Action Transducers and Timed Automata*

Frits Vaandrager[1] ** and Nancy Lynch[2]

[1] Department of Software Technology, CWI
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands
fritsv@cwi.nl
[2] MIT Laboratory for Computer Science
Cambridge, MA 02139, USA
lynch@theory.lcs.mit.edu

Abstract. The *timed automaton model* of [13, 12] is a general model for timing-based systems. A notion of *timed action transducer* is here defined as an automata-theoretic way of representing operations on timed automata. It is shown that two timed trace inclusion relations are substitutive with respect to operations that can be described by timed action transducers. Examples are given of operations that can be described in this way, and a preliminary proposal is given for an appropriate language of operators for describing timing-based systems. Finally, justification is given for the definition of implementation based on inclusion of timed trace sets; this is done in terms of a notion of *feasibility* which says that a timed automaton cannot prevent the passage of time.

1 Introduction

The *timed automaton model* of [13, 12] is a general model for timing-based systems. It is intended as a basis for formal reasoning about such systems, in particular, for verification of their correctness and for analysis of their complexity. In [13, 12], we develop a full range of *simulation* proof methods for timed automata; in this paper, we continue the development by studying *process algebras* for the same model. Eventually, we envision using a combination of proof methods, perhaps even using several in the verification of single system.

A timed automaton is an automaton (or labelled transition system) with some additional structure. There are three types of actions: *time-passage actions*, *visible actions* and the special *internal action* τ. All except the time-passage actions are thought of as occurring instantaneously. To specify times, a dense time domain is used, specifically, the nonnegative reals, and no lower bounds are imposed on the times between events. Two notions of external behavior are considered. First, as the finite behaviors, we take the *finite timed traces*, each of which consists of a finite sequence of timed visible actions together with a final time. Second, as the

* This work was supported by ONR contracts N00014-85-K-0168 and N00014-91-J-1988, by NSF grant CCR-8915206, and by DARPA contract N00014-89-J-1988.

** Most of the work on this paper was done while the first author was employed by the Ecole des Mines, CMA, Sophia Antipolis, France.

infinite behaviors, we take the *admissible timed traces*, each of which consists of a sequence of timed visible actions that can occur in an execution in which time grows unboundedly.

The timed automaton model permits description of algorithms and systems at different levels of abstraction. We say that a "low-level" timed automaton A *implements* a "high-level" timed automaton B if the sets of finite and admissible timed traces of A are included in the corresponding sets of B. To justify this notion of implementation, we must argue that the timed trace sets for A are not trivial. We can do this by classifying the visible actions of A as *input actions* or *output actions*, as in the I/O automaton model of [11]. We then require A to be *input enabled*, i.e., willing to accept each input action in each state, and *I/O feasible*, which means that each finite execution can be extended to an admissible execution via an execution fragment that contains no input actions. If A is input enabled and I/O feasible, then it can generate an admissible execution for any "non-Zeno" pattern of inputs, as follows. A starts with an admissible execution containing no inputs. When an input arrives, A performs a transition labelled by that input and continues from the resulting state with another admissible execution fragment containing no inputs until the next input arrives, etc. Thus, A must have a rich set of admissible timed traces.

In the 'classical' untimed case, bisimulation equivalences have been reasonably successful as a notion of implementation between transition systems [5, 15]. Consequently, bisimulation equivalences have also been put forward as a central notion in many studies on real-time process algebras [4, 9, 16, 17, 23]. However, we do not believe that bisimulation equivalences will be very useful as implementation relations in the timed case. The problem is that bisimulation equivalences do not allow one to abstract in specifications from the often very complex timing behavior of implementations (see [9] for an example).

Often, the design or verification of an algorithm includes several levels of abstraction, each of which implements the one above it. Note that it is only necessary to show that the trace sets at the *lowest level* are nontrivial. Thus, only at the lowest level does one require the I/O classification with its accompanying properties of input enabledness and I/O feasibility. Fortunately, at this level, the two properties are generally quite easy to achieve, since they correspond to the receptive and non-Zeno nature of physical machines.

Since we believe that timed trace inclusion does form a good notion of implementation, we are interested in identifying operations on timed automata for which the timed trace inclusion relation is substitutive. This substitutivity is a prerequisite for the compositional verification of systems using timed automata. It should also enable verification of systems using a combination of compositional methods and methods based on levels of abstraction.

We represent operations by automaton-like objects that we call *action transducers*, rather than, for example, using SOS specifications [20]. The importance of transducers for process algebra and concurrency theory was first noted by Larsen and Xinxin [10], who introduced a certain type of transducer, which they called a *context system*, to study compositionality questions in the setting of process algebra. For an example of an action transducer, consider the operation $\|$ of interleaving parallel composition. It can be described by an automaton with a single state s and

transitions (one for each action a):

$$s \xrightarrow[(1,a)]{a} s \quad \text{and} \quad s \xrightarrow[(2,a)]{a} s.$$

The left transition says that if the first argument performs an a-action the composition will also perform an a-action, while the right transition says that if the second argument performs an a-action the composition will also perform an a-action. Together, the transitions say that the automaton $A \parallel B$ can do an a-step whenever one of its arguments can do so. In the SOS approach, the same operator $\parallel$ can be described by inference rules (one for each action a):

$$\frac{x \xrightarrow{a} x'}{x \parallel y \xrightarrow{a} x' \parallel y} \quad \text{and} \quad \frac{y \xrightarrow{a} y'}{x \parallel y \xrightarrow{a} x \parallel y'}.$$

The two styles of describing operators, SOS and action transducers, are quite similar.

However, action transducers are more convenient for our purposes. First, although it is easy to see how SOS specifications determine automata, it is less clear how to regard them as defining *operations* on automata. For action transducers, this correspondence is more direct. Second, as noted by Larsen and Xinxin [10], action transducers are a convenient tool for studying compositionality questions, and their use tends to simplify proofs. Third, action transducers can easily be defined to allow multiple start states. Multiple start states have turned out to be useful in untimed automaton formalisms for concurrency such as the I/O automaton model, and we would like to include them. We do not know how to handle start states in the setting of SOS.

A major result of our paper is that the timed trace inclusion relation is substitutive with respect to all operations that can be described by our action transducers, provided they satisfy a number of conditions that concern the handling of internal and time-passage steps.

A condition that is required of our action transducers for timed automata is that time passes uniformly (i.e., at the same rate) for the transducer and all active holes, and it does not pass at all for the inactive holes. This uniformity condition (along with some other technical constraints) is used in our proof of substitutivity. We note that this time uniformity condition is not necessary for substitutivity to hold. For instance, the timed trace inclusion relation is substitutive for a "speedup" operation that doubles the rate at which its timed automaton argument operates. However, we do not have a clean generalization of the uniformity condition that applies to the speedup operation and still guarantees substitutivity.

Having proved substitutivity for a general class of operations, we describe many examples of specific operations that fall into this class. In our view, an appropriate language for describing timing-based systems should consist of a small number of basic operations, both timed and untimed, out of which more complex operations can be built. The basic and derived operations together should be sufficient to describe most interesting timing-based systems. As a starting point, we believe that such a language ought to include the basic untimed operations that are already

well understood and generally accepted. Nicollin and Sifakis [18] describe a simple and general construction to transform any untimed operation into a timed one that behaves essentially the same and moreover does not use or constrain the time. By applying this construction to the well-known untimed operations, we obtain a collection of corresponding timed operations that we believe should be included in a real-time process language.

