Formal Aspects of Computing (1995) 3: 1–000

Action Transducers and Timed Automata[1]

Nancy Lynch[2,4] and Frits Vaandrager[3,5]

[4]MIT, LCS, Cambridge, MA 02139, USA, lynch@theory.lcs.mit.edu
[5]CWI, P.O. Box 94079, 1090 GB Amsterdam, The Netherlands, fritsv@cwi.nl

Keywords: Real-time; process algebra; action transducers; timed automata; timed trace inclusion; congruence properties

Abstract. The *timed automaton model* of [LV92, LV93] 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.

1. Introduction

The *timed automaton model* of [LV92, LV93] 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 [LV92, LV93], we develop a full range of *simulation* proof methods for timed automata; these methods are used in [LLSA93, BPV94, HL94] to verify

[1] A preliminary version of this paper appeared in W.R. Cleaveland, editor, *Proceedings CONCUR'92*, Stony Brook, New York. LNCS 630, pages 436–455. Springer-Verlag, 1992.
[2] Supported by ONR contracts N00014-85-K-0168 and N00014-91-J-1988, by NSF grant CCR-8915206, and by ARPA contracts N00014-89-J-1988 and N00014-92-J-4033.
[3] Supported by ESPRIT BRA 7166 CONCUR2 and by the HCM network EXPRESS. Part of the work on this paper was done while the author was employed by the Ecole des Mines, CMA, Sophia Antipolis, France.
Correspondence and offprint requests to: Frits Vaandrager

the correctness of timed protocols for communication, audio control and real-time process control, respectively. 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.

Timed automata 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 behaviour are considered. First, as the finite behaviours, 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 infinite behaviours, we take the *admissible timed traces*, each of which consists of a sequence of timed visible actions that can occur in an *admissible execution*, i.e., 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 one timed automaton A *implements* another timed automaton B if the sets of finite and admissible timed traces of A are included in the corresponding trace sets of B. Justification for the use of trace inclusions to define "implementation" appears, for example, in the work of Gawlick, Segala, Søgaard-Andersen and Lynch [GSSL94]. Basically, this justification amounts to showing that the set of admissible timed traces of A is not trivial. Doing this depends on a classification of the visible actions of A as *input actions* or *output actions*, as in the I/O automaton model of [LT87]. Then A is required to have the property that each of its finite executions can be extended to an admissible execution in a way that includes any given "non-Zeno" input pattern. Showing that this property holds for a given timed automaton A is an interesting problem, but we do not address this problem in this paper.

In the untimed setting, bisimulation equivalences have been reasonably successful as notions of implementation between transition systems [BW90, Mil89]. Consequently, bisimulation equivalences have also been proposed as implementation relations for the timed setting [BB91, Klu93, MT90, NS94, Yi90]. However, we do not believe that bisimulations will turn out to be very useful as implementation relations in the timed case. The problem is that they do not allow one to abstract in specifications from the often very complex timing behaviour of implementations (see Chapter 10 of [Klu93] for an example).

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.

Action transducers We represent operations by automaton-like objects that we call *action transducers*, rather than, for example, using SOS specifications [Plo81]. 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 the composition can perform an a action when its first argument performs an a-action, while the right transition says that the composition can perform an a action when its second argument does so. 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. In fact, it is shown in [Vaa93] how SOS specifications in a variant of a format proposed by De Simone [Sim85] can be translated to equivalent action transducers, and vice versa.

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 [LX90], 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 model start states in the setting of SOS.

As mentioned above, the action transducers we consider have multiple start states. They also include *holes*, which describe locations for holding argument automata. In fact, our action transducers also allow holes to be *coloured*, which allows us to express the condition that several holes (those of the same colour) must hold copies of the same automaton. The concepts of multiple start states and of coloured holes are not present in [LX90].

Results The 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.

After proving substitutivity for a general class of operations, we describe many examples of specific operations that fall into this class. These include most of the usual untimed operations from process algebra, in particular, sequential and parallel composition, external choice, action hiding and renaming. Other untimed operations included are an interrupt operation similar to those used in Extended LOTOS [Bri88] and CSP [Hoa85], disjoint union, and a binary version of Kleene's star. We also describe several timed operations as timed action transducers: a CLOCK operation directly inspired by the clock variables of [AD94, AH94], a BOUND operation that can block the passage of time, and a RATE operation that can change the speed of its argument. On the other hand, there are several operators that have been proposed in the literature that do not fit our format of action transducers, in particular, the CCS-style choice operation present in [BB91, MT90, NS94, Yi90]. This operation cannot be expressed as a timed action transducer because the timed trace inclusion relation is not substitutive with respect to it.

We briefly consider the design of an appropriate language of operators for describing timing-based systems. Such a language 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. To this end, we describe a simple and general construction, inspired by Nicollin and Sifakis [NS92], 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 use and constrain time explicitly. Of the many possibilities, we would like to identify a small number that can be used for constructing all the others. For this purpose, we tentatively propose our CLOCK, BOUND and RATE operations mentioned above. Using only these operations and untimed operations, we can construct many of the other timed operations appearing in the literature, including a very general timer similar to that used in the timed ω-automata model of Alur and Dill [AD94], the timeout construct of Timed CSP [RR88, DS89], and the execution delay operation of the timed process algebra ATP [NS94]. We can also define a minor variant of Alur and Dill's timed automata [AD94], as well as the finite-state subcase of the timed automaton model of Merritt, Modugno and Tuttle [MMT91]. All of this provides evidence of the power of our proposed language.

The decidability and closure properties of Alur-Dill automata suggest that they can be regarded as a real-time analogue 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 [NSY93] 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 real-time ACP language of Baeten and Bergstra [BB91] is sufficiently expressive to allow for a direct encoding of Alur-Dill automata.

We present our definitions and results for timed systems by first presenting related definitions and results for untimed systems, and then building upon those to obtain the corresponding timed concepts. Thus, byproducts of our results for timed systems include a definition and a substitutivity theorem for untimed action transducers, as well as a demonstration that the most commonly used untimed operations can be expressed as action transducers. These byproducts may be of some interest in themselves.

In summary, we believe that 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 presentation of a large number of interesting operators, timed and untimed, as action transducers, and (4) 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

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 suppress the subscript A where no confusion is likely. Automaton A is called *finite* if all its components are finite sets.

The term *event* will be used to refer to an occurrence of an action in a sequence.

2.2. Executions and Traces

An *execution fragment* of an automaton 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. A state s of A is *reachable* if it is the last state of some finite execution of A.

