Liveness in Timed and Untimed Systems*

Roberto Segala[†] Rainer Gawlick[‡] Jørgen Søgaard-Andersen[§] Nancy Lynch[¶]

Abstract

When proving the correctness of algorithms in distributed systems, one generally considers *safety* conditions and *liveness* conditions. The Input/Output (I/O) automaton model and its timed version have been used successfully, but have focused on safety conditions and on a restricted form of liveness called fairness. In this paper we develop a new I/O automaton model, and a new timed I/O automaton model, that permit the verification of general liveness properties on the basis of existing verification techniques. Our models include a notion of *receptiveness* which extends the idea of *receptiveness* of other existing formalisms, and enables the use of compositional verification techniques. The presentation includes an *embedding* of the untimed model into the timed model which preserves all the interesting attributes of the untimed model. Thus, our models constitute a *coordinated framework* for the description of concurrent and distributed systems satisfying general liveness properties.

Keywords: Automata, timed automata, I/O automata, liveness, receptiveness, formal verification, simulation techniques.

---

*Supported by NSF grant CCR-89-15206, by DARPA contracts N00014-89-J-1988 and N00014-92-J-4033, by ONR contract N00014-91-J-1046, and by ARPA contract F19628-95-C-0118. Also supported in part at the Technical University of Denmark by the Danish Technical Research Council.

[†]Department of Computer Science, University of Bologna, Italy.

[‡]Laboratory for Computer Science, Massachusetts Institute of Technology and McKinsey & Co.

[§]Department of Computer Science, Technical University of Denmark, DK-2800 Lyngby, Denmark.

[¶]Laboratory for Computer Science, Massachusetts Institute of Technology, 545 Technology Square, Cambridge, MA 02139.

Contents

1 Introduction

The increasing need for reliable software has led the scientific community to develop many formalisms for verification. Particularly important are formalisms that can model distributed and concurrent systems and those that can model real time systems, i.e., systems that rely on time constraints in order to guarantee correct behavior. Formalisms should be able to support verification of both *safety* and *liveness* properties [AS85]. Roughly speaking, a liveness property specifies that certain desirable events will eventually occur, while a safety property specifies that undesirable events will never occur.

In this paper, we present a *coordinated framework* that permits modeling and verification of safety and liveness properties for both timed and untimed systems. The framework consists of two models, one timed and one untimed, with an *embedding* of the untimed model into the timed model. Both models come equipped with notions of external behavior and of implementation, which are based simply on *traces*. The framework is intended to support a variety of verification techniques, including *simulation methods*, *compositional reasoning*, *algebraic methods*, and *temporal logic methods*.

A successful technique for the verification of safety properties and some special liveness properties is based on the *simulation method* of [AL91a, LV91, LV93, LV95, Jon91], applied to the *Input/Output automaton model* of [LT87] and to its generalization to the timed case [MMT91]. I/O automata are state machines with a labeled transition relation where the labels, also called *actions*, model communication. A key feature of I/O automata is the explicit distinction between their *input* and *output* actions, which characterize the events under the control of the environment and those under the control of the automaton, respectively. I/O automata can handle general safety properties and can also deal with a special kind of liveness, called *fairness*. Fairness captures the intuitive idea that each subcomponent of a composed system has fair chances to make progress. The notion of *implementation* for I/O automata, i.e., the way a concrete system is said to implement a more abstract specification, is expressed through *fair trace inclusion*, where a fair trace of an I/O automaton is a sequence of actions that can occur whenever the I/O automaton respects its fairness property. I/O automata can be composed in parallel, i.e., they can interact together so that they can be viewed as a single large system. An important property of I/O automata is that the implementation relation is *compositional* in the sense that it is always correct to replace a subcomponent in a large system with one of its implementations. Compositionality is needed for modular design techniques.

Despite its success, the I/O automaton model is not general enough to handle some recent verification work in [SLL93b, SLL93a]. In particular, [SLL93b, SLL93a] provide examples where fairness is not adequate to express liveness naturally. Moreover, the work in [SLL93b, SLL93a] has shown the need for a connection between timed and untimed models to prove that an implementation that uses timing constraints correctly implements an untimed specification. The mutual exclusion algorithm of Fischer [Fis85, AL91b] is another instance of a timed implementation for an untimed specification.

This motivates a generalization of the I/O automaton model and its timed version to handle general liveness properties in such a way that the simulation based proof method still applies.

A simple and natural generalization is motivated by [AL93], which models a machine as a pair (A, L) consisting of an automaton and A and a subset L of its behaviors satisfying the desired liveness property. The implementation notion can then be expressed by *live trace inclusion* just as fair trace inclusion expresses implementation for I/O automata. The use of live trace inclusion as the implementation notion is motivated by the fact that the simulation based proof method is known to work for implementation notions based on some form of trace inclusion. Unfortunately, if L is not restricted, simple examples show that live trace inclusion is not compositional (cf. Examples 3.4 and 3.5).

In this paper we identify the appropriate restrictions on L, in both the untimed model and the timed model, so that live trace inclusion is compositional for the pair (A, L). A pair (A, L) satisfying these restrictions on L is called a *live I/O automaton* in the untimed model and a *live timed I/O automaton* in the timed model. The restrictions on L are given by a property called *receptiveness*[1], which captures the intuitive idea that a live (timed) I/O automaton must not constrain its environment. The receptiveness property is defined, using ideas from [Dil88], by means of a two-person game between a live (timed) I/O automaton and its environment. Specifically, the environment provides arbitrary inputs while the system tries to react so that it behaves according to its liveness condition. A live (timed) I/O automaton (A, L) has a *winning strategy* against its environment if A can respond to any environment move in such a way that it will always lead to a behavior of L. If a live (timed) I/O automaton has a winning strategy, then it is said to be receptive.

The definitions of receptiveness in the untimed and the timed model are closely related. In particular, the receptiveness property for the timed model is a natural extension of the receptiveness property for the untimed model up to some technical details involving the so called *Zeno behaviors*. The close relationship between the receptiveness property in the untimed and the timed model allows the models to be tied together, thus permitting the verification of timed implementations of untimed specifications. Specifically, the paper presents a *patient* operator [NS92, VL92] that converts (untimed) live I/O automata into live timed I/O automata without timing constraints. The patient operator preserves receptiveness and the live trace preorder relation of the untimed model. Thus, the patient operator provides the mechanism by which the timed and untimed models are unified into a coordinated framework.

Our models generalize several existing models. The fairness condition of I/O automata satisfies the receptiveness property; thus, live I/O automata are a proper generalization of I/O automata. Receptiveness also implies feasibility as defined in [LS89]. The failure free complete trace structures of [Dil88] are also properly generalized by our model. In the timed case, our model generalizes [MMT91] and the notion of *strong I/O feasibility* introduced in [VL92]. Finally, in contrast to [AL91b], our timed model does not give either the system or the environment control over the passage of time.

We believe that our coordinated untimed and timed models comprise a good general framework for verification of concurrent systems. Besides the fact that our models generalize several

[1] In our original work [GSSL93, GSSL94] we used the term *environment-freedom*. Due to the close connection between environment-freedom in our untimed model and receptiveness in other existing models, we have uniformed our terminology to the existing literature.

others, our models support the simulation based proof method of [AL91a, LV91, LV93, LV95, Jon91]. In [GSSL93] we show how the simulation based proof method can be used to handle liveness by means of an *Execution Correspondence Theorem*, which extracts from a simulation relation more information than just trace inclusion. Our models have already been used in [SLL93b, SLL93a] to verify a non-trivial communication protocol used in the Internet, and the verifications require all the new expressiveness provided in this paper and the simulation tools provided in [GSSL93].

After some preliminary definitions, given in Section 2, the paper is divided into three main sections. Section 3 presents the untimed model, Section 4 presents the timed model, and Section 5 embeds the untimed model into the timed model by means of the *patient* operator. The presentation of both the untimed and timed models starts with a general automaton model with liveness conditions in the style of [AL91b]; then the I/O distinction is introduced together with the receptiveness property and the proof of compositionality. The presentation of the untimed model also includes several examples that motivate the definition of receptiveness and show that there does not seem to be any trivial generalization of receptiveness that still leads to the compositionality of the live trace preorder. Once live (timed) I/O automata are defined for each model, the paper introduces the corresponding notions of implementation and compares our model with other existing models. The paper ends with some considerations on the generality of receptiveness and additional considerations for further work.

2 Preliminaries

We use "list" and "sequence" synonymously. The empty sequence is denoted by ε. A finite sequence $l_1 = e_1 \ldots e_n$ and a sequence $l_2 = e_{n+1}e_{n+2}\ldots$ can be *concatenated.* The concatenation, written $l_1 l_2$, is the sequence $e_1 \ldots e_n e_{n+1} e_{n+2} \ldots$. A sequence l_1 is a *prefix* of a sequence l_2, written $l_1 \leq l_2$, if either $l_1 = l_2$, or l_1 is finite and there exists a sequence l_1' such that $l_2 = l_1 l_1'$. For any non-empty sequence $l = e_1 e_2 e_3 \ldots$, define $head(l)$ to be e_1, the first element of l, and $tail(l)$ to be the sequence $e_2 e_3 \ldots$, the rest of l. For any sequence l define $|l|$, the *length* of l, to be the number of elements that occur in l. If l is infinite, then $|l| = \infty$.

3 Untimed Systems

The discussion of untimed systems is organized as follows. Section 3.1 defines *automata*, without an Input/Output distinction. Section 3.2 introduces *live automata*, without an I/O distinction. Section 3.3 defines *safe I/O automata* by adding an I/O distinction to automata, and introduces the parallel composition operator. Section 3.4 introduces *receptiveness*, defines *live I/O automata*, extends parallel composition to live automata, and shows that the parallel composition of two live I/O automata is a live I/O automaton. Section 3.5 defines two preorder relations, the safe preorder and the live preorder, and shows in what sense the live preorder can express a notion of implementation. Section 3.6 compares our model with existing work.

3.1 Automata

We define automata using the presentation style of [LT87]. Essentially, an automaton is a labeled transition system [Plo81].

Definition 3.1 (Automaton) An *automaton* A consists of four components:

- a set $states(A)$ of states.
- a nonempty set $start(A) \subseteq states(A)$ of start states.
- an action signature $sig(A) = (ext(A), int(A))$ where $ext(A)$ and $int(A)$ are disjoint sets of external and internal actions, respectively. Denote by $acts(A)$ the set $ext(A) \cup int(A)$.
- a transition relation $steps(A) \subseteq states(A) \times acts(A) \times states(A)$. ∎

Thus, an automaton is a state machine with labeled steps. Its action signature describes the interface with the environment. It specifies which actions model events that are visible from the environment and which actions model internal events.

An action a of automaton A is said to be *enabled* in state s if there exists a state s' such that the step (s, a, s') is an element of $steps(A)$.

An *execution fragment* α of an automaton A is a (finite or infinite) sequence of alternating states and actions starting with a state and, if the execution fragment is finite, ending in a state,

$$\alpha = s_0 a_1 s_1 a_2 s_2 \cdots,$$

where each triple (s_i, a_{i+1}, s_{i+1}) is an element $steps(A)$. Denote by $fstate(\alpha)$ the first state of α and, if α is finite, denote by $lstate(\alpha)$ the last state of α. Furthermore, denote by $frag^*(A), frag^\omega(A)$ and $frag(A)$ the sets of finite, infinite and all execution fragments of A, respectively. An *execution* is an execution fragment whose first state is a start state. Denote by $exec^*(A)$, $exec^\omega(A)$ and $exec(A)$ the sets of finite, infinite and all executions of A, respectively. A state s of A is *reachable* if there exists a finite execution of A that ends in s.

A finite execution fragment $\alpha_1 = s_0 a_1 s_1 \cdots a_n s_n$ of A and an execution fragment $\alpha_2 = s_n a_{n+1} s_{n+1} \cdots$ of A can be *concatenated*. In this case the concatenation, written $\alpha_1 \frown \alpha_2$, is the execution fragment $s_0 a_1 s_1 \cdots a_n s_n a_{n+1} s_{n+1} \cdots$. An execution fragment α_1 of A is a *prefix* of an execution fragment α_2 of A, written $\alpha_1 \leq \alpha_2$, if either $\alpha_1 = \alpha_2$, or α_1 is finite and there exists an execution fragment α_1' of A such that $\alpha_2 = \alpha_1 \frown \alpha_1'$.

The *trace* of an execution fragment α of an automaton A, written $trace_A(\alpha)$, or just $trace(\alpha)$ when A is clear from context, is the list obtained by restricting α to the set of external actions of A, i.e., $trace(\alpha) = \alpha \upharpoonright ext(A)$, where $\upharpoonright$ is the standard restriction operator on lists. For a set S of executions of an automaton A, denote by $traces_A(S)$, or just $traces(S)$ when A is clear from context, the set of traces of the executions in S. We say that β is a trace of an automaton A if there exists an execution α of A with $trace(\alpha) = \beta$. Denote by $traces^*(A), traces^\omega(A)$ and $traces(A)$ the sets of finite, infinite and all traces of A, respectively. Note, that a finite trace might be the trace of an infinite execution.

3.2 Live Automata

The automaton A of Definition 3.1 can be thought of as expressing the *safety* properties of a system [AS85], i.e, what always holds, or equivalently what is never supposed to happen. The *liveness* properties of a system [AS85], i.e., what must eventually happen, can be expressed by a subset L of the executions of its safe part A, as proposed in [AL93]. Thus, informally, a *live automaton* is a pair (A, L) where A is an automaton and L is a subset of its executions. The executions of L, which satisfy both the safety and liveness requirements of (A, L), are the only ones that can occur in the described system. However, in order to ensure that the set L of executions does not introduce any more safety than is already given by A, it should not be possible to violate L in a finite number of steps. As a consequence, any finite execution of A must be extendible to an execution in L. In fact, if the safe part A of live automaton (A, L) has a finite execution α that cannot be extended to an execution in L, then α cannot occur in the system described by (A, L), and thus L introduces the additional safety property that α cannot occur. Our restriction on the pair (A, L) implies that the pair $(\mathit{exec}(A), L)$ is *machine-closed* as defined in [AL93].

Definition 3.2 (Live automaton) A *liveness condition* L for an automaton A is a subset of the executions of A such that any finite execution of A has an extension in L, i.e., for each $\alpha \in \mathit{exec}^*(A)$ there exists an $\alpha' \in \mathit{frag}(A)$ such that $\alpha \frown \alpha' \in L$.

A *live automaton* is a pair (A, L), where A is an automaton and L is a liveness condition for A. The executions of L are called the *live executions* of (A, L). ■

Informally, a liveness condition can be used to express (at least) two intuitively different sorts of requirements. First, a liveness condition can be used to specify assumptions about the long-term behavior of a system that are based on its physical structure. For example, it is reasonable to assume that two independent processes running in parallel are both allowed to make progress infinitely often. In a physical system this is ensured by executing the two processes on separate processors or by using a fair scheduler in a multiprogramming environment. The notion of fairness of I/O automata [LT87] exactly captures this particular physical assumption. Second, a liveness condition can be used to specify additional properties that a system is required to satisfy. For example, in a mutual exclusion problem we may require a process to eventually exit the critical region whenever it enters it.

Even though a liveness condition can express many specific intuitive ideas, for the purpose of this paper a liveness condition simply represents the set of executions that a system can exhibit whenever it is "working properly".

3.3 Safe I/O Automata

Our notion of safe I/O automaton is the same as the "unfair" I/O automaton of [LT87], i.e., the automaton obtained by removing the partition of the locally-controlled actions from an I/O automaton of [LT87].

Definition 3.3 (Safe I/O automaton) A *safe I/O automaton* A is an automaton augmented with an *external action signature*, $esig(A) = (in(A), out(A))$, which partitions $ext(A)$ into input and output actions. In each state, each input action must be enabled. A is said to be *input-enabled.*

The internal and output actions of a safe I/O automaton A are referred to as the *locally-controlled* actions of A, written $local(A)$. Thus, $local(A) = int(A) \cup out(A)$. ∎

The interaction between safe I/O automata is specified by the parallel composition operator. We use the synchronization style of [Hoa85, LT87], where automata synchronize on their common actions and evolve independently on the others. We also retain the constraint of [LT87] that each action is under the control of at most one automaton by defining parallel composition only for *compatible* safe I/O automata. Compatibility requires that each action be an output action of at most one safe I/O automaton. Furthermore, to avoid action name clashes, compatibility requires that internal action names be unique. Note that compatible automata are allowed to share input actions.

Definition 3.4 (Parallel composition) Two safe I/O automata A_0 and A_1 are *compatible* if the following conditions hold:

1. $out(A_0) \cap out(A_1) = \emptyset$
2. $int(A_0) \cap acts(A_1) = int(A_1) \cap acts(A_0) = \emptyset$.

The *parallel composition* $A_0 \parallel A_1$ of two compatible safe I/O automata A_0 and A_1 is the safe I/O automaton A such that

1. $states(A) = states(A_0) \times states(A_1)$
2. $start(A) = start(A_0) \times start(A_1)$
3. $out(A) = out(A_0) \cup out(A_1)$
4. $in(A) = (in(A_0) \cup in(A_1)) - out(A)$
5. $int(A) = int(A_0) \cup int(A_1)$
6. $((s_0, s_1), a, (s'_0, s'_1)) \in steps(A)$ iff for all $i \in \{0, 1\}$
   (a) if $a \in acts(A_i)$ then $(s_i, a, s'_i) \in steps(A_i)$
   (b) if $a \notin acts(A_i)$ then $s_i = s'_i$. ∎

The executions of the parallel composition of compatible safe I/O automata A_0 and A_1 can be characterized alternatively as those alternating sequences of states and actions of A that, when projected onto any component A_i, yield an execution of A_i. In particular, let $A = A_0 \parallel A_1$. First let s be a state of A. Then, for any $i \in \{0, 1\}$, define $s\lceil A_i$ to be the projection of s onto the i^{th} component. Now, let $\alpha = s_0 a_1 s_1 a_2 s_2 \cdots$ be an alternating sequence of states and actions such that $s_k \in states(A)$ and $a_k \in acts(A)$, for all k, and α ends in a state if it is a finite sequence. Define $\alpha\lceil A_i$ to be the sequence obtained from α by projecting the states onto their i^{th} component and by removing each action not in $acts(A_i)$ together with its following state.

Lemma 3.5 *Let $A = A_0 \parallel A_1$. Let $\alpha = s_0 a_1 s_1 a_2 s_2 \cdots$ be an alternating sequence of states and actions such that $s_k \in states(A)$ and $a_k \in acts(A)$, for all k, and α ends in a state if it is a finite sequence. Then $\alpha \in exec(A)$ iff, for each $i \in \{0, 1\}$, $\alpha\lceil A_i \in exec(A_i)$ and for each $j > 0$, if $a_j \notin acts(A_i)$, then $s_{j-1}\lceil A_i = s_j\lceil A_i$.*

Proof. Direct consequence of Corollary 8 of [LT87]. ■

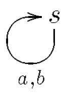
Parallel composition is typically used to build complex systems based on simpler components. Two other operators are defined and used for I/O automata: hiding, which transforms some output actions into internal actions, and renaming, which changes the name to some actions. Hiding and renaming can be handled trivially by extending slightly the theory developed in this paper, and thus we omit their definition.

3.4 Live I/O Automata

In defining live I/O automata one could follow the approach of Definition 3.2 and define a live I/O automaton to be a pair (A, L) where A is a safe I/O automaton and L is a liveness condition for A. However, such a naive definition would not capture the fact that a live I/O automaton should behave properly independently of the inputs provided by its environment. Given the structure of our liveness conditions, such independence from the environment will prove to play a fundamental role in the proofs for the closure of live I/O automata under parallel composition and the substitutivity of our trace based preorders.

Example 3.1 Let A be a the safe I/O automaton described by the diagram,

$A:$



where a is an input action and b is an output action. Let L be the set of executions of A containing at least five occurrences of action a. L is trivially a liveness condition for A; however, the pair (A, L) would not behave properly if the environment does not provide more than four a actions (recall that behaving properly means being an execution of L). ■

Some of the problems arising from the requirement that a live I/O automaton should behave properly independently of the inputs provided by its environment are addressed in [Dil88, AL93]. Their solutions lead to the notion of *receptiveness*. Intuitively a system is receptive if it behaves properly independently of the inputs provided by its environment, or equivalently, if it does not constrain its environment. The interaction between a system and its environment is represented as a two-person game where each environment move consists of providing an arbitrary finite number of inputs, i.e., in our model, a finite number of input actions, and the system moves consist of performing at most one local step, i.e., in our model, at most one locally-controlled step. A system is *receptive* if it has a way to win the game (i.e., to behave properly) independently of the moves of its environment. The fact that an environment move can include at most a finite number of actions represents the natural requirement that the environment cannot be infinitely faster than the system.

The behavior of the system during the game is determined by a *strategy*. In our model a strategy consists of a pair of functions (g, f). The function g decides which of the possible states the system reaches in response to any given input action; the function f determines the next move of the system. The move can be a local step or no step ($\perp$ move).

Definition 3.6 (Strategy) Consider any safe I/O automaton A. A *strategy* defined on A is a pair of functions (g, f) where $g : exec^*(A) \times in(A) \to states(A)$ and $f : exec^*(A) \to (local(A) \times states(A)) \cup \{\perp\}$ such that

1. $g(\alpha, a) = s$ implies $\alpha a s \in exec^*(A)$
2. $f(\alpha) = (a, s)$ implies $\alpha a s \in exec^*(A)$. ∎

In the game between the environment and the system the moves of the environment are represented as an infinite sequence $\mathcal{I}$, called an *environment sequence*, of input actions interleaved with infinitely many λ symbols. The symbol λ represents the points at which the system is allowed to move. The occurrence of infinitely many λ symbols in an environment sequence guarantees that each environment move consists of only finitely many input actions.

Suppose the game starts after a finite execution α. Then the *outcome* of a strategy (g, f), given α and an environment sequence $\mathcal{I}$, is the extension of α obtained by applying g at each input action in $\mathcal{I}$ and f at each λ in $\mathcal{I}$.

Definition 3.7 (Outcome of a strategy) Let A be a safe I/O automaton and (g, f) a strategy defined on A. Define an *environment sequence* for A to be any infinite sequence of symbols from $in(A) \cup \{\lambda\}$ with infinitely many occurrences of λ. Then define $R_{(g,f)}$, the *next-function induced by* (g, f) as follows: for any finite execution α of A and any environment sequence $\mathcal{I}$ for A,

$$R_{(g,f)}(\alpha, \mathcal{I}) = \begin{cases} (\alpha a s, \mathcal{I}') & \text{if } \mathcal{I} = \lambda\mathcal{I}',\ f(\alpha) = (a, s) \\ (\alpha, \mathcal{I}') & \text{if } \mathcal{I} = \lambda\mathcal{I}',\ f(\alpha) = \perp \\ (\alpha a s, \mathcal{I}') & \text{if } \mathcal{I} = a\mathcal{I}',\ g(\alpha, a) = s. \end{cases}$$

Let α be any finite execution of A and $\mathcal{I}$ any environment sequence for A. The *outcome sequence of* (g, f) *given* α *and* $\mathcal{I}$ is the unique infinite sequence $(\alpha^n, \mathcal{I}^n)_{n \geq 0}$ that satisfies:

- $(\alpha^0, \mathcal{I}^0) = (\alpha, \mathcal{I})$ and
- for all $n > 0$, $(\alpha^n, \mathcal{I}^n) = R_{(g,f)}(\alpha^{n-1}, \mathcal{I}^{n-1})$.