The untimed operations alone are not enough, however; a real-time process language also must include operations that constrain time explicitly. Of the many possibilities, we would like to identify only one or two that can be used for constructing all the others. For this purpose, we tentatively propose a *timer* operation, derived from Alur and Dill [2]. Using only this timer operation and untimed operations, we can construct the timeout construct of Timed CSP [21, 7], and the execution delay operation of ATP [17]. Also, because this timer operation is derived from [2], we are able to use it to define a minor variant of Alur and Dill's ω-automata. Moreover, we can also use it to define the timed automata of Merritt, Modugno and Tuttle [14]. All of this provides evidence that our timer (or something very similar) may be appropriate to use as the sole timed operation in a real-time process language.

The decidability and closure properties of Alur-Dill automata suggest that they can be regarded as a real-time analog of classical finite automata. In the untimed setting, a crucial characteristic of algebras like CCS is that they can easily describe finite automata. Thus by analogy, a natural requirement for a real-time process language is that it can easily describe Alur-Dill automata. Nicollin, Sifakis and Yovine [19] give a translation from ATP into Alur-Dill automata, but do not investigate the reverse translation. In fact it appears that, besides our language, only the BPAr$\rho\delta$I-language of Baeten and Bergstra [4] is sufficiently expressive to allow for a direct encoding of Alur-Dill automata.

As discussed earlier, the appropriateness of the timed trace inclusion relation as a notion of implementation depends upon the lowest level implementation satisfying the input enabling and I/O feasibility properties. In the final section of our paper, we address the question of how to ensure that these properties hold. Although the input enabling condition can be required explicitly, I/O feasibility will typically require a proof. Since we envision the lowest level implementation being described in terms of operators from our real-time language, the way we would like to prove I/O feasibility is by proving it for the basic components and arguing that it is preserved by each of the operators. But examples (e.g., one studied in [1]) show that I/O feasibility is *not* preserved, e.g., by parallel composition. Therefore, we propose a condition, which we call *strong I/O feasibility*, which implies I/O feasibility, is preserved by operations expressed appropriately as timed action transducers, and appears to be satisfied by "low-level" automata, i.e., those that are close to physical machines. This condition can be used as the basis of a proof method for demonstrating I/O feasibility of implementations.

We present our definitions and results for timed systems by first presenting related definitions and results for untimed systems, and then building upon those wherever we can.

In summary, the main contributions of the paper are: (1) the definitions of action transducers and timed action transducers, (2) the substitutivity results for traces and

timed traces, (3) the notion of strong I/O feasibility, (4) the presentation of a large number of interesting operators, timed and untimed, as action transducers, and (5) a preliminary proposal for a process language for timed systems. We see these all as pieces of a unified proof methodology for timed systems.

2 The Untimed Setting

We begin by describing action transducers for the untimed setting. Later, the concepts needed for the timed setting will be defined in terms of corresponding concepts for the untimed setting.

2.1 Automata and Traces

An (untimed) *automaton* A consists of:

- a set $states(A)$ of *states*,
- a nonempty set $start(A) \subseteq states(A)$ of *start states*,
- a set $acts(A)$ of *actions* that includes the *internal action* τ, and
- a set $steps(A) \subseteq states(A) \times acts(A) \times states(A)$ of *steps*.

We let $s, s', u, u', ..$ range over states, and $a, ..$ over actions. The set $ext(A)$ of *external actions* is defined by $ext(A) \triangleq acts(A) - \{\tau\}$. We write $s' \xrightarrow{a}_A s$ as a shorthand for $(s', a, s) \in steps(A)$. We will suppress the subscript A where no confusion is likely. An *execution fragment* of A is a finite or infinite alternating sequence $s_0 a_1 s_1 a_2 s_2 \cdots$ of states and actions of A, beginning with a state, and if it is finite also ending with a state, such that for all i, $s_i \xrightarrow{a_{i+1}} s_{i+1}$. An *execution* of A is an execution fragment that begins with a start state.

For $\alpha = s_0 a_1 s_1 a_2 s_2 \cdots$ an execution $trace(\alpha)$ is defined as the sequence obtained from $a_1 a_2 \cdots$ by removing all τ's. A finite or infinite sequence β of actions is a *trace* of A if A has an execution α with $\beta = trace(\alpha)$. We write $traces^*(A)$, $traces^\omega(A)$ and $traces(A)$ for the sets of finite, infinite and all traces of A, respectively. These notions induce three preorders on automata: we define $A \leq_* B \triangleq traces^*(A) \subseteq traces^*(B)$, $A \leq_\omega B \triangleq traces^\omega(A) \subseteq traces^\omega(B)$, and $A \leq B \triangleq traces(A) \subseteq traces(B)$.

2.2 Action Transducers

We now define a notion of *action transducer*, as an explicit representation of certain extensional operations on automata. We consider operations with a possibly infinite set of arguments. As placeholders for these arguments, an action transducer contains a set of *colors*. Sometimes we will find it useful to make several copies of an argument automaton.[3] [4] To this end a transducer is equipped with a set of *holes*, and a

[3] The idea of copying arguments of transducers is not present in the work of Larsen and Xinxin [10].

[4] Note that, since we always start copies of an argument automaton from a start state, our copying operations are different from those of Bloom, Istrail and Meyer [6], which also allow copying from intermediate states. As a consequence, the trace preorder is substitutive for our operations, whereas it is not substitutive in general for the operations of [6].

mapping that associates a color to each hole. The idea is that we plug into each hole the argument automaton for which the color of the hole serves as placeholder. As a useful analogy one can consider the way in which a term with free variables determines an operation on terms: here the variables play the role of colors, and the occurrences of variables serve as holes. As the rest of its "static" description, a transducer has an associated global set of actions, and, for each color, a local set of actions. The "dynamic" part of a transducer is essentially an automaton: a set of states, a nonempty set of start states, and a step relation. The elements of the step relation are 4-tuples of source state, action, trigger and target state. Here the trigger is a function that tells, for each hole, whether the argument automaton in that hole is supposed to idle, or whether it has to participate in the step, and if so by which action. Finally, each state of the transducer has an associated set of *active holes*, and these are the only ones that can participate in the steps from that state. Formally, an *(action) transducer* T consists of:

- a set $states(T)$ of *states*,
- a nonempty set $start(T) \subseteq states(T)$ of *start states*,
- a set $holes(T)$ of *holes*,
- for each state s, a set $active_T(s) \subseteq holes(T)$ of holes that are *active* in s,
- a set $colors(T)$ of *colors*,
- a map $c\text{-}map_T : holes(T) \to colors(T)$, the *coloring* of the holes,
- a set $acts(T)$ of *actions* that includes τ,
- for each color m, a set $acts_T(m)$ of *actions* that includes τ,
- a set $steps(T) \subseteq states(T) \times acts(T) \times triggers(T) \times states(T)$, where $triggers(T)$ is the set of maps $\eta : holes(T) \to (\bigcup_m acts_T(m) \cup \{\bot\})$ such that, for all i, $\eta(i) \in acts_T(c\text{-}map_T(i)) \cup \{\bot\}$. We require that if $(s', a, \eta, s) \in steps(T)$ and $\eta(i) \neq \bot$, then $i \in active_T(s')$.