For $\alpha = s_0 a_1 s_1 a_2 s_2 \cdots$ an execution fragment, $trace(\alpha)$ is defined as the sequence obtained from $a_1 a_2 \cdots$ by removing all τ's. A 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 $A \leq B \triangleq traces(A) \subseteq traces(B)$. Recall that the *kernel* of a preorder $\sqsubseteq$ is the equivalence $\equiv$ defined by $x \equiv y \triangleq x \sqsubseteq y \wedge y \sqsubseteq x$. We denote by $\equiv_*$, $\equiv_\omega$ and $\equiv$, the respective kernels of these preorders.

2.3. Action Transducers

We now define a notion of *action transducer*, as an explicit representation of certain 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 *colours*. Sometimes we will find it useful to make several copies of an argument automaton. To this end an action transducer is equipped with a set of *holes* and a mapping that associates a colour to each hole. The idea is that we plug into each hole a copy of the argument automaton for which the colour 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 colours, and the occurrences of variables serve as holes. As the rest of its "static" description, an action transducer has an associated global set of actions, and, for each colour, a local set of actions.

The "dynamic" part of an action 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 idles or participates in the step, and if it participates, by which action.

2.3.1. Definition

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*,
- a set $colours(T)$ of *colours*,
- for each hole i, a colour $col(T, i)$,
- a set $acts(T)$ of *actions* that includes τ,
- for each colour c, a set $acts(T, c)$ of *actions* that includes τ but excludes the *noaction* symbol 0,
- a set $steps(T) \subseteq states(T) \times acts(T) \times triggers(T) \times states(T)$ of *steps*, where $triggers(T)$ is the set of maps η that assign to each hole i either 0 or an action in $acts(T, col(T, i))$.

We say that hole i *participates in* a step (s', a, η, s) if $\eta(i) \neq 0$; hole i is *active* in s' if it participates in some step starting with s'. For each state s', we define $active(T, s')$ as the set of holes that are active in s'.

We define the sets of *external actions* of T by $ext(T) \triangleq acts(T) - \{\tau\}$, and, for each c, $ext(T,c) \triangleq acts(T,c) - \{\tau\}$. We write $s' \xrightarrow[\eta]{a}_T s$ instead of $(s', a, \eta, s) \in steps(T)$, and suppress the argument T when no confusion is likely. We often represent a trigger η by the set $\{(i, a) \mid \eta(i) = a \neq 0\}$.

2.3.2. Executions and traces

An *execution fragment* of an action transducer T is a finite or infinite alternating sequence $s_0 a_1 \eta_1 s_1 a_2 \eta_2 s_2 \cdots$ 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.

For $\gamma = s_0 a_1 \eta_1 s_1 a_2 \eta_2 s_2 \cdots$ an execution fragment and i a hole, we define

$$\begin{aligned} trace(\gamma) &\triangleq (a_1 a_2 \cdots) \lceil ext(T), \\ trace(\gamma, i) &\triangleq (\eta_1(i) \eta_2(i) \cdots) \lceil ext(T, col(T, i)). \end{aligned}$$

2.3.3. Relation with automata

We view action transducers as a generalisation of automata. Specifically, if A is an automaton, then the associated action transducer $trans(A)$ has the same states, start states and actions as A, empty sets of holes and colours, and its step relation given by:

$$s' \xrightarrow[\eta]{a}_{trans(A)} s \quad \triangleq \quad \eta = \emptyset \wedge s' \xrightarrow{a}_A s.$$

In this way, automata are embedded into the class of action transducers. We will frequently identify an automaton with its corresponding action transducer.

Conversely, if T is an action transducer, then we can define an associated automaton, $aut(T)$. Namely, $aut(T)$ inherits the sets of states, start states and actions of T, and has its step relation defined by

$$s' \xrightarrow{a}_{aut(T)} s \quad \triangleq \quad \exists \eta : s' \xrightarrow[\eta]{a}_T s.$$

It is not hard to see that, for any automaton A, $aut(trans(A)) = A$, and for any action transducer T with an empty set of holes, $trans(aut(T)) = T$.

2.3.4. Combining action transducers and automata

We define the meaning of an action transducer as an operation on automata. First, define an *automaton assignment* for T to be a function ζ that maps each colour c of T to an automaton in such a way that $acts(\zeta(c)) = acts(T, c)$. Suppose ζ is an automaton assignment for T, and let Z be the function that associates an automaton to each hole, by the rule $Z(i) = \zeta(col(T, i))$. Then $T(\zeta)$ is the automaton A given by:

- $states(A) = \{(s, z) \mid s \in states(T),\ z$ maps holes i of T to states of $Z(i)\}$,
- $start(A) = \{(s, z) \mid s \in start(T),\ z$ maps holes i of T to start states of $Z(i)\}$,
- $acts(A) = acts(T)$, and
- $(s', z') \xrightarrow{a}_A (s, z)$ if and only if

$$\exists \eta : s' \xrightarrow[\eta]{a}_T s \wedge \forall i : [\textbf{if } \eta(i){=}0 \textbf{ then } z'(i){=}z(i) \textbf{ else } z'(i) \xrightarrow{\eta(i)}_{Z(i)} z(i)].$$

Thus, the steps of the automaton $T(\zeta)$ are just those that are allowed by the action transducer T, using triggers that describe steps allowed by the automata in the holes.

It is useful to have explicit terminology for the sequence of triggers that are used to justify the steps in an execution of $T(\zeta)$. Thus, suppose that

$$\alpha \quad = \quad (s_0, z_0)a_1(s_1, z_1)a_2(s_2, z_2)\cdots$$

is an execution of $T(\zeta)$. Suppose that for each hole i and each $j \geq 1$, $s_{j-1} \xrightarrow[\eta_j]{a_j}_T s_j$ and if $\eta_j(i) = 0$ then $z_{j-1}(i) = z_j(i)$ else $z_{j-1}(i) \xrightarrow{\eta_j(i)}_{Z(i)} z_j(i)$. Then we say that the sequence $\eta_1\eta_2\cdots$ is a *trigger sequence* for α. By definition of $T(\zeta)$ every execution has at least one trigger sequence (there may be more than one).

Lemma 2.1. Suppose that T is an action transducer, ζ is an automaton assignment for T, and $\alpha = (s_0, z_0)a_1(s_1, z_1)a_2(s_2, z_2)\cdots$ is an execution of $T(\zeta)$ with trigger sequence $\eta = \eta_1\eta_2\cdots$. Then $\gamma = s_0 a_1 \eta_1 s_1 a_2 \eta_2 s_2 \cdots$ is an execution of T, and for each hole i of T, $trace(\gamma, i) \in traces(\zeta(col(T, i)))$.