Note, that $(\alpha^n)_{n\geq 0}$ forms a chain ordered by *prefix*.

The *outcome* $\mathcal{O}_{(g,f)}(\alpha, \mathcal{I})$ of the strategy (g, f) given α and $\mathcal{I}$ is the execution $\lim_{n\to\infty} \alpha^n$, where $(\alpha^n, \mathcal{I}^n)_{n\geq 0}$ is the outcome sequence of (g, f) given α and $\mathcal{I}$ and the limit is taken under prefix ordering. ∎

Lemma 3.8 *Let A be a safe I/O automaton and (g, f) a strategy defined on A. Then for any finite execution α of A and any environment sequence $\mathcal{I}$ for A, the outcome $\mathcal{O}_{(g,f)}(\alpha, \mathcal{I})$ is an execution of A such that $\alpha \leq \mathcal{O}_{(g,f)}(\alpha, \mathcal{I})$.*

Proof. Simple analysis of the definitions. ∎

The concepts of strategies and outcomes are used to define formally the receptiveness property, i.e., the property that a system does not constrain its environment. Informally, receptiveness requires the existence of a strategy, called a *receptive strategy*, that allows the system to win every game against its environment. In other words, every outcome of the receptive strategy should be an element of L. An important feature of the definition of receptiveness is that it considers outcomes where the receptive strategy for (A, L) is applied after any finite execution of A. Example 3.2 shows that this feature leads to a clean separation of safety and liveness properties.

Definition 3.9 (Receptiveness) Let A be a safe I/O automaton and $L \subseteq exec(A)$. A strategy (g, f) defined on A is called a *receptive strategy* for (A, L) if for any finite execution α of A and any environment sequence $\mathcal{I}$ for A, the outcome $\mathcal{O}_{(g,f)}(\alpha, \mathcal{I})$ is an element of L. The pair (A, L) is *receptive* if there exists a receptive strategy for (A, L). ∎

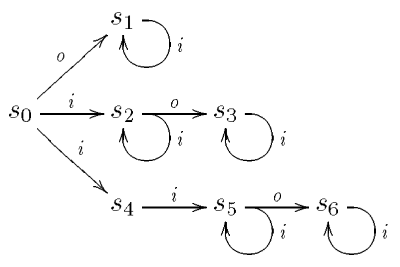
Lemma 3.10 *Consider the pair (A, L), where A is a safe I/O automaton and $L \subseteq exec(A)$. If (A, L) is receptive, then L is a liveness condition for A.*

Proof. Consider any receptive strategy (g, f) for (A, L), any finite execution α of A, and any environment sequence $\mathcal{I}$ for A. Then, since (g, f) is a receptive strategy for (A, L), the outcome $\mathcal{O}_{(g,f)}(\alpha, \mathcal{I})$ is an element of L. Furthermore, by Lemma 3.8, $\mathcal{O}_{(g,f)}(\alpha, \mathcal{I})$ is an extension of α. Hence, any finite execution of A has an extension in L. ∎

Definition 3.11 (Live I/O automaton) A *live I/O automaton* is a pair (A, L), where A is a safe I/O automaton and $L \subseteq exec(A)$, such that (A, L) is receptive. ∎

Example 3.2 Consider the safe I/O automaton A described by the transition diagram below.

The unique start state of A is s_0. Action i is an input action and action o is an output action. Let L be the liveness condition for A consisting of the set of executions of A with at least



one occurrence of action o. The pair (A, L) is not receptive. Specifically, consider the finite execution $\alpha = s_0 i s_4$ and the environment sequence $\mathcal{I} = \lambda\lambda\lambda\cdots$. Performing action o after reaching state s_4 requires receiving an input i. Therefore, there is no strategy whose outcome given α and $\mathcal{I}$ is an execution in L.

Define a new automaton A' from A by removing states s_4, s_5, s_6, and let L' be the set of executions of A' containing at least one occurrence of action o. Then the pair (A', L') is receptive. Function f chooses to perform action o whenever applied to an execution ending in s_0 or s_2 and chooses $\perp$ otherwise; function g always moves to the only possible next state. ∎

Remark 3.3 The definition of a receptive pair shows why we include the input-enabled property in our definition of a safe I/O automaton. Namely, consider any reachable state s of A and any finite execution α of A leading to state s. Since a receptive strategy must allow α to be extended in response to any possible input action, each input action must be enabled in s. Thus, a receptive strategy can only exist for a pair (A, L) for which all inputs are enabled in all reachable states. ∎

The parallel composition operator can now be extended to live I/O automata by using the result of Lemma 3.5.

Definition 3.12 (Parallel composition) Two live I/O automata (A_0, L_0) and (A_1, L_1) are *compatible* iff the safe I/O automata A_0 and A_1 are compatible. The *parallel composition* $(A_0, L_0) \parallel (A_1, L_1)$ of two compatible live I/O automata (A_0, L_0) and (A_1, L_1) is defined to be the pair (A, L) where $A = A_0 \parallel A_1$ and $L = \{\alpha \in \mathit{exec}(A) \mid \alpha\lceil A_0 \in L_0 \text{ and } \alpha\lceil A_1 \in L_1\}$. ∎

The parallel composition operator is closed for live I/O automata in the sense that it produces a new live I/O automaton whenever applied to live I/O automata. The proof of this result, however, is not trivial and needs some preliminary lemmas. Given $(A, L) = (A_0, L_0) \parallel (A_1, L_1)$, it is easy to see that A is a safe I/O automaton since its definition is based on the parallel composition of safe I/O automata. However, it is not as easy to see that the pair (A, L) is receptive, and hence a live I/O automaton. The proof that (A, L) is receptive uses a strategy (g, f) for (A, L) based on receptive strategies (g_0, f_0) and (g_1, f_1) for (A_0, L_0) and (A_1, L_1), respectively, and shows that (g, f) is a receptive strategy for (A, L).

Function g should compute, given input a, the next state according to the g_i functions of those components of A for which a is an input action, and simply leave the state unchanged for those components where a is not an action.

Function f must ensure that every component of A gets a chance to control a step of A infinitely often. This fact accounts for much of the complexity in the definition of (g, f). Ensuring that each component of A gets a chance to control a step infinitely often would most naturally be done by assigning the control of steps to the two components in an alternating way. The alternating approach, however, would give rise to a technical problem in the definition of f: since the only argument to f is a finite execution α, the component whose turn it is to control the step in the alternating schedule must be determined from α. Unfortunately, the finite execution α does not include enough information to make this determination. Consider the following scenario. Assume that it is component A_i's turn to control the step after a finite execution α. Assume further that A_i decides to perform a $\perp$ move and that the next input is a λ symbol. In this case α will not change and, thus, it will again be A_i's turn to control the next step. Therefore, the alternating protocol is violated. The problem is, of course, that $\perp$ and λ moves are "invisible" in α. One solution to this problem would be to let f be a function of "extended" executions that contain information about $\perp$ and λ moves. The problem with this solution, however, is that it becomes messy due to the fact that this new notion of execution must keep track of $\perp$ and λ moves of subcomponents of components, and so on. An alternative solution, adopted in our definition of f, uses the parity of the number of locally-controlled actions in α to determine which component has priority for a step. If the component having priority for a step wants to perform a $\perp$ move but the other component wants to perform a local step, then the other component gets to perform a step even though it does not have priority. Only if both components want to perform $\perp$ moves, does f yield a $\perp$ move.

One final technicality in the definition of f is that it uses the g_i functions. In particular, if a component performs a local step with action a, action a might be an input action of the other component. In this case, the definition of f will need the g_i function of the other component.

Definition 3.13 (Parallel composition of strategies) Let $A = A_0 \parallel A_1$ be the parallel composition of two compatible safe I/O automata A_0 and A_1. For each finite execution $\alpha \in exec^*(A)$, let $l(\alpha)$ be the number of occurrences of locally-controlled actions of A in α, i.e., $l(\alpha) = |\alpha \restriction local(A)|$, and let $p(\alpha) = l(\alpha) \bmod 2$. Let (g_0, f_0) and (g_1, f_1) be strategies defined on A_0 and A_1, respectively. The *parallel composition* $(g_0, f_0) \parallel (g_1, f_1)$ of the strategies (g_0, f_0) and (g_1, f_1) is the pair of functions (g, f) defined as follows.

Function $g : exec^*(A) \times in(A) \to states(A)$ is defined as $g(\alpha, a) = s$ where, for each $i \in \{0, 1\}$,

$$s\lceil A_i = \begin{cases} g_i(\alpha\lceil A_i, a) & \text{if } a \in in(A_i) \\ lstate(\alpha)\lceil A_i & \text{otherwise.} \end{cases}$$

Function $f : exec^*(A) \to (local(A) \times states(A)) \cup \{\perp\}$ is defined as follows: if $f_0(\alpha\lceil A_0) = \perp$ and $f_1(\alpha\lceil A_1) = \perp$, then $f(\alpha) = \perp$. Otherwise, let k be $p(\alpha)$ if $f_{p(\alpha)}(\alpha\lceil A_{p(\alpha)}) \neq \perp$, and let k

be $1-p(\alpha)$ if $f_{p(\alpha)}(\alpha\lceil A_{p(\alpha)}) = \perp$. Let (a, s_k) denote $f_k(\alpha\lceil A_k)$, and define $f(\alpha) = (a, s)$ where, for each $i \in \{0, 1\}$,

$$s\lceil A_i = \begin{cases} s_k & \text{if } i = k \\ g_i(\alpha\lceil A_i, a) & \text{if } a \in in(A_i) \\ lstate(\alpha)\lceil A_i & \text{otherwise.} \end{cases}$$

■

Lemma 3.14 *Let A_0 and A_1 be two compatible safe I/O automata and let (g_0, f_0) and (g_1, f_1) be strategies defined on A_0 and A_1, respectively. Then $(g_0, f_0) \parallel (g_1, f_1)$ is a strategy defined on $A_0 \parallel A_1$.*

Proof. Simple cases analysis on the different cases of Definition 3.13. In fact, for each one of those cases, it is sufficient to show that f and g give legal steps of $A_0 \parallel A_1$. ■

The following lemma is the key step for proving that the strategy of Definition 3.13 is receptive if the component strategies are receptive. The lemma shows that the projection of an outcome of the composed strategy onto any A_i is an outcome of the strategy (g_i, f_i). Intuitively, this means that, even though the composed system uses its composed strategy to find its outcome, it still looks to each component as if it was using its own component strategy.

Lemma 3.15 *Let A_0, A_1 be compatible safe I/O automata and let (g_0, f_0) and (g_1, f_1) be strategies defined on A_0 and A_1, respectively. Let $A = A_0 \parallel A_1$ and let $(g, f) = (g_0, f_0) \parallel (g_1, f_1)$. Let α be an arbitrary finite execution of A, $\mathcal{I}$ be an arbitrary environment sequence for A, and i be either 0 or 1. Then, there exists an environment sequence $\mathcal{I}_i$ for A_i such that $\mathcal{O}_{(g,f)}(\alpha, \mathcal{I})\lceil A_i = \mathcal{O}_{(g_i,f_i)}(\alpha\lceil A_i, \mathcal{I}_i)$.*

Proof. Let $R_{(g,f)}$ and $R_{(g_i,f_i)}$ be the next-functions induced by (g, f) and (g_i, f_i), respectively. Let $(\alpha^n, \mathcal{I}^n)_{n\geq 0}$ be the outcome sequence of (g, f) given α and $\mathcal{I}$. Then $\mathcal{O}_{(g,f)}(\alpha, \mathcal{I}) = \lim_{n\to\infty} \alpha^n$. For any finite execution $\alpha' \in exec^*(A)$, let $l(\alpha')$ be the number of occurrences of locally-controlled actions of A in α', i.e., $l(\alpha') = |\alpha' \restriction local(A)|$, and let $p(\alpha') = (l(\alpha') \bmod 2)$. (Cf. Definition 3.13.)

The first step of the proof consists of constructing an environment sequence $\mathcal{I}_i$ for A_i such that $\mathcal{O}_{(g,f)}(\alpha, \mathcal{I})\lceil A_i = \mathcal{O}_{(g_i,f_i)}(\alpha\lceil A_i, \mathcal{I}_i)$. The sequence $\mathcal{I}_i$ is defined as $\mathcal{I}_i^1\mathcal{I}_i^2\cdots$, where each $\mathcal{I}_i^j$ consists of 0, 1, or 2 symbols and is defined below. Along with the definition of $\mathcal{I}_i^j$ we prove the following property:

P1 For every environment sequence $\mathcal{I}'$ for A_i, $(\alpha^n\lceil A_i, \mathcal{I}') = R^*_{(g_i,f_i)}(\alpha^{n-1}\lceil A_i, \mathcal{I}_i^n, \mathcal{I}')$,

where, for any finite execution fragment α of A_i, any finite sequence $\mathcal{I}$ of elements from $in(A_i) \cup \{\lambda\}$, and any environment sequence $\mathcal{J}$ for A_i, $R^*_{(g_i,f_i)}(\alpha, \mathcal{I}, \mathcal{J})$ is defined as follows: $R^*_{(g_i,f_i)}(\alpha, \epsilon, \mathcal{J}) = (\alpha, \mathcal{J})$, and if $|\mathcal{I}| \geq 1$, then $R^*_{(g_i,f_i)}(\alpha, \mathcal{I}, \mathcal{J}) = R^*_{(g_i,f_i)}(\alpha', \mathcal{I}', \mathcal{J})$ where $(\alpha', \mathcal{I}'\mathcal{J}) = R_{(g_i,f_i)}(\alpha, \mathcal{I}\mathcal{J})$. Informally, $R^*_{(g_i,f_i)}(\alpha, \mathcal{I}, \mathcal{J})$ is the result of applying $R_{(g_i,f_i)}$ from $(\alpha, \mathcal{I}\mathcal{J})$ for a number of times equal to the length of $\mathcal{I}$. In the rest of the proof we let $\mathcal{I}'$ denote a generic environment sequence for A_i. Let $n > 0$. The definition of $R_{(g,f)}$ suggests three cases which are considered in order.

Case 1 $(\alpha^n, \mathcal{I}^n) = (\alpha^{n-1}as, tail(\mathcal{I}^{n-1}))$ where $f(\alpha^{n-1}) = (a, s)$ and $head(\mathcal{I}^{n-1}) = \lambda$.

The definition of f in Definition 3.13 suggests the following subcases:

Case 1.1 $p(\alpha^{n-1}) = i$ and $a \notin acts(A_i)$.

Define $\mathcal{I}_i^n = \lambda$. Since $p(\alpha^{n-1}) = i$ and $a \notin acts(A_i)$, the definition of f shows that $f_i(\alpha^{n-1}\lceil A_i) = \bot$. Furthermore, since $a \notin acts(A_i)$, $\alpha^n\lceil A_i = \alpha^{n-1}\lceil A_i$. By case 2 of the definition of $R_{(g_i,f_i)}$, $(\alpha^n\lceil A_i, \mathcal{I}') = R_{(g_i,f_i)}(\alpha^{n-1}\lceil A_i, \mathcal{I}_i^n\mathcal{I}')$. Thus, $(\alpha^n\lceil A_i, \mathcal{I}') = R^*_{(g_i,f_i)}(\alpha^{n-1}\lceil A_i, \mathcal{I}_i^n, \mathcal{I}')$.

Case 1.2 $p(\alpha^{n-1}) = i$ and $a \in in(A_i)$.

Define $\mathcal{I}_i^n = \lambda a$. Since $p(\alpha^{n-1}) = i$ and $a \in in(A_i)$, the definition of f shows that $f_i(\alpha^{n-1}\lceil A_i) = \bot$. By case 2 of the definition of $R_{(g_i,f_i)}$, $(\alpha^{n-1}\lceil A_i, a\mathcal{I}') = R_{(g_i,f_i)}(\alpha^{n-1}\lceil A_i, \lambda a\mathcal{I}')$. Since $a \in in(A_i)$, the definition of f shows that $g_i(\alpha^{n-1}\lceil A_i, a) = s\lceil A_i$. By case 3 of the definition of $R_{(g_i,f_i)}$, $(\alpha^n\lceil A_i, \mathcal{I}') = R_{(g_i,f_i)}(\alpha^{n-1}\lceil A_i, a\mathcal{I}')$. Thus, $(\alpha^n\lceil A_i, \mathcal{I}') = R^*_{(g_i,f_i)}(\alpha^{n-1}\lceil A_i, \mathcal{I}_i^n, \mathcal{I}')$.

Case 1.3 $a \in local(A_i)$.

Define $\mathcal{I}_i^n = \lambda$. Since $a \in local(A_i)$, the definition of f shows that $f_i(\alpha^{n-1}\lceil A_i) = (a, s\lceil A_i)$. By case 1 of the definition of $R_{(g_i,f_i)}$, $(\alpha^n\lceil A_i, \mathcal{I}') = R_{(g_i,f_i)}(\alpha^{n-1}\lceil A_i, \mathcal{I}_i^n\mathcal{I}')$. Thus, $(\alpha^n\lceil A_i, \mathcal{I}') = R^*_{(g_i,f_i)}(\alpha^{n-1}\lceil A_i, \mathcal{I}_i^n, \mathcal{I}')$.

Case 1.4 $p(\alpha^{n-1}) \neq i$ and $a \in in(A_i)$.

Define $\mathcal{I}_i^n = a$. Since $a \in in(A_i)$ the definition of f shows that $g_i(\alpha^{n-1}\lceil A_i, a) = s\lceil A_i$. By case 3 of the definition of $R_{(g_i,f_i)}$, $(\alpha^n\lceil A_i, \mathcal{I}') = R_{(g_i,f_i)}(\alpha^{n-1}\lceil A_i, \mathcal{I}_i^n\mathcal{I}')$. Thus, $(\alpha^n\lceil A_i, \mathcal{I}') = R^*_{(g_i,f_i)}(\alpha^{n-1}\lceil A_i, \mathcal{I}_i^n, \mathcal{I}')$.

Case 1.5 $p(\alpha^{n-1}) \neq i$ and $a \notin acts(A_i)$.

Define $\mathcal{I}_i^n = \varepsilon$. Observe that $\alpha^n\lceil A_i = \alpha^{n-1}\lceil A_i$. Thus, trivially $(\alpha^n\lceil A_i, \mathcal{I}') = R^*_{(g_i,f_i)}(\alpha^{n-1}\lceil A_i, \mathcal{I}_i^n\mathcal{I}')$.

Case 2 $(\alpha^n, \mathcal{I}^n) = (\alpha^{n-1}, tail(\mathcal{I}^{n-1}))$ where $f(\alpha^{n-1}) = \bot$ and $head(\mathcal{I}^{n-1}) = \lambda$.

Define $\mathcal{I}_i^n = \lambda$. Since $f(\alpha^{n-1}) = \bot$, the definition of f shows that $f_i(\alpha^{n-1}\lceil A_i) = \bot$. By case 2 of the definition of $R_{(g_i,f_i)}$, $(\alpha^n\lceil A_i, \mathcal{I}') = R_{(g_i,f_i)}(\alpha^{n-1}\lceil A_i, \mathcal{I}_i^n\mathcal{I}')$. Thus, $(\alpha^n\lceil A_i, \mathcal{I}') = R^*_{(g_i,f_i)}(\alpha^{n-1}\lceil A_i, \mathcal{I}_i^n, \mathcal{I}')$.

Case 3 $(\alpha^n, \mathcal{I}^n) = (\alpha^{n-1}as, tail(\mathcal{I}^{n-1}))$ where $g(\alpha^{n-1}, a) = s$ and $head(\mathcal{I}^{n-1}) = a$.

The definition of g in Definition 3.13 suggests the following subcases:

Case 3.1 $a \in in(A_i)$.

Define $\mathcal{I}_i^n = a$. The definition of g shows that $g_i(\alpha^{n-1}\lceil A_i, a) = s\lceil A_i$. By case 3 of the definition of $R_{(g_i,f_i)}$, $(\alpha^n\lceil A_i, \mathcal{I}') = R_{(g_i,f_i)}(\alpha^{n-1}\lceil A_i, \mathcal{I}_i^n\mathcal{I}')$. Thus, $(\alpha^n\lceil A_i, \mathcal{I}') = R^*_{(g_i,f_i)}(\alpha^{n-1}\lceil A_i, \mathcal{I}_i^n, \mathcal{I}')$.

Case 3.2 $a \notin in(A_i)$.

Define $\mathcal{I}_i^n = \varepsilon$. Observe that $\alpha^n\lceil A_i = \alpha^{n-1}\lceil A_i$. Thus, trivially $(\alpha^n\lceil A_i, \mathcal{I}') = R^*_{(g_i,f_i)}(\alpha^{n-1}\lceil A_i, \mathcal{I}_i^n\mathcal{I}')$.

The second step of the proof consists of showing that $\mathcal{I}_i$ is indeed an environment sequence for A_i. Showing that $\mathcal{I}_i$ is an environment sequence for A_i induces two proof obligations:

1. Each element of $\mathcal{I}_i$ is in $in(A_i) \cup \{\lambda\}$.

   This follows immediately from the definition of the $\mathcal{I}_i^j$'s.

2. There are infinitely many λ's in $\mathcal{I}$.

   For each $n > 0$, all the cases of the definition above except for 1.4, 1.5, 3.1, and 3.2 add a new λ to $\mathcal{I}$. Thus, the proof obligation is met as long as there exists no $n_o \geq 0$ such that for all $n > n_o$ the sequence $\mathcal{I}_i^n$ is defined according to cases 1.4, 1.5, 3.1, or 3.2. For a contradiction assume such an n_o exists. Observe the following: if $\mathcal{I}_i^n$ is defined according to cases 3.1 or 3.2, then $l(\alpha^n) = l(\alpha^{n-1})$; if $\mathcal{I}_i^n$ is defined according to cases 1.4 or 1.5, then $l(\alpha^n) = l(\alpha^{n-1})+1$. Furthermore, cases 1.4 and 1.5 require that $p(\alpha^{n-1}) \neq i$. Thus, there can be at most one $n_1 > n_0$ such that $\mathcal{I}_i^n$ is defined according to cases 1.4 or 1.5. In other words, there exists a number $n_1 > n_0$ such that for each $n > n_1$ $\mathcal{I}_i^n$ is defined according to cases 3.1 or 3.2. However, since $\mathcal{I}$ is an environment sequence, for infinitely many n such that $n > n_1$, $head(\mathcal{I}^{n-1}) = \lambda$. This is a contradiction since the $\mathcal{I}_i^n$ cannot be defined according to cases 3.1 or 3.2 when $head(\mathcal{I}^{n-1}) = \lambda$.