We define the sets of *external actions* of T by $ext(T) \triangleq acts(T) - \{\tau\}$, and, for each m, $ext_T(m) \triangleq acts_T(m) - \{\tau\}$. We write $s' \xrightarrow[\eta]{a}_T s$ instead of $(s', a, \eta, s) \in steps(T)$. We call s' the *source* of the step, s the *target*, a the *action*, and η the *trigger*. Often we will suppress the subscript T. We often represent a trigger η by the set $\{(i, a) \mid \eta(i) = a \neq \bot\}$. An *execution fragment* of T is a finite or infinite alternating sequence $\alpha = s_0 a_1 \eta_1 s_1 a_2 \eta_2 s_2 \cdots$ of states, actions and triggers of T, beginning with a state, and if it is finite also ending with a state, such that for all i, $s_i \xrightarrow[\eta_{i+1}]{a_{i+1}} s_{i+1}$. An *execution* of T is an execution fragment that begins with a start state.

We view action transducers as a generalization of automata. Thus we will frequently identify an action transducer having an empty set of holes with its underlying automaton.

2.3 Combining Transducers and Automata

We now define the meaning of a transducer as an operation on automata.[5] First, define an *automaton assignment* for T to be a function ζ that maps each color m

[5] In fact, it is often useful to interpret transducers in a more general (and somewhat more complex) way, as operations on transducers.

of T to an automaton in such a way that $acts(\zeta(m)) = acts_T(m)$. Suppose ζ is an automaton assignment for T, and let Z be the composition $\zeta \circ c\text{-}map_T$ (so Z associates an automaton to each hole). Then $T(\zeta)$ is the automaton A given by:

- $states(A) = \{(s, z) \mid s \in states(T)$ and z maps holes i of T to states of $Z(i)\}$,
- $start(A) = \{(s, z) \mid s \in start(T)$ and z maps holes i of T to start states of $Z(i)\}$,
- $acts(A) = acts(T)$, and
- $steps(A)$ is the least relation such that

$$s' \xrightarrow[\eta]{a}_T s \ \wedge\ \forall i : [\text{if } \eta(i) = \perp \text{ then } z'(i) = z(i) \text{ else } z'(i) \xrightarrow{\eta(i)}_{Z(i)} z(i)] \\ \Rightarrow \quad (s', z') \xrightarrow{a}_A (s, z).$$

2.4 Substitutivity

Now we state our substitutivity result for untimed action transducers. A relation R on a class of automata $\mathcal{A}$ is *substitutive* for an action transducer T if for all automaton assignments ζ, ζ' for T with range $\mathcal{A}$,

$$\forall m \in colors(T) : \zeta(m)\ R\ \zeta'(m) \quad \Rightarrow \quad T(\zeta)\ R\ T(\zeta').$$

An action transducer T is *τ-respecting* if it satisfies the following constraints:

1. For each state s and for each hole i that is active in s, T contains a *clearing step*, i.e., a step $s \xrightarrow[\{(i,\tau)\}]{\tau} s$.
2. If $s' \xrightarrow[\eta]{a} s$ and $\eta(i) = \tau$, then $s' \xrightarrow[\eta]{a} s$ is a clearing step for s' and i.
3. Only finitely many holes participate in each step, i.e., if $s' \xrightarrow[\eta]{a} s$ then $\{i | \eta(i) \neq \perp\}$ is finite.

Theorem 1. *The relations $\leq_*$ and $\leq$ on automata are substitutive for all τ-respecting action transducers.*

In the full paper we give an example to show that $\leq_\omega$ is not substitutive, even for τ-respecting action transducers. The converse of Theorem 1 does not hold: there are many examples of non-τ-respecting action transducers for which $\leq_*$ and $\leq$ are substitutive.

2.5 Examples

We give examples of operations that can be expressed as action transducers; these examples include variants of most of the usual operations considered in (untimed) process algebras ([8, 15, 5]).

We first describe a number of conventions so that, in most cases, we do not have to specify the static part of transducers explicitly. Since we only use the ability to make copies of arguments in one of our operations, we adopt the convention that, unless otherwise specified, the sets of holes and colors are equal, and the coloring function is the identity. Often, the set of holes will be an initial fragment $\{1, \ldots, n\}$

of the natural numbers. All action transducers that we define are parametrized by the actions sets of their arguments. By default, there are no restrictions on these action sets. The global action set of a transducer can be obtained by taking the set of all actions that occur in steps of the transducer. Finally, the set of active holes of a state will be implicitly defined as the set of holes that participate in one of the steps starting from that state.

In our language, it is convenient to allow each external action to be structured as a nonempty finite set of *labels*. Sometimes, for uniformity, it will be convenient to identify τ with the empty set. For each transducer T we define $labels(T) \triangleq \bigcup ext(T)$. Similarly we define, for each color m of T, $labels_T(m) \triangleq \bigcup ext_T(m)$. Often we will denote the singleton set $\{l\}$ by the symbol l.

The simplest action transducer is STOP_H. It is parametrized by a set H of actions, has no holes, no colors, no steps, just a single state, which is also start state.

Transducer SKIP denotes the process that terminates successfully. The transducer starts in state s_1, does action $\{\surd\}$ and then stops in state s_2.

$$s_1 \xrightarrow[\emptyset]{\surd} s_2$$

Here $\surd$ is a special label denoting *successful termination*. In accordance with this terminology, our language has been designed in such a way that no further transitions are possible after a transition whose label contains $\surd$. Also, $\surd$ does not occur in the process language itself.

For any nonempty set a of labels with $\surd \notin a$, transducer a starts in state s_1, performs action $a \cup \{\surd\}$ and then stops in state s_2.

$$s_1 \xrightarrow[\emptyset]{a \cup \{\surd\}} s_2$$

Transducer ";" describes the binary operation of *sequential composition*. This operation runs its first argument up to successful termination and then runs its second argument. The transducer has two states s_1, s_2, of which s_1 is the start state, and steps (with $\emptyset$ denoting τ):

$$s_1 \xrightarrow[\{(\overrightarrow{1},a)\}]{a} s_1 \text{ for } \surd \notin a \in acts_;(1)$$

$$s_1 \xrightarrow[\{(\overrightarrow{1},a)\}]{a-\{\surd\}} s_2 \text{ for } \surd \in a \in acts_;(1)$$

$$s_2 \xrightarrow[\{(\overrightarrow{2},a)\}]{a} s_2 \text{ for } a \in acts_;(2)$$

Parametrized by an index set I, the *external choice operation* $\Box$ waits for the first external action of any of its arguments and then runs that argument. The transducer has distinct states s_i, for each $i \in I$, plus an additional state s, which is the only start state. The steps are:

$$s \xrightarrow[\{(\overrightarrow{i},\tau)\}]{\tau} s \text{ for } i \in I$$

$$s \xrightarrow[\{(\overrightarrow{i},a)\}]{a} s_i \text{ for } i \in I \wedge a \in ext_\Box(i)$$

$$s_i \xrightarrow[\{(\overrightarrow{i},a)\}]{a} s_i \text{ for } i \in I \wedge a \in acts_\Box(i)$$

Parametrized by an index set I, transducer $\sqcup$ takes the *disjoint union* of automata indexed by I. Operationally it behaves as the *internal choice operation* $\sqcap$ of CSP [8]: the operation runs one, nondeterministically chosen argument.[6] For each $i \in I$, the transducer has a distinct state s_i, which is also a start state, and steps:

$$s_i \xrightarrow[\{(\overrightarrow{i,a})\}]{a} s_i \text{ for } i \in I \wedge a \in acts_{\sqcup}(i)$$

Transducer $\|$, which is parametrized by a *finite* index set I, describes the operation of *parallel composition*. The transducer has a single state s, which is the start state, and steps:

$$s \xrightarrow[\{(\overrightarrow{i,\tau})\}]{\tau} s \text{ for } i \in I$$

$$s \xrightarrow[\eta]{a} s \quad \text{for } \emptyset \neq a \subseteq \bigcup_{j \in I} labels_{\|}(j) \wedge \forall i : [\eta(i) = \perp \wedge a \cap labels_{\|}(i) = \emptyset] \vee [\eta(i) = a \cap labels_{\|}(i) \in ext_{\|}(i)]$$

We require that $\surd$ is either in the label set of all arguments or in the label set of none of them.