2.3.5. Remarks

The importance of action transducers for process algebra and concurrency theory was first noted by Larsen and Xinxin [LX90], who introduced a certain type of action transducer, which they call *context systems*, to study compositionality questions in the setting of process algebra. Our action transducers generalise those of Larsen and Xinxin [LX90] in several respects: the distinction between colours and holes, which allows us to copy arguments, is new here. Also, Larsen and Xinxin [LX90] only consider operations with a finite number of arguments, and a setting where automata just have one start state and no explicit set of associated actions.

Note that, since we always start copies of an argument automaton from a start state, our notion of copying is different from that of Bloom, Istrail and Meyer [BIM88], who 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 [BIM88].

In this section we have defined the semantics of an action transducer as an operation on automata. In fact, it is often useful to interpret action transducers in a more general (and somewhat more complex) way, as operations on action transducers. We leave this generalisation to the reader.

2.4. Substitutivity

We say that 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 $\mathcal{A}$,

$$\forall c \in colours(T) : \zeta(c)\ R\ \zeta'(c) \quad \Rightarrow \quad T(\zeta)\ R\ T(\zeta').$$

In this subsection we present two substitutivity results for untimed action transducers. These results depend on certain additional assumptions involving the internal steps of the arguments. We express these assumptions in the following definition of the subclass of τ-*respecting* action transducers. Then we show that $\leq_*$ and $\leq$ are substitutive for all action transducers in this class.

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. The only steps with τ in the range of the trigger are clearing steps, i.e., 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., $s' \xrightarrow[\eta]{a} s$ implies that $\{i \mid \eta(i) \neq 0\}$ is finite.

Condition 1 says that the action transducer must permit the component automata to take internal steps, by means of special clearing steps of the action transducer, whereas Condition 2 says that clearing steps are the only steps of the action transducer that permit internal steps of the components. Condition 3 does not explicitly mention internal steps; however, this condition is needed in the substitutivity proof because of complications caused by internal steps. Conditions 1 and 2 slightly strengthen similar constraints that are presented in [Vaa91] in the setting of SOS. Condition 3 does not occur in [Vaa91] because there only operations with a finite number of arguments are considered. However, a similar constraint appears in the I/O automaton model of [LT87].

Theorem 2.2. The relations $\leq_*$ and $\leq$ on automata are substitutive for all τ-respecting action transducers.

Proof Let T be a τ-respecting action transducer. We show that $\leq$ is substitutive for T. The proof that $\leq_*$ is substitutive for T is similar but slightly simpler.

Suppose ζ, ζ' are automaton assignments for T such that, for all c, $\zeta(c) \leq \zeta'(c)$, and suppose that $\beta \in traces(T(\zeta))$. We must prove that $\beta \in traces(T(\zeta'))$. For this, define $Z \triangleq \lambda i.\zeta(col(T,i))$ and $Z' \triangleq \lambda i.\zeta'(col(T,i))$. Then $Z(i) \leq Z'(i)$ for each hole i.

Since $\beta \in traces(T(\zeta))$, $T(\zeta)$ has a (possibly finite) execution

$$\alpha \quad = \quad (s_0, z_0)a_1(s_1, z_1)a_2(s_2, z_2)\cdots$$

with $trace(\alpha) = \beta$. Let $\eta_1\eta_2\cdots$ be a trigger sequence for α, and let

$$\gamma \quad = \quad s_0 a_1 \eta_1 s_1 a_2 \eta_2 s_2 \cdots.$$

By Lemma 2.1, γ is an execution of T, and $\beta_i \triangleq trace(\gamma, i) \in traces(Z(i))$, for all i. Since $Z(i) \leq Z'(i)$, we obtain $\beta_i \in traces(Z'(i))$, for all i. Therefore for each i, $Z'(i)$ has an execution α_i with $trace(\alpha_i) = \beta_i$. Let γ_0 be the sequence obtained from γ by removing all clearing steps. Then γ_0 is a execution of T and $trace(\gamma_0) = \beta$.

Informally speaking, our job is to "paste" together γ_0 and the α_i to obtain an execution of $T(\zeta')$. We construct an automaton A that describes several allowable ways to do this pasting and that generates executions of $T(\zeta')$ with the required properties. The set of states of A consists of all valuations of the following state variables in their domains:

- a variable *frag* ranging over the set of execution fragments of T. This variable denotes the part of γ_0 that still has to be dealt with. The initial value of *frag* is γ_0.
- for each hole i, a variable $frag_i$ ranging over execution fragments of $Z'(i)$. This variable denotes the part of α_i that still has to be pasted together with (the remainder of) γ_0. The initial value of $frag_i$ is α_i.
- a variable *exec* ranging over finite executions of $T(\zeta')$. The limit of the values of *exec* will be the execution of $T(\zeta')$ in which we are interested. The initial value of *exec* is the trivial execution consisting of the state composed from the first states of γ_0 and the first states of the α_i.

Automaton A has actions *CLEARING* and *BASIC*, which correspond to the two different types of actions of $T(\zeta')$: clearing steps, and "basic" steps. The transitions of A are defined using precondition/effect style in Figure 1[4]. The

[4] Here and elsewhere we use Lamport's [Lam94] list notation for conjunction. In this notation the formula $b_1 \wedge b_2 \cdots \wedge b_n$ is written as the aligned list
$$\begin{array}{ll} \wedge & b_1 \\ \wedge & b_2 \\ & \vdots \\ \wedge & b_n \end{array}$$

BASIC
Precondition
$\wedge$ *frag* begins with $sa\eta$
$\wedge$ for all holes i that participate in the first step of *frag*:
$frag_i$ begins with an $\eta(i)$ step
Effect
remove the first step from *frag*;
for each hole i that participates in the first step of *frag* **do**
remove the first step from $frag_i$;
append to *exec* an a followed by the state of $T(\zeta')$ composed
from the first states of *frag* and all the $frag_i$

CLEARING
Precondition
$\wedge$ *frag* contains at least one step
$\wedge$ hole i_0 participates in the first step of *frag*
$\wedge$ $frag_{i_0}$ begins with a τ step
Effect
remove the first step from $frag_{i_0}$;
append to *exec* a τ followed by the state of $T(\zeta')$ composed
from the first states of *frag* and all the $frag_i$

Fig. 1. Algorithm for pasting together γ_0 and the α_i.

intuition is that, while building an execution of $T(\zeta')$, automaton A peels off initial steps of γ_0 and the α_i. If the remainder of γ_0 starts with an a step and, for each hole i that participate in this step, the remainder of α_i starts with the action required for hole i, then A can perform a *BASIC* step. If, for some hole i, the remainder α_i starts with a τ step then A can perform a corresponding *CLEARING* step, provided that i participates in the next step of γ_0.