From the construction above and from **P1**, $\mathcal{O}_{(g_i,f_i)}(\alpha\lceil A_i, \mathcal{I}_i) = \lim_{n\to\infty} \alpha_n\lceil A_i$. From the continuity of the projection operator, $\lim_{n\to\infty} \alpha_n\lceil A_i = (\lim_{n\to\infty} \alpha_n)\lceil A_i$. Thus, $\mathcal{O}_{(g,f)}(\alpha, \mathcal{I})\lceil A_i = \mathcal{O}_{(g_i,f_i)}(\alpha\lceil A_i, \mathcal{I}_i)$. ■

Lemma 3.16 *Let (A_0, L_0) and (A_1, L_1) be two compatible live I/O automata and let (g_0, f_0) and (g_1, f_1) be receptive strategies for (A_0, L_0) and (A_1, L_1), respectively. Then $(g_0, f_0) \parallel (g_1, f_1)$ is a receptive strategy for $(A_0, L_0) \parallel (A_1, L_1)$.*

Proof. Let $(A, L) = (A_0, L_0) \parallel (A_1, L_1)$ and $(g, f) = (g_0, f_0) \parallel (g_1, f_1)$. Consider any environment sequence $\mathcal{I}$ for A and any finite execution α of A. By Lemma 3.15 there exists for all A_i an environment sequence $\mathcal{I}_i$ such that $\mathcal{O}_{(g,f)}(\alpha, \mathcal{I})\lceil A_i = \mathcal{O}_{(g_i,f_i)}(\alpha\lceil A_i, \mathcal{I}_i)$. Since (g_i, f_i) is a receptive strategy for (A_i, L_i), $\mathcal{O}_{(g_i,f_i)}(\alpha\lceil A_i, \mathcal{I}_i) \in L_i$. Consequently, $\mathcal{O}_{(g,f)}(\alpha, \mathcal{I})\lceil A_i \in L_i$ for all (A_i, L_i). By Definition 3.12, $\mathcal{O}_{(g,f)}(\alpha, \mathcal{I}) \in L$. ■

Theorem 3.17 (Closure of parallel composition) *Let (A_0, L_0) and (A_1, L_1) be compatible live I/O automata. Then $(A_0, L_0) \parallel (A_1, L_1)$ is a live I/O automaton.*

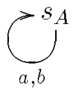
Proof. Let $(A, L) = (A_0, L_0) \parallel (A_1, L_1)$. By Definition 3.4, we know that A is a safe I/O automaton. Furthermore, by Definition 3.12, Lemma 3.5, and the fact that each $L_i \subseteq exec(A_i)$, the set L is a subset of $exec(A)$.

Let (g_0, f_0) and (g_1, f_1) be receptive strategies for (A_0, L_0) and (A_1, L_1), respectively. By Lemma 3.16 the strategy $(g, f) = (g_0, f_0) \parallel (g_1, f_1)$ is a receptive strategy for (A, L). Therefore, the pair (A, L) is receptive. Thus, by Definition 3.11, (A, L) is a live I/O automaton. ∎

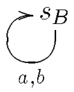
Receptiveness is a crucial property of live I/O automata since it guarantees that no pair of compatible live I/O automata constrain each other's environments. In particular, if a pair (A, L) is not receptive, the parallel composition operator may generate pairs that are not even live automata.

Example 3.4 Consider safe I/O automata A and B described by the diagrams below.

A :



B :



For A, action b is an input action, and action a is an output action; for B, action a is an input action and action b is an output action. Let the liveness condition L_A for A be the set of executions α of A such that $trace(\alpha)$ ends in $(ab)^\infty$ or a^∞, and let the liveness condition L_B for B be the set of executions α of B such that $trace(\alpha)$ ends in $(aabb)^\infty$ or b^∞.

The pairs (A, L_A) and (B, L_B) are not receptive. To see that (A, L_A) is not receptive consider the environment sequence $\mathcal{I} = bb\lambda bb\lambda \cdots$; to see that (B, L_B) is not receptive consider the environment sequence $\mathcal{I} = aaa\lambda aaa\lambda \cdots$.

Let $(C, L_C) = (A, L_A) \parallel (B, L_B)$. In this case, $L_C = \emptyset$. Thus L_C is not a liveness condition for C, which means that (C, L_C) is not even a live automaton. ∎

Example 3.4 also exposes the flaw in a simpler and more intuitive definition for receptiveness we originally considered for this paper. The simpler definition, which is a natural generalization of the fairness condition of [LT87] and is also discussed in [LS89], states that "a pair (A, L) is receptive if for each finite execution α of A and each (finite or infinite) sequence β of input actions there is an execution fragment α' of A such that $\alpha' \lceil in(A) = \beta$ and $\alpha \frown \alpha' \in L$." It is easy to see that the pairs (A, L_A) and (B, L_B) of Example 3.4 are both receptive based on the simpler definition. However, the example shows that their composition cannot be a live I/O automaton. The problem with the simpler definition is that it allows the system to choose its relative speed with respect to the environment, and it allows the system to base its decisions on the future behavior of the environment. Example 3.4 shows that the simpler definition thus gives the system too much power for parallel composition to be closed.

3.5 Preorder Relations for Live I/O Automata

In [LT87, Dil88, AL93] the notion of implementation is expressed through some form of trace inclusion. Similar notions of implementation can be defined on live I/O automata. In particular it is possible to identify two preorder relations, the safe and the live preorders, which aim at capturing the safety and liveness aspects of live I/O automata, respectively.

Definition 3.18 (Trace preorders) Given two live I/O automata (A_1, L_1) and (A_2, L_2) such that $esig(A_1) = esig(A_2)$, define the following preorders:

Safe: $(A_1, L_1) \sqsubseteq_{\mathrm{S}} (A_2, L_2)$ iff $traces(A_1) \subseteq traces(A_2)$;
Live: $(A_1, L_1) \sqsubseteq_{\mathrm{L}} (A_2, L_2)$ iff $traces(L_1) \subseteq traces(L_2)$. ■

The safe preorder is the same as the unfair preorder of I/O automata [LT87], while the live preorder is a generalization of the fair preorder of [LT87]. In particular, the live preorder coincides with the fair preorder if, for each live I/O automaton (A, L), L is chosen to be the set of fair executions of A. The conformation preorder of [Dil88], which expresses the notion of implementation for complete trace structures, coincides with the live preorder when dealing with failure free complete trace structures. Finally, the notion of implementation of [AL93], which works in a state based model, coincides with the live preorder up to a different notion of traces arising from the state structure of the model. In [AL93], a system M_1 implements a system M_2 iff the set of "traces" of the *realizable* part of M_1 is a subset of the set of "traces" of the realizable part of M_2. Furthermore, if a system M is receptive, then M is equal to its realizable part. Thus, for receptive systems, the implementation notion of [AL93] is just the live trace preorder. The reader is referred to Section 3.6 for more discussion of realizability.

Note that the live preorder implies the safe preorder whenever the involved automata have finite internal nondeterminism. On the other hand, if the involved automata do not have finite internal nondeterminism, the live preorder only implies finite trace inclusion. Essentially, finite internal nondeterminism requires that a live I/O automaton has a finite internal branching structure. In particular, a finite trace can lead to at most finitely many states.

Definition 3.19 (Finite internal nondeterminism) An automaton A has *finite internal nondeterminism* (FIN) iff, for each finite trace $\beta \in traces^*(A)$, the set $\{lstate(\alpha) \mid \alpha \in exec^*(A), trace(\alpha) = \beta\}$ is finite. ■

Proposition 3.20 *Let (A_1, L_1) and (A_2, L_2) be two live I/O automata with $esig(A_1) = esig(A_2)$.*

1. *If $(A_1, L_1) \sqsubseteq_{\mathrm{L}} (A_2, L_2)$ then $traces^*(A_1) \subseteq traces^*(A_2)$*
2. *If A_2 has FIN and $(A_1, L_1) \sqsubseteq_{\mathrm{L}} (A_2, L_2)$, then $(A_1, L_1) \sqsubseteq_{\mathrm{S}} (A_2, L_2)$*

Proof. Let β be a finite trace of A_1. By definition of trace, there is an execution α_1 of A_1 such that $trace(\alpha_1) = \beta$. By definition of a live I/O automaton there exists an execution α_1' of A_1 such that $\alpha_1 \leq \alpha_1'$ and $\alpha_1' \in L_1$. Since $(A_1, L_1) \sqsubseteq_{\mathrm{L}} (A_2, L_2)$, there exists an execution α_2' of L_2 such that $trace(\alpha_1') = trace(\alpha_2')$. By definition of a live I/O automaton, α_2' is an execution of A_2, and, since the set of executions of an automaton is closed under prefix, there is a prefix α_2 of α_2' such that α_2 is an execution of A_2 and $trace(\alpha_2) = \beta$, i.e., β is a trace of A_2. This shows part 1. For part 2 we need to show infinite trace inclusion as well, which follows from finite trace inclusion, closure under prefix of trace sets, and the fact that trace sets of automata with finite internal nondeterminism are closed under prefix ordering limit [LV91]. ■

The proof of Proposition 3.20 supports the requirement of our definition of a liveness condition (Definition 3.2) that every safe execution be extendible to a live execution. Without this requirement, the live preorder could not be used to infer the safe preorder, i.e., neither part of Proposition 3.20 would hold.

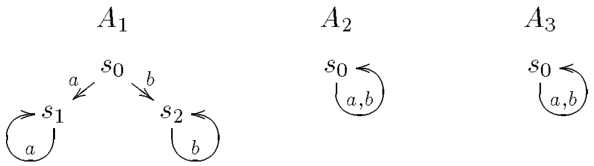
An important goal of this paper is the substitutivity of the safe and live preorders for parallel composition. This means that an implementation of a system made up of several parallel components can be obtained by implementing each component separately.

Theorem 3.21 (Substitutivity) *Let $(A_i, L_i), (A'_i, L'_i)$, $i \in \{0,1\}$ be live I/O automata, and let $\sqsubseteq_X$ be either $\sqsubseteq_{\mathrm{S}}$ or $\sqsubseteq_{\mathrm{L}}$. If, for each i, $(A_i, L_i) \sqsubseteq_X (A'_i, L'_i)$, (A_0, L_0) and (A_1, L_1) are compatible, and (A'_0, L'_0) and (A'_1, L'_1) are compatible, then $(A_0, L_0)\|(A_1, L_1) \sqsubseteq_X (A'_0, L'_0)\|(A'_1, L'_1)$.*

Proof. The substitutivity results for the safe trace preorder are already proven in [LT87]. The substitutivity results for the live trace preorder follow directly from the definition of the parallel composition operator after observing, as it is proved in Corollary 8 of [LT87], that parallel composition of execution sets preserves trace equivalence. ∎

The following example shows that the absence of receptiveness can lead to situations where the substitutivity result of Theorem 3.21 breaks.

Example 3.5 Consider the safe I/O automata A_1, A_2, and A_3 with the transition diagrams below.



where a and b are output actions for A_1 and A_2 and are input actions for A_3. Let L_1 (resp. L_2) be the set of executions of A_1 (resp. A_2) containing at least one action and let L_3 be the set of executions of A_3 whose trace contains the subsequence ab. It is easy to check that (A_1, L_1) and (A_2, L_2) are both receptive, and that (A_3, L_3) is not receptive since it requires at least one input.

Observe that $(A_1, L_1) \sqsubseteq_{\mathrm{L}} (A_2, L_2)$ and that $(A_2, L_2)\|(A_3, L_3)$ is receptive and thus a live I/O automaton. One might want to conclude that $(A_1, L_1)\|(A_3, L_3) \sqsubseteq_{\mathrm{L}} (A_2, L_2)\|(A_3, L_3)$. Unfortunately, this conclusion is false. In particular, let $(A, L) = (A_1, L_1)\|(A_3, L_3)$. Then, the set L is not a liveness condition since A_1 can never perform an action a followed by an action b. Thus, the fact that (A_3, L_3) is not receptive causes situations where the parallel composition with (A_3, L_3) fails to lead to a pair (A, L) where L is a liveness condition. This in turn causes the substitutivity of the parallel composition operator to fail. ∎

There are several ways to justify the live preorder as an adequate notion of implementation for live I/O automata. Since the live preorder captures the implementation notions of [LT87, Dil88, AL93] it can rest on the justifications provided for these implementation notions. For example, the fair preorder of [LT87] is justified by two observations. First, the fact that I/O automata are input-enabled guarantees that a system must respond to any environment. In our model the same property is guaranteed by the concept of receptiveness. Second, by restricting attention to fair traces the correctness of an implementation is based only on executions where the system behaves fairly. In our model this property is guaranteed by restricting attention to live traces.

An additional justification for the live preorder as a notion of implementation is based on the concepts of safety and liveness properties. It is easy to see that the safe preorder preserves the safety properties of a system, i.e., the safe preorder guarantees that an implementation cannot do anything that is not allowed by the specification. The live preorder, on the other hand, preserves the liveness properties of a system, thus guaranteeing that an implementation must do something whenever it is required to by the specification. Informally, if after a sequence of actions β something has to happen, β is not a live trace of the specification, and thus not a live trace of the implementation. Therefore, even in the implementation something has to happen after β has occurred. If the involved systems have finite internal nondeterminism, then the live preorder implies the safe preorder. Thus the live preorder guarantees both safety and liveness properties.

It is well known that simulation based proof techniques [LV93] can be used for implementation notions based on trace inclusion. In [GSSL93] simulation based proof techniques are extended to the live preorder, and in [SLL93b] the new proof techniques are used to verify nontrivial communication protocols.

3.6 Comparison with Other Models

This section compares our model with the models of [Dil88, LT87, AL93] and the work of [RWZ92].

The model of complete trace structures of [Dil88] is a special case of our model. Specifically, the model of [Dil88] does not include a state structure, so that the safe part of a live automaton in [Dil88] is given by a set of traces. Since there is no notion of a state in a complete trace structure, a strategy for a system is simpler than our strategies in the sense that function g is not necessary and that function f simply picks up a locally-controlled action based on previous environment moves. By ignoring the state structure of a system, the model in [Dil88] may erroneously view as receptive a state machine that is not receptive based on our model since its traces may be receptive. Thus, complete trace structures are not adequate whenever the state structure of a system is important.

The I/O automaton model of [LT87] is also a special case of our model. An I/O automaton M of [LT87] can be represented in our model as the receptive pair (A, L), where A is the I/O automaton M without the partition of its locally-controlled actions and L is the set of fair executions of M. The receptive strategy (g, f) for (A, L) is defined so that g picks up any possible next state in response to an input action, while f gives fair turns to proceed (say in a round robin way) to all the components of M that are continuously willing to perform some locally-controlled action. Thus [LT87] can only express some special cases of our general liveness conditions.

The model of [AL93] is based on unlabeled state transition systems and is suitable for the modeling of shared memory systems. An action in [AL93] is identified with a set of transitions, and transitions are partitioned into environment transitions and system transitions. The environment moves by performing an arbitrary finite number of environment transitions and the system responds by performing zero or one system transitions. Function g is not necessary in a strategy for a system of [AL93] since the environment chooses the next shared state in its move and does not modify the internal state. Function f chooses a new transition based on the past history of the system.

In this paper we have defined receptiveness by requiring the existence of a strategy that can "win the game" after any finite execution α. In [AL93] a weaker property called *realizability* is considered, where the requirement is the existence of a strategy that can win starting from any start state. The realizable part of a system of [AL93] is the set of behaviors that can be the outcome of some strategy. A system that coincides with its realizable part is called *receptive*. The notion of receptiveness of [AL93] corresponds to our notion of receptiveness, as can be derived easily from Proposition 9 of [AL93].

Example 3.2 shows a live automaton (A, L) which is not receptive. However, (A, L) is realizable, and (A', L'), which is defined in the same example, is the realizable part of (A, L). In [AL93] systems are compared based on their realizable parts. Thus, it is necessary to determine the realizable part of a system *before* its safety properties can be determined, and for this reason realizable systems are closed under parallel composition in [AL93]. In other words, L can add new safety properties to A. However, later in [AL93] a notion of *machine-realizability* is introduced which separates safety and liveness properties and requires receptiveness just like our live I/O automata.

Finally, it is easy to show, given our definition of receptiveness, that the set of live traces of any live I/O automaton is union-game realizable according to [RWZ92], and thus describable by means of a standard I/O automaton of [LT87]. However, in general the I/O automaton description would involve a lot of encoding and would be extremely unnatural. That is, even though the I/O automata of [LT87] and our live I/O automata are formally equivalent, fairness is not adequate to describe general liveness.

4 Timed Systems

The notion of liveness discussed in the previous section is now extended to the timed model. Section 4.1 introduces *timed automata* along with *timed executions* and *timed traces*, and shows the relationship between the new timed executions and the ordinary executions from the untimed model. Section 4.2 introduces *live timed automata*. Section 4.3 defines *safe timed I/O automata* by introducing the Input/Output distinction. Section 4.4 extends the notion of *receptiveness* to the timed model and defines *live timed I/O automata*. Section 4.5 introduces several preorders on live timed I/O automata, one of which is used to express a notion of implementation. Finally, Section 4.6 compares our model with existing work. Most of the discussion for the untimed model applies to the timed model as well. In particular, Examples 3.1, 3.2, 3.4, and 3.5 apply equally to the timed model. In the rest of the paper our discussion focuses on issues specific to the timed model.

4.1 Timed Automata

The following definition of a timed automaton is the same as the corresponding definition in [LV95] except for the fact that our definition allows multiple internal actions. Also, the notions of timed executions and timed traces are the same as the definitions of [LV95]. The definitions are repeated here but the reader is referred to [LV95] for further details. Times are specified using a *dense* time domain T. In this paper, as in [LV95], let T be $\mathbb{R}^{\geq 0}$, the set of non-negative reals.

Definition 4.1 (Timed automaton) A *timed automaton* A is an automaton whose set of external actions contains a collection of special *time-passage* actions $\{\nu(t) \mid t \in \mathbb{R}^{>0}\}$. Define the set of *visible* actions to be $vis(A) \triangleq ext(A) - \{\nu(t) \mid t \in \mathbb{R}^{>0}\}$. The automaton A must satisfy the following two axioms.

S1 If $(s, \nu(t), s') \in steps(A)$ and $(s', \nu(t'), s'') \in steps(A)$, then $(s, \nu(t+t'), s'') \in steps(A)$.

To be able to state the second axiom, the following auxiliary definition is needed. Let I be an interval of $\mathbb{R}^{\geq 0}$ with minimum element 0. Then a function $\omega : I \rightarrow states(A)$ is an *A-trajectory*, sometimes called *trajectory* when A is clear from context, if for all $t, t' \in I$ with $t < t'$, $(\omega(t), \nu(t'-t), \omega(t')) \in steps(A)$. That is, ω assigns a state to each time t so that time-passage steps can span between any pair of states in the range of ω. Denote $sup(I)$ by $ltime(\omega)$. Denote $\omega(0)$ by $fstate(\omega)$, and if I is right closed, then denote $\omega(ltime(\omega))$ by $lstate(\omega)$. If I is closed, then ω is said to be an *A-trajectory from $fstate(\omega)$ to $lstate(\omega)$*. An A-trajectory ω whose domain $dom(\omega)$ is the point interval $[0,0]$ is called a *point trajectory* and is also denoted by the set $\{\omega(0)\}$. The range of ω is denoted by $rng(\omega)$.

The second axiom then becomes

S2 If $(s, \nu(t), s') \in steps(A)$ then there is an A-trajectory from s to s' with domain $[0, t]$. ∎

Axioms **S1** and **S2** state natural properties of time, namely that if time can pass in two steps, then it can also pass in a single step, and if time t can pass, then it is possible to associate states with all times in the interval $[0, t]$ in a consistent way. In [LV95] axiom **S2** is explained further and compared to the weaker axiom that says the following: if time can pass in one step, then it can pass in two steps with the time of the intermediate state being any time in the interval.

Timed Executions

Section 3 introduced the notions of execution and trace for automata. These notions carry over to timed automata with the addition of one new idea. In particular, the notion of execution for automata allows one to associate states with only a countable number of points in time, whereas the trajectory axiom **S2** allows one to associate states with all real times. Also, the intuition about the execution of a timed system is that visible actions occur at points in time, and that time passes "continuously" between these points. These observations lead to the definition of a *timed execution*. The definition is close to the notion of *hybrid computation* of [MMP91] where continuous changes and discrete events alternate during the execution of a system.

A *timed execution fragment* Σ of a timed automaton A is a (finite or infinite) sequence of alternating A-trajectories and actions in $vis(A) \cup int(A)$, starting in a trajectory and, if the sequence is finite, ending in a trajectory

$$\Sigma = \omega_0 a_1 \omega_1 a_2 \omega_2 \cdots$$

such that the following holds for each index i:

1. If ω_i is not the last trajectory in Σ, then its domain is a closed interval.

2. If ω_i is not the last trajectory of Σ, then $(lstate(\omega_i), a_{i+1}, fstate(\omega_{i+1})) \in steps(A)$.

A *timed execution* is a timed execution fragment $\omega_0 a_1 \omega_1 a_2 \omega_2 \cdots$ for which $fstate(\omega_0)$ is a start state. If Σ is a timed execution fragment, then define $fstate(\Sigma)$ to be $fstate(\omega_0)$, where ω_0 is the first trajectory of Σ. Also, define $ltime(\Sigma)$ to be the sum of the suprema of the domains of the trajectories of Σ. That is, $ltime(\omega_0 a_1 \omega_1 a_2 \omega_2 \cdots a_n \omega_n) = \sum_{0 \leq i \leq n} ltime(\omega_i)$, and $ltime(\omega_0 a_1 \omega_1 a_2 \omega_2 \cdots) = \sum_{i \geq 0} ltime(\omega_i)$. Finally, if Σ is a finite sequence where the domain of the last trajectory ω is a closed interval, define $lstate(\Sigma)$ to be $lstate(\omega)$.

Finite, Admissible, and Zeno Timed Executions

The timed executions and timed execution fragments of a timed automaton can be partitioned into *finite*, *admissible*, and *Zeno* timed executions and timed execution fragments.

A timed execution (fragment) Σ is defined to be *finite*, if it is a finite sequence and the domain of the last trajectory is a closed interval. A timed execution (fragment) Σ is *admissible* if $ltime(\Sigma) = \infty$. Finally, a timed execution (fragment) Σ is *Zeno* if it is neither finite nor admissible. Denote by $t\text{-}frag^*(A)$, $t\text{-}frag^\infty(A)$, $t\text{-}frag^Z(A)$, and $t\text{-}frag(A)$ the sets of finite, admissible, Zeno, and all timed execution fragments of A. Similarly, denote by $t\text{-}exec^*(A)$, $t\text{-}exec^\infty(A)$, $t\text{-}exec^Z(A)$, and $t\text{-}exec(A)$ the sets of finite, admissible, Zeno, and all timed executions of A.

There are basically two types of Zeno timed executions: those containing infinitely many occurrences of non-time-passage actions, and those containing finitely many occurrences of non-time-passage actions and for which the domain of the last trajectory is right-open. Thus, Zeno timed executions represent executions of a timed automaton where an infinite amount of activity occurs in a bounded period of time. (For the second type of Zeno timed executions, the infinitely many time-passage steps needed to span the right-open interval should be thought of as an "infinite amount of activity".)