The postfix *hiding* operation $\backslash L$ hides all labels from the set L by removing them from the actions. The steps are (with $\emptyset$ denoting τ):

$$s \xrightarrow[\{(\overrightarrow{1,a})\}]{a-L} s \text{ for } a \in acts_{\backslash L}(1)$$

For each function f on labels such that $f(l) = \surd$ implies that $l = \surd$, we introduce a unary *relabeling* operation "f" that renames actions of its argument according to f. The steps are (with f lifted to sets of labels and $\emptyset$ denoting τ):

$$s \xrightarrow[\{(\overrightarrow{1,a})\}]{f(a)} s \text{ for } a \in acts_{f}(1)$$

Transducer $\wedge$ describes the binary *interrupt* operation of CSP. The transducer runs its first argument until the second argument performs an external action; after that the first argument is disabled and the second argument takes over. The transducer has two states s_1, s_2, of which s_1 is the start state, and steps:

$$s_1 \xrightarrow[\{(\overrightarrow{1,a})\}]{a} s_1 \text{ for } a \in acts_{\wedge}(1)$$

$$s_1 \xrightarrow[\{(\overrightarrow{2,\tau})\}]{\tau} s_1$$

$$s_1 \xrightarrow[\{(\overrightarrow{2,a})\}]{a} s_2 \text{ for } a \in ext_{\wedge}(2)$$

$$s_2 \xrightarrow[\{(\overrightarrow{2,a})\}]{a} s_2 \text{ for } a \in acts_{\wedge}(2)$$

In order to guarantee that after a $\surd$-step no other transitions are possible, we require that $\surd \notin labels_{\wedge}(1)$.

[6] We have tried to use CSP notation where possible. However, here we have not followed CSP since we found it confusing to use an intersection-like symbol for the operation of disjoint union.

Now we describe a construct that exploits the ability of transducers to copy their arguments. The construct is inspired by the process creation mechanism of the object-oriented language POOL-T [3]. Transducer CREATE$(C, I, .)$ is parametrized by a set C of *classes* and a nonempty subset $I \subset C$ of *initial classes*. The transducer takes colors from C. Initially the transducer starts up a single instance of one of the initial classes. Then, each time some process does an action containing a label $new(X)$, for some class X, a new instance of X is created. Formally, the states of CREATE are finite multisets over C, i.e., functions $M : C \to \mathsf{N}$ that are 0 almost everywhere. The set of start states of the transducer is $\{\{X\} | X \in I\}$, it takes holes from the set $C \times \mathsf{N}^+$, and the color of hole (X, j) is X. The steps are (with $\emptyset$ denoting τ):

$$M \xrightarrow[\{((X,j),a)\}]{a-\{new(Y)|Y\in C\}} M \cup \{Y | new(Y) \in a\} \quad \text{for} \quad M(X) \geq j \wedge a \in acts_{\mathsf{CREATE}}(X)$$

The CREATE construct adds a lot of expressive power to our language, and plays a role similar to the recursion constructs in process algebras such as ACP [5], CCS [15] and CSP [8]. By an example we show how this construct allows us to specify arbitrary automata *inside* our language up to isomorphism. As a useful notation, define

$$a \,.\, X \triangleq a \cup \{new(X)\} \,;\, \mathsf{STOP}$$

for X a class and a a nonempty set of non-*new* labels. Now the expression[7]

$$\begin{aligned} SWITCH \triangleq\ & \mathsf{CREATE}(\{OFF, ON\}, \{OFF\}, \\ & OFF \mapsto sw\_on \,.\, ON \\ & ON \mapsto sw\_on \,.\, ON \,\Box\, sw\_off \,.\, OFF \ ) \end{aligned}$$

denotes a finite automaton that describes an automatic switch off mechanism. The system allows a lamp to be switched on at any time; once it has been switched on, it can be switched off (automatically). In Section 3 we will come back to this example and show how we can add real-time constraints to make it more interesting.

Using the CREATE operation with a single argument, we can define the *looping* operation ω, which restarts its argument upon each successful termination.

$$(A)^\omega \triangleq \mathsf{CREATE}(\{X\}, \{X\}, f(A)),$$

where f relabels $\surd$ to $new(X)$ and leaves all other labels unchanged.

Note that all of the operations described in this subsection are τ-respecting. Therefore, it follows from Theorem 1 that the preorders $\leq_*$ and $\leq$ are substitutive for all these operations.

[7] We warn the reader that, even though the notation we use here is very close to the standard notation in process algebra for systems of recursion equations, the operational semantics is quite different in the case of nonlinear systems.

3 The Timed Setting

3.1 Timed Automata

We use a slight variant of the *timed automaton* model from [12].[8] A *timed automaton* A is an automaton whose set of actions is a superset of R^+, the set of positive reals. Actions from R^+ are referred to as *time-passage actions*. We let $d, d', \ldots$ range over time-passage actions and, more generally, over the set R of real numbers. The set of *visible* actions is defined by $vis(A) \triangleq ext(A) - \mathsf{R}^+$. We assume that a timed automaton satisfies the following axioms:

S1 If $s' \xrightarrow{d} s''$ and $s'' \xrightarrow{d'} s$, then $s' \xrightarrow{d+d'} s$.
S2 If $s' \xrightarrow{d} s$ then there exists a trajectory from s' to s of length d.

Here a *trajectory from s' to s of length d* is a function $w : [0, d] \to states(A)$ such that $w(0) = s'$, $w(d) = s$, and $w(d_1) \xrightarrow{d_2 - d_1} w(d_2)$ for all $d_1, d_2 \in [0, d]$ with $d_1 < d_2$.

3.2 Timed Traces

Suppose $\beta = a_0 a_1 a_2 \cdots$ is a trace of a timed automaton A. For each i, we define the *time of occurrence* t_i of event a_i by:

$$t_0 = 0$$
$$t_{i+1} = \textbf{if } a_i \in \mathsf{R}^+ \textbf{ then } t_i + a_i \textbf{ else } t_i$$

The *limit time* of β, notation $\beta.ltime$, is the smallest element of $\mathsf{R}^{\geq 0} \cup \{\infty\}$ larger than or equal to (i.e., the supremum of) all the t_i. The *timed trace* $t\text{-}trace(\beta)$ associated with β is defined by:

$$t\text{-}trace(\beta) \triangleq (((a_0, t_0)(a_1, t_1)(a_2, t_2)\cdots) \lceil (vis(A) \times \mathsf{R}^{\geq 0}), \beta.ltime)$$

So $t\text{-}trace(\beta)$ records the visible events of β paired with their time of occurrence, as well as the limit time of the sequence. The above definitions are lifted to executions in the obvious way. For α an execution of A, $\alpha.ltime \triangleq trace(\alpha).ltime$ and $t\text{-}trace(\alpha) \triangleq t\text{-}trace(trace(\alpha))$. Execution α is *admissible* if $\alpha.ltime = \infty$.

