in the model gives rise to some simplifications in the theory (compared to HIOA). For example, proving that the composition of two timed automata is a well-defined automaton is simpler without

external variables: technical "strong compatibility" conditions that are needed in the general HIOA framework are not needed to obtain the result for TIOAs. Similarly, the treatment of receptiveness is simpler.

2. Describing Timed System Behavior

In this section, we list the basic notions that are used in describing the behavior of a timed system, including both discrete and continuous changes. We simply sketch this material, leaving the reader to consult [14] for the details.

The time domains we use is the set R of real numbers (in [14] also other time domains are considered). States of automata will consist of valuations of *variables*. Each variable has both a *static type*, which defines the set of values it may assume, and a *dynamic type*, which gives the set of trajectories it may follow. We assume that dynamic types are closed under some simple operations: shifting the time domain, taking subintervals and pasting together intervals. We call a variable *discrete* if its dynamic type equals the pasting-closure of a set of constant-valued functions (i.e., the step-functions), and *analog* if its dynamic type equals the pasting-closure of a set of continuous functions (i.e., the piecewise-continuous functions).

A *valuation* for a set V of variables is a function that associates with each variable $v \in V$ a value in its static type. We write $val(V)$ for the set of all valuations for V. A *trajectory* for a set V of variables describes the evolution of the variables in V over time; formally, it is a function from a time interval that starts with 0 to valuations of V, that is, a trajectory defines a value for each variable at each time in the interval. A *point trajectory* is one with the trivial domain $\{0\}$. The *limit time* of a trajectory τ, $\tau.ltime$, is the supremum of the times in its domain. $\tau.fval$ is defined to be the first valuation of τ, and if τ is right-closed, $\tau.lval$ is the last valuation. Suppose τ and τ' are trajectories for V, with τ closed. The *concatenation* of τ and τ', denoted by $\tau \frown \tau'$, is the trajectory obtained by taking the union of the first trajectory and the function obtained by shifting the domain of the second trajectory until the start time agrees with the limit time of the first trajectory; the last valuation of the first trajectory, which may not be the same as the first valuation of the second trajectory, is the one that appears in the concatenation.

The notion of a *hybrid sequence* is used to model a combination of changes that occur instantaneously and changes that occur over intervals of time. Our definition is parameterized by a set A of discrete actions and a set V of variables. Thus, an (A, V)-*sequence* is a finite or infinite alternating sequence, $\tau_0\, a_1\, \tau_1\, a_2\, \tau_2 \ldots$, of trajectories over V and actions in A. A *hybrid sequence* is any (A, V)-sequence. Since the trajectories in a hybrid sequence can be point trajectories, our notion of hybrid sequence allows a sequence of discrete actions to occur at the same real time, with corresponding changes of variable values. The *limit time* of a hybrid sequence α, denoted by $\alpha.ltime$, is defined by adding the limit times of all its trajectories. The first and last valuations, $\alpha.fval$ and $\alpha.lval$, are defined in the natural way. Hybrid sequence α is defined to be *time-bounded* if $\alpha.ltime$ is finite, *admissible* if $\alpha.ltime = \infty$, *closed* if α is finite and ends with a trajectory whose domain is a closed interval, and *Zeno* if it is neither closed nor admissible. That is, a Zeno hybrid sequence spans only a finite amount of time, but either contains an infinite number of actions, or else it ends with a right-open trajectory. Like trajectories, hybrid sequences can be concatenated, and one can be a prefix of another. A hybrid sequence can also be restricted to smaller sets of actions and variables: the (A', V')-*restriction* of an (A, V)-sequence α is obtained by first projecting all trajectories of α on the variables in V', then removing the actions not in A', and finally concatenating all adjacent trajectories.

3. Timed Automata

A timed automaton is a state machine whose states are divided into *variables*, and that has a set of discrete *actions*, which are classified as internal or external. We postpone the classification of external actions as input or output until Section 5. The reason for this delay is simply that many of the basic results about timed I/O automata do not depend on the distinction. The state of a timed automaton may change in two ways: by *discrete transitions*, which change the state atomically, and by *trajectories*, which describe the evolution of the state over intervals of time. Discrete transitions are labeled with actions; these are used to synchronize transitions of different automata when the automata are composed in parallel.

Timed automata definition. A timed automaton is exactly a hybrid automaton in the sense of [17] that has no external variables. Formally, a *timed automaton (TA)* consists of:

- A set X of *internal variables*.
- A set $Q \subseteq val(X)$ of *states*.
- A nonempty set $\Theta \subseteq Q$ of *start states*.
- A set E of *external actions* and a set H of *internal actions*. We write A for the set $E \cup H$ of all actions.
- A set $\mathcal{D} \subseteq Q \times A \times Q$ of *discrete transitions*. We use $\mathbf{x} \xrightarrow{a} \mathbf{x}'$ as shorthand for $(\mathbf{x}, a, \mathbf{x}') \in \mathcal{D}$. We say that a is *enabled* in $\mathbf{x}$ if $\mathbf{x} \xrightarrow{a} \mathbf{x}'$ for some $\mathbf{x}'$.
- A set $\mathcal{T}$ of trajectories for X such that $\tau(t) \in Q$ for every $\tau \in \mathcal{T}$ and every t in the domain of τ.

We require that the set of trajectories be closed under the operations of prefix, suffix, and concatenation.

The definition above differs from previous definitions of timed automata [22, 21, 29] in two major respects. First,

Automaton $Channel(b, M)$ **where** $b \in \mathsf{R}^+$

Variables X :	**discrete** $queue \in (M \times \mathsf{R})^*$ **initially** `empty` **analog** $now \in \mathsf{R}$ **initially** 0
States Q :	$val(X)$ **where** $\forall (m, u) \in queue.\ (now \leq u)$
Actions A :	**external** $send(m), receive(m)$ **where** $m \in M$
Transitions $\mathcal{D}$:	**external** $send(m)$ **effect** add $(m, now + b)$ to $queue$ **external** $receive(m)$ **precondition** $\exists u.\ (m, u)$ is first element of $queue$ **effect** remove first element of $queue$
Trajectories $\mathcal{T}$:	**satisfies** **constant**$(queue)$ $\mathbf{d}(now) = 1$

Figure 1. Time-bounded channel

the states are structured through the presence of typed variables. And second, the set of trajectories appears explicitly as a component of the automaton. In the previous definitions, time-passage is represented by special time-passage actions and trajectories are defined only as as auxiliary functions used in describing the effects of the time passage actions on states. Also note that we allow the set Q of states to be a subset of all possible valuations for the state variables (rather than all such valuations). This is because when modeling actual systems one often encounters valuations which are not reachable from any initial state, and which in fact one prefers not to consider as states. Typical examples are the valuations that do not satisfy the "location invariants" of Uppaal style timed automata [3, 15].