A finite timed execution fragment $\Sigma_1 = \omega_0 a_1 \omega_1 \cdots a_n \omega_n$ of A and a timed execution fragment $\Sigma_2 = \omega'_n a_{n+1} \omega_{n+1} a_{n+2} \omega_{n+2} \cdots$ of A can be *concatenated* if $lstate(\Sigma_1) = fstate(\Sigma_2)$. The concatenation, written $\Sigma_1 \frown \Sigma_2$, is defined to be $\Sigma = \omega_0 a_1 \omega_1 \cdots a_n (\omega_n \frown \omega'_n) a_{n+1} \omega_{n+1} a_{n+2} \omega_{n+2} \cdots$, where, for any trajectories ω and ω' with $lstate(\omega) = fstate(\omega')$,

$$\omega \frown \omega'(t) \triangleq \begin{cases} \omega(t) & \text{if } t \leq ltime(\omega) \\ \omega'(t - ltime(\omega)) & \text{otherwise.} \end{cases}$$

It is easy to see that Σ is a timed execution fragment of A.

The notion of prefix for timed execution fragments is defined as follows. A timed execution fragment Σ_1 of A is a *prefix* of a timed execution fragment Σ_2 of A, written $\Sigma_1 \leq_t \Sigma_2$, if either $\Sigma_1 = \Sigma_2$ or Σ_1 is finite and there exists a timed execution fragment Σ'_1 of A such that $\Sigma_2 = \Sigma_1 \frown \Sigma'_1$. Likewise, Σ_1 is a *suffix* of Σ_2 if there exists a finite timed execution fragment Σ'_1 such that $\Sigma_2 = \Sigma'_1 \frown \Sigma_1$. For a finite timed execution fragment Σ_1 and a timed execution fragment Σ_2 with $\Sigma_1 \leq_t \Sigma_2$, define $\Sigma_2 - \Sigma_1$ to be the (unique) timed execution fragment Σ'_1 such that $\Sigma_2 = \Sigma_1 \frown \Sigma'_1$.

Define $\Sigma \triangleleft t$, read "Σ before t", for all $t \geq 0$, to be the prefix of Σ that includes exactly all states with times not bigger than t. Formally,

$$\Sigma \triangleleft t \triangleq \begin{cases} \Sigma & \text{if} \quad t \geq ltime(\Sigma) \\ \Sigma' & \text{if} \quad t < ltime(\Sigma) \text{ and there exists } \Sigma'' = \omega''_0 a''_1 \omega''_1 \cdots \text{ such that} \\ & \qquad \Sigma = \Sigma' \frown \Sigma'' \text{ and } ltime(\Sigma') = t \text{ and } \omega''_0 \text{ is not a point trajectory.} \end{cases}$$

Likewise, define $\Sigma \triangleright t$, read "Σ after t", for all $t < ltime(\Sigma)$ or all $t \leq ltime(\Sigma)$ when Σ is finite, to be the suffix of Σ that includes exactly all states with times not smaller than t. Formally,

$$\Sigma \triangleright t \triangleq \begin{cases} \Sigma' & \text{if} \quad \text{there exists } \Sigma'' = \omega''_0 a''_1 \omega''_1 \cdots \omega''_n \text{ such that} \\ & \qquad \Sigma = \Sigma'' \frown \Sigma' \text{ and } ltime(\Sigma'') = t \text{ and } \omega''_n \text{ is not a point trajectory.} \end{cases}$$

Observe that $\Sigma \triangleleft t$ and $\Sigma \triangleright t$ include also all the actions that occur at time t. In this paper we apply the operators $\triangleleft$ and $\triangleright$ mostly to trajectories. By specializing the definitions above, $\omega \triangleleft t$ is the restriction of ω to the interval $[0, t]$, while $\omega \triangleright t$ is a trajectory ω' such that, for each $t' \geq 0$, $\omega'(t') = \omega(t' - t)$.

Timed Traces

In the untimed model automata are compared based on their traces. This turns out to be inadequate in the timed model, since time is invisible in a trace (cf. [LV95] for more details). This leads to timed traces, which consist of visible actions paired with their time of occurrence (timed sequences) together with a time of termination.

A *timed sequence* over a set K is defined to be a (finite or infinite) sequence δ over $K \times \mathbb{R}^{\geq 0}$ in which the second components of every pair (the *time* components) are nondecreasing. Define δ to be *Zeno* if it is infinite and the limit of the time components is finite. For any nonempty timed sequence δ, define $ftime(\delta)$ to be the time component of the first pair in δ.

A *timed sequence pair* over K is a pair $\gamma = (\delta, t)$, where δ is a timed sequence over K and $t \in \mathsf{T} \cup \{\infty\}$, such that t is greater than or equal to all time components in δ. Let $seq(\gamma)$ and $ltime(\gamma)$ denote the two respective components of γ. Then define $ftime(\gamma)$ to be equal $ftime(seq(\gamma))$ in case $seq(\gamma)$ is nonempty, and equal to $ltime(\gamma)$ otherwise. Denote by $tsp(K)$ the set of timed sequence pairs over K. A timed sequence pair γ is said to be *finite* if both $seq(\gamma)$ and $ltime(\gamma)$ are finite, and *admissible* if $seq(\gamma)$ is not Zeno and $ltime(\gamma) = \infty$.

Let $\Sigma = \omega_0 a_1 \omega_1 a_2 \omega_2 \cdots$ be a timed execution fragment of a timed automaton A. For each a_i, define the *time of occurrence* t_i to be $\sum_{0 \leq j < i} ltime(\omega_j)$. Then, define $t\text{-}seq(\Sigma)$ to be the sequence consisting of the actions in Σ paired with their time of occurrence:

$$t\text{-}seq(\Sigma) = (a_1, t_1)(a_2, t_2) \cdots$$

Then $t\text{-}trace(\Sigma)$, the *timed trace* of Σ, is defined to be the timed sequence pair over $vis(A)$:

$$t\text{-}trace(\Sigma) \triangleq (t\text{-}seq(\Sigma) \lceil (vis(A) \times \mathsf{T}), ltime(\Sigma)).$$

Thus, $t\text{-}trace(\Sigma)$ records the occurrences of *visible* actions together with their time of occurrence, and the limit time of the timed execution fragment. A timed trace suppresses both internal and time-passage actions.

Let $t\text{-}traces^*(A)$, $t\text{-}traces^\infty(A)$, $t\text{-}traces^Z(A)$, and $t\text{-}traces(A)$ denote the sets of timed traces of A obtained from finite, admissible, Zeno, and all timed executions of A, respectively.

4.2 Live Timed Automata

The definition of a live timed automaton is similar to the definition of a live automaton (Definition 3.2) except for the fact that the liveness condition is a set of *timed executions*.

Definition 4.2 (Live timed automaton) A *liveness condition* L for a timed automaton A is a subset of the timed executions of A such that any finite timed execution of A has an extension in L. Formally, $L \subseteq t\text{-}exec(A)$ such that for all $\Sigma \in t\text{-}exec^*(A)$ there exists a $\Sigma' \in t\text{-}frag(A)$, such that $\Sigma \frown \Sigma' \in L$.

A *live timed automaton* is a pair (A, L), where A is a timed automaton and L is a liveness condition for A. The timed executions of L are called the *live timed executions* of A. ■

4.3 Safe Timed I/O Automata

Definition 4.3 (Safe timed I/O automaton) A *safe timed I/O automaton* is a timed automaton augmented with a *visible action signature*, $vsig(A) = (in(A), out(A))$, which partitions $vis(A)$ into input and output actions. A must be *input-enabled*.

The internal and output actions of a safe timed I/O automaton A are referred to as the *locally-controlled* actions of A, written $local(A)$. Thus, $local(A) = int(A) \cup out(A)$. ■

Parallel composition of safe timed I/O automata is defined similarly to the way it is defined for the untimed model (Definition 3.4). All the time-passage actions synchronize. Thus, time is only allowed to pass by a certain amount in the composition if all components allow the same amount of time to pass.

Definition 4.4 (Parallel composition) Two safe timed I/O automata A_0 and A_1 are *compatible* if the following conditions hold:

1. $out(A_0) \cap out(A_1) = \emptyset$

2. $int(A_0) \cap acts(A_1) = int(A_1) \cap acts(A_0) = \emptyset$.

The *parallel composition* $A_0 \| A_1$ of two compatible safe timed I/O automata A_0 and A_1 is the safe timed I/O automaton A such that

1. $states(A) = states(A_0) \times states(A_1)$

2. $start(A) = start(A_0) \times start(A_1)$

3. $out(A) = out(A_0) \cup out(A_1)$

4. $in(A) = (in(A_0) \cup in(A_1)) - out(A)$

5. $int(A) = int(A_0) \cup int(A_1)$

6. $((s_0, s_1), a, (s'_0, s'_1)) \in steps(A)$ iff for all $i \in \{0, 1\}$

   (a) if $a \in acts(A_i)$ then $(s_i, a, s'_i) \in steps(A_i)$

   (b) if $a \notin acts(A_i)$ then $s_i = s'_i$. ■

Note how Condition 6 of Definition 4.4 captures both time-passage steps (where all components participate) and other steps (where a subset of the components participate).

Lemma 3.5 carries over to the timed case. However, a new definition of projection is needed for timed executions. Specifically, let $A = A_0 \| A_1$. For any A-trajectory ω, define $\omega \lceil A_i$ to be obtained from ω by projecting every state in the range of ω to A_i. Let $\Sigma = \omega_0 a_1 \omega_1 a_2 \omega_2 \cdots$ be an alternating sequence of A-trajectories and actions from $acts(A) - \{\nu(t) \mid t \in \mathbb{R}^{>0}\}$. We say that Σ is *well-formed* if Σ does not end in an action if it is a finite sequence, the domain of each trajectory ω_j that is not the last function of Σ is closed, and, for each A_i and each j such that $a_j \notin acts(A_i)$, $lstate(\omega_{j-1}) \lceil A_i = fstate(\omega_j) \lceil A_i$. Then, if Σ is well-formed, the projection $\Sigma \lceil A_i$ of Σ onto A_i is obtained by projecting each ω_k of Σ onto A_i, removing each action a_j that is not an action of A_i, and concatenating each pair of (projected) functions ω_k, ω_{k+1} whose interleaved action is removed. The next lemma is the analog of Lemma 3.5 in the untimed model.

Lemma 4.5 *Let $A = A_0 \| A_1$. Let $\Sigma = \omega_0 a_1 \omega_1 a_2 \omega_2 \cdots$ be a well-formed alternating sequence of A-trajectories and actions from $acts(A) - \{\nu(t) \mid t \in \mathbb{R}^{>0}\}$. Then,*

1. *$\Sigma \lceil A_i \in t\text{-}exec^*(A_i)$, for all A_i, iff $\Sigma \in t\text{-}exec^*(A)$.*

2. *$\Sigma \lceil A_i \in t\text{-}exec^\infty(A_i)$, for all A_i, iff $\Sigma \in t\text{-}exec^\infty(A)$.*

3. *$\Sigma \lceil A_i \in t\text{-}exec(A_i)$, for all A_i, iff $\Sigma \in t\text{-}exec(A)$.*

4. *If $\Sigma \in t\text{-}exec(A)$ then, for all i, $ltime(\Sigma) = ltime(\Sigma \lceil A_i)$.* ■

4.4 Live Timed I/O Automata

In order to define live timed I/O automata, we generalize the notion of receptiveness to timed systems. As for the untimed model, a live timed I/O automaton is receptive if it can behave properly independently of the behavior of the environment. Specifically, a game is set up between a timed automaton and its environment and the timed automaton is receptive iff it has a winning strategy against its environment. The notion of strategy is similar to the one used for the untimed model. However, the presence of time has a strong impact on the type of interactions that can occur between a timed automaton and its environment.

In the untimed model the environment is allowed to provide any finite number of input actions at each move, and the system is allowed to perform at most one of its locally-controlled actions at each move. Thus, the fact that the environment can be arbitrarily fast with respect to the system, but not infinitely fast, is reflected in the structure of the environment moves. This structure is not needed in the timed model since actions in the timed model are associated with specific times. In particular, the relative speeds of the system and the environment are given directly by their timing constraints. The behavior of the environment during the game can be represented simply as a timed sequence over input actions.

In the untimed model a strategy is not allowed to base its decisions on any future input actions from the environment. In the timed model, not only is the strategy not allowed to know about the occurrence of future input actions, but the strategy is also not allowed to know anything about the *timing* of such input actions, e.g., that no inputs will arrive in the next ϵ time units. Thus, if a strategy in the timed model decides to let time pass, it is required to specify explicitly all intermediate states. By specifying all states at intermediate times for a time-passage step, the current state of the system will always be known should the time-passage step be interrupted by an input action. This leads us to the main idea behind the definition of a timed strategy, that is, the system lets time pass by providing a trajectory.

As in the untimed model, a strategy in the timed model is a pair of functions (g, f). Function f takes a finite timed execution and decides how the system behaves till its next locally-controlled action under the assumption that no input are received in the meantime; function g decides what state to reach whenever some input is received.

Definition 4.6 (Strategy) Consider any safe timed I/O automaton A. A *strategy* defined on A is a pair of functions (g, f) where $g : t\text{-}exec^*(A) \times in(A) \rightarrow states(A)$ and $f : t\text{-}exec^*(A) \rightarrow (traj(A) \times local(A) \times states(A)) \cup traj(A)$, where $traj(A)$ is the set of A-trajectories, such that

1. $g(\Sigma, a) = s$ implies $\Sigma a\{s\} \in t\text{-}exec^*(A)$,
2. $f(\Sigma) = (\omega, a, s)$ implies $\Sigma \frown \omega a\{s\} \in t\text{-}exec^*(A)$,
3. $f(\Sigma) = \omega$ implies $\Sigma \frown \omega \in t\text{-}exec^{\infty}(A)$,
4. f is *consistent*, i.e., if $f(\Sigma) = (\omega, a, s)$, then, for each $t \leq ltime(\omega)$, $f(\Sigma \frown (\omega \triangleleft t)) = (\omega \triangleright t, a, s)$, and, if $f(\Sigma) = \omega$, then, for each $t < ltime(\omega)$, $f(\Sigma \frown (\omega \triangleleft t)) = \omega \triangleright t$.

For notational convenience define

$$f(\Sigma).trj \triangleq \begin{cases} \omega & \text{if } f(\Sigma) = (\omega, a, s) \\ \omega & \text{if } f(\Sigma) = \omega. \end{cases}$$

■

Condition 1 of Definition 4.6 states that g returns a "legal" next state given an input. Conditions 2 and 3 describe the two possible system moves given by f: either f specifies time-passage followed by a local step, or f specifies that the system simply lets time pass forever. Note that f specifies all states during time passage. The consistency condition (Condition 4) for f says that, if after a finite timed execution Σ the system decides to behave according to $\omega a\{s\}$ or ω, then after performing a part of ω the system decides to behave according to the rest of $\omega a\{s\}$ or ω. In other words, a strategy decision cannot change in the absence of any inputs. The consistency condition is required for the closure of the composition operator.

The game between the system and the environment works as follows. The environment can provide any input at any time, while the system lets time pass and provides locally-controlled actions based on its strategy. At any point in time the system decides its next move using function f. If an input comes, the system performs its current step just until the time the input occurs, and then uses function g to compute the state reached as a result of the input.

A new problem arises when the system decides to perform an action at the same time the environment is providing some input. Our model does not rule out such race conditions. Practical examples of such situations arise whenever the system has some timeout mechanism and the input occurs exactly when the timeout period expires. The race conditions are modeled as nondeterministic choices. As a consequence, the *outcome*, that is, the result of the game, for a timed strategy is a set of timed executions.

The following definition of the outcome of a strategy for safe timed I/O automata parallels the corresponding definition in the untimed model.

Definition 4.7 (Outcome of a strategy) Let A be a safe timed I/O automaton and (g, f) a strategy defined on A. Define a *timed environment sequence* for A to be a timed sequence over $in(A)$, and define a timed environment sequence $\mathcal{I}$ for A to be *compatible* with a timed execution fragment Σ of A if either $\mathcal{I}$ is empty, or Σ is finite and $ltime(\Sigma) \leq ftime(\mathcal{I})$. Then define $R_{(g,f)}$, the *next-relation induced by* (g, f), as follows: for any $\Sigma, \Sigma' \in t\text{-}exec(A)$ and any $\mathcal{I}, \mathcal{I}'$ compatible with Σ, Σ', respectively, $((\Sigma, \mathcal{I}), (\Sigma', \mathcal{I}')) \in R_{(g,f)}$ iff

$$(\Sigma', \mathcal{I}') = \begin{cases} (\Sigma \frown \omega a\{s\}, \mathcal{I}) & \text{if } \Sigma \text{ is finite, } \mathcal{I} = \varepsilon,\ f(\Sigma) = (\omega, a, s), \\[2ex] (\Sigma \frown \omega, \mathcal{I}) & \text{if } \Sigma \text{ is finite, } \mathcal{I} = \varepsilon,\ f(\Sigma) = \omega, \\[2ex] (\Sigma \frown \omega a\{s\}, \mathcal{I}) & \text{if } \Sigma \text{ is finite, } \mathcal{I} = (b, t)\mathcal{I}'', f(\Sigma) = (\omega, a, s), \\ & \quad ltime(\Sigma \frown \omega) \leq t, \\[2ex] (\Sigma \frown \omega' a\{s'\}, \mathcal{I}'') & \text{if } \Sigma \text{ is finite, } \mathcal{I} = (a, t)\mathcal{I}'',\ f(\Sigma).trj = \omega, \\ & \quad ltime(\Sigma \frown \omega) \geq t,\ \omega' = \omega \triangleleft (t - ltime(\Sigma)),\ s' = g(\Sigma \frown \omega', a), \\[2ex] (\Sigma, \mathcal{I}) & \text{if } \Sigma \text{ is not finite.} \end{cases}$$

Let Σ be a finite timed execution of A, and $\mathcal{I}$ be a timed environment sequence for A *compatible* with Σ.

An *outcome sequence of* (g, f) *given* Σ *and* $\mathcal{I}$ is an infinite sequence $(\Sigma^n, \mathcal{I}^n)_{n \geq 0}$ that satisfies:

- $(\Sigma^0, \mathcal{I}^0) = (\Sigma, \mathcal{I})$ and
- for all $n > 0$, $((\Sigma^{n-1}, \mathcal{I}^{n-1}), (\Sigma^n, \mathcal{I}^n)) \in R_{(g,f)}$.

Note, that $(\Sigma^n)_{n \geq 0}$ forms a chain ordered by *prefix*.

The *outcome* $\mathcal{O}_{(g,f)}(\Sigma, \mathcal{I})$ of the strategy (g, f) given Σ and $\mathcal{I}$ is the set of timed executions Σ' for which there exists an outcome sequence $(\Sigma^n, \mathcal{I}^n)_{n \geq 0}$ of (g, f) given Σ and $\mathcal{I}$ such that $\Sigma' = \lim_{n \to \infty} \Sigma^n$. ■

The set of outcome sequences of (g, f) given some Σ and $\mathcal{I}$ is determined step by step using the next-relation $R_{(g,f)}$. The first case of the definition of $R_{(g,f)}$ deals with the situation where no input occurs and the system performs an action; the second case deals with the situation where no input occurs and the system lets time pass forever; the third case deals with the situation where both the environment and the system provide some action and the system does not provide its action after the environment does; the fourth case deals with the situation where both the environment and the system provide some action and the environment does not provide its action after the system does; the fifth case is needed for technical convenience, since the second case produces an admissible timed execution. Note, that the third and fourth cases may both be applicable whenever the next input action of $\mathcal{I}$ and the local action chosen by f occur at the same time. This is why the outcome is a *set* of timed executions.

The following lemma states that an outcome set is never empty and that an element of an outcome cannot be finite. Furthermore, if an element of an outcome is Zeno, it contains infinitely many actions (other than the implicit time-passage actions).

Lemma 4.8 *Let A be a safe timed I/O automaton, (g, f) a strategy defined on A, Σ a finite timed execution of A, and $\mathcal{I}$ a timed environment sequence for A compatible with Σ. Then $\mathcal{O}_{(g,f)}(\Sigma, \mathcal{I}) \neq \emptyset$ and $\mathcal{O}_{(g,f)}(\Sigma, \mathcal{I}) \subseteq (t\text{-}exec^{\infty}(A) \cup t\text{-}exec^{Z}(A))$. Furthermore, if $\Sigma' \in \mathcal{O}_{(g,f)}(\Sigma, \mathcal{I})$ and $\Sigma' \in t\text{-}exec^{Z}(A)$, then $|\Sigma' \upharpoonright acts(A)| = \infty$.*

Proof. Let $R_{(g,f)}$ be the next-relation induced by (g, f). Construct an outcome sequence of (g, f) given Σ and $\mathcal{I}$ inductively as follows. Define $(\Sigma^0, \mathcal{I}^0) = (\Sigma, \mathcal{I})$. For any $n > 0$, assume $(\Sigma^{n-1}, \mathcal{I}^{n-1})$ has been defined. Then it is easy to see that the condition of at least one case in the definition of $R_{(g,f)}$ is satisfied. Thus, define $(\Sigma^n, \mathcal{I}^n)$ to be any pair such that $((\Sigma^{n-1}, \mathcal{I}^{n-1}), (\Sigma^n, \mathcal{I}^n)) \in R_{(g,f)}$. This inductively defined outcome sequence gives rise to an element in $\mathcal{O}_{(g,f)}(\Sigma, \mathcal{I})$. That proves that $\mathcal{O}_{(g,f)}(\Sigma, \mathcal{I})$ is not empty.

Let $(\Sigma^n, \mathcal{I}^n)$ be an arbitrary outcome sequence of (g, f) given Σ and $\mathcal{I}$. Clearly, $\Sigma^0 = \Sigma \in t\text{-}exec(A)$. Assume, that $\Sigma^{n-1} \in t\text{-}exec(A)$. Then, by the four conditions of Definition 4.6, it is easy to see that also $\Sigma^n \in t\text{-}exec(A)$. Thus, by induction, $\Sigma^n \in t\text{-}exec(A)$ for all $n \geq 0$. Suppose by contradiction that $\Sigma' = \lim_{n \to \infty}(\Sigma^n) \notin t\text{-}exec(A)$. Then there must be a finite *prefix* Σ'' of Σ' such that $\Sigma'' \notin$ *t-exec*$^*(A)$. Also, Σ'' must be a *prefix* of Σ^n for some n. However, this contradicts the fact that $\Sigma^n \in$ *t-exec*(A). Thus, $\Sigma' \in$ *t-exec*(A).

Now, assume by contradiction that Σ' is finite. Then there exists a number n' such that for all $n > n'$, $\Sigma^n = \Sigma^{n-1} = \Sigma'$, but this contradicts the definition of $R_{(g,f)}$, since $\Sigma^n = \Sigma^{n-1}$ only if Σ^{n-1} is admissible. Thus, $\mathcal{O}_{(g,f)}(\Sigma, \mathcal{I}) \subseteq (t\text{-}exec^{\infty}(A) \cup t\text{-}exec^{Z}(A))$.