We leave it to the reader to check that the definition of A is type correct, in the sense that each variable is only assigned values in its domain.

Pick an arbitrary maximal execution $\delta = u_0 b_1 u_1 b_2 u_2 \cdots$ of A. Since the only way that *exec* is modified is by appending values, we can define α' as the limit of the values of *exec* along δ. By construction, α' is an execution of $T(\zeta')$. We claim that $trace(\alpha') = \beta$.

In order to see this, we first establish that A satisfies the following invariant properties. Here we write $u.v$ for the value of state variable v of A in state u.

1. For all reachable states u, $trace(u.exec)\ trace(u.frag) = \beta$.
2. For all reachable states u and for all holes i, $trace(u.frag_i) = trace(u.frag, i)$.

Proof By simple inductive arguments. ■

Using Invariants 1 and 2, we next prove two claims.

Claim 1. Suppose u is a reachable state of A and $u.frag$ is not a single state execution fragment. Then u has an outgoing step.

Proof Let $s' \xrightarrow[\eta]{a} s$ be the first step of $u.frag$. If, for some hole i that participates in this first step, $u.frag_i$ begins with a τ-step, then a *CLEARING* action is enabled in u. If, for no hole i that participates in the first step, $frag_i$ starts with a τ step, then it follows by Invariant 2 that, for each of these holes i, $frag_i$ starts with an $\eta(i)$ step. But this means that a *BASIC* action is enabled in u. ■

Claim 2. Execution δ has no infinite suffix that consists of *CLEARING* steps only.

Proof Suppose that starting from some state u_n, execution δ consists entirely of *CLEARING* steps. That is, from u_n onwards all the steps of δ simulate τ steps of components that participate in the first step of $u_n.frag$. Because T is τ-respecting, there are only finitely many such participants. Consider any individual participant i. By Invariant 2, $u_n.frag_i$ contains an $\eta(i)$ step after finitely many τ steps. Therefore, only finitely many *CLEARING* steps in δ correspond to τ steps of i. Thus, δ contains only a finite number of consecutive *CLEARING* starting from u_n, a contradiction. ■

Now we return to the proof that $trace(\alpha') = \beta$. Again we consider cases.

1. Suppose δ contains only finitely many *BASIC* actions. By Claim 2, execution δ does not have an infinite suffix that consists of *CLEARING* steps only, so δ is finite.[5] Suppose u_n is the final state of δ. Then, by Claim 1, $u_n.frag$ consists of a single state execution fragment. In combination with Invariant 1, this gives $trace(u_n.exec) = \beta$. But α' is defined as the limit of $u_n.exec$, so $\alpha' = u_n.exec$. Hence $trace(\alpha') = \beta$.
2. Suppose δ contains infinitely many *BASIC* actions. Since *frag* is initially γ_0, and each *BASIC* step removes a step from γ_0, it follows that γ_0 is infinite. By Invariant 1, $trace(u_j.exec)$ is a prefix of β for each j. Since each step of γ_0 is eventually simulated in α', $trace(\alpha') = \beta$.

Hence, $\beta \in t\text{-}traces(T(\zeta'))$, as required. This completes the proof of the theorem. ■

In Section 3.4, we give an example to show that $\leq_\omega$ is not substitutive, even for τ-respecting action transducers. The converse of Theorem 2.2 does not hold: there are many examples of non-τ-respecting action transducers for which $\leq_*$ and $\leq$ are substitutive. We give one example in Section 3.4.

3. An Untimed Process Algebra

In this section, we give several examples of operations that can be expressed as action transducers; all these operations are directly inspired by operations from well-known "untimed" process algebras such as CSP [Hoa85], CIRCAL [Mil85], CCS [Mil89], Extended LOTOS [Bri88] and ACP [BW90]. Our motivation for presenting these examples is twofold: first, they serve as an illustration of how familiar process algebraic operations can be defined using action transducers, and second, the resulting language $\mathcal{L}_u$ will form the basis of a timed process algebra that we will define in Section 5.

3.1. Preliminaries

We first describe a number of conventions so that, in most cases, we do not have to specify the static part of action transducers explicitly. To begin with,

[5] At this point the proof for $\leq_*$ is simpler, since all infinite executions can be excluded trivially.

we adopt the convention that, unless otherwise specified, the sets of holes and colours are the same, and the colouring function is the identity. Often, the set of colours will be an initial fragment $\{1, \ldots, n\}$ of the natural numbers. In this case we write $T(A_1, \ldots, A_n)$ for $T(\lambda c.A_c)$. We also use infix notation in the case of binary operations. All action transducers that we define are parameterised by the action sets of their arguments. Some of the action transducers also have other parameters. Unless stated otherwise, the (global) action set of an action transducer can be obtained by taking the set of all actions that occur in steps of the action transducer.

We find it convenient to structure external actions as nonempty finite sets of *labels*, and to identify τ with the empty set of labels. This will permit a component automaton to perform several activities (labels) together, which the action transducer can handle separately. For instance, the sequential composition action transducer, described below, takes advantage of composite actions: a component can perform an arbitrary label simultaneously with a termination label, and the action transducer handles these two in different ways. The idea to choose sets of labels as the structure of actions was first introduced in CIRCAL, but is used in other algebras as well, for instance in Extended LOTOS. Typically, the generalisation to multiple label actions increases the expressive power of a process algebra.

We regard non-composite external actions as a special case of composite actions, identifying the non-composite action a with the set $\{a\}$. For each action transducer T we define $labels(T) \triangleq \bigcup ext(T)$. Similarly we define, for each colour c, $labels(T, c) \triangleq \bigcup ext(T, c)$.

In our language we assume a special label $\surd$ to indicate *successful termination* and to transfer control to a subsequent process. Symbol $\surd$ is in the label set of all action transducers in the language as well as in the label sets of all their colours. The language has been designed such that any (closed) expression denotes an automaton in which no further transitions are possible after a transition whose label contains $\surd$.

3.2. Operators

3.2.1. Actions

For any finite set a of labels with $\surd \notin a$, we introduce an action transducer a. This action transducer performs the composite action consisting of a together with the termination label $\surd$, and then halts. The action transducer has two states s and t: it starts in s, performs action $a \cup \{\surd\}$, and then terminates in t:

$$s \xrightarrow[\emptyset]{a \cup \{\surd\}} t$$

By the correspondence described earlier, action transducer a can equally well be regarded as an automaton.