Example 3.1 (Time-bounded channel) The automaton in Figure 1 is the specification of a reliable FIFO channel that delivers its messages within a certain time bound, represented by the automaton parameter b, which is a positive real number. The other automaton parameter M represents the type of messages communicated by the channel.

The states of the automaton are valuations of the state variables $queue$ and now. The discrete variable $queue$ holds a finite sequence of pairs consisting of a message that has been sent and its delivery deadline. The analog variable now records the current real time. The restriction on states that the value of now is less than the delivery deadline of any message in $queue$ implies that, within a trajectory, time cannot pass beyond the point where now becomes equal to the delivery deadline of some message in the queue.

Automaton $Sync(u, \rho)_i$ **where** $u \in \mathsf{R}^+, 0 \leq \rho < 1, i \in I$

Variables X :	**analog** $physclock \in \mathsf{R}$ **initially** 0 **discrete** $nextsend \in \mathsf{R}$ **initially** 0 **discrete** $maxother \in \mathsf{R}$ **initially** 0
Derived variables:	$logclock = max(maxother, physclock)$
States Q :	$val(X)$
Actions A :	**external** $send(m)_i, receive(m)_{j,i}$ **where** $m \in \mathsf{R}, j \in I, j \neq i$
Transitions $\mathcal{D}$:	**external** $send(m)_i$ **precondition** $m = physclock$ $physclock = nextsend$ **effect** $nextsend := nextsend + u$ **external** $receive(m)_{j,i}$ **effect** $maxother := max(maxother, m)$
Trajectories $\mathcal{T}$:	**satisfies** **constant**$(nextsend)$ **constant**$(maxother)$ $1 - \rho \leq \mathbf{d}(physclock) \leq 1 + \rho$ **stops when** $physclock = nextsend$

Figure 2. Clock synchronization

Every $send(m)$ transition adds to the queue a new pair whose first component is m and whose second component is the deadline $now + b$. A $receive(m)$ transition can occur only when m is the first message in the queue and it results in the removal of the first message from the queue.

The trajectory specification shows that the discrete variable $queue$ is kept constant by trajectories and that the variable now increases with rate 1, that is, at the same rate as real time. ■

Example 3.2 (Clock synchronization algorithm) The code in Figure 2 describes a timed automaton, $Sync(u, \rho)_i$, representing a single process participating in a clock synchronization algorithm. Each process has a physical clock, represented by the variable $physclock$, and generates a logical clock, $logclock$. The physical clocks may drift from the real time with a drift rate bounded by ρ. The goal of the algorithm is to achieve "agreement" and "validity" among the logical clock values. Agreement means that the logical clocks are close to one another. Validity means that the logical clocks are within the range of the physical clocks.

The algorithm is based on the exchange of physical clock values between processes in the system. The parameter u determines the frequency of sending messages. The variable $nextsend$ records the next time at which the process is supposed to send its physical clock to the other processes.

The process uses the variable $maxother$ to keep track of the largest physical clock value of the other processes in the system. The logical clock, $logclock$, is defined to be the maximum of $maxother$ and $physclock$. Formally $logclock$ is a *derived variable*, which is a function whose value is defined in terms of the state variables.

A $send$ transition is enabled when $nextsend = physclock$. The message sent is the current $physclock$. This transition updates the value of $nextsend$ so that the next $send$ can occur when $physclock$ has advanced by u time units. The transition definition for $receive(m)_{j,i}$ specifies the effect of receiving a message from another process j in the system. Upon receiving a message m from j, i sets $maxother$ to the maximum of m and the current value of $maxother$, thereby updating its knowledge of the largest physical clock value of other processes.

The trajectory specification expresses the fact that the discrete variables do not change, and $physclock$ drifts with a rate bounded by ρ. Periodic sending is enforced through a **stops when** clause: if the predicate $physclock = nextsend$ in this clause becomes true at a point t in time, then t must be the limit time of the trajectory. An alternative way to enforce periodic sending would have been to restrict the set of states to those that satisfy $physclock \leq nextsend$. ■

Executions and traces. An *execution fragment* of a TA $\mathcal{A}$ is an (A, V)-sequence $\alpha = \tau_0\ a_1\ \tau_1\ a_2\ \tau_2 \ldots$, where A and V are all the actions and variables of $\mathcal{A}$, respectively, where each τ_i is a trajectory of $\mathcal{A}$, and for every i, $\tau_i.lval \overset{a_{i+1}}{\rightarrow} \tau_{i+1}.fval$. An execution fragment records what happens during a particular run of a system, including all the discrete state changes and all the changes that occur while time advances. We write $\alpha.fstate$ for the first state of α, and if α is a closed hybrid sequence then we write $\alpha.lstate$ for the last state of α. An *execution* is an execution fragment whose first state is a start state of $\mathcal{A}$.

The external behavior of a TA is captured by the set of "traces" of its execution fragments, which record external actions and the intervening passage of time. Formally, the *trace* of an execution fragment α is the $(E, \emptyset)$-restriction of α. Thus, a trace is a hybrid sequence consisting of external actions of $\mathcal{A}$ and trajectories over the empty set of variables. The only interesting information contained in these trajectories is the amount of time that elapses. A *trace fragment* of $\mathcal{A}$ is the trace of an execution fragment of $\mathcal{A}$, and a *trace* of $\mathcal{A}$ is the trace of an execution of $\mathcal{A}$. In some earlier timed automaton models [22, 29], an execution fragment is defined in a similar style to the one presented here, that is, as an alternating sequence of trajectories and actions. However, a trace is defined differently—as a sequence of actions paired with their times of occurrence. We do not see any advantages of this style of definition, whereas the new definition clearly increases uniformity.

Implementation relationships. Timed automata $\mathcal{A}$ and $\mathcal{B}$ are *comparable* if they have the same external actions. If $\mathcal{A}$ and $\mathcal{B}$ are comparable then $\mathcal{A}$ *implements* $\mathcal{B}$, denoted by $\mathcal{A} \leq \mathcal{B}$, if the traces of $\mathcal{A}$ are a subset of the traces of $\mathcal{B}$.

Simulation relations provide sufficient conditions for showing that one automaton implements another. In [14], we define several types of simulation relations for timed automata, including forward simulations, backward simulations, history and prophecy relations. Here, we define only the most important type, a forward simulation relation.