Finally, it is easy to see that if $\Sigma' \in t\text{-}exec^{Z}(A)$, then Σ' is an infinite sequence of trajectories and actions. Only the second case in the definition of $R_{(g,f)}$ can lead to a finite sequence, but in this case the outcome would be admissible (cf. Definition 4.6 Condition 3). ■

Another problem due to the explicit presence of time in the model is the capability of a system to block time. Under the reasonable assumption that it is natural for a system to require time to advance forever, a timed automaton that blocks time cannot be receptive. Thus, we could assume that finite and Zeno timed executions are not live and that the environment cannot block time. However, as is illustrated in the following example due to Lamport, Zeno timed executions cannot be ignored completely.

Example 4.1 Consider two safe timed I/O automata A, B such that $in(A) = out(B) = \{b\}$ and $out(A) = in(B) = \{a\}$. Let A start by performing its output action a and let B start by waiting for some input. Furthermore, let both A and B reply to their n^{th} input with an output action exactly $1/2^n$ time units after the input has occurred.

Consider the following definition of receptiveness, which assumes that the environment does not behave in a Zeno manner: a pair (A, L) is receptive iff there exists a strategy (g, f) defined on A such that for each finite timed execution Σ of A and any admissible timed environment sequence $\mathcal{I}$ for A compatible with Σ we have $\mathcal{O}_{(g,f)}(\Sigma, \mathcal{I}) \subseteq L$. Then it is easy to observe that, if L_A and L_B are defined to be the set of admissible timed executions of A and B, respectively, the pairs (A, L_A) and (B, L_B) are receptive. However, the parallel composition of A and B yields no admissible executions, rather it only yields a Zeno timed execution, which blocks time. Thus, the parallel composition of (A, L_A) and (B, L_B) constrains the environment. Observe that (A, L_A) and (B, L_B) "unintentionally" collaborate to generate a Zeno timed execution: each pair looks like a Zeno environment to the other. ■

To eliminate the problem of Example 4.1 one must ensure that a system does not collaborate with its environment to generate a Zeno timed execution. We call those timed executions where the environment is Zeno but the system does not collaborate with the environment to generate the Zeno timed execution *Zeno-tolerant*.

Definition 4.9 (Special types of timed executions) Given a safe timed I/O automaton A, and given a timed execution Σ of A,

- Σ is said to be *environment-Zeno* if Σ is a Zeno timed execution that contains infinitely many input actions;
- Σ is said to be *system-Zeno* if Σ is a Zeno timed execution that either contains infinitely many locally-controlled actions or contains finitely many actions;

- Σ is said to be *Zeno-tolerant* if it is an environment-Zeno, non-system-Zeno timed execution; equivalently, Σ is Zeno-tolerant if

  1. $ltime(\Sigma)$ is finite,
  2. Σ contains infinitely many input actions, and
  3. Σ contains finitely many locally-controlled actions.

  Denote by $t\text{-}exec^{Zt}(A)$ the set of Zeno-tolerant timed executions of A. ■

The notion of environment-Zenoness captures the fact that the environment contributes to the Zenoness of a timed execution. The environment can contribute only by providing infinitely many actions in a finite time. The notion of system-Zenoness captures the fact that the system contributes to the Zenoness of a timed execution. The system can contribute either by providing infinitely many actions in a finite time, or by letting time pass in a Zeno way, without producing any action, even though the environment does not provide any more actions. The notion of Zeno-tolerance captures the fact that only the environment contributes to the Zenoness of a timed execution.

In Example 4.1 the unique execution of $A\|B$ that contains infinitely many actions is an example of an environment-Zeno and system-Zeno timed execution. We define a strategy to be Zeno-tolerant if it guarantees that the system never chooses to block time in order to win its game against the environment. That is, a Zeno-tolerant strategy produces Zeno timed executions only when applied to a Zeno timed environment sequence $\mathcal{I}$, and in these cases the outcome is Zeno-tolerant. Thus, the system does not respond to Zeno inputs by behaving in a Zeno fashion.

Definition 4.10 (Zeno-tolerant strategy) A strategy (g, f) defined on a safe timed I/O automaton A is said to be *Zeno-tolerant* if, for every finite timed execution $\Sigma \in t\text{-}exec^*(A)$ and every timed environment sequence $\mathcal{I}$ for A compatible with Σ, $\mathcal{O}_{(g,f)}(\Sigma, \mathcal{I}) \subseteq t\text{-}exec^{\infty}(A) \cup t\text{-}exec^{Zt}(A)$. ■

We can now define receptiveness by requiring a system to behave according to its liveness condition under non-Zeno environments and in a Zeno-tolerant way under Zeno environments.

Definition 4.11 (Receptiveness) Let A be a safe timed I/O automaton and $L \subseteq t\text{-}exec(A)$. A timed strategy (g, f) defined on A is called a *receptive strategy* for (A, L) if (g, f) is Zeno-tolerant and for each finite timed execution Σ of A and each timed environment sequence $\mathcal{I}$ for A compatible with Σ, $\mathcal{O}_{(g,f)}(\Sigma, \mathcal{I}) \subseteq L \cup t\text{-}exec^{Zt}(A)$. The pair (A, L) is *receptive* if there exists a receptive strategy for (A, L). ■

A pair (A, L) is receptive if, after any finite timed execution and with any (Zeno or non-Zeno) sequence of input actions, it can generate some admissible timed execution in L or some Zeno-tolerant timed execution. Also, A must never generate one of its finite or system-Zeno timed executions, since it would constrain its environment in this case. Thus liveness conditions should not include any finite or system-Zeno timed execution. Zeno-tolerant timed executions are used only to handle illegal interactions, and therefore also should not be included in liveness conditions. This leads to the definition of *live timed I/O automata*, where the liveness condition contains only admissible timed executions, but the strategy is allowed to yield *Zeno-tolerant* outcomes when given a Zeno timed environment sequence.

Definition 4.12 (Live timed I/O automaton) A *live timed I/O automaton* is a pair (A, L), where A is a safe timed I/O automaton and $L \subseteq \textit{t-exec}^{\infty}(A)$, such that the pair (A, L) is receptive. ■

Lemma 4.13 *If (A, L) is a live timed I/O automaton, then L is a liveness condition for A.*

Proof. Given a finite timed execution Σ of A, consider a receptive strategy (g, f) for (A, L). Consider any timed execution $\Sigma \frown \Sigma' \in \mathcal{O}_{(g,f)}(\Sigma, \varepsilon)$. Such a timed execution exists according to Lemma 4.8. The timed execution $\Sigma \frown \Sigma'$ is not Zeno-tolerant since it contains finitely many input actions. Therefore $\Sigma \frown \Sigma'$ is a timed execution of L, that is, Σ can be extended to a timed execution of L. ■

There is an interesting property that connects Zeno-tolerance, receptiveness, and admissibility. This property emphasizes the importance of admissible timed executions in the timed model.

Proposition 4.14 *Let A be a timed I/O automaton and (g, f) be a timed strategy defined on A. Then (g, f) is receptive for $(A, \textit{t-exec}^{\infty}(A))$ iff (g, f) is Zeno-tolerant.*

Proof. Follows trivially from the definitions. ■

As in the untimed model, the parallel composition operator defined for safe timed I/O automata is extended to live timed I/O automata.

Definition 4.15 (Parallel composition) Two live timed I/O automata (A_0, L_0) and (A_1, L_1) are *compatible* iff the safe timed I/O automata A_0 and A_1 are compatible. The *parallel composition* $(A_0, L_0)\|(A_1, L_1)$ of compatible live timed I/O automata (A_0, L_0) and (A_1, L_1) is defined to be the pair (A, L) where $A = A_0\|A_1$ and $L = \{\Sigma \in \textit{t-exec}(A) \mid \Sigma\lceil A_0 \in L_0 \text{ and } \Sigma\lceil A_1 \in L_1\}$. ■

As expected, parallel composition is closed for live timed I/O automata in the sense that it produces a new live timed I/O automaton; however, the proof of closure is quite complex. For compatible live timed I/O automata (A_0, L_0) and (A_1, L_1), let (A, L) denote the parallel composition $(A_0, L_0)\|(A_1, L_1)$. In order to prove that (A, L) is a live timed I/O automaton we must show that (A, L) is receptive, which, in turn, requires finding a receptive strategy for (A, L).

The proof proceeds by first defining a strategy (g, f) for (A, L) based on a strategy (g_i, f_i) for each (A_i, L_i), and then proving that (g, f) is a receptive strategy for (A, L). Function g computes, given input a, the next state according to the g_i functions of those components of

A for which a is an input action, and simply leaves the state unchanged for those components for which a is not an action. Function f determines, using the f_i functions, which component is allowed to execute the next locally-controlled action. Say this is component k and it wishes to perform action a at time t. Then each component A_i evolves based on f_i up to time t. Furthermore, at time t, A_k takes a step based on f_k and, if a is an input action of $A_{(1-k)}$, $A_{(1-k)}$ takes a step based on $g_{(1-k)}$. If at time t both A_0 and A_1 want to take a step, then priority is given to A_0. We do not need to enforce any specific tie breaking policy in the timed case: the fact that time must elapse ensures that both A_0 and A_1 have a chances take steps under non-Zeno environments.

Definition 4.16 (Parallel composition of (timed) strategies) Let $A = A_0 \| A_1$ be the parallel composition of two compatible safe timed I/O automata A_0 and A_1, and let (g_0, f_0) and (g_1, f_1) be strategies defined on A_0 and A_1, respectively. The *parallel composition* $(g_0, f_0)\|(g_1, f_1)$ of the strategies (g_0, f_0) and (g_1, f_1) is the pair of functions (g, f)

$$
\begin{aligned}
&g : \textit{t-exec}^*(A) \times \textit{in}(A) \to \textit{states}(A)\\
&f : \textit{t-exec}^*(A) \to (\textit{traj}(A) \times \textit{local}(A) \times \textit{states}(A)) \cup \textit{traj}(A)
\end{aligned}
$$

such that

$$
g(\Sigma, a) = s \quad \text{where, for all } i \in \{0,1\}, \quad s\lceil A_i = \begin{cases} g_i(\Sigma\lceil A_i, a) & \text{for } a \in \textit{in}(A_i) \\ \textit{lstate}(\Sigma)\lceil A_i & \text{for } a \notin \textit{acts}(A_i) \end{cases}
$$

and f is defined as follows: for all $i \in \{0,1\}$, define ω_i to be $f_i(\Sigma\lceil A_i).\textit{trj}$. Pick the smallest k such that $\textit{ltime}(\omega_k) = \min(\textit{ltime}(\omega_0), \textit{ltime}(\omega_1))$. Define ω such that

$$
\omega\lceil A_i = \begin{cases} \omega_k & \text{if } i = k \\ \omega_i \lhd \textit{ltime}(\omega_k) & \text{if } i \neq k. \end{cases}
$$

Distinguish two cases.

1. If $f_k(\Sigma\lceil A_k) = (\omega_k, a, s_k)$ then $f(\Sigma) = (\omega, a, s)$,

   where, for all $i \in \{0,1\}$, $s\lceil A_i = \begin{cases} s_k & \text{if } i = k \\ g_i((\Sigma \frown \omega)\lceil A_i, a) & \text{if } i \neq k \text{ and } a \in \textit{in}(A_i) \\ \textit{lstate}(\omega)\lceil A_i & \text{if } i \neq k \text{ and } a \notin \textit{acts}(A_i). \end{cases}$

2. If $f_k(\Sigma\lceil A_k) = \omega_k$ then $f(\Sigma) = \omega$. ∎

Lemma 4.17 *Let A_0 and A_1 be compatible safe timed I/O automata and let (g_0, f_0) and (g_1, f_1) be strategies defined on A_0 and A_1, respectively. Then $(g_0, f_0)\|(g_1, f_1)$ is a strategy defined on $A_0\|A_1$.*

Proof. Let $(g, f) = (g_0, f_0)\|(g_1, f_1)$. To prove that (g, f) is a strategy defined on A, the four conditions of Definition 4.6 must be checked. Conditions 1–3 are trivial given the definitions of g and f, and the fact that (g_0, f_0) and (g_1, f_1) are strategies defined on A_0 and A_1, respectively. Condition 4 (consistency) needs more analysis.

Let $\Sigma \in t\text{-}exec^*(A)$, and suppose that $f(\Sigma) = (\omega, a, s)$. We leave to the reader the case for $f(\Sigma) = \omega$ since it is simpler. Let t be an arbitrary time such that $t \leq ltime(\omega)$. We show that $f(\Sigma \frown (\omega \lhd t)) = (\omega \rhd t, a, s)$. By the definition of f and by the compatibility of A_0 and A_1, there is a unique index i such that $f_i(\Sigma\lceil A_i) = (\omega\lceil A_i, a, s\lceil A_i)$. Let w_i denote $w\lceil A_i$ and s_i denote $s\lceil A_i$. Let j denote $1 - i$. Then, $f_j(\Sigma\lceil A_j).trj = \omega_j$ for some trajectory w_j such that $w\lceil A_j \leq w_j$, and $s\lceil A_j$ is either $g_j((\Sigma\lceil A_j) \frown (\omega_j \lhd ltime(\omega)), a)$ or $\omega_j(ltime(\omega))$ depending on whether a is an input action of A_j. Since f_j is consistent, $f_j((\Sigma\lceil A_j) \frown (\omega_j \lhd t)).trj = w_j \rhd t$. Furthermore, since f_i is consistent, $f_i((\Sigma\lceil A_i) \frown (\omega \lhd t)\lceil A_i) = ((\omega_i \rhd t), a, s_i)$. By the definition of (g, f), $f(\Sigma \frown (\omega \lhd t)) = (\omega \rhd t, a, s)$. ■

The following lemma is the key step for showing that the strategy of Definition 4.16 is receptive if the component strategies are receptive. Specifically, up to a technical condition, the projection of an outcome of (g, f) onto a component A_i is an outcome of (g_i, f_i). Intuitively this means that even though the composed system uses its composed strategy to find possible outcomes, up to a technical restriction it still looks to each component as if it is using its own component strategy. The restriction says that the projection of a Zeno execution onto A_i contains infinitely many actions. This restriction does not hurt the applicability of the lemma later in Lemma 4.19. The proof of Lemma 4.18 is more complex than the proof of the analogous result for the untimed case (cf. Lemma 3.15).

Lemma 4.18 *Let A_0 and A_1 be compatible safe timed I/O automata and let (g_0, f_0) and (g_1, f_1) be strategies defined on A_0 and A_1. Let $A = A_0\|A_1$ and $(g, f) = (g_0, f_0)\|(g_1, f_1)$. Let Σ be an arbitrary finite timed execution of A, $\mathcal{I}$ be an arbitrary timed environment sequence for A compatible with Σ, Σ' be an arbitrary timed execution of $\mathcal{O}_{(g,f)}(\Sigma, \mathcal{I})$, and i be either 0 or 1. Assume that $|\Sigma' \upharpoonright acts(A_i)| = \infty$ if Σ' is Zeno. Then there exists a timed environment sequence $\mathcal{I}_i$ for A_i compatible with $\Sigma\lceil A_i$, such that $\Sigma'\lceil A_i \in \mathcal{O}_{(g_i,f_i)}(\Sigma\lceil A_i, \mathcal{I}_i)$.*

Proof. Let $R_{(g,f)}$ and $R_{(g_i,f_i)}$ be the next-relations induced by (g, f) and (g_i, f_i), respectively, and let $(\Sigma^n, \mathcal{I}^n)_{n\geq 0}$ be an outcome sequence of (g, f) given Σ and $\mathcal{I}$ such that $\Sigma' = \lim_{n\to\infty} \Sigma^n$. Since $(\Sigma^n)_{n\geq 0}$ forms an infinite chain ordered by prefix and $\Sigma^0 = \Sigma$, $\Sigma \leq_t \Sigma'$. Define $\mathcal{I}_i = t\text{-}seq(\Sigma' - \Sigma) \upharpoonright (in(A_i) \times \mathbb{R}^{\geq 0})$. Then either $\mathcal{I}_i$ is empty or $ftime(\mathcal{I}_i) \geq ltime(\Sigma) = ltime(\Sigma\lceil A_i)$. Thus, $\mathcal{I}_i$ is compatible with $\Sigma\lceil A_i$. For each $n > 0$ define $\mathcal{I}_i^n = t\text{-}seq(\Sigma' - \Sigma^n) \upharpoonright (in(A_i) \times \mathbb{R}^{\geq 0})$.

Define a *sentence* to be a finite timed execution that ends with a point trajectory, i.e., a trajectory whose domain consists of a singleton set. For each $n > 0$, define $sentence(\Sigma^n\lceil A_i)$ to be the maximum between $\Sigma\lceil A_i$ and the maximum prefix of $\Sigma^n\lceil A_i$ that is a sentence. Since for each $n > 0$ the number of actions in Σ^n and Σ^{n-1} differ by at most 1, and since $\Sigma_0 = \Sigma$, it is easy to show that for each $n \geq 0$ there exists $m \leq n$ such that $sentence(\Sigma^n\lceil A_i) = \Sigma^m\lceil A_i$. Denote the minimum such m by $m(n)$. Observe that $m(n)$ is monotonic non-decreasing. Finally, for each $n > 0$, since in $\Sigma^n - \Sigma^{m(n)}$ no action from A_i occurs, $\mathcal{I}_i^n = \mathcal{I}_i^{m(n)}$.

We prove the following facts by induction on n:

P1 If $n > 0$ and $\Sigma^n\lceil A_i$ is not a sentence and is finite, then $\Sigma^n\lceil A_i = (\Sigma^{m(n-1)}\lceil A_i \frown f_i(\Sigma^{m(n-1)}\lceil A_i).trj) \lhd ltime(\Sigma^n)$;

P2 If $n > 0$ and $\Sigma^n\lceil A_i$ is either a sentence or admissible, then either $\Sigma^n\lceil A_i = \Sigma^{n-1}\lceil A_i$ and $\Sigma^n\lceil A_i$ is a sentence, or $((\Sigma^{m(n-1)}\lceil A_i, \mathcal{I}_i^{m(n-1)})), (\Sigma^n\lceil A_i, \mathcal{I}_i^n)) \in R_{(g_i,f_i)}$.

The base case is trivial. For the inductive step assume that properties **P1** and **P2** hold for each $j < n$. Observe first that, if Σ^n is finite, then

P3 $(\Sigma^{n-1}\lceil A_i) \frown f_i(\Sigma^{n-1}\lceil A_i).trj = (\Sigma^{m(n-1)}\lceil A_i) \frown f_i(\Sigma^{m(n-1)}\lceil A_i).trj$.

In fact, if $n = 1$ or $\Sigma^{n-1}\lceil A_i$ is a sentence, then $m(n-1) = n-1$ and **P3** holds trivially; if $\Sigma^{n-1}\lceil A_i$ is not a sentence, then $m(n-1) = m(n-2)$, and thus

$$\begin{aligned}
& (\Sigma^{n-1}\lceil A_i) \frown f_i(\Sigma^{n-1}\lceil A_i).trj \\
\stackrel{1}{=}\ & (((\Sigma^{m(n-2)}\lceil A_i) \frown f_i(\Sigma^{m(n-2)}\lceil A_i).trj) \triangleleft ltime(\Sigma^{n-1})) \frown \\
& f_i(((\Sigma^{m(n-2)}\lceil A_i) \frown f_i(\Sigma^{m(n-2)}\lceil A_i)) \triangleleft ltime(\Sigma^{n-1})).trj \\
\stackrel{2}{=}\ & (\Sigma^{m(n-2)}\lceil A_i) \frown f_i(\Sigma^{m(n-2)}\lceil A_i).trj \\
\stackrel{3}{=}\ & (\Sigma^{m(n-1)}\lceil A_i) \frown f_i(\Sigma^{m(n-1)}\lceil A_i).trj,
\end{aligned}$$

where step 1 follows by induction and from the fact that Σ^{n-1} is not a sentence, step 2 follows from consistency of f_i (cf. Condition 4 of Definition 4.6), and step 3 follows from the fact that $m(n-1) = m(n-2)$. We now distinguish the following cases.

Case 1 Σ^{n-1} is not finite.

Then Σ^n is not finite, and statement **P1** is satisfied trivially. Since $((\Sigma^{n-1}, \mathcal{I}^{n-1}), (\Sigma^n, \mathcal{I}^n)) \in R_{(g,f)}$ and Σ^{n-1} is not finite, then $(\Sigma^{n-1}, \mathcal{I}^{n-1}) = (\Sigma^n, \mathcal{I}^n)$.
By induction, $((\Sigma^{m(n-1)}\lceil A_i, \mathcal{I}_i^{m(n-1)}), (\Sigma^{n-1}\lceil A_i, \mathcal{I}_i^{n-1})) \in R_{(g_i,f_i)}$.
Since $\Sigma^n = \Sigma^{n-1}$, $((\Sigma^{m(n-1)}\lceil A_i, \mathcal{I}_i^{m(n-1)}), (\Sigma^n\lceil A_i, \mathcal{I}_i^n)) \in R_{(g_i,f_i)}$.

Case 2 Σ^{n-1} is finite and Σ^n is not finite.

Since Σ^n is not finite, statement **P1** is satisfied trivially. Since $((\Sigma^{n-1}, \mathcal{I}^{n-1}), (\Sigma^n, \mathcal{I}^n)) \in R_{(g,f)}$, by Definition 4.7, $\Sigma^n = \Sigma^{n-1} \frown \omega$, where $\omega = f(\Sigma^{n-1})$, and $\mathcal{I}_i^n = \mathcal{I}_i^{n-1} = \varepsilon$. Observe that

$$\begin{aligned}
\Sigma^n\lceil A_i &\stackrel{1}{=} (\Sigma^{n-1}\lceil A_i) \frown (\omega\lceil A_i) \\
&\stackrel{2}{=} (\Sigma^{n-1}\lceil A_i) \frown f_i(\Sigma^{n-1}\lceil A_i) \\
&\stackrel{3}{=} (\Sigma^{m(n-1)}\lceil A_i) \frown f_i(\Sigma^{m(n-1)}\lceil A_i)
\end{aligned}$$

where step 1 is trivial, step 2 follows from definition of (g, f) (Definition 4.16) and the fact that $ltime(\omega) = \infty$ (because (g, f) is a strategy), and step 3 follows from **P3**. Thus, by case 2 of Definition 4.7, $((\Sigma^{m(n-1)}\lceil A_i, \mathcal{I}_i^{m(n-1)}), (\Sigma^n\lceil A_i, \mathcal{I}_i^n)) \in R_{(g_i,f_i)}$.

Case 3 Σ^{n-1} and Σ^n are finite.

The definition of $R_{(g,f)}$ gives three cases to consider: the first, third, and fourth cases in Definition 4.7. We consider the first and third cases together.

Case 3.1 First and third cases.

By the definition of $R_{(g,f)}$, $\Sigma^n = \Sigma^{n-1} \frown \omega a\{s\}$, where $f(\Sigma^{n-1}) = (\omega, a, s)$.

Case 3.1.1 $a \notin acts(A_i)$.