3.2.2. Sequential composition

Transducer ";" describes the binary *sequential composition* operation of Extended LOTOS. The action transducer has two states s_1 and s_2. In the start state s_1, the action transducer runs its first argument up to successful termination, and then in state s_2 the action transducer runs its second argument. The steps are (for all actions a, b of the first and second argument, respectively):

$$s_1 \xrightarrow[\{(1,a)\}]{a} s_1 \quad \text{if } \surd \notin a$$
$$s_1 \xrightarrow[\{(1,a)\}]{a-\{\surd\}} s_2 \quad \text{if } \surd \in a$$
$$s_2 \xrightarrow[\{(2,b)\}]{b} s_2$$

Note that, unlike in ACP, $a \; ; \tau$ is different from a (for $a \neq \tau$), because in the second automaton successful termination occurs simultaneously with a whereas in the first automaton it occurs after the a.

3.2.3. External choice

The *external choice* operation $\Box$ is taken from CSP. This operation, which is parameterised by a finite index set I, waits for the first external action of any of its arguments and then runs that argument. The action transducer has distinct states s_i, for each $i \in I$, plus an additional state s, which is the start state. The steps are (for all i and all actions a of the i-th argument):

$$s \xrightarrow[\{(i,\tau)\}]{\tau} s$$
$$s \xrightarrow[\{(i,a)\}]{a} s_i \quad \text{if } a \neq \tau$$
$$s_i \xrightarrow[\{(i,a)\}]{a} s_i$$

We write STOP for external choice over an empty index set. STOP is the simplest action transducer from our language. It has no holes, no colours, no steps, a single state, a single action $\surd$, and no steps. STOP represents the inactive agent, capable of no action whatsoever.

3.2.4. Disjoint union

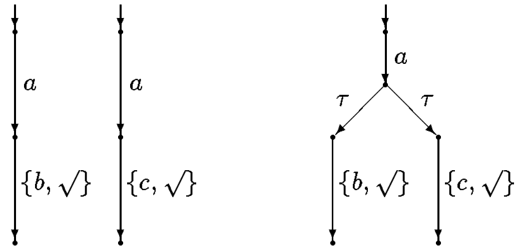
Parameterised by a finite index set I, action transducer $\sqcup$ takes the *disjoint union* of automata indexed by I. The $\sqcup$ construct exploits the feature of multiple start states. For each $i \in I$, the action transducer has a distinct state s_i, which is also a start state, and steps (for all actions a of the i-th argument):

$$s_i \xrightarrow[\{(i,a)\}]{a} s_i$$

Operation $\sqcup$ behaves in a similar way to the *internal choice operation* $\sqcap$ of CSP: it runs one, nondeterministically chosen argument. An interesting difference between the operational semantics of $\sqcup$ and $\sqcap$ is that in $a \; ; (b \sqcup c)$ the choice between b and c is made before execution of the a, whereas in $a \; ; (b \sqcap c)$ this choice is made after the a has been done. This becomes apparent from the automata for these expressions, which are displayed in Figure 2. Modulo trace equivalence, the differences between the two operations disappear: for all automata A and B, $A \sqcup B \equiv A \sqcap B$.

3.2.5. Relabeling

For each function f on labels such that $f(l) = \surd$ iff $l = \surd$, we introduce a unary *relabeling* operation f that renames actions of its argument according to f. The



$a\ ;\ (b \sqcup c)$ $\qquad$ $a\ ;\ (b \sqcap c)$

Fig. 2. The difference between $\sqcup$ and $\sqcap$.

action transducer has a single state s, which is the start state, and steps (for all actions a of the argument, and with f lifted to sets of labels):

$$s \xrightarrow[\{(1,a)\}]{f(a)} s$$

3.2.6. Parallel composition

The binary action transducer $||$, which describes the binary operation of *parallel composition*, is a slight variant of the *dot* operation of CIRCAL. The operation $||$ generalises the usual definition of composition, taking into account the composite nature of actions. In the case where all actions of the arguments are singletons or τ, the operator behaves just as the composition operator of CSP and the I/O automata model. The additional power of our composition operator is used in the proof of Theorem 3.1 and is indispensable in the timed extension of $\mathcal{L}_u$ in Section 5, where actions do not only contain synchronisation labels but also labels expressing timing constraints.

The action transducer $||$ has a single state s, which is the start state, and steps (for all actions a, b of the first and second argument, respectively):

$$\begin{array}{rl} s \xrightarrow[\{(1,a)\}]{a} s & \text{if } a \cap \mathit{labels}(||, 2) = \emptyset \\ s \xrightarrow[\{(2,b)\}]{b} s & \text{if } b \cap \mathit{labels}(||, 1) = \emptyset \\ s \xrightarrow[\{(1,a),(2,b)\}]{a \cup b} s & \text{if } a \cap \mathit{labels}(||, 2) = b \cap \mathit{labels}(||, 1) \neq \emptyset \end{array}$$

The restriction to nonempty sets of labels in the last step is not present in CIRCAL. There, independent actions from different components may occur simultaneously without synchronisation. We have excluded such behavior here in order to keep our composition operation compatible with the one of CSP and the I/O automata model.

When specifying systems it is often convenient to use a derived operator $||_H$ that only requires its arguments to synchronise on a set of labels $H \cup \{\surd\}$. Suppose A and B are automata with label sets L_A and L_B, respectively, and suppose $H \subseteq L_A \cap L_B$ is a set of non-$\surd$ labels. We define

$$A ||_H B \quad \triangleq \quad \mathit{Untag}(\mathit{Tag}_1(A) || \mathit{Tag}_2(B)),$$

where *Untag* and Tag_i $(i = 1, 2)$ are relabeling functions given by:

$$Tag_i(l) \triangleq \begin{cases} l_i & \text{if } l \in (L_A \cap L_B) - H \\ l & \text{otherwise} \end{cases}$$

$$Untag(l) \triangleq \begin{cases} k & \text{if } l := k_j, k \in (L_A \cap L_B) - H, j \in \{1, 2\} \\ l & \text{otherwise} \end{cases}$$

The idea behind this definition is that first the functions tag_1 and tag_2 rename those labels of A and B on which we do not want to synchronise so that they are distinct. Then after the resulting automata have been composed in parallel, the function *Untag* renames the tagged labels back to what they were originally.

Note that the $||$ and $||_H$ operators are commutative and associative.