Let $\mathcal{A}$ and $\mathcal{B}$ be comparable TAs. A *forward simulation* from $\mathcal{A}$ to $\mathcal{B}$ is a relation R from states of $\mathcal{A}$ to states of $\mathcal{B}$ satisfying the following conditions, for all states $\mathbf{x}_\mathcal{A}$ and $\mathbf{x}_\mathcal{B}$ of $\mathcal{A}$ and $\mathcal{B}$:

1. If $\mathbf{x}_\mathcal{A} \in \Theta_\mathcal{A}$ then there exists a state $\mathbf{x}_\mathcal{B} \in \Theta_\mathcal{B}$ such that $\mathbf{x}_\mathcal{A}\ R\ \mathbf{x}_\mathcal{B}$.
2. If $\mathbf{x}_\mathcal{A}\ R\ \mathbf{x}_\mathcal{B}$ and α is an execution fragment of $\mathcal{A}$ consisting of one action surrounded by two point trajectories, with $\alpha.fstate = \mathbf{x}_\mathcal{A}$, then $\mathcal{B}$ has a closed execution fragment β with $\beta.fstate = \mathbf{x}_\mathcal{B}$, $trace(\beta) = trace(\alpha)$, and $\alpha.lstate\ R\ \beta.lstate$.
3. If $\mathbf{x}_\mathcal{A}\ R\ \mathbf{x}_\mathcal{B}$ and α is an execution fragment of $\mathcal{A}$ consisting of a single closed trajectory, with $\alpha.fstate = \mathbf{x}_\mathcal{A}$, then $\mathcal{B}$ has a closed execution fragment β with $\beta.fstate = \mathbf{x}_\mathcal{B}$, $trace(\beta) = trace(\alpha)$, and $\alpha.lstate\ R\ \beta.lstate$.

Theorem 3.3 *If $\mathcal{A}$ and $\mathcal{B}$ are comparable TAs and there is a forward simulation from $\mathcal{A}$ to $\mathcal{B}$, then $\mathcal{A}$ implements $\mathcal{B}$.*

Example 3.4 (A simulation relation) In this example, we define a forward simulation from $Sync(u, \rho)_i$ of Figure 2 to an automaton $SendVal(u, \rho)_i$ that simply sends multiples of u. Code for this automaton is given in Figure 3.

The natural number typed variable $counter$ keeps track of the multiple u to be sent next, and variable now contains the current time. The automaton parameter ρ is used in the precondition of the $send$ and in the stopping condition to enforce bounds on the times of occurrence of $send$.

We now define a forward simulation R from the automaton $Sync(u, \rho)_i$ to $SendVal(u, \rho)_i$ where u and ρ are actual parameters. If $\mathbf{x}$ is a state of $Sync(u, \rho)_i$ and $\mathbf{y}$ is a state of $SendVal(u, \rho)_i$, then $\mathbf{x}\ R\ \mathbf{y}$ provided that the following conditions are satisfied:

1. $\mathbf{y}(now)(1 - \rho) \leq \mathbf{x}(physclock) \leq \mathbf{y}(now)(1 + \rho)$.
2. $\mathbf{y}(counter) = \mathbf{x}(nextsend)/u$. ■

Composition. The composition operation for timed automata allows an automaton representing a complex system to be constructed by composing automata representing individual system components. Our composition operation identifies external actions with the same

Automaton $SendVal(u,\rho)_i$ **where** $u \in \mathsf{R}^+, 0 \le \rho < 1, i \in I$

Variables X :	**discrete** $counter \in \mathsf{N}$ **initially** 0 **analog** $now \in \mathsf{R}$ **initially** 0
States Q :	$val(X)$
Actions A :	**external** $send(m)_i, receive(m)_{j,i}$ **where** $m \in \mathsf{R}, j \in I, j \neq i$
Transitions $\mathcal{D}$:	**external** $send(m)_i$ **precondition** $m = counter \times u$ $counter \times u/(1+\rho) \le now$ **effect** $counter := counter + 1$ **external** $receive(m)_{j,i}$
Trajectories $\mathcal{T}$:	**satisfies** **constant**$(counter)$ **d**$(now) = 1$ **stops when** $now = counter \times u/(1-\rho)$

Figure 3. Automaton $SendVal$

name in different component automata. When any component performs a discrete transition involving an action a, so do all components that have a as an external action. All the components perform trajectories together, allowing the same amount of time to pass. The composition operator for timed automata is simpler than the one for general hybrid automata [17] since all the variables in a timed automaton are internal.[1]

We say that timed automata $\mathcal{A}_1$ and $\mathcal{A}_2$ are *compatible* if they have no state variables in common, and if neither automaton has an internal action that is an action of the other automaton. If $\mathcal{A}_1$ and $\mathcal{A}_2$ are compatible then their *composition* $\mathcal{A}_1 \| \mathcal{A}_2$ is defined formally to be the timed automaton $\mathcal{A} = (X, Q, \Theta, E, H, \mathcal{D}, \mathcal{T})$ where:

- $X = X_1 \cup X_2$.
- $Q = \{\mathbf{x} \in val(X) \mid \mathbf{x} \lceil X_1 \in Q_1 \wedge \mathbf{x} \lceil X_2 \in Q_2\}$.
- $\Theta = \{\mathbf{x} \in Q \mid \mathbf{x} \lceil X_1 \in \Theta_1 \wedge \mathbf{x} \lceil X_2 \in \Theta_2\}$.
- $E = E_1 \cup E_2$ and $H = H_1 \cup H_2$.
- For each $\mathbf{x}, \mathbf{x}' \in Q$ and each $a \in A$, $\mathbf{x} \xrightarrow{a}_{\mathcal{A}} \mathbf{x}'$ iff for $i \in \{1,2\}$, either (1) $a \in A_i$ and $\mathbf{x} \lceil X_i \xrightarrow{a}_i \mathbf{x}' \lceil X_i$, or (2) $a \notin A_i$ and $\mathbf{x} \lceil X_i = \mathbf{x}' \lceil X_i$.
- $\mathcal{T} = \{\tau \in trajs(X) \mid \tau \downarrow X_1 \in \mathcal{T}_1 \wedge \tau \downarrow X_2 \in \mathcal{T}_2\}$.

The following fundamental theorem relates the set of traces of a composed automaton to the sets of traces of its components. Set inclusion in one direction expresses the idea that a trace of a composition "projects" to yield traces of the components. Set inclusion in the other direction expresses the idea that traces of components can be "pasted" to yield a trace of the composition.

1 The composition operation for general hybrid automata requires external variables to be identified as well as external actions. When any component automaton follows a particular trajectory for an external variable v, then so do all component automata of which v is an external variable.