Then

$$\begin{aligned} \Sigma^n \lceil A_i &\stackrel{1}{=} (\Sigma^{n-1} \frown \omega a\{s\})\lceil A_i \\ &\stackrel{2}{=} \Sigma^{n-1}\lceil A_i \frown \omega\lceil A_i \\ &\stackrel{3}{=} \Sigma^{n-1}\lceil A_i \frown (f_i(\Sigma^{n-1}\lceil A_i).trj \\ &\stackrel{4}{=} (\Sigma^{m(n-1)}\lceil A_i) \frown f_i(\Sigma^{m(n-1)}\lceil A_i).trj \end{aligned}$$

where step 1 is trivial, step 2 follows from the fact that $a \notin acts(A_i)$, step 3 follows from the definition of (g, f) (cf. Definition 4.16), and step 4 follows from **P3**. This is sufficient to show statement **P1**. For statement **P2**, either $\Sigma^n\lceil A_i$ is not a sentence, or $\Sigma^n\lceil A_i$ is a sentence, but in this case $\Sigma^n\lceil A_i = \Sigma^{n-1}\lceil A_i$, since ω would be a point trajectory.

Case 3.1.2 $a \in local(A_i)$.

Statement **P1** is satisfied trivially since $\Sigma^n\lceil A_i$ is a sentence. By the definition of (g, f) (Definition 4.16) and the fact that $a \in local(A_i)$, $f_i(\Sigma^{n-1}\lceil A_i) = (\omega\lceil A_i, a, s\lceil A_i)$. Observe that

$$\begin{aligned} \Sigma^n\lceil A_i &\stackrel{1}{=} \Sigma^{n-1}\lceil A_i \frown (\omega\lceil A_i)a\{s\lceil A_i\} \\ &\stackrel{2}{=} \Sigma_i^{m(n-1)} \frown (f_i(\Sigma^{m(n-1)}\lceil A_i).trj)a\{s\lceil A_i\} \end{aligned}$$

where step 1 follows from the fact that $a \in acts(A_i)$ and step 2 follows from **P3**. By consistency of (g, f), $f_i(\Sigma^{m(n-1)}\lceil A_i) = (f_i(\Sigma^{m(n-1)}\lceil A_i).trj, a, \{s\lceil A_i\})$. Thus, by cases 1 or 3 of Definition 4.7, $((\Sigma^{m(n-1)}\lceil A_i, \mathcal{I}_i^{m(n-1)}), (\Sigma^n\lceil A_i, \mathcal{I}_i^n)) \in R_{(g_i,f_i)}$.

Case 3.1.3 $a \in in(A_i)$.

Statement **P1** is satisfied trivially since $\Sigma^n\lceil A_i$ is a sentence. Let $t = ltime(\Sigma^{n-1} \frown \omega)$. Observe that $\mathcal{I}_i^{m(n-1)} = (a, t)\mathcal{I}_i^n$. Furthermore,

$$\begin{aligned} \Sigma^n\lceil A_i &\stackrel{1}{=} \Sigma^{n-1}\lceil A_i \frown (\omega\lceil A_i)a\{s\lceil A_i\} \\ &\stackrel{2}{=} (((\Sigma^{n-1}\lceil A_i) \frown f_i(\Sigma^{n-1}\lceil A_i).trj) \triangleleft t)\, a \\ &\quad\; g_i(((\Sigma^{n-1}\lceil A_i) \frown f_i(\Sigma^{n-1}\lceil A_i).trj) \triangleleft t, a) \\ &\stackrel{3}{=} (((\Sigma^{m(n-1)}\lceil A_i) \frown f_i(\Sigma^{m(n-1)}\lceil A_i).trj) \triangleleft t)\, a \\ &\quad\; g_i(((\Sigma^{m(n-1)}\lceil A_i) \frown f_i(\Sigma^{m(n-1)}\lceil A_i).trj) \triangleleft t, a). \end{aligned}$$

where step 1 is trivial, step 2 follows from the definition of (g, f), and step 3 follows from **P3**. By case 4 of Definition 4.7, $((\Sigma^{m(n-1)}\lceil A_i, \mathcal{I}_i^{m(n-1)}), (\Sigma^n\lceil A_i, \mathcal{I}_i^n)) \in R_{(g_i,f_i)}$.

Case 3.2 Fourth case.

The definition of $R_{(g,f)}$ gives us $\Sigma^n = \Sigma^{n-1} \frown \omega' a\{s'\}$, $\mathcal{I}^{n-1} = (a, t)\mathcal{I}^n$, $f(\Sigma^{n-1}).trj = \omega$, $ltime(\Sigma^{n-1} \frown \omega) \geq t$, $\omega' = \omega \triangleleft t$, and $g(\Sigma^{n-1} \frown \omega', a) = s'$. Distinguish three subcases.

Case 3.2.1 $a \notin acts(A_i)$.

Similar to subcase 3.1.1.

Case 3.2.2 $a \in local(A_i)$.

This situation cannot occur since $a \in in(A)$ (cf. the definition of parallel composition).

Case 3.2.3 $a \in in(A_i)$.

Similar to subcase 3.1.3.

Let $k_0, k_1, k_2, k_3, \ldots$ be the sequence of indices such that $k_0 = 0$ and for each $n > 0$, $\Sigma^{k_n}\lceil A_i$ is either a sentence or is an admissible timed execution. By statements **P1** and **P2**, the sequence $(\Sigma^{k_0}\lceil A_i, \mathcal{I}_i^{k_0}), (\Sigma^{k_1}\lceil A_i, \mathcal{I}_i^{k_1}), (\Sigma^{k_2}\lceil A_i, \mathcal{I}_i^{k_2}), \ldots$ is a prefix of an outcome sequence of (g_i, f_i) given $\Sigma\lceil A_i$ and $\mathcal{I}_i$, possibly with repeated elements. We distinguish the following cases.

1. There exists $n' > 0$ such that $\Sigma^{n'}$ is not finite.

   Then, by definition of $R_{(g,f)}$, there exists a number $n' > 0$ such that $\Sigma^{n'}$ is admissible, and for all $n > n'$, $\Sigma^n = \Sigma^{n'} = \Sigma'$. In particular, the k_j's are infinite, and there exists $n'' > 0$ such that for each $n \geq n''$, $\Sigma^{k_n}\lceil A_i = \Sigma'\lceil A_i$. Thus, $\lim_{n\to\infty}(\Sigma^{k_n}\lceil A_i) = \Sigma'\lceil A_i$. By case 2 of Definition 4.7, for each $n \geq n''$, $((\Sigma^{k_n}\lceil A_i, \mathcal{I}_i^{k_n}), (\Sigma^{k_{n+1}}\lceil A_i, \mathcal{I}_i^{k_{n+1}}) \in R_{(g_i,f_i)}$. Therefore, $\Sigma'\lceil A_i \in \mathcal{O}_{(g_i,f_i)}(\Sigma\lceil A_i, \mathcal{I}_i)$.

2. All the Σ^n's are finite and there are finitely many k_j's.

   Then there is a number n' such that for all $n \geq n'$, $\Sigma^{n'}\lceil A_i$ is not a sentence nor admissible. This means that $\Sigma' - \Sigma^{n'}$ contains no actions from $acts(A_i)$, which implies that $|\Sigma' \upharpoonright acts(A_i)| = |\Sigma^{n'} \upharpoonright acts(A_i)| \neq \infty$ since $\Sigma^{n'}$ is finite. Thus, by hypothesis Σ' is not Zeno. Lemma 4.8 then implies that Σ' is admissible.

   Let $\bar{k}$ be the index of the maximum of the k_j's. Then, for each $n > k_{\bar{k}}$, since $\Sigma^n\lceil A_i$ is not a sentence nor admissible, $\Sigma^n\lceil A_i = (\Sigma^{k_{\bar{k}}}\lceil A_i \frown f_i(\Sigma^{k_{\bar{k}}}\lceil A_i).trj) \triangleleft ltime(\Sigma^n))$. Since Σ' is admissible, $\lim_{n\to\infty} ltime(\Sigma^n) = \infty$ which implies that $ltime(f_i(\Sigma^{k_{\bar{k}}}\lceil A_i).trj) = \infty$. Thus, $f_i(\Sigma^{\bar{k}}) = \omega$ for some admissible trajectory ω. Furthermore, for each $n > \bar{k}$, since $\Sigma' - \Sigma^n$ does not contain any action from $acts(A_i)$, $\mathcal{I}_i^n = \varepsilon$. Thus, the sequence $(\Sigma^{k_0}\lceil A_i, \mathcal{I}_i^{k_0}), (\Sigma^{k_1}\lceil A_i, \mathcal{I}_i^{k_1}), (\Sigma^{k_2}\lceil A_i, \mathcal{I}_i^{k_2}), \ldots (\Sigma^{k_{\bar{k}}}\lceil A_i, \varepsilon), (\Sigma^{k_{\bar{k}}}\lceil A_i \frown \omega, \varepsilon), (\Sigma^{k_{\bar{k}}}\lceil A_i \frown \omega, \varepsilon), (\Sigma^{k_{\bar{k}}}\lceil A_i \frown \omega, \varepsilon), (\Sigma^{k_{\bar{k}}}\lceil A_i \frown \omega, \varepsilon), \ldots$ is an outcome sequence of (g_i, f_i) given $\Sigma\lceil A_i$ and $\mathcal{I}_i$, possibly with repeated elements. The limit of the timed executions of such sequence is $\Sigma^{k_{\bar{k}}}\lceil A_i \frown \omega$, which is given by $\lim_{n\to\infty}(\Sigma^{\bar{k}}\lceil A_i \frown f_i(\Sigma^{\bar{k}}\lceil A_i).trj) \triangleleft ltime(\Sigma^n)$. Since for each $n > \bar{k}$ $\Sigma^n\lceil A_i$ is not a sentence nor admissible, the limit above is the same as $\lim_{n\to\infty} \Sigma^n\lceil A_i$, which in turn is $\Sigma'\lceil A_i$. Thus, $\Sigma'\lceil A_i \in \mathcal{O}_{(g_i,f_i)}(\Sigma\lceil A_i, \mathcal{I}_i)$.

3. All the Σ^n's are finite and there are infinitely many k_j's.

   In this case $(\Sigma_i^{k_n}\lceil A_i, \mathcal{I}_i^n)_{n\geq 0}$ is an outcome sequence of (g_i, f_i) given $\Sigma\lceil A_i$ and $\mathcal{I}_i$, possibly with repeated elements. Then $\Sigma'\lceil A_i = (\lim_{n\to\infty} \Sigma^n)\lceil A_i = \lim_{n\to\infty}(\Sigma^n\lceil A_i)$, which means that $\Sigma'\lceil A_i \in \mathcal{O}_{(g_i,f_i)}(\Sigma\lceil A_i, \mathcal{I}_i)$. ■

Lemma 4.19 *Let (A_0, L_0) and (A_1, L_1) be compatible live timed I/O automata and let (g_i, f_i), $i \in \{0,1\}$, be a receptive strategy for (A_i, L_i). Furthermore, let $(A, L) = (A_0, L_0) \| (A_1, L_1)$. Then $(g, f) = (g_0, f_0) \| (g_1, f_1)$ is a receptive strategy for (A, L).*

Proof. We need to show that $\mathcal{O}_{(g,f)}(\Sigma, \mathcal{I}) \subseteq L \cup \textit{t-exec}^{Zt}(A)$, for all $\Sigma \in \textit{t-exec}^*(A)$ and all timed environment sequences $\mathcal{I}$ for A that are compatible with Σ.

Let $\Sigma \in \textit{t-exec}^*(A)$ be an arbitrary finite timed execution of A and $\mathcal{I}$ be an arbitrary timed environment sequence for A that is compatible with Σ. Since (g_i, f_i) is a receptive strategy for (A_i, L_i), (g_i, f_i) is, by Definition 4.11, a Zeno-tolerant strategy defined on A_i. Let Σ' be an arbitrary element of the outcome $\mathcal{O}_{(g,f)}(\Sigma, \mathcal{I})$. By Lemma 4.8, Σ' is either Zeno or admissible. We distinguish the two cases.

1. Σ' is Zeno.

   By Lemma 4.8, Σ' contains infinitely many actions ($|\Sigma' \upharpoonright \textit{acts}(A)| = \infty$). Assume Σ' is not Zeno-tolerant. Then $|\Sigma' \upharpoonright \textit{local}(A)| = \infty$. Since each locally-controlled action in Σ' belongs to the locally-controlled actions of either A_0 or A_1, there exists an i such that $|\Sigma' \upharpoonright \textit{local}(A_i)| = \infty$, which also implies $|(\Sigma' \lceil A_i) \upharpoonright \textit{local}(A_i)| = \infty$. Thus, Lemma 4.18 is applicable. Lemma 4.18 now implies the existence of a timed sequence $\mathcal{I}_i$ over $\textit{in}(A_i)$ compatible with $\Sigma \lceil A_i$ such that $\Sigma' \lceil A_i \in \mathcal{O}_{(g_i,f_i)}(\Sigma \lceil A_i, \mathcal{I}_i)$. By Lemma 4.5, since Σ' is Zeno, $\textit{ltime}(\Sigma' \lceil A_i)$ is finite. Furthermore, since $|(\Sigma' \lceil A_i) \upharpoonright \textit{local}(A_i)| = \infty$, $\Sigma' \lceil A_i$ is Zeno but not Zeno-tolerant. This contradicts the fact that (g_i, f_i) is Zeno-tolerant. Thus, $\Sigma' \in \textit{t-exec}^{Zt}(A)$.

2. Σ' is admissible.

   By Lemma 4.18, for each $i \in \{0,1\}$ there exists a timed sequence $\mathcal{I}_i$ over $\textit{in}(A_i)$ compatible with $\Sigma \lceil A_i$, such that $\Sigma' \lceil A_i \in \mathcal{O}_{(g_i,f_i)}(\Sigma \lceil A_i, \mathcal{I}_i)$. By Lemma 4.5, $\Sigma' \lceil A_i$ is admissible. Since (g_i, f_i) is a receptive strategy for the pair (A_i, L_i), $\Sigma' \lceil A_i \in L_i$. This implies, by Definition 4.15, that $\Sigma' \in L$. ■

Theorem 4.20 (Closure of parallel composition) *Let (A_0, L_0) and (A_1, L_1) be compatible live timed I/O automata. Then the parallel composition $(A_0, L_0) \| (A_1, L_1)$ is a live timed I/O automaton.*

Proof. Let $(A, L) = (A_0, L_0) \| (A_1, L_1)$. By the definition of parallel composition, A is a safe timed I/O automaton. Furthermore, since $L_i \subseteq \textit{t-exec}^\infty(A_i)$, Lemma 4.5 and Definition 4.15 show that $L \subseteq \textit{t-exec}^\infty(A)$.

For each $i \in \{0,1\}$, let (g_i, f_i) be a receptive strategy for (A_i, L_i). By Lemma 4.19 the strategy $(g, f) = (g_0, f_0) \| (g_1, f_1)$ is a receptive strategy for (A, L). Therefore, the pair (A, L) is receptive. By Definition 4.12 of a live timed I/O automaton, (A, L) is a live timed I/O automaton. ■

4.5 Preorder Relations for Live Timed I/O Automata

For safe timed I/O automata there are several ways of defining a timed trace preorder that depend upon which kinds of traces are being considered. A naive choice would be to consider all the timed traces of a safe timed I/O automaton; however, one might not be interested in, e.g., the Zeno timed traces of a system. For the live preorder, on the other hand, there is a unique natural choice.

Definition 4.21 (Timed trace preorders) Given two live timed I/O automata (A_1, L_1) and (A_2, L_2) such that $esig(A_1) = esig(A_2)$ define the following preorders:

Safe:	$(A_1, L_1) \sqsubseteq_{\mathrm{St}} (A_2, L_2)$	iff	$t\text{-}traces(A_1) \subseteq t\text{-}traces(A_2)$.
Safe-finite:	$(A_1, L_1) \sqsubseteq_{\mathrm{St}}^{*} (A_2, L_2)$	iff	$t\text{-}traces^{*}(A_1) \subseteq t\text{-}traces^{*}(A_2)$.
Safe-admissible:	$(A_1, L_1) \sqsubseteq_{\mathrm{St}}^{\infty} (A_2, L_2)$	iff	$t\text{-}traces^{\infty}(A_1) \subseteq t\text{-}traces^{\infty}(A_2)$.
Safe-non-Zeno:	$(A_1, L_1) \sqsubseteq_{\mathrm{St}}^{\mathrm{nz}} (A_2, L_2)$	iff	$(A_1, L_1) \sqsubseteq_{\mathrm{St}}^{*} (A_2, L_2)$ and $(A_1, L_1) \sqsubseteq_{\mathrm{St}}^{\infty} (A_2, L_2)$.
Live:	$(A_1, L_1) \sqsubseteq_{\mathrm{Lt}} (A_2, L_2)$	iff	$t\text{-}traces(L_1) \subseteq t\text{-}traces(L_2)$. ■

The safe-non-Zeno preorder is the relation that is used in [VL92]. This preorder is used in [VL92] instead of the more natural safe-admissible preorder since finite timed traces are needed for substitutivity of a sequential composition operator.

Note that the live preorder implies the safe preorder whenever the involved safe timed I/O automata have *timed finite internal nondeterminism*. On the other hand, if the involved safe timed I/O automata do not have timed finite internal nondeterminism, then the live preorder only implies finite timed trace inclusion. Essentially, timed finite internal nondeterminism requires that a timed automaton has a finite internal branching structure. In particular, a finite timed trace can lead to at most finitely many states.

Definition 4.22 (Timed finite internal nondeterminism) A timed automaton A has *timed finite internal nondeterminism* (t-FIN) iff, for each trace $\gamma \in t\text{-}traces^{*}(A)$, the set $\{lstate(\Sigma) \mid t\text{-}trace(\Sigma) = \gamma\}$ is finite. ■

Proposition 4.23 *Let (A_1, L_1) and (A_2, L_2) be two live timed I/O automata with $vsig(A_1) = vsig(A_2)$.*

1. *If $(A_1, L_1) \sqsubseteq_{\mathrm{St}}^{\infty} (A_2, L_2)$ then $(A_1, L_1) \sqsubseteq_{\mathrm{St}}^{*} (A_2, L_2)$.*
2. *If A_2 has t-FIN and $(A_1, L_1) \sqsubseteq_{\mathrm{St}}^{*} (A_2, L_2)$ then $(A_1, L_1) \sqsubseteq_{\mathrm{St}} (A_2, L_2)$.*
3. *If $(A_1, L_1) \sqsubseteq_{\mathrm{Lt}} (A_2, L_2)$ then $(A_1, L_1) \sqsubseteq_{\mathrm{St}}^{*} (A_2, L_2)$.*

Proof.

1. Let γ be a finite timed trace of A_1. By definition of timed trace, there is a timed execution Σ_1 of A_1 such that $t\text{-}trace(\Sigma_1) = \gamma$. By definition of live timed I/O automaton there exists an admissible timed execution Σ_1' of A_1 such that $\Sigma_1 \leq_t \Sigma_1'$ and $t\text{-}trace(\Sigma_1') \in L_1$ (just apply any receptive strategy for (A_1, L_1) to Σ_1 and to an admissible timed environment

sequence for A compatible with Σ_1). By definition of live timed I/O automaton, Σ'_1 is a timed execution of A_1. Since $(A_1, L_1) \sqsubseteq_{\mathrm{St}}^{\infty} (A_2, L_2)$, there exists a timed execution Σ'_2 of A_2 such that $t\text{-}trace(\Sigma'_1) = t\text{-}trace(\Sigma'_2)$. Since the set of timed executions of a timed I/O automaton is closed under prefix, there is a prefix Σ_2 of Σ'_2 such that Σ_2 is a timed execution of A_2 and $t\text{-}trace(\Sigma_2) = \gamma$, i.e., γ is a timed trace of A_2.

2. This is a standard result that appears in [LV91].

3. Similar to the proof of Proposition 3.19, part 1. Use timed executions and timed traces instead of executions and traces, respectively. ■

The important property of the safe and live preorders is that they are substitutive for parallel composition. This means that an implementation of a system made up of several parallel components can be obtained by implementing each component separately.

Theorem 4.24 (Substitutivity) *Let $(A_i, L_i), (A'_i, L'_i)$, $i \in \{0, 1\}$ be live timed I/O automata, and let $\sqsubseteq_X$ be one relation among $\sqsubseteq_{\mathrm{St}}$, $\sqsubseteq_{\mathrm{St}}^{*}$, $\sqsubseteq_{\mathrm{St}}^{\infty}$, $\sqsubseteq_{\mathrm{St}}^{\mathrm{nz}}$ and $\sqsubseteq_{\mathrm{Lt}}$. If $(A_0, L_0), (A_1, L_1)$ are compatible, $(A'_0, L'_0), \ldots, (A'_1, L'_1)$ are compatible, and, for each i, $(A_i, L_i) \sqsubseteq_X (A'_i, L'_i)$, then $(A_0, L_0)\| \cdots \|(A_1, L_1) \sqsubseteq_X (A'_0, L'_0)\| \cdots \|(A'_1, L'_1)$.*

Proof. The substitutivity result is a direct consequence of Lemma 4.5 and the observation, analogous to the one of the untimed model, that parallel composition of timed execution sets preserve timed trace equivalence. ■

It is well known that simulation based proof techniques [LV91, LV95] can be used for implementation notions based on trace inclusion. In [GSSL93] simulation based proof techniques are extended to live preorder, and in [SLL93b] the new proof techniques are used to verify nontrivial communication protocols.

4.6 Comparison with Other Timed Models

This section compares our timed model with the work of [AL91b, MMT91, VL92].

The formalism that is used in [AL91b] is the Temporal Logic of Actions (TLA) [Lam91] extended with a new variable *now* that models time. A specification S consists of the conjunction of three formulas $Init \wedge \Pi \wedge L$ where $Init$ represents the initial configurations of S, Π is a *safety property*, and L is a *liveness property*. The subformula $Init \wedge \Pi$ corresponds to our safe timed I/O automata, while the subformula L corresponds to our timed liveness conditions. In [AL91b] L can also be satisfied by finite or Zeno executions or by executions that do not satisfy $Init \wedge \Pi$. The formula L is a liveness condition for $Init \wedge \Pi$ based on our definition iff the pair $(Init \wedge \Pi, L)$ is machine-closed based on the definition in [AL91b].

There is a special formula NZ in [AL91b] that is used to express non-Zenoness, i.e., that time advances forever. Time blocking or Zeno behaviors are undesirable in [AL91b] as well as in our model; however, it is possible for the safety part of a specification to describe systems for which time cannot advance past a given upper bound whenever a particular state is reached. Such a situation is eliminated in [AL91b] by requiring the pair (Π, NZ) to be machine-closed. In our model, on the other hand, the same situation is eliminated by the fact that system-Zeno executions are not allowed in the liveness part of a live timed I/O automaton and that a live timed I/O automaton is machine-closed by definition.

A major difference between our notion of receptiveness and the notion of receptiveness of [AL91b] is in the role of time: in our model no one is allowed to have control over time; in [AL91b] either the system or its environment must have control over time. We believe that it is more reasonable to assume that no one has control over time.