3.2.7. Hiding

The unary *hiding* operation $\backslash L$ hides all elements from a set L of labels by removing them from all actions of its argument. The action transducer has a single state s, which is the start state, and steps (for all a):

$$s \xrightarrow[\{(1,a)\}]{a-L} s$$

3.2.8. Interrupts

The binary action transducer $\wedge$ is very similar to the *disruption composition* of Extended LOTOS and the *interrupt* operation of CSP. The action transducer has three states s_1, s_2 and t. In start state s_1, the action transducer runs its first argument until the second argument performs an external action; if and when this occurs, the action transducer moves to state s_2 in which the first argument is disabled and the second argument takes over. If in state s_1 the first argument terminates successfully, the action transducer moves to the termination state t. The steps are (for all actions a, b of the first and second argument, respectively):

$$\begin{array}{llll} s_1 \xrightarrow[\{(1,a)\}]{a} s_1 & \text{if } \surd \notin a & s_1 \xrightarrow[\{(2,\tau)\}]{\tau} s_1 & \\ s_1 \xrightarrow[\{(1,a)\}]{a} t & \text{if } \surd \in a & s_1 \xrightarrow[\{(2,b)\}]{b} s_2 & \text{if } b \neq \tau \\ & & s_2 \xrightarrow[\{(2,b)\}]{b} s_2 & \end{array}$$

3.2.9. Iteration

We introduce iteration in our language by means of a binary version of Kleene's star operator: $A * B$ is the automaton that chooses between A and B, and upon successful termination of A has this choice again. A key identity satisfied by the operator is

$$A * B \quad \equiv \quad A \, ; (A * B) \, \Box \, B.$$

Kleene's star operation is best known in its unary form, but in fact the original operator introduced by Kleene in [Kle56] was binary. Recently, the binary star has been studied in the context of ACP in [BBP94, FZ94].

The iteration construct exploits the ability of action transducers to copy their arguments: it uses an infinite number of copies of both the first and the second argument. Formally, the action transducer has colours $\{1,2\}$, holes $\{1,2,\ldots\} \cup \{1',2',\ldots\}$, and a colouring function that, for $i \in \mathsf{N}^+$, maps hole i to colour 1 and hole i' to colour 2. The action transducer has states $\{s_i, l_i, r_i \mid i \in \mathsf{N}^+\}$. In state s_i, the action transducer chooses between execution of the i-th copy of the first argument or execution of the i-th copy of the second argument. In state l_i, the action transducer is running the i-th copy of the first argument, and in state r_i the action transducer runs the i-th copy of the second argument. The initial state is s_1, and the steps are (for all actions a and b of the first and second argument, respectively):

$$
\begin{array}{llll}
s_i \xrightarrow[\{(i,a)\}]{a} l_i & \text{if } \surd \notin a \neq \tau & l_i \xrightarrow[\{(i,a)\}]{a} l_i & \text{if } \surd \notin a \\
s_i \xrightarrow[\{(i,a)\}]{a-\{\surd\}} s_{i+1} & \text{if } \surd \in a & l_i \xrightarrow[\{(i,a)\}]{a-\{\surd\}} s_{i+1} & \text{if } \surd \in a \\
s_i \xrightarrow[\{(i',b)\}]{b} r_i & \text{if } b \neq \tau & r_i \xrightarrow[\{(i',b)\}]{b} r_i & \\
s_i \xrightarrow[\{(i,\tau)\}]{\tau} s_i & & s_i \xrightarrow[\{(i',\tau)\}]{\tau} s_i &
\end{array}
$$

Using the $*$ operator, we can easily define the unary *looping* operator ω, which restarts its argument upon each successful termination:

$$A^\omega \quad \triangleq \quad A * \mathsf{STOP}.$$

Despite what the notation might suggest, operator ω does not run A a finite number of times and then stop! In a choice context the STOP process should be viewed as the absence of an alternative: each time the action transducer $*$ is faced with a choice between A and STOP, it must choose the A.

As an example of the iteration and looping constructs, consider the following expression, which describes an automatic switch-off mechanism:

$$SWITCH \quad \triangleq \quad (sw\_on \; ; (sw\_on * sw\_off))^\omega.$$

The system allows the environment to switch on a lamp at any time by pushing some button; once the lamp has been switched on, it will remain on, even if the button is pushed again, until it is switched off by the system. In Section 5, we will come back to this example and show how we can add real-time constraints to make it more interesting.

3.3. Expressivity of $\mathcal{L}_u$

We define $\mathcal{L}_u$ to be the language consisting of all (closed) expressions built with the operations of Section 3.2. Since all the corresponding action transducers are τ-respecting, it follows from Theorem 2.2 that the preorders $\leq_*$ and $\leq$ are substitutive for all the operations in $\mathcal{L}_u$.

The automata denoted by expressions in $\mathcal{L}_u$ are always acyclic but need not be finite. In particular, each nontrivial use of the iteration construct gives rise to an automaton with an infinite number of reachable states. However, under the condition that no $\sqcup$ occurs within the first argument of a $*$-operator, each expression in $\mathcal{L}_u$ has a tree unfolding which is isomorphic to the tree unfolding of a finite automaton. In the case of expressions where $\sqcup$ occurs within the first argument of a $*$-operator, the underlying automaton will still be trace equivalent to a finite automaton, but no longer "tree equivalent" (consider the automaton denoted by $(a \sqcup b)^\omega$: this automaton has infinitely many start states, one for each infinite sequence over $\{a,b\}$). All automata denoted by $\mathcal{L}_u$-expressions further have the property that after a transition with a label containing $\surd$, no further steps are possible. The following theorem states that $\mathcal{L}_u$ is universally expressive for the class of finite automata with this property. In the proof of this result all operators of the language play a role.

Theorem 3.1. Suppose that A is a finite automaton in which no further steps are possible after a transition whose label contains $\surd$. Then the tree unfolding of A is isomorphic to the tree unfolding of the automaton associated to some expression in $\mathcal{L}_u$.

Proof (Sketch) Without loss of generality, we may assume that A only has a single start state: any finite automaton with $n > 1$ start states is tree equivalent to the disjoint union of n copies of this automaton in which the set of start states is restricted to a singleton.

Also without loss of generality, we may assume that A has no self-loops, i.e., steps of the form $s \xrightarrow{a} s$: for each finite automaton with such self-loops one can construct an equivalent finite automaton without them, for instance by adding a boolean "history variable" that records whether the number of transitions thus far is even.