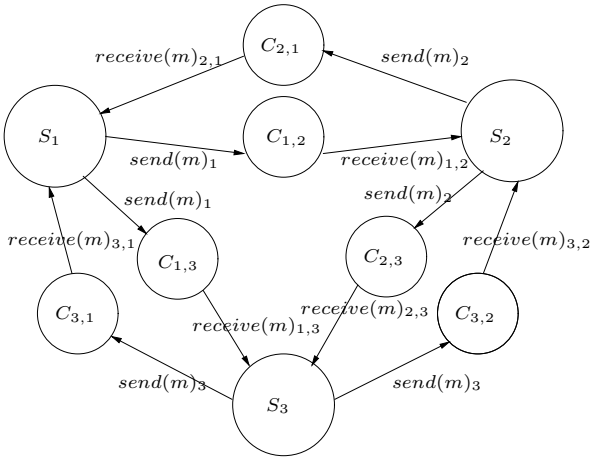

Figure 4. Clock synchronization network

Theorem 3.5 *Let $\mathcal{A} = \mathcal{A}_1 \| \mathcal{A}_2$. Then the set of traces of $\mathcal{A}$ is exactly the set of $(E, \emptyset)$-sequences whose restrictions to $\mathcal{A}_1$ and $\mathcal{A}_2$ are traces of $\mathcal{A}_1$ and $\mathcal{A}_2$, resp.*

The following theorem is a standard kind of "substitutivity" result:

Theorem 3.6 *Suppose $\mathcal{A}_1$, $\mathcal{A}_2$, and $\mathcal{B}$ are TAs, $\mathcal{A}_1$ and $\mathcal{A}_2$ have the same external actions, and each of $\mathcal{A}_1$ and $\mathcal{A}_2$ is compatible with $\mathcal{B}$. If $\mathcal{A}_1 \le \mathcal{A}_2$ then $\mathcal{A}_1 \| \mathcal{B} \le \mathcal{A}_2 \| \mathcal{B}$.*

Example 3.7 (Clock synchronization) We consider a system defined as the composition of a finite set of clock synchronization automata, one for each $i \in I$, plus time-bounded channel automata connecting all the pairs of clock synchronization automata. We assume that the channel automata are defined as in Example 3.1 where $receive$ and $send$ actions in each instance are renamed so that they can be shared with the right clock synchronization automata. Figure 4 illustrates the composed system in the case of three clock synchronization automata. Here S_i abbreviates $Synch(u,\rho)_i$, and C abbreviates $Channel(b, \mathsf{R}^+)$.

In [14], we state and prove several invariants expressing interesting properties for the composed system. For example, the difference between any physical clock and the real time at time t is no more than $t\rho$, and consequently, the difference between any two physical clocks is at most $2t\rho$:

Invariant 1 : *In any reachable state* $\mathbf{x}$*, at any time* t*, for all* i, j*:*

1. $|\mathbf{x}(S_i.physclock) - t| \leq t\rho$.
2. $|\mathbf{x}(S_i.physclock) - \mathbf{x}(S_j.physclock)| \leq 2t\rho$.

The following invariant expresses the main *validity property* for the algorithm: that the logical clock values of all the processes are always between the minimum and maximum physical clock values in the system.

Invariant 2 : *In any reachable state* $\mathbf{x}$ *there exist* j, k *such that for all* i*:*

$$\begin{aligned} \mathbf{x}(S_j.physclock) &\leq \mathbf{x}(S_i.logclock) \\ &\leq \mathbf{x}(S_k.physclock). \end{aligned}$$

It follows that all the logical clocks differ from real time at time t by at most $t\rho$:

Invariant 3 : *In any reachable state* $\mathbf{x}$*, at any time* t*, for any* i*:* $|\mathbf{x}(S_i.logclock) - t| \leq t\rho$.

Finally, we show an *agreement property* for logical clocks. It says that the difference between two logical clocks is always bounded by a constant (which depends on the message-sending interval and the bounds on clock drift and message delay).

Invariant 4 : *In any reachable state* $\mathbf{x}$*, for any* i, j*:* $|\mathbf{x}(S_i.logclock) - \mathbf{x}(S_j.logclock)| \leq u + b(1 + \rho)$.

■

4. Properties for Timed Automata

A *property* P for a TA $\mathcal{A}$ is defined to be any subset of the execution fragments of $\mathcal{A}$. In this section, we define what it means for a property to be a safety or a liveness property and present some basic results that relate these special classes of properties. The classification of properties as safety or liveness properties is important because such properties are proved using different techniques.

Safety and liveness properties. A property P for a TA $\mathcal{A}$ is said to be a *safety* property if it is closed under prefix and limits of execution fragments. In other words, if an execution fragment satisfies a safety property P, then so do all its prefixes, and if all the executions in a "chain" of successive extensions satisfy P, then so does the "limit" of the chain.

A property P for $\mathcal{A}$ is defined to be a *liveness* property provided that for any closed execution fragment α of $\mathcal{A}$, there exists an execution fragment β such that $\alpha \frown \beta \in P$. In other words, no matter how $\mathcal{A}$ behaves for a finite period of time, it is still possible for it to continue in some way and satisfy P.

These definitions of safety and liveness are considered to be standard for untimed systems [2, 6, 8], and they have also been adopted in some models for timed systems (e.g., [29, 13, 1]). The following results show standard facts about safety and liveness: that no property of a TA is both a safety property and a liveness property, and that every property can be expressed as the intersection of a safety and a liveness property. The proofs are given in [14].

Theorem 4.1 *Let* $\mathcal{A}$ *be a TA. If* P *is both a safety property and a liveness property for* $\mathcal{A}$*, then* P *is the set of all execution fragments of* $\mathcal{A}$*.*

Theorem 4.2 *Let* $\mathcal{A}$ *be a TA. If* P *is a property for* $\mathcal{A}$*, then there exists a safety property* S *and a liveness property* L *for* $\mathcal{A}$ *such that* $P = S \cap L$*.*

Fairness properties. Proving interesting liveness properties requires some assumptions saying that certain activities in the system get "enough" chances to try to make progress [20, 1, 28]. Fairness properties are special kinds of liveness properties that express this informal idea. Two important notions of fairness can be formulated for TAs as follows.

Let $\mathcal{A}$ be a TA and let C be a subset of the actions of $\mathcal{A}$. Let α be an execution fragment of $\mathcal{A}$. Then:

1. α is *weakly fair* for C if (at least) one of the following conditions holds:
   (a) α contains infinitely many events from C.
   (b) There is no suffix β of α such that C is enabled in all states of β.
2. α is *strongly fair* for C if (at least) one of the following conditions holds:
   (a) α contains infinitely many events from C.
   (b) There is some suffix β of α such that C is disabled in all states of β.