A pair p is a *timed trace* of A if it is the timed trace of some finite or admissible execution of A. Thus, we explicitly exclude the timed traces that originate from *Zeno* executions, i.e., infinite executions with a finite limit time. We write $t\text{-}traces(A)$ for the set of all timed traces of A, $t\text{-}traces^*(A)$ for the set of *finite* timed traces, i.e., those that originate from a finite execution, and $t\text{-}traces^\infty(A)$ for the *admissible* timed traces, i.e., those that originate from an admissible execution. These notions induce three preorders on timed automata: $A \leq^t B \triangleq t\text{-}traces(A) \subseteq t\text{-}traces(B)$, $A \leq^t_* B \triangleq t\text{-}traces^*(A) \subseteq t\text{-}traces^*(B)$, and $A \leq^t_\infty B \triangleq t\text{-}traces^\infty(A) \subseteq t\text{-}traces^\infty(B)$. The kernels of these preorders are denoted by $\equiv^t$, $\equiv^t_*$ and $\equiv^t_\infty$, respectively.

[8] The difference is just the explicit indication of the amount of elapsed time in the time-passage action instead of using a *.now* function that associates the current time to a state.

3.3 Timed Action Transducers

A *timed action transducer* T is a transducer with $\mathsf{R}^+ \subseteq acts(T)$ and, for all colors m, $\mathsf{R}^+ \subseteq acts_T(m)$. The sets of *visible* actions are defined by $vis(T) \triangleq ext(T) - \mathsf{R}^+$ and, for all m, $vis_T(m) \triangleq ext_T(m) - \mathsf{R}^+$. We assume that T satisfies the following axioms.

T1 If $s' \xrightarrow[\eta]{d} s$ then $\eta(i) = d$ for all $i \in active_T(s')$.

(As a consequence of axiom **T1**, the trigger of a d step is fully determined. This justifies our convention below to write $s' \xrightarrow{d} s$ instead of $s' \xrightarrow[\eta]{d} s$).

T2 If $s' \xrightarrow[\eta]{a} s$ and $\eta(i) = d$, then $a = d$.

T3 If $s' \xrightarrow{d} s$ then $active_T(s') = active_T(s)$.

We also assume that T satisfies the axioms **S1** and **S2** for timed automata (with triggers added implicitly according to the convention above).

An automaton assignment ζ for a timed action transducer T is called *timed* if it maps each color to a timed automaton.

Lemma 2. *Suppose T is a timed action transducer and ζ is a timed automaton assignment for T. Then $T(\zeta)$ is a timed automaton.*

3.4 Substitutivity

A timed action transducer T is *Zeno respecting* if for each execution $s_0 a_1 \eta_1 s_1 a_2 \eta_2 \cdots$ of T in which the sum of the time passage actions in $a_1 a_2 \cdots$ grows to ∞, and for each hole i, either the sum of the time passage actions in $\eta_1(i)\eta_2(i)\cdots$ grows to ∞ as well, or there exists an index n such that $\eta_m(i) = \perp$ for all $m \geq n$.

Transducers that are not Zeno respecting can turn a Zeno execution of an argument into a non-Zeno execution, by deactivating the argument infinitely often while advancing time. Since timed trace inclusion does not preserve Zeno traces, this means that $\leq^t$ will in general not be substitutive for such transducers. However, we do have the following result.

Theorem 3. *The relations $\leq^t_*$ and $\leq^t$ on timed automata are substitutive for all Zeno- and τ-respecting timed action transducers.*

3.5 Examples

We give examples of operations that can be expressed as timed action transducers.

Timed Transducers from Untimed Transducers An important collection of timed transducers can be obtained from untimed transducers. In this subsection we present a simple but important construction, essentially due to Nicollin and Sifakis [18], that transforms an untimed action transducer into a timed one, by inserting arbitrary time-passage steps. Suppose T is an (untimed) action transducer whose

sets of actions are disjoint with R^+. Then the structure *patient*(T) is obtained from T by adding R^+ to all the action sets, and also adding, for each state s of T and each $d \in \mathsf{R}^+$, a step

$$s \xrightarrow[\eta]{d} s, \quad \text{where, for all } i, \eta(i) = \textbf{if } i \in \mathit{active}_T(s) \textbf{ then } d \textbf{ else } \perp .$$

It is straightforward to check that *patient*(T) is indeed a timed action transducer.

If T is τ-respecting then *patient*(T) is not τ-respecting in general. For instance, the transducer $\Box$ for external choice over an infinite index set of Section 2.5 is τ-respecting, but its patient timed version is not. The following simple lemma characterizes the situations in which the *patient* operation preserves the property of being τ-respecting.

Lemma 4. *Suppose T is a τ-respecting action transducer whose sets of actions are disjoint with R^+. Then patient(T) is Zeno respecting. Moreover, patient(T) is τ-respecting iff in each state of T only finitely many holes are active.*

Except for external choice over an infinite index set, all the patient timed versions of the transducers of Section 2.5 are τ-respecting, by Lemma 4. Thus, by Theorem 3, the timed trace preorders $\leq^{\mathrm{t}}_*$ and $\leq^{\mathrm{t}}$ are substitutive for the *patient* variants of all the transducers of Section 2.5 except for infinitary external choice.

The timed transducers obtained by the *patient* construction turn out to be quite useful, so in the subsequent sections we will adopt the convention that T means *patient*(T) for any of the transducers of Section 2.5.

Timers We consider a set X of *timer variables*, ranged over by $x, y, \ldots$. The set of *timer constraints* ϕ is defined inductively by:

$$\phi ::= x < d \mid x = d \mid \phi \wedge \phi' \mid \neg\phi$$

Note that constraints such as true, $5 < 4$, $x \geq d$, $x \in [2,5)$ can be defined as abbreviations.

A *time assignment* $\xi : X \to \mathsf{R}^{\geq 0}$ assigns a nonnegative real value to each timer variable. We say that a time assignment ξ for *satisfies* a timer constraint ϕ, notation $\xi \models \phi$, iff ϕ evaluates to true using the values given by ξ. We say that ϕ is a *tautology* iff for all time assignments ξ, $\xi \models \phi$. We say that ϕ is *satisfiable* iff there exists a time assignment ξ such that $\xi \models \phi$. We denote by $\phi[d/x]$ the formula obtained from ϕ by replacing all occurrences of x by d.