The model of [MMT91] is an extension to the timed model of the I/O automaton model of [LT87]. The locally-controlled actions of an automaton are partitioned into classes, each one of which is associated with a lower bound (possibly 0 but not ∞) and an upper bound (possibly ∞ but not 0). Actions from one class with lower bound c_1 and upper bound c_2 must stay enabled for at least c_1 time units before one of them can be performed, and cannot stay enabled more than c_2 time units without any one of them being performed.

An automaton M of [MMT91] can be represented in our model as a pair (A, L) where A is a safe timed I/O automaton with a transition relation that satisfies all the timing constraints of M, and L is the set of all admissible executions of A. It is easy to check that (A, L) is receptive and that admissible timed trace inclusion in [MMT91] coincides with live trace inclusion in our model. However, there are liveness conditions that can be represented in our model but cannot be represented naturally in the model of [MMT91].

The work in [VL92] does not deal with general liveness properties, and argues that finite and admissible timed traces inclusion is generally sufficient to express a useful notion of implementation whenever time is involved. The work in [SLL93b], however, has shown that liveness is useful even in a timed model. In general, the automata of [VL92] are not receptive; however, in order to avoid trivial implementations, [VL92] assumes some form of I/O distinction and some form of receptiveness at the lower level of implementation. There is a very close connection between the technical definitions of *I/O feasibility* and *strong I/O feasibility* of [VL92] and our notion of receptiveness. It is possible to represent each timed I/O automaton A of [VL92] with the pair (A, L) where L is the set of admissible executions of A. The notion of I/O feasibility of [VL92], which requires each finite timed execution of A to be extendible to an admissible timed execution of A using locally-controlled actions only, is stronger than requiring that L is a liveness condition for A and weaker than requiring that (A, L) is a live timed I/O automaton. In order to have closure under parallel composition, [VL92] introduces a stronger requirement on I/O automata called strong I/O feasibility. Strong I/O feasibility adds to I/O feasibility the requirement that the safe part of an I/O automaton A does not exhibit any system-Zeno execution. However, receptiveness, which is weaker than strong I/O feasibility since the safe part of a live timed I/O automaton is allowed to exhibit system-Zeno behaviors, is sufficient to guarantee closure under parallel composition and hence substitutivity.

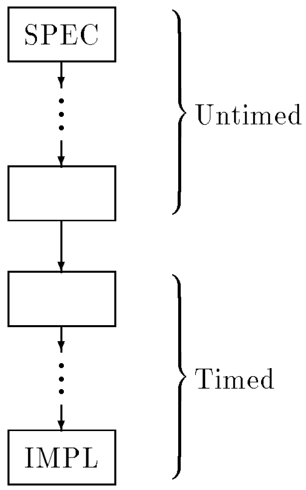


Figure 1: A stepwise development from an untimed specification to a timed implementation.

5 Embedding the Untimed Model in the Timed Model

The untimed model, presented in Section 3, is used to specify systems where the amount of time that passes between actions is considered unimportant. Many problems in distributed computing can be stated and solved using this model. However, it is not possible to state anything about, e.g., response times in the untimed model. It is implicitly assumed that the final implementation on a physical machine is "fast enough" for practical use.

An untimed system can be thought of as a timed system that allows arbitrary time-passage. This indicates that the timed model is, in some sense, more general than the untimed model, and that one could use the timed model in situations where one would usually use the untimed model. However, the timed model is more complicated than the untimed model; furthermore, it does not seem natural to be required to deal with time, when the problem to be solved does not mention time.

Thus, one would like to work in the untimed model as much as possible and only switch to the timed model when it is needed. Sometimes, however, an algorithm that uses time implements a specification that does not use time. For example, [SLL93a] shows how an untimed specification (of the at-most-once message delivery problem) is implemented by a system that assumes upper time bounds on certain process steps and channel delays. Fischer's mutual exclusion algorithm [Fis85, AL91b] is another such example. Figure 1 depicts the stepwise development one would use for an implementation proof like the one in [SLL93a]. The stepwise development in Figure 1, however, raises the issue of what it means to implement an untimed specification with a timed implementation. Our approach to this issue is to convert the untimed systems in the stepwise development to timed systems by applying a *patient* operator that adds arbitrary time-passage steps. The patient operator we use is similar to the one of [NS92, VL92]. To complement the *patient* operator, this section proves the *Embedding Theorem* which states that a concrete level implements an abstract level in the untimed model if and only if the *patient* version of the concrete level implements the *patient* version of the abstract level in the timed model. Thus, the first part of the stepwise development of Figure 1 can be carried out entirely in the simpler untimed model, and the last part in the timed model. In the intermediate development step which goes from untimed to timed, one must prove that the timed level implements the *patient* version of the untimed level. The embedding theorem can then be applied to show that the implementation IMPL implements the *patient* version of the specification SPEC.

Definition 5.1 (Patient operator on safe I/O automata) Let A be a safe (untimed) I/O automaton such that $\{\nu(t) \mid t \in \mathbb{R}^{>0}\} \cap acts(A) = \emptyset$. Then define $patient(A)$ to be the safe timed I/O automaton with

- $states(patient(A)) = states(A)$.
- $start(patient(A)) = start(A)$.
- $ext(patient(A)) = ext(A) \cup \{\nu(t) \mid t \in \mathbb{R}^{>0}\}$.
- $(in(patient(A)), out(patient(A)), int(patient(A))) = (in(A), out(A), int(A))$.
- $steps(patient(A)) = steps(A) \cup \{(s, \nu(t), s) \mid t \in \mathbb{R}^{>0}\}$. ■

The following lemma states a simple but important property of the patient operator. That is, the state of an automaton $patient(A)$ does not change during any trajectory.

Lemma 5.2 *Let A be a safe I/O automaton such that $\{\nu(t) \mid t \in \mathbb{R}^{>0}\} \cap acts(A) = \emptyset$, and let $\Sigma = \omega_0 a_1 \omega_1 a_2 \omega_2 \cdots$ be a timed execution of $patient(A)$. Then, for all i, $|rng(\omega_i)| = 1$.* ■

In order to state what it means to apply the *patient* operator to a live I/O automaton, the following auxiliary definition of what it means to *untime* a timed execution is needed.

Definition 5.3 *Let A be a safe I/O automaton with such that $\{\nu(t) \mid t \in \mathbb{R}^{>0}\} \cap acts(A) = \emptyset$, and let $\Sigma = \omega_0 a_1 \omega_1 a_2 \omega_2 \cdots$ be a timed execution of $patient(A)$. Then define*

$$untime(\Sigma) = fstate(\omega_0) a_1 fstate(\omega_1) a_2 fstate(\omega_2) \cdots$$

Similarly, let $\gamma = ((a_1, t_1)(a_2, t_2) \cdots, t)$ be a timed trace of $patient(A)$. Then define

$$untime(\gamma) = a_1 a_2 \cdots$$ ■

Lemma 5.4 *Let A be a safe I/O automaton such that $\{\nu(t) \mid t \in \mathbb{R}^{>0}\} \cap acts(A) = \emptyset$. Then $\Sigma \in t\text{-}exec(patient(A))$ iff $untime(\Sigma) \in exec(A)$. Furthermore, if Σ is finite, then $untime(\Sigma)$ is finite.*

Proof. Follows trivially from Lemma 5.2, Definition 5.1, and the definition of *untime*. ∎

The patient operator can now be extended to live I/O automata. For any live I/O automaton (A, L), the patient live I/O automaton of (A, L) should be the live timed I/O automaton whose safety part is $patient(A)$ and whose liveness part consists of all those admissible executions that, when made *untime*d, are in L. Thus, the liveness condition of the patient live I/O automaton allows time to pass arbitrarily, as long as the liveness prescribed by L is satisfied.

Definition 5.5 (Patient operator on live I/O automaton) Let (A, L) be a live I/O automaton such that $\{\nu(t) \mid t \in \mathbb{R}^{>0}\} \cap acts(A) = \emptyset$. The *patient live I/O automaton* of (A, L), denoted by $patient(A, L)$, is the pair $(patient(A), patient_A(L))$, where $patient_A(L)$ is the set $\{\Sigma \in t\text{-}exec^{\infty}(patient(A)) \mid untime(\Sigma) \in L\}$. ∎

We prove that for any live I/O automaton (A, L), $patient(A, L)$ is a live timed I/O automaton. This means showing the existence of a receptive strategy for the pair $(patient(A), patient_A(L))$. This is accomplished by defining the *patient strategy* of an (untimed) strategy (g, f) defined on A, and showing that the patient strategy of (g, f) is receptive for (A_p, L_p), where $(A_p, L_p) = patient(A, L)$, if (g, f) is receptive for (A, L). To ensure that the patient strategy of (g, f) is Zeno-tolerant, which is required for receptiveness, the patient strategy of (g, f) insists on letting time pass for at least δ time units between local steps.

Definition 5.6 For any safe timed I/O automaton A and any finite timed execution Σ of A, define $lloctime(\Sigma)$ to be the time of occurrence of the last locally-controlled action in Σ, or 0 if no such action exists. Formally, let $\Sigma = \omega_0 a_1 \omega_1 \cdots a_n \omega_n$. If $a_1, \ldots, a_n \notin local(A)$, then define $lloctime(\Sigma) = 0$; otherwise, define $lloctime(\Sigma) = ltime(\omega_0 a_1 \omega_1 \cdots a_k \omega_k)$ where $a_k \in local(A)$ and $a_{k+1}, \ldots, a_n \notin local(A)$.

Given a positive real number δ, let $nloctime_\delta(\Sigma)$ denote $max(0, lloctime(\Sigma) + \delta - ltime(\Sigma))$. That is, $nloctime_\delta(\Sigma)$ is the minimum time that must elapse after Σ before performing any local action so that the minimum distance δ between any two local actions is preserved. ∎

Definition 5.7 (Patient strategy) Let A be a safe I/O automaton such that $\{\nu(t) \mid t \in \mathbb{R}^{>0}\} \cap acts(A) = \emptyset$, and let (g, f) be an (untimed) strategy defined on A. Furthermore, let $A_p = patient(A)$. Then define the *patient strategy* of (g, f) with respect to some positive real number δ, written $patient_\delta(g, f)$, to be the pair of functions

$$g_p : t\text{-}exec^*(A_p) \times in(A_p) \to states(A_p)$$

$$f_p : t\text{-}exec^*(A_p) \to (traj(A_p) \times local(A_p) \times states(A_p)) \cup traj(A_p)$$

defined as follows:

$$g_p(\Sigma, a) \triangleq g(untime(\Sigma), a)$$

$$f_p(\Sigma) \triangleq \begin{cases} (\omega, a, s) & \text{if } f(untime(\Sigma)) = (a, s), \text{ where } rng(\omega) = \{lstate(\Sigma)\} \text{ and} \\ & \quad ltime(\omega) = nloctime_\delta(\Sigma) \\ \omega & \text{if } f(untime(\Sigma)) = \bot, \text{ where } rng(\omega) = \{lstate(\Sigma)\} \text{ and} \\ & \quad ltime(\omega) = \infty. \end{cases}$$ ∎

For a finite timed execution Σ of A_p, Lemma 5.4 implies that $untime(\Sigma)$ is a finite execution of A. Also, by Definition 5.1, A and A_p have the same input, output, and internal actions. Thus, in the definition of (g_p, f_p), the domains and ranges of g and f are compatible with the usage of g and f. The following lemma states that the patient strategy is indeed a strategy.

Lemma 5.8 *Let A be a safe I/O automaton such that $\{\nu(t) \mid t \in \mathbb{R}^{>0}\} \cap acts(A) = \emptyset$, and let (g, f) be an (untimed) strategy defined on A, and δ be any positive real number. Then $patient_\delta(g, f)$ is a (timed) strategy defined on $patient(A)$.*

Proof. Let $(g_p, f_p) = patient_\delta(g, f)$ and $A_p = patient(A)$. We verify that (g_p, f_p) satisfies the four conditions of Definition 4.6.

1. Let $\Sigma \in t\text{-}exec^*(A_p)$ and $a \in in(A_p)$. Let s denote, $g_p(\Sigma, a)$. By the definition of g_p and the fact that (g, f) is a strategy defined on A (cf. Definition 3.6), $(lstate(untime(\Sigma)), a, s) \in steps(A)$. By the definition of $untime$ and Lemma 5.2, $lstate(untime(\Sigma)) = lstate(\Sigma)$. Thus, $(lstate(\Sigma), a, s) \in steps(A)$. By Definition 5.1, $(lstate(\Sigma), a, s) \in steps(A_p)$.

2. Let $\Sigma \in t\text{-}exec^*(A_p)$ and let (ω, a, s) denote $f_p(\Sigma)$. Similar to the first condition, it is easy to see that $(lstate(\omega), a, s)$ is a step of A_p. Then, by the definition of ω and the fact that A_p allows time to pass arbitrarily, $\omega a\{s\}$ is a timed execution fragment of A_p and $fstate(\omega) = lstate(\Sigma)$. Thus, $\Sigma \frown \omega a\{s\} \in t\text{-}exec^*(A_p)$.

3. The argument parallels that for Condition 2.

4. We consider only the case where $f_p(\Sigma) = (\omega, a, s)$, and we leave to the reader the similar and simpler case where $f_p(\Sigma) = \omega$.

   Let $t \leq ltime(\omega)$. By definition of f_p, $f(untime(\Sigma)) = (a, s)$, $ltime(\omega) = nloctime_\delta(\Sigma)$ and $rng(\omega) = \{lstate(\Sigma)\}$. By definition of $untime$, $untime(\Sigma \frown (\omega \triangleleft t)) = untime(\Sigma)$, which implies $f(untime(\Sigma \frown (\omega \triangleleft t))) = f(untime(\Sigma))$. Thus, $f_p(\Sigma \frown (\omega \triangleleft t)) = (\omega', a, s)$, where $ltime(\omega') = nloctime_\delta(\Sigma \frown (\omega \triangleleft t))$ and $rng(\omega') = \{lstate(\Sigma)\}$. We need to show that $ltime(\omega') = ltime(\omega) - t$, i.e., that $nloctime_\delta(\Sigma \frown (\omega \triangleleft t)) = nloctime_\delta(\Sigma) - t$. Observe that, since ω does not contain any action, $lloctime(\Sigma \frown (\omega \triangleleft t)) = lloctime(\Sigma)$. Then

$$\begin{aligned} nloctime_\delta(\Sigma \frown (\omega \triangleleft t)) &\overset{1}{=} lloctime(\Sigma \frown (\omega \triangleleft t)) + \delta - ltime(\Sigma \frown (\omega \triangleleft t)) \\ &\overset{2}{=} lloctime(\Sigma) + \delta - ltime(\Sigma) - t \\ &\overset{3}{=} nloctime_\delta(\Sigma) - t \end{aligned}$$

   where steps 1 and 3 follow from the definition of $nloctime_\delta()$, and step 2 follows from $lloctime(\Sigma \frown (\omega \triangleleft t)) = lloctime(\Sigma)$ and from $ltime(\Sigma \frown (\omega \triangleleft t)) = ltime(\Sigma) + t$. ■

The proof that for any receptive (untimed) strategy (g, f) for a live I/O automaton (A, L), and any positive δ, the patient strategy $patient_\delta(g, f)$ is a receptive (timed) strategy for (A_p, L_p), where $(A_p, L_p) = patient(A, L)$, is based on two technical lemmas. The first of these lemmas states that if Σ' is an *admissible* timed execution of an outcome of $patient_\delta(g, f)$, then

$untime(\Sigma')$ is an outcome of (g, f). This expresses the intuitive idea that the only significant difference between (g, f) and $patient(g, f)$ is due to time-passage. The second lemma states that the difference in the time of occurrence of any two *locally-controlled* actions in a timed execution of an outcome of $patient_\delta(g, f)$, is at least δ. This is, of course, due to the fact that $patient_\delta(g, f)$ insists on letting time pass for at least δ time units between local steps.

Lemma 5.9 *Let A be a safe I/O automaton such that $\{\nu(t) \mid t \in \mathbb{R}^{>0}\} \cap acts(A) = \emptyset$, and let (g, f) be an (untimed) strategy defined on A. Let $A_p = patient(A)$ and $(g_p, f_p) = patient_\delta(g, f)$ for some arbitrary positive real number δ. Then, for all $\Sigma \in t\text{-}exec^*(A_p)$, all timed environment sequences $\mathcal{I}_p$ for A_p compatible with Σ, and all admissible $\Sigma' \in \mathcal{O}_{(g_p,f_p)}(\Sigma, \mathcal{I}_p)$, there exists an environment sequence $\mathcal{I}$ for A such that $untime(\Sigma') = \mathcal{O}_{(g,f)}(untime(\Sigma), \mathcal{I})$.*

Proof. Let $\Sigma \in t\text{-}exec^*(A_p)$ be an arbitrary finite timed execution of A, $\mathcal{I}_p$ an arbitrary timed environment sequence for A_p compatible with Σ, and Σ' be an arbitrary admissible timed execution of the outcome $\mathcal{O}_{(g_p,f_p)}(\Sigma, \mathcal{I}_p)$. Let $R_{(g_p,f_p)}$ be the next-relation induced by (g_p, f_p) and $R_{(g,f)}$ be the next-function induced by (g, f). Also, let $(\Sigma^n, \mathcal{I}_p^n)_{n\geq 0}$ be an outcome sequence of (g_p, f_p) given Σ and $\mathcal{I}_p$ such that $\Sigma' = \lim_{n\to\infty} \Sigma^n$. We first define a sequence $\mathcal{I}$ as $\mathcal{I}^1\mathcal{I}^2\cdots$, and for each $n > 0$ we prove the following:

P1 For each environment sequence $\mathcal{I}'$ for A, $(untime(\Sigma^n), \mathcal{I}') \in R^*_{(g,f)}(untime(\Sigma^{n-1}), \mathcal{I}^n, \mathcal{I}')$.

In the rest of the proof we let $\mathcal{I}'$ denote a generic environment sequence for A. We distinguish the five cases that appear in the definition of $R_{(g_p,f_p)}$.

Case 1 Define $\mathcal{I}^n = \lambda$. Here $(\Sigma^n, \mathcal{I}_p^n) = (\Sigma^{n-1} \frown \omega a\{s\}, \mathcal{I}_p^{n-1})$ with $f_p(\Sigma^{n-1}) = (\omega, a, s)$. Then, by definition of (g_p, f_p), $f(untime(\Sigma^{n-1})) = (a, s)$. Observe that $untime(\Sigma^n) = untime(\Sigma^{n-1}) \frown untime(\omega a\{s\})$. Since $rng(w) = \{lstate(\Sigma^{n-1})\}$, $\Sigma^n = \Sigma^{n-1} \frown lstate(\Sigma^{n-1})\, a\, s$. Thus, $(untime(\Sigma^n), \mathcal{I}') \in R^*_{(g,f)}(untime(\Sigma^{n-1}), \mathcal{I}^n, \mathcal{I}')$.

Case 2 Define $\mathcal{I}^n = \varepsilon$. Here $\Sigma^n = \Sigma^{n-1} \frown \omega$ where $\omega = f_p(\Sigma^{n-1})$. Furthermore, $untime(\Sigma^n) = untime(\Sigma^{n-1} \frown \omega) = untime(\Sigma^{n-1})$. Thus, $(untime(\Sigma^n), \mathcal{I}') \in R^*_{(g,f)}(untime(\Sigma^{n-1}), \mathcal{I}^n, \mathcal{I}')$.

Case 3 This case is handled in the same way as case 1.

Case 4 Let $(a, t) = head(\mathcal{I}_p^{n-1})$. Then, $(\Sigma^n, \mathcal{I}_p^n) = (\Sigma^{n-1} \frown \omega a\{s\}, tail(\mathcal{I}_p^{n-1}))$, where $\omega = (f_p(\Sigma^{n-1}).trj) \triangleleft (t - ltime(\Sigma^{n-1}))$ and $s = g_p(\Sigma^{n-1} \frown \omega, a)$. By the definition of (g_p, f_p), since $rng(w) = \{lstate(\Sigma^{n-1})\}$, $g(untime(\Sigma^{n-1} \frown \omega), a) = g(untime(\Sigma^{n-1}), a)$. We distinguish two cases.

Case 4.1 $f_p(\Sigma^{n-1}) = \omega$.
Define $\mathcal{I}^n = \lambda a$. By the definition of f_p, $f(untime(\Sigma^{n-1})) = \perp$. Thus, by case 2 of the definition of $R_{(g,f)}$, $(untime(\Sigma^{n-1}), a\mathcal{I}') = R_{(g,f)}(untime(\Sigma^{n-1}), \lambda a\mathcal{I}')$. By the definition of g_p, $g(untime(\Sigma^{n-1}), a) = s$. By case 3 of the definition of $R_{(g,f)}$, $(untime(\Sigma^n), \mathcal{I}') = R_{(g,f)}(untime(\Sigma^{n-1}), a\mathcal{I}')$. This means that $(untime(\Sigma^n), \mathcal{I}') \in R^*_{(g,f)}(untime(\Sigma^{n-1}), \mathcal{I}^n, \mathcal{I}')$.

Case 4.2 $f_p(\Sigma^{n-1}) = (\omega, b, s)$.

Define $\mathcal{I}^n = a$. By the definition of g_p, $g(untime(\Sigma^{n-1}), a) = s$. By case 3 of the definition of $R_{(g,f)}$, $(untime(\Sigma^n), \mathcal{I}') \in R^*_{(g,f)}(untime(\Sigma^{n-1}), \mathcal{I}^n, \mathcal{I}')$.

Case 5 Define $\mathcal{I}^n = \lambda$. In this case $(\Sigma^n, \mathcal{I}^n_p) = (\Sigma^{n-1}, \mathcal{I}^{n-1}_p)$. By definition of $(\Sigma^n)_{n\geq 0}$, there exists an $n' < n$ such that $\Sigma^{n'}$ is finite, $f_p(\Sigma^{n'}) = \omega$, for some admissible trajectory ω, and $\Sigma^{n'+1} = \Sigma^{n'+2} = \cdots = \Sigma^{n-1} = \Sigma^n = \Sigma^{n'} \frown \omega$. Then, by the definition of f_p, $f(untime(\Sigma^{n'})) = \bot$. Since $untime(\Sigma^{n'}) = untime(\Sigma^{n-1})$, $f(untime(\Sigma^{n-1})) = \bot$. This implies that $(untime(\Sigma^n), \mathcal{I}') \in R^*_{(g,f)}(untime(\Sigma^{n-1}), \mathcal{I}^n, \mathcal{I}')$.

We now argue that $\mathcal{I}$ is an environment sequence for A. It is immediate to observe that each element of $\mathcal{I}$ is either λ or an input action of A. Suppose by contradiction that $\mathcal{I}$ does not contain infinitely many occurrences of λ. Then there exists a number n' such that for all $n > n'$, the definition of $\mathcal{I}^{n'}$ is handled by case 4.2 above (case 2 occurs at most once). Let, for each $n \geq n'$, $f_p(\Sigma^n) = (\omega^n, a^n, s^n)$. Then by definition of f_p, $ltime(\omega^n) = max(0, lloctime(\Sigma^n) + \delta - ltime(\Sigma^n))$ which, since case 4.2 adds only input actions, equals $max(0, lloctime(\Sigma^{n'}) + \delta - ltime(\Sigma^n))$.