History-independent properties. A property P of a TA $\mathcal{A}$ is said to be *history-independent* provided that the following holds: If α' is a suffix of an execution fragment α, then α satisfies P if and only if α' does. In other words, whether or not α satisfies P is determined only by what happens in its suffixes—it is not affected by what happens in any initial portion of α. If a property P is known to be history-independent, then one can prove that an execution fragment α satisfies P by considering the portion of α from some point onward. Weak fairness and strong fairness (as defined just above) and admissibility (as defined in Section 2) are all history-independent properties:

Theorem 4.3 *For any TA* $\mathcal{A}$*, and a subset* C *of its actions, the sets of weakly fair execution fragments for* C *and strongly fair execution fragments for* C *are history-independent.*

Theorem 4.4 *For any TA $\mathcal{A}$, the set of admissible execution fragments is history-independent.*

5. Timed I/O Automata

In this section we refine the timed automaton model of Section 3 by distinguishing input and output actions. We extend the results on simulation relations and composition from Section 3 to this new setting. We also introduce special kinds of timed I/O automata: I/O feasible, progressive, and receptive TIOAs.

Timed I/O automata definition. A timed I/O automaton is a timed automaton in the sense of Section 3, where the set of actions is partitioned into a set of input actions, which are used to model the actions performed by the environment, and a set of output actions, which are under the control of the automaton. Formally, a *timed I/O automaton (TIOA)* $\mathcal{A}$ is a tuple $(\mathcal{B}, I, O)$ where

- $\mathcal{B} = (X, Q, \Theta, E, H, \mathcal{D}, \mathcal{T})$ is a timed automaton.
- I and O partition E into *input* and *output actions*, respectively. Actions in $L \triangleq H \cup O$ are called *locally controlled*; as before we write $A \triangleq E \cup H$.

We require two additional axioms, which express input-enabling conditions for actions and for time-passage, respectively:

E1 *Input action enabling:* For every $\mathbf{x} \in Q$ and every $a \in I$, there exists $\mathbf{x}' \in Q$ such that $\mathbf{x} \xrightarrow{a} \mathbf{x}'$.

E2 *Time-passage enabling:* For every $\mathbf{x} \in Q$, there exists $\tau \in \mathcal{T}$ such that $\tau.fstate = \mathbf{x}$ and either

1. $\tau.ltime = \infty$, or
2. τ is closed and some $l \in L$ is enabled in $\tau.lstate$.

E1 is the usual input enabling condition of ordinary I/O automata [20]; it says that a TIOA is able to accomodate an input action whenever it arrives. **E2** says that a TIOA either allows time to advance forever, or it allows time to advance for a while, up to a point where it is prepared to react with some locally controlled action. Because TIOAs have no external variables, **E1** and **E2** are slightly simpler than the corresponding axioms for HIOAs.

Example 5.1 The time-bounded channel described in Example 3.1 can be turned into a TIOA by classifying the $send$ actions as inputs, and the $receive$ actions as outputs. Since there is no precondition for $send$ actions, they are enabled in each state, so clearly the input enabling condition **E1** holds. It is also easy to see that axiom **E2** holds: in each state either $queue$ is nonempty, in which case a $receive$ output action is enabled after a point trajectory, or $queue$ is empty, in which case time can advance forever.

The clock synchronization automaton of Example 3.2 can be turned into a TIOA by classifying the $send$ actions as outputs, and the $receive$ actions as inputs. Axiom **E1** then holds trivially. Axiom **E2** holds since from each state either time can advance forever, or we have an outgoing trajectory (possibly of length 0) to a state in which $physclock = nextsend$, and from there a $send$ output action is enabled. ■

Execution and traces. An *execution fragment*, *execution*, *trace fragment*, or *trace* of a TIOA $\mathcal{A}$ is defined to be an execution fragment, execution, trace fragment, or trace of its underlying timed automaton, respectively.

We introduce one new definition here, to capture the idea that, in a particular execution fragment, the automaton itself produces Zeno behavior. Namely, we say that an execution fragment of a TIOA is *locally-Zeno* if it is Zeno and contains infinitely many locally controlled actions, or equivalently, if it has finite limit time and contains infinitely many locally controlled actions.

Implementation relationships. TIOAs $\mathcal{A}$ and $\mathcal{B}$ are *comparable* if they have the same input actions and the same output actions. If $\mathcal{A}$ and $\mathcal{B}$ are comparable then we say that $\mathcal{A}$ *implements* $\mathcal{B}$, denoted by $\mathcal{A} \leq \mathcal{B}$, if the traces of $\mathcal{A}$ are a subset of the traces of $\mathcal{B}$. The definition of forward simulation for TIOAs is the same as for TAs.

Theorem 5.2 *If $\mathcal{A}$ and $\mathcal{B}$ are comparable TIOAs and there is a forward simulation from $\mathcal{A}$ to $\mathcal{B}$, then $\mathcal{A}$ implements $\mathcal{B}$.*

Composition. The definition of composition for TIOAs is based on the corresponding definition for TAs, but also takes the input/output structure into account. Namely, we say that TIOAs $\mathcal{A}_1$ and $\mathcal{A}_2$ are *compatible* if they satisfy the compatible conditions for TAs, and also they have no output actions in common. A consequence of these conditions is that each action is controlled by at most one component.

If $\mathcal{A}_1$ and $\mathcal{A}_2$ are compatible TIOAs then their *composition* $\mathcal{A}_1 \| \mathcal{A}_2$ is defined to be the TIOA $\mathcal{A} = (\mathcal{B}, I, O)$ where $\mathcal{B}$ is the parallel composition $\mathcal{B}_1 \| \mathcal{B}_2$, $I = (I_1 \cup I_2) - (O_1 \cup O_2)$, and $O = O_1 \cup O_2$. That is, an external action of the composition is classified as an output if it is an output of one of the component automata, and otherwise it is classified as an input.

In the TIOA setting, it is straightforward to show that the composition of compatible TIOAs is in fact a TIOA. This is less straightforward in the more general HIOA setting. Theorem 6.12 of [17], which asserts the corresponding fact for HIOAs, requires an additional hypothesis: that HIOAs $\mathcal{A}_1$ and $\mathcal{A}_2$ be "strongly compatible". This extra condition is needed to rule out dependencies between external variables that may prevent the component automata from evolving together. The absence of external variables in TIOA elimi-

nates this kind of problematic behavior and gives rise to a simpler theory of composition.

The projection and pasting theorem and the substitutivity theorem for timed automata, Theorems 3.5 and 3.6, apply to TIOAs as well.

I/O feasibility. We define *I/O feasibility*, which is a basic requirement that reasonable TIOAs should satisfy. It says that the automaton is capable of providing some response from any state, for any sequence of input actions and any amount of intervening time-passage. In particular, it should allow time to pass to infinity if the environment does not submit any input actions. Formally, we define a TIOA to be *I/O feasible* provided that, for each state $\mathbf{x}$ and each $(I, \emptyset)$-sequence β, there is some execution fragment α from $\mathbf{x}$ such that $\alpha \lceil (I, \emptyset) = \beta$. That is, an I/O feasible TIOA accommodates arbitrary input actions occurring at arbitrary times. The given $(I, \emptyset)$-sequence β describes the inputs and the amounts of intervening times.