The transducer TIMER^x_d models the behavior of a timer called x. The argument $d \in \mathsf{R}^{\geq 0} \cup \{\infty\}$ gives a *bound* beyond which time cannot proceed.[9] The state set of the transducer is $\mathsf{R}^{\geq 0} \times (\mathsf{R}^{\geq 0} \cup \{\infty\})$ and the initial state is $(0, d)$. There is just one argument, that is, one hole and one color. The argument can reset timer x at

[9] For simplicity, we do not consider timers with strict bounds. Such timers can be defined by parametrizing the transducer with an additional boolean that tells whether the time bound is strict or not. Alternatively, one can follow a suggestion of Abadi and Lamport [1], and introduce, as additional elements of the time domain, the set of all 'infinitesimally shifted' real numbers r^-, where $t \leq r^-$ iff $t < r$, for any reals t and r.

any moment via a label $x := 0$; similarly the upper bound can be modified via a label $x :\leq d$. Besides assigning values to the timer, an automaton can use timer constraints as labels to test the values of the various timers in whose scope it occurs. TIMER_d^x has, for instance, a step

$$(1,10) \xrightarrow[\{(\overrightarrow{1,8.5})\}]{8.5} (9.5,10), \quad \text{but not a step} \quad (1,10) \xrightarrow[\{(\overrightarrow{1,9.5})\}]{9.5} (10.5,10),$$

because that would violate the time bound. If $a = \{sw\_off, 9 \leq x, x \leq 10, x :\leq \infty\}$ and $b = \{sw\_off\}$, then TIMER_d^x also has a step

$$(9.5,10) \xrightarrow[\{(\overrightarrow{1,a})\}]{b} (9.5,\infty), \quad \text{but not a step} \quad (1,10) \xrightarrow[\{(\overrightarrow{1,a})\}]{b} (1,\infty),$$

since a contains a constraint $9 \leq x$.

In order to define the step relation formally it is convenient to define some auxiliary functions. For x a timer, $d_1, d_2 \in \mathsf{R}^+$ and a a set of labels, $\mathcal{A}(x, d_1, a)$ is obtained from a by first removing all labels $x := 0$ and $x :\leq d$ (for all d) then replacing each time constraint ϕ in a by $\phi[d_1/x]$, and finally removing all tautologies from the result. We say $\mathcal{A}(x, d_1, a)$ is *satisfiable* if all time constraints contained in it are satisfiable. We also define

$$\mathcal{V}(x, d_1, a) \triangleq \textbf{if } x := 0 \in a \textbf{ then } 0 \textbf{ else } d_1,$$
$$\mathcal{B}(x, d_2, a) \triangleq \textbf{if } \{d | x :\leq d \in a\} = \emptyset \textbf{ then } d_2 \textbf{ else } \min\{d | x :\leq d \in a\}.$$

Now the steps of TIMER_d^x can be defined by (with $\emptyset$ denoting τ):

$$(d_1, d_2) \xrightarrow[\{(\overrightarrow{1,d})\}]{d} (d_1 + d, d_2) \qquad \text{for } d_1 + d \leq d_2,$$
$$(d_1, d_2) \xrightarrow[\{(\overrightarrow{1,a})\}]{b} (\mathcal{V}(x, d_1, a), \mathcal{B}(x, d_2, a)) \quad \text{for } b = \mathcal{A}(x, d_1, a) \text{ satisfiable.}$$

The reader can easily check that TIMER_d^x is Zeno- and τ-respecting. Thus relations $\leq_*^t$ and $\leq^t$ are substitutive for this transducer.

Our definition of a timer is similar to the one proposed by Alur and Dill [2] for their timed ω-automata. However, instead of a Büchi style acceptance criterion we use bounds to specify *urgency*, i.e., properties that say that certain actions *must* occur at a certain time. We suppose that for some applications, it will be useful to have a more general definition. One can, for instance, extend the set of time constraints with formulas like $x + y < 1$, or allow for assignments of the form $x := y + 4$, or introduce operations that ask the timer to emit its current time. The important point here is that explicit timers constitute an important and useful construct in real-time process algebra. Our specific choice of operations is just an example, subject to modification.

A Timed Process Algebra We think that a reasonable proposal for a timed process algebra might include *only* (1) the timed transducers obtained by applying the *patient* operation to the untimed transducers discussed earlier, and (2) the *timers* described above. A justification for this is that these are sufficient to implement all the other timed operators we have thought of. In this section, we give some of these derivations.

Using a single timer, we can define the process WAIT d of Timed CSP [21, 7], which waits time d and then terminates successfully.

$$\mathsf{WAIT}\ d \triangleq \mathsf{TIMER}^x_d(x = d)$$

More generally, we can specify a process that terminates successfully after waiting some nondeterministically chosen time from the closed interval $[d_1, d_2]$.

$$\mathsf{WAIT}\ [d_1, d_2] \triangleq \mathsf{TIMER}^x_{d_2}(x \geq d_1)$$

Using a timer with deadline 0, we can restrict any automaton to its behavior at time 0. We define

$$\overline{A} \triangleq \mathsf{TIMER}^x_0(A)$$

The above construction is useful for defining the *timeout* construct of Timed CSP. For a given delay d this operator is defined, as in [7], by

$$A \overset{d}{\triangleright} B \triangleq (A \,\Box\, (\mathsf{WAIT}\ d\ ;\ \overline{abort}\ ;\ B))\backslash\{abort\}$$

If, at time d, A has not performed any visible action, an interrupt occurs and automaton B is started. Note that we need the auxiliary label *abort* (not in the label set of A and B) to force the choice between A and B at time d.

The *execution delay* operator of ATP [17] is given by:

$$\begin{aligned}\lceil A \rceil^d(B) \quad \triangleq \quad & (\mathsf{TIMER}^x_d((A \wedge (\{abort, x = d\}\ ;\ B))\|C))\backslash\{abort, cancel\} \\ & \text{where } C = \{cancel, x :\leq \infty\}\ ;\ \mathsf{STOP} \,\Box\, \{abort, x :\leq \infty\}\ ;\ \mathsf{SKIP}\end{aligned}$$

$\lceil A \rceil^d(B)$ behaves as A until time d; at time d, A is interrupted and B is started. However, if A performs a special label *cancel*, then the interrupt is cancelled and A can continue to run forever. The auxiliary process C takes care that once A has done *cancel*, it can no longer be interrupted by B. Also C removes deadline d after a *cancel* or *abort* action. We assume that A and B do not have *abort* in their label set, nor any label referring to timer x. For simplicity we also assume that A does not contain the termination symbol $\surd$. The labels *cancel* and *abort* are hidden so that they cannot communicate with any other action. A minor difference between our execution delay operator and the one from ATP is that ours allows machine A to perform visible actions *at* time d.

As an illustration of the use of the timed operators presented thus far, we specify a timed version of the automatic switch-off mechanism we described in Section 2.5. The system allows a lamp to be switched on at any time; then between 9 and 10 time units after the last time the lamp has been switched on, it will be switched off.

$$\begin{aligned}T\_SWITCH \triangleq\ & \mathsf{TIMER}^x_\infty(\mathsf{CREATE}(\{OFF, ON\}, \{OFF\}, \\ & OFF \mapsto \{sw\_on, x := 0, x :\leq 10\}\ .\ ON \\ & ON \mapsto \{sw\_on, x := 0\}\ .\ ON \,\Box\, \{sw\_off, 9 \leq x \leq 10, x :\leq \infty\}\ .\ OFF\ ))\end{aligned}$$

Counterexamples Although the converse of Theorem 3 does not hold, our result appears to be quite sharp: for many examples of timed transducers that are not τ-respecting, the timed trace preorders are indeed not substitutive. The best example is the CCS-like choice operator $+$ that plays a dominant role in many real-time process calculi (TCCS [16], the timed extension of CCS proposed in [23], ATP [17], and ACPρ [4]). This operator can be viewed as the *patient* version of the choice operator from CCS. Relation $\leq_*$ is not substitutive for $+$ because

$$\mathsf{WAIT}\ 2 + \mathsf{WAIT}\ 1.5 \quad \not\leq^{\mathrm{t}}_{*} \quad (\mathsf{WAIT}\ 1\ ;\ \mathsf{WAIT}\ 1) + \mathsf{WAIT}\ 1.5$$

The first process will always terminate at time 1.5, whereas the second process will always terminate at time 2.