Let $states(A) = \{s_0, \ldots, s_n\}$, let $start(A) = \{s_0\}$, and let S be short for $steps(A)$. In the $\mathcal{L}_u$-expression that encodes A, we use elements of S as auxiliary labels. The expression is

$$(((X_0 \,\|_{S_1} X_1) \,\|_{S_2} X_2) \cdots \|_{S_n} X_n) \backslash S,$$

where, for $i > 0$,

$$\begin{aligned}
S_i &\triangleq \text{the set of all steps between } s_i \text{ and states in } \{s_0, ..., s_{i-1}\} \\
&\quad \cup \text{ the set of all steps } t \text{ with } \surd \in action(t) \\
X_0 &\triangleq ((non_final_step_0 \;;\; wait_0)\,^*\, final_step_0) \wedge final_step_other_0, \\
X_i &\triangleq [wait_i \;;\; ((non_final_step_i \;;\; wait_i)\,^*\, final_step_i)] \wedge final_step_other_i,
\end{aligned}$$

where, for $i \geq 0$,

$$\begin{aligned}
wait_i &\triangleq \square_{\{t \in S \mid target(t) = s_i \wedge \surd \notin action(t)\}} \{t\}, \\
non_final_step_i &\triangleq \square_{\{t \in S \mid source(t) = s_i \wedge \surd \notin action(t)\}} \{t\} \cup action(t), \\
final_step_i &\triangleq \square_{\{t \in S \mid source(t) = s_i \wedge \surd \in action(t)\}} \{t\} \cup (action(t) - \{\surd\}), \\
final_step_other_i &\triangleq \square_{\{t \in S \mid source(t) \neq s_i \wedge \surd \in action(t)\}} \{t\}.
\end{aligned}$$

■

3.4. Counterexamples

An example of an operation for which $\leq_*$ is not substitutive is parallel composition over an infinite index set I. We have $a \equiv_* \tau \;;\; a$ but not

$$\|_{i \in I}(a) \leq_* \|_{i \in I}(\tau \;;\; a),$$

because the automaton on the left has a trace a, which the automaton on the right does not have, since it has do an infinite number of τ-actions "first". Another example is the version of (binary) parallel composition obtained by requiring the argument automata to synchronise on τ. Here one loses substitutivity since $a \equiv_* \tau ; a$ but not $a||a \leq_* (\tau ; a)||a$, because the automaton on the left has a trace a, which the automaton on the right does not have, since the initial τ-action of the first argument cannot synchronise with a τ-action of the second argument. Note that neither of these two examples is τ-respecting.

It is not the case that preorder $\leq_\omega$ is substitutive for all τ-respecting action transducers. For instance, we have $\tau \equiv_\omega \mathsf{STOP}$ but not $\tau ; a^\omega \leq_\omega \mathsf{STOP} ; a^\omega$.

As an example of a non-τ-respecting action transducer for which $\leq_*$ and $\leq$ are substitutive, consider the choice operation $+$ from CCS. The action transducer for this operation can be obtained by removing all clearing steps from the initial state of the action transducer for $\Box$, and instead allowing a to range over τ in the second equation as well, so that τ-steps can force the choice. The resulting action transducer is clearly not τ-respecting. In Section 5.3, we will show that the timed trace preorders are not substitutive for the timed version of the CCS choice operation.

4. The Timed Setting

Now we extend the notions described in Section 2 to the case of timed systems. We follow the same general outline, introducing time systematically into all of the definitions and results.

4.1. Timed Automata

We use a slight variant of the *timed automaton* model from [LV93].[6] A *timed automaton* A is an automaton whose set of actions includes R^+, the set of positive reals. Actions from R^+ are referred to as *time-passage actions*. We let $d, d', \ldots$ range over R^+ and, more generally, $t, t', \ldots$ 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 $s' \xrightarrow{d+d'} s$.

For the second axiom, an auxiliary definition is needed. A *trajectory* for a step $s' \xrightarrow{d} s$ is a function $w : [0, d] \rightarrow states(A)$ such that $w(0) = s'$, $w(d) = s$, and

$$w(t) \xrightarrow{t'-t} w(t') \text{ for all } t, t' \in [0, d] \text{ with } t < t'.$$

Now we can state the second axiom.

S2 Each step $s \xrightarrow{d} s'$ has a trajectory.

Axiom **S1** gives a natural property of time, namely that if time can pass in two steps, then it can also pass in a single step. The *trajectory axiom* **S2** is a kind of converse to **S1**; it says that any time-passage step can be "filled in" with states for each intervening time, in a "consistent" way. For a further discussion of this axiom we refer to [LV93, JSV93].

[6] 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.

4.2. Timed Traces

Executions of timed automata correspond to what are called *sampling computations* in [MP93]: they provide information about a run of a system at a countable number of points in time. In [LV93], a notion of *timed execution* is also defined for timed automata: these are alternating sequences of trajectories and actions, which correspond to the *super-dense computations* of [MP93]. It can be argued that timed executions provide a more precise representation of the behaviour of real-time systems than (sampling) executions. However, our trajectory axiom **S2** guarantees that for each (sampling) execution of a timed automaton there exists a corresponding timed execution. This means that the full externally visible behaviour of timed automata can already be inferred from the technically much simpler (sampling) executions. This is done in the following definitions.

Suppose $\alpha = s_0 a_1 s_1 a_2 s_2 \cdots$ is an execution fragment of a timed automaton A. For each index j, let t_j be given by

$$\begin{array}{rcl} t_0 & = & 0, \\ t_{j+1} & = & \textbf{if } a_{j+1} \in \mathsf{R}^+ \textbf{ then } t_j + a_{j+1} \textbf{ else } t_j. \end{array}$$

The *limit time* of α, notation $ltime(\alpha)$, is the smallest element of $\mathsf{R}^{\geq 0} \cup \{\infty\}$ larger than or equal to all the t_j, i.e., we define $ltime(\alpha) \triangleq \sup_j(t_j)$. We say α is *admissible* if $ltime(\alpha) = \infty$, and *Zeno* if it is an infinite sequence but with a finite limit time. The *timed trace* $t\text{-}trace(\alpha)$ associated with α is defined by

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

Thus, $t\text{-}trace(\alpha)$ records the visible actions of α paired with their times of occurrence, as well as the limit time of the execution.

A pair β 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. 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 finite executions, and $t\text{-}traces^\infty(A)$ for the *admissible* timed traces, i.e., those that originate from admissible executions. These notions induce three preorders on timed automata: $A \leq^{\mathsf{t}} B \triangleq t\text{-}traces(A) \subseteq t\text{-}traces(B)$, $A \leq^{\mathsf{t}}_* B \triangleq t\text{-}traces^*(A) \subseteq t\text{-}traces^*(B)$, and $A \leq^{\mathsf{t}}_\infty B \triangleq t\text{-}traces^\infty(A) \subseteq t\text{-}traces^\infty(B)$. The kernels of these preorders are denoted by $\equiv^{\mathsf{t}}$, $\equiv^{\mathsf{t}}_*$ and $\equiv^{\mathsf{t}}_\infty$, respectively.