Unfortunately, it turns out that I/O feasibility is not preserved by composition of TIOAs:

Example 5.3 (I/O feasible TIOAs whose composition is not I/O feasible) Consider two I/O feasible TIOAs $\mathcal{A}$ and $\mathcal{B}$, where $O_{\mathcal{A}} = I_{\mathcal{B}} = \{a\}$ and $O_{\mathcal{B}} = I_{\mathcal{A}} = \{b\}$. Suppose that $\mathcal{A}$ performs its output a at time 0 and then waits, allowing time to pass, until it receives input b. If and when it receives b, it responds with output a without allowing any time to pass, and ignoring any inputs that occur before it has a chance to perform its output. On the other hand, $\mathcal{B}$ starts out waiting, allowing time to pass, until it receives input a. If and when it receives a, it responds with output b without allowing time to pass.

It is not difficult to see that each of $\mathcal{A}$ and $\mathcal{B}$ is I/O feasible. However, the composition $\mathcal{A}\|\mathcal{B}$ is not I/O feasible. To see this, consider the start state of $\mathcal{A}\|\mathcal{B}$ and the unique input hybrid sequence β with $\beta.ltime = \infty$; β contains no actions, but simply allows time to pass to infinity. The composition $\mathcal{A}\|\mathcal{B}$ has no way of accommodating this input, since it will never allow time to pass beyond 0. ■

The fact that the set of I/O feasible TIOAs is not preserved by composition is inconvenient; it means that we cannot verify this important property in a compositional manner. This motivated us to define the more restrictive notion of *receptiveness* for TIOAs. Receptiveness is a natural condition that implies I/O feasibility, and that also is preserved by composition.

We build our definition of receptiveness on a preliminary definition of *progressiveness* for TIOAs. Namely, we define a *strategy* for resolving nondeterministic choices, and define receptiveness in terms of the existence of a progressive strategy. The approach follows that in [17].

Progressiveness. A progressive TIOA is simply one that never generates infinitely many locally controlled actions in finite time. Formally, a TIOA is *progressive* if it has no locally-Zeno execution fragments. The following theorem says that a progressive TIOA is capable of allowing arbitrary input actions at arbitrary times.

Theorem 5.4 *Every progressive TIOA is I/O feasible.*

The idea behind the proof is the following. Given a state x and an $(I, \emptyset)$-sequence β, we construct the needed execution fragment α recursively, spanning one input trajectory at a time. To span each trajectory, we apply axiom **E2** repeatedly; the progressiveness assumption implies that we do not "get stuck" in a trajectory, performing infinitely many locally controlled actions. To process the input actions in between the trajectories, we use axiom **E1**.

The following theorem says that progressiveness is preserved by composition:

Theorem 5.5 *If $\mathcal{A}_1$ and $\mathcal{A}_2$ are compatible progressive TIOAs, then their composition is also progressive.*

The idea behind the proof is that a Zeno execution of $\mathcal{A}_1\|\mathcal{A}_2$ with infinitely many locally controlled actions contains infinitely many locally controlled actions of either $\mathcal{A}_1$ or $\mathcal{A}_2$. But this would violate progressiveness for one of the component automata.

Receptiveness. Finally, we can define *receptiveness*, a more general condition than progressiveness that implies I/O feasibility and is also preserved by composition. We define a *strategy* for a TIOA $\mathcal{A}$ to be a TIOA $\mathcal{A}'$ that differs from $\mathcal{A}$ only in that $\mathcal{D}' \subseteq \mathcal{D}$ and $\mathcal{T}' \subseteq \mathcal{T}$. That is, a strategy selects a subset of the discrete transitions and trajectories of the original automaton. Our strategies are nondeterministic and memoryless. They provide a way of choosing some of the evolutions that are possible from each state $\mathbf{x}$ of $\mathcal{A}$. The fact that the state set Q' of $\mathcal{A}'$ is the same as the state set Q of $\mathcal{A}$ implies that $\mathcal{A}'$ chooses evolutions from every state of $\mathcal{A}$. We define a TIOA to be *receptive* if it has a progressive strategy.

In previous studies of receptiveness for timed automata [9, 1, 29], strategies are defined in terms of two-player games, and receptiveness is defined in terms of the outcome of a strategy. Our new definitions of strategies and receptiveness capture similar ideas in a simpler way.

The following theorem says that a receptive TIOA provides some response from any state, for any sequence of discrete input actions at any times.

Theorem 5.6 *Every receptive TIOA is I/O feasible.*

This result follows easily from Theorem 5.4. The following theorem follows from the definition of composition and strategies.

Theorem 5.7 *Let $\mathcal{A}_1$ and $\mathcal{A}_2$ be compatible TIOAs with strategies $\mathcal{A}'_1$ and $\mathcal{A}'_2$, resp. Then $\mathcal{A}'_1\|\mathcal{A}'_2$ is a strategy for $\mathcal{A}_1\|\mathcal{A}_2$.*

Finally, we can state the main result of this section, which follows from the previous two theorems. It says that the receptiveness is preserved by composition.

Theorem 5.8 *Let $\mathcal{A}_1$ and $\mathcal{A}_2$ be compatible receptive TIOAs with progressive strategies $\mathcal{A}_1'$ and $\mathcal{A}_2'$, resp. Then $\mathcal{A}_1 \| \mathcal{A}_2$ is a receptive TIOA with progressive strategy $\mathcal{A}_1' \| \mathcal{A}_2'$.*

Thus, the TIOA model has simpler and stronger composition theorems than the general HIOA model. In particular, the main compositionality result for receptive HIOAs has a more complicated statement and proof than ours. The result for HIOAs makes an assumption about the existence of strongly compatible strategies and uses an additional technical lemma about strongly compatible strategies.

Example 5.9 The time-bounded channel automaton described in Example 3.1 is not progressive since it allows for an infinite execution in which $send$ and $receive$ actions alternate without any passage of time in between. The time-bounded channel automaton is receptive, however, as we may construct a progressive strategy for it by adding a condition $u = now$ to the precondition of the $receive$ action. In this way we enforce that the channel operates maximally slow and messages are only delivered at their delivery deadline. The clock synchronization automaton of Example 3.2 is progressive (and therefore receptive) since it can only generate a locally controlled action once every u time units. Theorem 5.8 now implies that the network of clock synchronization automata described in Example 3.7 is also receptive, and hence (Theorem 5.6) I/O feasible. ■

6. Properties for Timed I/O Automata

A *property* for a TIOA $\mathcal{A} = (\mathcal{B}, I, O)$ is defined to be a property of its underlying TA, that is, it is a subset of the execution fragments of $\mathcal{B}$. The most comprehensive study to date of properties for I/O automaton models can be found in [29]. In that work, as in similar works for other models [9, 1, 29, 18], receptive strategies are used to describe how a system interacts with its environment to guarantee that the outcome of the interaction satisfies a liveness property. Although the HIOA modeling framework includes simpler definitions for strategies and receptiveness, it does not address general liveness properties. In this section, we present new definitions and results about receptiveness for properties, which we think are simpler than those found in prior work [29, 18]. We show that receptiveness implies liveness and that it is compositional. We refer the reader to [14] for the proofs.