4 Nontriviality of Implementations

As explained in the Introduction, we exclude trivial implementations by distinguishing input and output actions and requiring implementations to be input enabled and I/O feasible. In a process algebraic setting, a natural way to prove the property $\phi \triangleq$ *input enabled* $\wedge$ *I/O feasible*, is to describe a timed automaton purely in terms of operators that preserve some property ψ that implies ϕ. In this section we describe such a ψ, i.e., a property that implies input enabledness and I/O feasibility and that is preserved by an interesting class of operators.

We begin by defining *timed I/O automata* and *I/O feasibility*.

4.1 Timed I/O Automata

A *timed I/O automaton* A is an timed automaton together with a set $inp(A) \subseteq vis(A)$ of *input actions*. We require that input actions always be enabled, i.e., for each state s and for each $a \in inp(A)$ there is a step $s \xrightarrow{a}_A s'$.

We define the set $out(A)$ of *output actions* by $out(A) \triangleq vis(A) - inp(A)$, and the set $loc(A)$ of *locally controlled actions* by $loc(A) \triangleq out(A) \cup \{\tau\}$. A step $s' \xrightarrow{a} s$ is said to be *under control of* A if $a \in loc(A)$. By $t\text{-}AUT(A)$ we denote the timed automaton that underlies timed I/O automaton A. Conversely, for each timed automaton A and $IN \subseteq vis(A)$ a set of actions that are enabled in each state of A, $IO_{IN}(A)$ denotes the corresponding timed I/O automaton.

Timed I/O automata constitute a variant of the I/O automata model of Lynch and Tuttle [11] within our timed setting.[10]

4.2 I/O Feasibility

A timed I/O automaton is *I/O feasible* if each finite execution can be extended to an admissible execution via an execution fragment that does not contain input actions. I/O feasibility strengthens the notion of *feasibility* used in [13, 12], which simply requires that each finite execution can be extended to an admissible one.

[10] They do not include the class structure, used in I/O automata to model fairness.

This means that all the results of [13, 12] for feasible automata are valid for I/O feasible automata. However, the requirement of feasibility is too weak to exclude trivial implementations, because it may be that a feasible implementation can only produce an admissible execution in case the environment provides certains inputs. Thus if the environment does not offer these inputs, a Zeno execution is produced. To illustrate the problem, we consider as a specification the timed switch of the previous section, with as additional "feature" that it can break down at any moment and stop functioning:

$$IO_{\{sw\_on, br\_down\}}(T\_SWITCH \wedge (br\_down \,;\, (sw\_on \,\Box\, br\_down)^{\omega}))$$

This specification is implemented by the following feasible timed I/O automaton:

$$IO_{\{sw\_on, br\_down\}}((\overline{sw\_on})^{\omega} \wedge (br\_down \,;\, (sw\_on \,\Box\, br\_down)^{\omega}))$$

But this is an undesirable implementation because, first, the machine will never turn the switch off, even if no breakdown occurs, and moreover, the machine will not allow time to advance unless there is an immediate breakdown. The notion of I/O feasibility excludes this type of implementation.

4.3 Timed I/O Transducers

An I/O automaton should always be willing to engage in input actions, because these are thought of as being under the control of the environment. When defining operations on I/O automata, it is natural to require, as a dual property, that the environment (i.e., the operation) cannot block output actions, as these are under the control of the automata. In the untimed setting the nonblocking condition for outputs can be motivated technically because it is needed to obtain substitutivity of the *quiescent* and *fair* trace preorders [22]. Interestingly, the nonblocking condition also has a technical motivation in the timed case, but a different one: it is needed to preserve I/O feasibility. To illustrate this, we consider a trivial example of two I/O feasible timed I/O automata, A and B, whose parallel composition is not I/O feasible.

$$A = IO_{\emptyset}((\mathsf{WAIT}\ 1 \,;\, \overline{tick})^{\omega}) \qquad B = IO_{\emptyset}(\mathsf{STOP}_{\{tick\}})$$

Machine A describes a perfect clock that perform a *tick* action after each unit of time. Machine B does nothing except allow time to pass. Both machines are trivially input enabled (since there are no inputs) and I/O feasible. Still, in the parallel composition $A \| B$, time cannot proceed past 1, because B refuses to synchronize with the output action of A. Time deadlocks like this can be avoided if we do not allow for contexts (environments) that can block output actions. This leads us to the following definition.

A *timed I/O transducer* T is a timed transducer together with a set $inp(T) \subseteq vis(T)$ and, for each m, a set $inp_T(m) \subseteq vis_T(m)$. Analogous to the automaton case, we define derived sets $out(T) \triangleq vis(T) - inp(T)$, $loc(T) \triangleq out(T) \cup \{\tau\}$, and, for each m, $out_T(m) \triangleq vis_T(m) - inp_T(m)$ and $loc_T(m) \triangleq out_T(m) \cup \{\tau\}$. We require

- *Input enabling.* For all states s and for all $a \in inp(T)$, T has a step $s \xrightarrow[\eta]{a} s'$, such that, for all i, $\eta(i) \in inp_T(c\text{-}map_T(i)) \cup \{\perp\}$.

– *No output blocking.* For all states s, all active holes i of s, and all actions $a \in out_T(c\text{-}map_T(i))$, T has a step $s \xrightarrow[\eta]{b} s'$, with $b \in loc(T)$, $\eta(i) = a$, and, for all $i' \neq i$, $\eta(i') \in inp_T(c\text{-}map_T(i')) \cup \{\perp\}$.

By $t\text{-}TRANS(T)$ we denote the timed action transducer that underlies timed I/O transducer T. We say that a step $s' \xrightarrow[\eta]{a} s$ is *under control of* T if $a \in loc(T)$ and for all i, $\eta(i) \in inp_T(c\text{-}map_T(i)) \cup \{\perp\}$.

The conditions of input enabling and no output blocking for timed action transducers are direct translations of similar constraints presented in [22]. An important example of a timed I/O transducer can be obtained by taking the patient version of the composition operation of the I/O automaton model [11]. In fact, this operation can be viewed as a special case of of our parallel composition operator with additional requirements on the action sets of the arguments. These requirements are: (1) all visible actions are singletons, (2) the sets of output actions of different arguments are disjoint. The input actions of the composition are then defined as the union of the input actions of the arguments minus the union of the output actions of the arguments.

Suppose T is a timed I/O transducer. A *timed I/O automaton assignment* for T is a function ζ that maps each color m of T to a timed I/O automaton in such a way that $inp(\zeta(m)) = inp_T(m)$. Let T be a timed I/O transducer and let ζ be a timed I/O automaton assignment for T. Then we define $T(\zeta)$ to be the structure A given by $t\text{-}AUT(A) \triangleq t\text{-}TRANS(T)(t\text{-}TRANS \circ \zeta)$ and $inp(A) \triangleq inp(T)$.