We show by induction that for each $n \geq n'$, $ltime(\Sigma^n) \leq ltime(\Sigma^{n'}) + \delta$. The case for $n = n'$ is trivial. For the inductive step suppose by induction that $ltime(\Sigma^{n-1}) \leq ltime(\Sigma^{n'}) + \delta$. We have shown already that $ltime(\omega^{n-1}) = max(0, lloctime(\Sigma^{n'}) + \delta - ltime(\Sigma^{n-1}))$. If $ltime(\omega^{n-1}) = 0$, then $ltime(\Sigma^{n-1} \frown \omega^{n-1}) = ltime(\Sigma^{n-1}) \leq ltime(\Sigma^{n'}) + \delta$, where the last step follows by induction; if $ltime(\omega^{n-1}) = lloctime(\Sigma^{n'}) + \delta - ltime(\Sigma^{n-1})$, then $ltime(\Sigma^{n-1} \frown \omega^{n-1}) = ltime(\Sigma^{n-1}) + lloctime(\Sigma^{n'}) + \delta - ltime(\Sigma^{n-1}) = lloctime(\Sigma^{n'}) + \delta$. Thus, in both cases $ltime(\Sigma^{n-1} \frown \omega^{n-1}) \leq ltime(\Sigma^{n'}) + \delta$. Since, by definition of $R_{(g_p,f_p)}$, $ltime(\Sigma^n) \leq ltime(\Sigma^{n-1} \frown \omega^{n-1})$, $ltime(\Sigma^n) \leq ltime(\Sigma^{n'}) + \delta$.

Since $\Sigma' = \lim_{n\to\infty} \Sigma^n$, $ltime(\Sigma') \leq ltime(\Sigma^{n'}) + \delta$. Since $\Sigma^{n'}$ is finite, we contradict the hypothesis that Σ' is admissible. Therefore, $\mathcal{I}$ contains infinitely many occurrences of λ.

From the construction above, $\mathcal{O}_{(g,f)}(untime(\Sigma), \mathcal{I}) = \lim_{n\to\infty} untime(\Sigma^n)$. By the continuity of the untiming operator, $\lim_{n\to\infty} untime(\Sigma^n) = untime(\lim_{n\to\infty} \Sigma^n)$. Thus, $untime(\Sigma') = \mathcal{O}_{(g,f)}(untime(\Sigma), \mathcal{I})$. ■

Lemma 5.10 *Let A be a safe I/O automaton such that $\{\nu(t) \mid t \in \mathbb{R}^{>0}\} \cap acts(A) = \emptyset$, and let (g, f) be an (untimed) strategy defined on A. Let $A_p = patient(A)$ and $(g_p, f_p) = patient_\delta(g, f)$ for some arbitrary positive real number δ. Let $\Sigma \in t\text{-}exec^*(A_p)$ be an arbitrary finite timed execution of A_p, $\mathcal{I}$ an arbitrary timed environment sequence for A_p compatible with Σ, and Σ' an arbitrary timed execution of the outcome $\mathcal{O}_{(g_p,f_p)}(\Sigma, \mathcal{I})$. Then for any two elements (a_1, t_1) and (a_2, t_2) in $t\text{-}seq(\Sigma' - \Sigma) \upharpoonright (local(A_p) \times \mathsf{T})$, $|t_2 - t_1| \geq \delta$.*

Proof. Let (a_1, t_1) and (a_2, t_2) be two arbitrary pairs in $\gamma = t\text{-}seq(\Sigma' - \Sigma) \upharpoonright (local(A_p) \times \mathsf{T})$ and assume, without loss of generality, that (a_1, t_1) occurs before (a_2, t_2) in γ. This implies that $t_2 \geq t_1$. Furthermore, assume, again without loss of generality, that (a_1, t_1) and (a_2, t_2) are consecutive in γ. Let $(\Sigma^n, \mathcal{I}^n)_{n\geq 0}$ be an outcome sequence of (g_p, f_p) given Σ and $\mathcal{I}$ such that $\Sigma' = \lim_{n\to\infty} \Sigma^n$.

Definition 4.7 now implies the existence of a number n such that (a_2, t_2) is not in $t\text{-}seq(\Sigma^n - \Sigma) \upharpoonright (local(A_p) \times \mathsf{T})$ and $\Sigma^{n+1} = \Sigma^n \frown \omega a_2\{s\}$ with $f_p(\Sigma^n) = (\omega, a_2, s)$ and $ltime(\Sigma^n \frown \omega) = t_2$. Also, (a_1, t_1) must be in $t\text{-}seq(\Sigma^n - \Sigma) \upharpoonright (local(A_p) \times \mathsf{T})$ since otherwise it could not occur before (a_2, t_2) in γ. Let $t_l = lloctime(\Sigma^n)$. Since $a_1 \in local(A_p)$, $t_1 \leq t_l$.

By definition of f_p (Definition 5.7), $ltime(\omega) = max(0, lloctime(\Sigma^n) + \delta - ltime(\Sigma^n))$. Thus, $t_2 = ltime(\Sigma^n \frown \omega) \geq t_l + \delta \geq t_1 + \delta$, or equivalently, $t_2 - t_1 \geq \delta$. That suffices. ∎

It is now possible to prove that for any receptive strategy (g, f) for a live I/O automaton (A, L) and any positive δ, $patient_\delta(g, f)$ is a receptive (timed) strategy for (A_p, L_p), where $(A_p, L_p) = patient(A, L)$.

Lemma 5.11 *Let (A, L) be a live I/O automaton such that $\{\nu(t) \mid t \in \mathbb{R}^{>0}\} \cap acts(A) = \emptyset$, and let (g, f) be an (untimed) receptive strategy for (A, L). Furthermore, let $(A_p, L_p) = patient(A, L)$. Then, for any positive real number δ, $patient_\delta(g, f)$ is a (timed) receptive strategy for (A_p, L_p).*

Proof. Let δ be an arbitrary positive real number and let $(g_p, f_p) = patient_\delta(g, f)$. Note that by Lemma 5.8 (g_p, f_p) is a (timed) strategy defined on A_p. By Definition 4.11 we need to show that $\mathcal{O}_{(g_p, f_p)}(\Sigma, \mathcal{I}_p) \subseteq L_p \cup t\text{-}exec^{Zt}(A_p)$, for all $\Sigma \in t\text{-}exec^*(A_p)$ and all timed environment sequences $\mathcal{I}_p$ for A_p compatible with Σ.

Let $\Sigma \in t\text{-}exec^*(A_p)$ be an arbitrary finite timed execution of A_p and $\mathcal{I}_p$ be an arbitrary timed environment sequence for A_p compatible with Σ. Let $\Sigma' \in \mathcal{O}_{(g_p, f_p)}(\Sigma, \mathcal{I}_p)$ be an arbitrary element of the outcome. By Lemma 4.8, Σ' is either Zeno or admissible. We distinguish the two cases.

1. Σ' is Zeno.

   Then, by Lemma 5.10 there are only finitely many locally-controlled actions of A_p in Σ'. By Lemma 4.8, Σ' contains infinitely many input actions. Thus, $\Sigma \in t\text{-}exec^{Zt}(A_p)$.

2. Σ' is admissible.

   By Lemma 5.9 there exists an environment sequence $\mathcal{I}$ for A such that $untime(\Sigma') = \mathcal{O}_{(g,f)}(untime(\Sigma), \mathcal{I})$. Since (g, f) is a receptive strategy for (A, L), $untime(\Sigma') \in L$. Thus, by Definition 5.5, $\Sigma' \in L_p$. ∎

Finally, we can prove that for any live I/O automaton (A, L), $patient(A, L)$ is a live timed I/O automaton.

Theorem 5.12 *Let (A, L) be a live I/O automaton. Then $patient(A, L)$ is a live timed I/O automaton.*

Proof. Let $(A_p, L_p) = patient(A, L)$. Definition 5.1 implies that A_p is a safe timed I/O automaton. Furthermore, $L \subseteq t\text{-}exec^\infty(A_p)$ by Definition 5.5. Finally, Lemma 5.11 implies that the pair (A_p, L_p) is receptive. By Definition 4.12, this suffices. ∎

Now attention is turned to proving the Embedding Theorem, which states that the safe and live preorders of live I/O automata are preserved by the patient operator. A few simple preliminary lemmas are needed.

Lemma 5.13 *Let A be a safe I/O automaton such that $\{\nu(t) \mid t \in \mathbb{R}^{>0}\} \cap acts(A) = \emptyset$, and let $A_p = patient(A)$. Furthermore, let $\Sigma \in t\text{-}exec(A_p)$. Then,*

$$untime(t\text{-}trace_{A_p}(\Sigma)) = trace_A(untime(\Sigma)).$$ ■

Lemma 5.14 *Let (A, L) be a live I/O automaton such that $\{\nu(t) \mid t \in \mathbb{R}^{>0}\} \cap acts(A) = \emptyset$. Then,*

1. *If $\gamma \in t\text{-}traces(patient(A))$ then $untime(\gamma) \in traces(A)$.*
2. *If $\beta \in traces(A)$ and $\gamma \in tsp(ext(A))$ with $\beta = untime(\gamma)$ such that if $seq(\gamma)$ is Zeno, then $ltime(\gamma)$ is the limit of the times in $seq(\gamma)$, then $\gamma \in t\text{-}traces(patient(A))$.*
3. *If $\gamma \in t\text{-}traces(patient_A(L))$ then $untime(\gamma) \in traces(L)$.*
4. *If $\beta \in traces(L)$ and $\gamma \in tsp(ext(A))$ is admissible with $\beta = untime(\gamma)$, then $\gamma \in t\text{-}traces(patient_A(L))$.* ■

Theorem 5.15 (Embedding Theorem) *Let (A, L) and (B, M) be live I/O automata such that $\{\nu(t) \mid t \in \mathbb{R}^{>0}\} \cap (acts(A) \cup acts(B)) = \emptyset$. Then*

1. *$(A, L) \sqsubseteq_{\mathrm{S}} (B, M)$ iff $patient(A, L) \sqsubseteq_{\mathrm{St}} patient(B, M)$.*
2. *$(A, L) \sqsubseteq_{\mathrm{L}} (B, M)$ iff $patient(A, L) \sqsubseteq_{\mathrm{Lt}} patient(B, M)$.*

Proof. Let $(A_p, L_p) = patient(A, L)$ and $(B_p, M_p) = patient(B, M)$. The two parts of the lemma are considered separately.

1. $\Longrightarrow$: Let $\gamma \in t\text{-}traces(A_p)$. By Lemma 5.14 Part 1, $\beta = untime(\gamma) \in traces(A)$, which implies, since $(A, L) \sqsubseteq_{\mathrm{S}} (B, M)$, that $\beta \in traces(B)$. Now, the fact that γ is a timed sequence pair over $vis(A_p) = vis(B_p) = ext(B)$ and the fact that γ satisfies the property $seq(\gamma)$ being Zeno implies $ltime(\gamma)$ is the limit of the times in $seq(\gamma)$, Lemma 5.14 Part 2 implies that $\gamma \in t\text{-}traces(B_p)$, as required.

   $\Longleftarrow$: Let $\beta \in traces(A)$ and let γ be any, say, admissible timed sequence pair over $ext(A)$ such that $untime(\gamma) = \beta$. (Such a timed sequence pair clearly exists.) Then, by Lemma 5.14 Part 2, $\gamma \in t\text{-}traces(A_p)$. Thus, the assumption that $patient(A, L) \sqsubseteq_{\mathrm{St}} patient(B, M)$ implies $\gamma \in t\text{-}traces(B_p)$. Lemma 5.14 Part 1 shows that $\beta = untime(\gamma) \in traces(B)$, as required.

2. Similar to Part 1 by using Lemma 5.14 Parts 3 and 4. ■

Finally we prove a result which is important when doing specification and verification in a modular fashion. Namely, the *patient* operator commutes with the parallel composition operator on safe and live (timed) I/O automata. First, let $\equiv_{\mathrm{St}}$ and $\equiv_{\mathrm{Lt}}$ denote the *kernels* of the preorders $\sqsubseteq_{\mathrm{St}}$ and $\sqsubseteq_{\mathrm{Lt}}$, respectively.[2]

Proposition 5.16 *Let (A_0, L_0) and (A_1, L_1) be two compatible live I/O automata and let $\equiv_X$ be one of $\equiv_{\mathrm{St}}$ and $\equiv_{\mathrm{Lt}}$. Then,*

$$patient((A_0, L_0)\|(A_1, L_1)) \equiv_X patient(A_0, L_0)\|patient(A_1, L_1).$$

Proof. We show the proof for $\equiv_{\mathrm{St}}$. The proof for $\equiv_{\mathrm{Lt}}$ is similar. First note that, since (A_0, L_0) and (A_1, L_1) are compatible, then also $patient(A_0, L_0)$ and $patient(A_1, L_1)$ are compatible. Observe that for each timed execution Σ, $untime(\Sigma)\lceil A_i = untime(\Sigma\lceil A_i)$. Then,

	$\Sigma \in t\text{-}exec(patient(A_0\|A_1))$	
iff	$untime(\Sigma) \in exec(A_0\|A_1)$	Lemma 5.4
iff	$\forall_{i\in\{0,1\}} : untime(\Sigma)\lceil A_i \in exec(A_i)$	Lemma 3.5
iff	$\forall_{i\in\{0,1\}} : untime(\Sigma\lceil A_i) \in exec(A_i)$	observation above
iff	$\forall_{i\in\{0,1\}} : \Sigma\lceil A_i \in t\text{-}exec(patient(A_i))$	Lemma 5.4
iff	$\Sigma \in t\text{-}exec(patient(A_0)\|patient(A_1))$	Lemma 4.5. ■

6 Generality of Receptiveness

Receptiveness could be a severe restriction if very few protocols can be described within (timed) live I/O automata. In this section we argue that receptiveness is not severe by providing examples of sufficient conditions for receptiveness. Other examples are likely to be derived in the future based on new applications.

Ordinary I/O automata [LT87] are examples of receptive systems. That is, systems specified using weak fairness assumptions are receptive. Romijn and Vaandrager [RV96] provide an even stronger syntactic criterion for receptiveness in our model by introducing fair I/O automata. A fair I/O automaton is a safe I/O automaton A equipped with sets $wfair(A)$ and $sfair(A)$ of subsets of $local(A)$, called the weak fairness and strong fairness sets, respectively. The elements of $wfair(A)$ are sets of actions over which weak fairness is enforced, while the elements of $wfair(A)$ are sets of actions over which strong fairness is enforced. It is proven in [RV96] that a fair I/O automaton A is receptive if each reachable state in A enables at most countably many sets in $wfair(A) \cup sfair(A)$ and each set of $sfair(A)$ is input resistant, i.e., each set in $sfair(A)$ is never disabled by the occurrence of an input action.

In the timed case we have seen that the automata of [MMT91] are receptive, and we have mentioned that the strong I/O feasibility condition of [VL92] is a sufficient conditions for receptiveness. Furthermore, any patient construction over a live I/O automaton leads to a receptive pair. A more general sufficient condition for receptiveness is given in [BPV94], where

[2] The kernel of a preorder $\sqsubseteq$ is defined to be the equivalence $\equiv$ defined by $x \equiv y \ \triangleq\ x \sqsubseteq y \wedge y \sqsubseteq x$.

linear hybrid systems are introduced as a basic model for the study of an audio control protocol. Roughly speaking, a linear hybrid system is an automaton with discrete and continuous variables. The continuous variables are allowed to change during time passage with a rate that is bounded by a convex polyhedron. Furthermore the values of the continuous variables can be bounded to remain on one side of a hyperplane. Reaching a bound means forcing some action to occur before time can elapse.

7 Concluding Remarks

This paper extends I/O automata [LT87, MMT91] to handle general liveness properties in both the timed and untimed model, and creates a coordinated framework where timed and untimed systems can be analyzed. A key aspect of the models is the notion of *receptiveness*, which expresses the fact that a live (timed) I/O automaton does not constrain its environment. Moreover, [GSSL93] extends the simulation method of [AL91a, LV91, LV93, LV95, Jon91] to our model, making the results of this paper immediately applicable in practice. A substantial verification project using the model appears in [SLL93b, SLL93a]. In addition to generalizing the I/O automaton model [LT87] and its timed version [MMT91], our model generalizes the failure free complete trace structures of [Dil88] and the strong I/O feasibility notion of [VL92].

People familiar with process algebras might object to our model, arguing that receptiveness is too restrictive since it rules out several systems that might be of interest at a high level of abstraction. We recognize this objection and regard the generalization of the model as future work. In fact, our model is closer to the classical models of the process algebraic community (e.g., labeled transition systems) than the models of [AL93, AL91b], and thus may represent a natural starting point for possible generalizations. Some promising results come from [Seg93], which shows that there is a strong connection between the trace semantics of I/O automata and the MUST preorder of the theory of testing [DH84].

Another line of research consists of extending the current model to handle systems with probabilistic behaviors. The ultimate goal would be a model where probabilistic behaviors, timing constraints, safety properties, and liveness properties can be integrated together.

Acknowledgments

We thank Hans Henrik Løvengreen and Frits Vaandrager for their valuable criticism and useful comments on this paper.

References

[AL91a] M. Abadi and L. Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 2(82):253–284, 1991.

[AL91b] M. Abadi and L. Lamport. An old-fashioned recipe for real time. In de Bakker et al. [dBHRR91], pages 1–27.

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

[AS85] B. Alpern and F.B. Schneider. Defining liveness. *Information Processing Letters*, 21(4):181–185, 1985.

[BG91] J.C.M. Baeten and J.F. Groote, editors. *Proceedings of CONCUR 91,* Amsterdam, volume 527 of *Lecture Notes in Computer Science*. Springer-Verlag, 1991.

[BPV94] D.J.B. Bosscher, I. Polak, and F.W. Vaandrager. Verification of an audio control protocol. In Langmaack, de Roever, and Vytopil, editors, *Proceedings of the Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 863 of *Lecture Notes in Computer Science*, pages 170–192, 1994. Full version available as Report CS-R9445, CWI, Amsterdam, July 1994.

[Cle92] W.R. Cleaveland, editor. *Proceedings of CONCUR 92,* Stony Brook, NY, USA, volume 630 of *Lecture Notes in Computer Science*. Springer-Verlag, 1992.

[dBHRR91] J.W. de Bakker, C. Huizing, W.P. de Roever, and G. Rozenberg, editors. *Proceedings of the REX Workshop "Real-Time: Theory in Practice"*, volume 600 of *Lecture Notes in Computer Science*. Springer-Verlag, 1991.

[DH84] R. De Nicola and M. Hennessy. Testing equivalences for processes. *Theoretical Computer Science*, 34:83–133, 1984.

[Dil88] D. Dill. *Trace Theory for Automatic Hierarchical Verification of Speed-Independent Circuits.* ACM Distinguished Dissertations. MIT Press, 1988.

[Fis85] M. Fischer. Re: Where are you? E-mail message to Leslie Lamport. Arpanet message number 8506252257.AA07636@YALE-BULLDOG.YALE/ARPA (47 lines), June 25 1985.

[GSSL93] R. Gawlick, R. Segala, J.F. Søgaard-Andersen, and N.A. Lynch. Liveness in timed and untimed systems. Technical Report MIT/LCS/TR-587, MIT Laboratory for Computer Science, November 1993.

[GSSL94] R. Gawlick, R. Segala, J.F. Søgaard-Andersen, and N.A. Lynch. Liveness in timed and untimed systems. In S. Abiteboul and E. Shamir, editors, *Proceedings* 21^{th} *ICALP,* Jerusalem, volume 820 of *Lecture Notes in Computer Science*. Springer-Verlag, 1994. A full version appears as MIT Technical Report number MIT/LCS/TR-587.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice-Hall International, Englewood Cliffs, 1985.

[Jon91] B. Jonsson. Simulations between specifications of distributed systems. In Baeten and Groote [BG91], pages 346–360.

[Lam91] L. Lamport. The temporal logic of actions. Technical Report 79, Digital Equipment Corporation, Systems Research Center, December 1991.

[LS89] N.A. Lynch and E.W. Stark. A proof of the Kahn principle for Input/Output automata. *Information and Computation*, 82(1):81–92, 1989.

[LT87] N.A. Lynch and M.R. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *Proceedings of the* 6^{th} *Annual ACM Symposium on Principles of Distributed Computing*, pages 137–151, Vancouver, Canada, August 1987. A full version is available as MIT Technical Report MIT/LCS/TR-387.

[LV91] N.A. Lynch and F.W. Vaandrager. Forward and backward simulations for timing-based systems. In de Bakker et al. [dBHRR91], pages 397–446.

[LV93] N.A. Lynch and F.W. Vaandrager. Forward and backward simulations – part I: Untimed systems. Technical Report MIT/LCS/TM-486, MIT Laboratory for Computer Science, May 1993.

[LV95] Nancy Lynch and Frits Vaandrager. Forward and backward simulations – Part II: Timing-based systems. *Information and Computation*, 121(2):214–233, September 1995.

[MMP91] O. Maler, Z. Manna, and A. Pnueli. From timed to hybrid systems. In de Bakker et al. [dBHRR91], pages 447–484.

[MMT91] M. Merritt, F. Modugno, and M. Tuttle. Time constrained automata. In Baeten and Groote [BG91], pages 408–423.

[NS92] X. Nicollin and J. Sifakis. An overview and synthesis on timed process algebras. In K.G. Larsen and A. Skou, editors, *Proceedings of the Third Workshop on Computer Aided Verification,* Aalborg, Denmark, July 1991, volume 575 of *Lecture Notes in Computer Science*, pages 376–398. Springer-Verlag, 1992.

[Plo81] G.D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Computer science Department, Aarhus University, 1981.

[RV96] J.M.T. Romijn and F. Vaandrager. A note on fairness in I/O automata. *Information Processing Letters*, 59(5):245–250, 1996.

[RWZ92] N. Reingold, D.W. Wang, and L.D. Zuck. Games I/O automata play. In Cleaveland [Cle92], pages 325–339.

[Seg93] R. Segala. Quiescence, fairness, testing and the notion of implementation. In E. Best, editor, *Proceedings of CONCUR 93,* Hildesheim, Germany, volume 715 of *Lecture Notes in Computer Science*. Springer-Verlag, 1993.

[SLL93a] J.F. Søgaard-Andersen, B. Lampson, and N.A. Lynch. Correctness of at-most-once message delivery protocols. In *FORTE '93 - Sixth International Conference on Formal Description Techniques*, 1993.

[SLL93b] J.F. Søgaard-Andersen, N.A. Lynch, and B.W. Lampson. Correctness of communication protocols. a case study. Technical Report MIT/LCS/TR-589, MIT Laboratory for Computer Science, November 1993.

[VL92] F.W. Vaandrager and N.A. Lynch. Action transducers and timed automata. In Cleaveland [Cle92], pages 436–455.