I/O liveness properties. In Section 4, we defined general liveness properties for timed automata. We now refine our notion of a liveness property to take the input/output distinction into account. A property P for a TIOA $\mathcal{A}$ is defined to be an *I/O liveness* property provided that for each closed execution fragment α of $\mathcal{A}$ and each $(I, \emptyset)$-sequence β, there is some execution fragment α' such that $\alpha' \lceil (I, \emptyset) = \beta$ and $\alpha \frown \alpha' \in P$. In other words, no matter how $\mathcal{A}$ behaves for a finite period of time, and no matter what inputs arrive, it is still possible for $\mathcal{A}$ to continue in some way and satisfy P. The following theorem relates I/O feasibility and I/O liveness.

Theorem 6.1 *A TIOA is I/O feasible if and only if its set of execution fragments is an I/O liveness property.*

Receptiveness for properties. If we would define a live TIOA to be a pair $(\mathcal{A}, L)$ of a TIOA $\mathcal{A}$ coupled with an I/O liveness property L then the resulting class of systems would not be closed under composition. The problem, and this was noted already in previous studies of liveness properties for timed I/O automata such as [29], is that this definition allows a system to choose its relative speed with respect to the environment, and to base its decisions on the future behavior of the environment. As a result, the live preorder is not substitutive for parallel composition. To solve these problems, previous studies have introduced notions of *receptive strategies* to guarantee that a system does not constrain its environment. The TIOA framework incorporates a simpler (although less general) notion of strategy than those considered in previous work on timed I/O automata [29].

We begin with a definition of receptiveness for a property. Let $\mathcal{A}$ be a TIOA and let P be a property for $\mathcal{A}$, that is, a subset of the execution fragments of $\mathcal{A}$. Then we say that $\mathcal{A}$ is *receptive for* P provided that there exists a strategy $\mathcal{A}'$ for $\mathcal{A}$ such that every execution fragment of $\mathcal{A}'$ is in P. That is, $\mathcal{A}$ has a strategy that can always ensure that P is satisfied (regardless of the behavior of the environment).

Theorem 6.2 below says that if $\mathcal{A}$ is receptive for P and P is history-independent, then P must be a liveness property for $\mathcal{A}$. Theorem 6.3 strengthens this result: if we also know that P consists of non-locally-Zeno execution fragments, then P must be an I/O liveness property.

Theorem 6.2 *If $\mathcal{A}$ is receptive for P and P is history-independent, then P is a liveness property for $\mathcal{A}$.*

Theorem 6.3 *If $\mathcal{A}$ is receptive for P, P is history-independent, and P consists only of non-locally-Zeno execution fragments, then P is an I/O liveness property for $\mathcal{A}$.*

The need for the history-independence assumption for the two theorems above stems from the fact that our strategies are memoryless whereas liveness properties are defined in terms of extending closed execution fragments. It might be possible to avoid the history-independence assumption by allowing strategies to have memory, or by modifying the

definition of a liveness property; we leave this for future work.

Finally, we consider composition of TIOAs with properties. If $\mathcal{A}_1$ and $\mathcal{A}_2$ are two compatible timed automata and P_1 and P_2 are properties for $\mathcal{A}_1$ and $\mathcal{A}_2$, respectively, then we define $P_1 \| P_2$ to be the set of execution fragments α of $\mathcal{A}_1 \| \mathcal{A}_2$ such that $\alpha \lceil (A_i, X_i) \in P_i, i \in \{1, 2\}\}$. The following is a simple composition theorem for TIOAs and properties for which they are receptive.

Theorem 6.4 *If $\mathcal{A}_1$ is receptive for P_1 and $\mathcal{A}_2$ is receptive for P_2 then $\mathcal{A}_1 \| \mathcal{A}_2$ is receptive for $P_1 \| P_2$.*

7. Related Work

One of the widely-used formal frameworks for timed systems is that of Alur-Dill timed automata [5, 3]. An Alur-Dill automaton is a finite directed multigraph augmented with a finite set of clock variables. The semantics of such a timed automaton are defined as a state transition system in which each state consists of a location and a clock valuation. Clocks are assumed to change at the same time as real-time. The aim of facilitating automated verification based on reachability analysis seems to be the main motivation for the restrictions on the expressive power of the model. The timed automaton model presented in this paper is more expressive than the model of Alur-Dill automata. In our model, there are no finiteness assumptions and no restrictions imposed on the dynamic type of variables. In [14], we give a semantics for Alur-Dill automata by using a restricted class of our timed automata. Alur-Dill timed automata have been extensively studied from the perspective of model checking. Our focus, on the other hand, has been to develop a general formal framework with a well-defined notion of external behavior, parallel composition and abstraction that supports assertional reasoning with state invariants and simulation relations.

Uppaal [27, 15] is a widely-used modeling and verification tool for timed systems. It supports the description of systems as a network of Alur-Dill timed automata that communicate using CCS-style synchronization [25] and shared variables. In addition it supports other notions such as committed and urgent locations. Uppaal has a sophisticated model-checker that (symbolically) explores the state space of the modeled system to verify timing properties. Finiteness assumptions are built into the model to make such verification possible and the operations on clocks are restricted. For example, it is not possible to add the current value of a clock to messages as a timestamp when they are placed in a buffer. One of our plans for the near future is to work on translations between Uppaal and some variation of our restricted timed I/O automaton model. There are several small mismatches due to the style of communication and notions such as committed locations but we intend to investigate to what extent we can use the communication mechanisms of our automata to model these formally. We could, for example, allow a non-empty set of external variables with restricted dynamic types and seek restrictions on the use of shared variables in Uppaal which would allow us to view these variables as external variables in the HIOA sense.

A slight generalization of Alur-Dill timed automata are the linear hybrid automata of [4]. In this model, apart from clocks that progress with rate 1 one can also use continuous variables whose derivatives are contained in some arbitrary interval. A well-known model checking tool for lineae hybrid automata is HyTech [12]. The input language of HyTech can easily be translated into our TIOA model.

The TIOA framework presented in this paper can be used to express models that use lower and upper time bounds on tasks or actions [24, 23]. Our manuscript [14] presents an operation for adding time bounds on a subset of the actions of a timed automaton. As a result of this operation, lower bounds are transformed to appropriate preconditions for transitions and upper bounds are transformed to restrictions on the set of states.