Lemma 5. *Let T be a timed I/O transducer and let ζ be a timed I/O automaton assignment for T. Then $T(\zeta)$ is a timed I/O automaton.*

4.4 Feasibility Revisited

It turns out that I/O feasibility is not preserved by parallel composition. Consider the following two timed I/O automata.[11]

$$\begin{aligned} FAST\_SWITCH \triangleq\ & IO_{\{sw\_on\}}\ (\mathsf{CREATE}(\{OFF, ON\}, \{OFF\}, \\ & OFF \mapsto sw\_on\ .\ ON \\ & ON \mapsto sw\_on\ .\ ON \,\Box\, \overline{sw\_off\ .\ OFF}\ )) \end{aligned}$$

$$\begin{aligned} FAST\_USER \triangleq\ & \overline{IO_{\{sw\_off\}}}\ (\mathsf{CREATE}(\{OFF, ON\}, \{OFF\}, \\ & OFF \mapsto \overline{sw\_on\ .\ ON} \,\Box\, sw\_off\ .\ OFF \\ & ON \mapsto sw\_off\ .\ OFF\ )) \end{aligned}$$

The first expression denotes a fast version of the timed switch of Section 3: each time the switch gets turned on the machine turns it off immediately. The second expression describes a fast user of this switch, who initially turns the switch on, and then, each time it goes off, immediately turns it on again. Both systems are I/O

[11] A similar example is given by Abadi and Lamport [1].

feasible, but their parallel composition is not. The problem arises from the ability of both systems to respond immediately to a given input. One way to avoid this problem is to introduce a *system delay constant*, as in in Timed CSP: a minimum amount of time that has to pass between each pair of locally controlled actions. However, this seems too drastic and artificial to us, and the extra constants would probably complicate the task of analyzing system timing behavior. Fortunately, system delays are not needed to preserve I/O feasibility. Instead the following conditions suffice.

A timed I/O automaton A is *strongly I/O feasible* if it is I/O feasible and time grows unboundedly in any execution that contains infinitely many steps that are under control of A. This generalizes to transducers as follows. A timed I/O transducer T is *I/O feasible* if each finite execution can be extended to an admissible execution via an execution fragment that contains only time-passage steps and steps that are under the control of the transducer. T is *strongly I/O feasible* if in addition:

1. In any execution that contains infinitely many steps that are under control of T, time grows unboundedly.
2. In any execution for which the union over all states of their sets of active holes is infinite, time grows unboundedly.

The second condition is needed to exclude situations in which, each time an input arrives, a new machine is activated to produce an immediate output. We have the following result:

Theorem 6. *Suppose T is a strongly I/O feasible timed I/O transducer and ζ is a timed I/O automaton assignment for T that maps each color of T to a strongly I/O feasible timed I/O automaton. Then $T(\zeta)$ is a strongly I/O feasible timed I/O automaton.*

Note that the timer construct proposed above is *not* strongly I/O feasible, because it does not permit time to pass beyond its upper bound. However, the timed I/O transducer for parallel composition is strongly I/O feasible.

As a topic for future research it remains to find a collection of strongly I/O feasible I/O transducers that is sufficiently expressive to describe implementations of real-time systems. Ideally, these action transducers would be defined in terms of the action transducers of Section 3. Process algebraic and/or assertional proof techniques could then be applied to prove that a system described in terms of feasible I/O transducers implements a system described in terms of the higher-level primitives of Section 3.

References

1. M. Abadi and L. Lamport. An old-fashioned recipe for real time. In *Proceedings of the REX Workshop "Real-Time: Theory in Practice"*, LNCS 600. Springer-Verlag, 1992. To appear.
2. R. Alur and D.L. Dill. Automata for modeling real-time systems. In *Proceedings* 17^{th} *ICALP*, LNCS 443, pages 322–335. Springer-Verlag, July 1990.
3. P. America. POOL-T — A parallel object-oriented language. In A. Yonezawa and M. Tokoro, editors, *Object-Oriented Concurrent Systems*. MIT Press, 1986.

4. J.C.M. Baeten and J.A. Bergstra. Real time process algebra. *Journal of Formal Aspects of Computing Science*, 3(2):142–188, 1991.
5. J.C.M. Baeten and W.P. Weijland. *Process Algebra.* Cambridge Tracts in Theoretical Computer Science 18. Cambridge University Press, 1990.
6. B. Bloom, S. Istrail, and A.R. Meyer. Bisimulation can't be traced: preliminary report. In *Conference Record of the* 15^{th} *ACM Symposium on Principles of Programming Languages*, pages 229–239, 1988.
7. J. Davies and S. Schneider. An introduction to Timed CSP. Technical Monograph PRG-75, Oxford University Computing Laboratory, Programming Research Group, August 1989.
8. C.A.R. Hoare. *Communicating Sequential Processes.* Prentice-Hall International, Englewood Cliffs, 1985.
9. A.S. Klusener. Abstraction in real time process algebra. Report CS-R9144, CWI, Amsterdam, October 1991.
10. K.G. Larsen and L. Xinxin. Compositionality through an operational semantics of contexts. In *Proceedings* 17^{th} *ICALP*, LNCS 443, pages 526–539. Springer-Verlag, July 1990.
11. 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, August 1987. A full version is available as MIT Technical Report MIT/LCS/TR-387.
12. N.A. Lynch and F.W. Vaandrager. Forward and backward simulations – part II: Timing-based systems, 1992. In preparation.
13. N.A. Lynch and F.W. Vaandrager. Forward and backward simulations for timing based systems. *Proceedings of the REX Workshop "Real-Time: Theory in Practice"*, LNCS 600. Springer-Verlag, 1992. To appear.
14. M. Merritt, F. Modugno, and M. Tuttle. Time constrained automata. In *Proceedings CONCUR 91*, LNCS 527, pages 408–423. Springer-Verlag, 1991.
15. R. Milner. *Communication and Concurrency.* Prentice-Hall International, Englewood Cliffs, 1989.
16. F. Moller and C. Tofts. A temporal calculus of communicating systems. In *Proceedings CONCUR 90*, LNCS 458, pages 401–415. Springer-Verlag, 1990.
17. X. Nicollin and J. Sifakis. The algebra of timed processes ATP: Theory and application (revised version). Technical Report RT-C26, LGI-IMAG, Grenoble, France, November 1991.
18. X. Nicollin and J. Sifakis. An overview and synthesis on timed process algebras. In *Proceedings CAV 91*, LNCS 575, pages 376–398. Springer-Verlag, 1992.
19. X. Nicollin, J. Sifakis, and S. Yovine. From ATP to timed graphs and hybrid systems. *Proceedings of the REX Workshop "Real-Time: Theory in Practice"*, LNCS 600. Springer-Verlag, 1992. To appear.
20. G.D. Plotkin. A structural approach to operational semantics. Report DAIMI FN-19, Computer Science Department, Aarhus University, 1981.
21. G.M. Reed and A.W. Roscoe. A timed model for communicating sequential processes. *Theoretical Computer Science*, 58:249–261, 1988.
22. F.W. Vaandrager. On the relationship between process algebra and input/output automata. In *Proceedings* 6^{th} *Annual Symposium on Logic in Computer Science*, pages 387–398. IEEE Computer Society Press, 1991.
23. Wang Yi. Real-time behaviour of asynchronous agents. In *Proceedings CONCUR 90*, LNCS 458, pages 502–520. Springer-Verlag, 1990.

This article was processed using the LaTeX macro package with LLNCS style