8. Conclusions

In this paper, we have defined a new timed I/O automaton modeling framework for describing and analyzing the behavior of timed systems. This model is a special case of the recently presented hybrid I/O automaton modeling framework [17]. We used what we have learned in developing the HIOA framework to revise the earlier timed I/O automaton models. Our main motivation was to have a timed I/O automaton model that is compatible with the new HIOA model. We sought to benefit from the new style used in describing hybrid behavior in simplifying the prior definitions and results on timed automata. Moreover, we extended the work on the HIOA model by investigating safety and liveness properties and receptiveness for general liveness, not only for feasibility as in the HIOA framework.

Our paper [14] has a larger scope than we were able to cover in this paper. For example, we have definitions and soundness results for a wide range of simulation relations for timed automata, which we believe apply also to hybrid automata. These include liveness-preserving simulation relations for special kinds of liveness properties. Likewise, we have obtained "assume-guarantee" style compositionality results for our timed automata and are interested in showing that they also hold for hybrid automata. All these suggest that we are not that far from having a unified framework for timed and hybrid systems in which we can collect and summarize previous results of our own work. We are also establishing formal relationships with other models that are comparable to ours. The details of our work in progress can be found in the longer manuscript [14].

References

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

[2] B. Alpern and F.B. Schneider. Defining liveness. *Information Processing Letters*, 21:181–185, 1985.

[3] R. Alur. Timed automata. In *Proc. of 11th International Conference on Computer-Aided Verification (CAV)*, volume 1633 of *LNCS*, pages 8–22. Springer-Verlag, 1999. An earlier and longer version appears in NATO-ASI Summer School on Verification of Digital and Hybrid Systems.

[4] 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.

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

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

[7] H. Attiya and N.A. Lynch. Time bounds for real-time process control in the presence of timing uncertainty. In *Proceedings of the 10th IEEE Real-Time Systems Symposium*, pages 268–284, 1989.

[8] F. Dederichs and R. Weber. Safety and liveness from a methodological point of view. *Information Processing Letters*, 36(1):25–30, 1990.

[9] D. Dill. *Trace Theory for Automatic Hierarchical Verification of Speed-Independent Circuits*. ACM Distinguished Dissertations. MIT Press, 1988.

[10] A. Fehnker, F.W. Vaandrager, and M. Zhang. Modeling and verifying a Lego car using hybrid I/O automata. In M. Broy and M. Pizka, editors, *Models, Algebras, and Logic of Engineering Software*, volume 191 of *NATO ASI Series III*, pages 385–402. IOS Press, 2003.

[11] C. Heitmeyer and N.A. Lynch. The generalized railroad crossing: A case study in formal verification of a real-time system. In *Proceedings of the 15th IEEE Real-Time Systems Symposium*, pages 120–131, 1994.

[12] T. A. Henzinger, P.-H. Ho, and H. Wong-Toi. HYTECH: A Model Checker for Hybrid Systems. In O. Grumberg, editor, *Proceedings of the 9th International Conference on Computer Aided Verification*, volume 1254 of *LNCS*, pages 460–463. Springer-Verlag, 1997.

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

[14] D.K. Kaynar, N.A. Lynch, R. Segala, and F.W. Vaandrager. Timed I/O automata. Technical Report MIT/LCS/TR-917, MIT Laboraty for Computer Science, 2003. Available at `http://theory.lcs.mit.edu/tds/reflist.html`.

[15] K.G. Larsen, P. Pettersson, and W. Yi. Uppaal in a nutshell. *Journal of Software Tools for Technology Transfer*, 1–2:134–152, 1997.

[16] C. Livadas, J. Lygeros, and N.A. Lynch. High-level modeling and analysis of TCAS. In *Proceedings of the 20th IEEE Real-Time Systems Symposium*, pages 115–125, 1999.

[17] N.A. Lynch, R. Segala, and F.W. Vaandrager. Hybrid I/O automata. *Information and Computation*, 185(1):105–157, 2003.

[18] N.A. Lynch, R. Segala, F.W. Vaandrager, and H.B. Weinberg. Hybrid I/O automata. In R. Alur, T.A. Henzinger, and E.D. Sontag, editors, *Hybrid Systems III*, volume 1066 of *LNCS*, pages 496–510. Springer-Verlag, 1996.

[19] N.A. Lynch and N. Shavit. Timing-based mutual exclusion. In *Proceedings of the Real-Time Systems Symposium*, pages 2–11, 1992.

[20] N.A. Lynch and M.R. Tuttle. An introduction to input/output automata. *CWI Quarterly*, 2(3):219–246, September 1989.

[21] N.A. Lynch and F.W. Vaandrager. Action transducers and timed automata. *Formal Aspects of Computing*, 8(5):499–538, 1996.

[22] N.A. Lynch and F.W. Vaandrager. Forward and backward simulations — Part II: Timing-based systems. *Information and Computation*, 128(1):1–25, July 1996.

[23] O. Maler, Z. Manna, and A. Pnueli. From timed to hybrid systems. In J.W. de Bakker, C. Huizing, W.P. de Roever, and G. Rozenberg, editors, *Proceedings REX Workshop on Real-Time: Theory in Practice,* Mook, The Netherlands, June 1991, volume 600 of *LNCS*, pages 447–484. Springer-Verlag, 1992.

[24] M. Merritt, F. Modugno, and M. Tuttle. Time constrained automata. In J.C.M. Baeten and J.F. Groote, editors, *Proceedings CONCUR 91,* Amsterdam, volume 527 of *LNCS*, pages 408–423. Springer-Verlag, 1991.

[25] R. Milner. *A Calculus of Communicating Systems*, volume 92 of *LNCS*. Springer-Verlag, 1980.

[26] S. Mitra, Y. Wang, N.A. Lynch, and E. Feron. Safety verification of model helicopter controller using hybrid input/output automata. In *Hybrid Systems: Computation and Control (HSCC'03),* Prague, the Czech Republic, pages 259–273. Lecture Notes in Computer Science, Springer-Verlag, 2003.

[27] P. Petterson. *Modelling and Verification of Real-Time Systems Using Timed Automata:Theory and Practice*. PhD thesis, Department of Computer Systems, Uppsala University, 1999. Technical Report DoCs 99/101.

[28] J.M.T. Romijn and F.W. Vaandrager. A note on fairness in I/O automata. *Information Processing Letters*, 59(5):245–250, 1996.

[29] R. Segala, R. Gawlick, J.F. Søgaard-Andersen, and N.A. Lynch. Liveness in timed and untimed systems. *Information and Computation*, 141(2):119–171, March 1998.

[30] H. B. Weinberg and N.A. Lynch. Correctness of vehicle control systems - a case study. In *Proceedings of the 17th IEEE Real-Time Systems*, pages 62–72, 1996.