A *timed sequence* over a given alphabet K is a (finite or infinite) sequence δ over $K \times \mathsf{R}^{\geq 0}$ in which the time components are nondecreasing, i.e., $t \leq t'$ if (k, t) and (k', t') are consecutive elements in δ. A *timed sequence pair* over K is a pair $\beta = (\delta, t)$, where δ is a timed sequence over K and $t \in \mathsf{R}^{\geq 0} \cup \{\infty\}$, such that t is greater or equal than all time components in δ. We say that β is *finite* if δ is a finite sequence and $t < \infty$.

Clearly, all timed traces of a timed automaton A are timed sequence pairs over $ext(A)$. In particular, all finite timed traces are finite timed sequence pairs.

Suppose β and β' are timed sequence pairs such that β is finite. Let

$$\begin{array}{rcl} \beta & = & ((k_1, t_1)(k_2, t_2) \cdots (k_n, t_n), t), \\ \beta' & = & ((k'_1, t'_1)(k'_2, t'_2) \cdots, t'). \end{array}$$

Then we define $\beta \,;\, \beta'$ to be the timed sequence pair

$$((k_1, t_1)(k_2, t_2) \cdots (k_n, t_n)(k'_1, t + t'_1)(k'_2, t + t'_2) \cdots, t + t').$$

If β and β' are timed sequence pairs then β is a *prefix* of β', notation $\beta \leq \beta'$, if either $\beta = \beta'$, or β is finite and there exists a timed sequence pair β'' such that $\beta' = \beta \,;\, \beta''$.

4.3. Timed Action Transducers

In this section we introduce the notion of a timed action transducer, define what are the timed traces of a timed action transducer, and show how timed action transducers define operations on timed automata.

4.3.1. Definition

A *timed action transducer* T is an action transducer with $acts(T) \supseteq \mathsf{R}^+$ and, for all colours c, $acts(T,c) \supseteq \mathsf{R}^+$. The sets of *visible* actions are defined by $vis(T) \triangleq ext(T) - \mathsf{R}^+$ and, for all c, $vis(T,c) \triangleq ext(T,c) - \mathsf{R}^+$.

We assume that T satisfies five axioms.

T1 If $s' \xrightarrow[\eta]{a} s$ and $\eta(i) \in \mathsf{R}^+$, then $a \in \mathsf{R}^+$.

T2 If $s' \xrightarrow[\eta]{d} s$ and $i \in active(T, s')$, then $\eta(i) \in \mathsf{R}^+$.

T3 If $s' \xrightarrow[\eta]{d} s$ then $active(T, s') = active(T, s)$.

T4 If $s' \xrightarrow[\eta]{d} s''$ and $s'' \xrightarrow[\eta']{d'} s$, then $s' \xrightarrow[\eta+\eta']{d+d'} s$.

(Here addition on triggers is defined by pointwise extension; we identify the noaction symbol 0 and the real-number 0.)

Axiom **T1** says that non-time-passage steps do not change any of the local times. Axiom **T2** says that time-passage steps must cause an increase in the local times for all of the active holes; note that we permit different amounts of time to pass for the action transducer and the components. Axiom **T3** states that time-passage steps do not change the set of active holes. Axiom **T4** allows repeated time-passage steps to be combined into one step.

In order to state the last axiom, we first need the definition of a "transducer trajectory". The notion of a transducer trajectory is analogous to that of a trajectory, and describes restrictions on the state changes that can occur during time-passage. A *transducer trajectory* for a step $s' \xrightarrow[\eta]{d} s$ of T consists of:

1. a function $w : [0, d] \rightarrow states(T)$ with $w(0) = s'$ and $w(d) = s$, and
2. for each hole i, a continuous, monotonic function $tt_i : [0, d] \rightarrow [0, \eta(i)]$ with $tt_i(0) = 0$ and $tt_i(d) = \eta(i)$, such that

$$w(t) \xrightarrow[\lambda i.\, tt_i(t') - tt_i(t)]{t'-t} w(t') \quad \text{for all } t, t' \in [0, d] \text{ with } t < t'.$$

A transducer trajectory assigns, to each time t in interval $[0, d]$, a state $w(t)$. As before, this assignment allows time-passage steps to span between any pair of states in the range of w. The functions tt_i can be viewed as *time tables* that translate a global increase in time to a local increase in time. Note that for each inactive hole i, the time table function tt_i is constant 0, and for each active hole i, tt_i is strictly monotonic by axiom **T2**.

Now we can state the final axiom for a timed action transducer.

T5 Each step $s' \xrightarrow[\eta]{d} s$ has a transducer trajectory.

Axiom **T5** says that any time-passage step can be "filled in" with states for each intervening time, in a "consistent" way.

Note that, for each timed automaton A, $trans(A)$ is a timed action transducer, and conversely, for each timed action transducer T, $aut(T)$ is a timed automaton. As in the untimed case, for any timed automaton A, $aut(trans(A)) = A$, and for any timed action transducer T with an empty set of holes, $trans(aut(T)) = T$.

The definition of τ-respecting in Section 2.4 applies to timed action transducers, since they are a special case of action transducers. In this case, however, axiom **T2** combines with Condition 3 of the τ-respecting definition to yield the following:

Lemma 4.1. If T is a τ-respecting timed action transducer, and s is a state of T in which an action $d \in \mathsf{R}^+$ is enabled, then there are only finitely many holes active in state s.

4.3.2. Timed traces

Let $\gamma = s_0 a_1 \eta_1 s_1 a_2 \eta_2 s_2 \cdots$ be an execution fragment of timed action transducer T. For each index j, let t_j be given by

$$\begin{aligned} t_0 &= 0, \\ t_{j+1} &= \textbf{if } a_{j+1} \in \mathsf{R}^+ \textbf{ then } t_j + a_{j+1} \textbf{ else } t_j. \end{aligned}$$

Then we define $ltime(\gamma) \triangleq \sup_j(t_j)$. The notions of *Zeno* and *admissible* execution fragments are defined for timed action transducers as for timed automata. The *timed trace* of γ, is defined to be the pair

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

Thus, $t\text{-}trace(\gamma)$ records the visible events of γ paired with their times of occurrence, as well as the limit time of the sequence. Also, for each index j and each hole i, we define the *local time of occurrence* $t_{j,i}$ by:

$$\begin{aligned} t_{0,i} &= 0, \\ t_{j+1,i} &= \textbf{if } \eta_{j+1}(i) \in \mathsf{R}^+ \textbf{ then } t_{j,i} + \eta_{j+1}(i) \textbf{ else } t_{j,i}. \end{aligned}$$

For each hole i, we let $hltime(i, \gamma) \triangleq \sup_j(t_{j,i})$; this is the largest local time for hole i.

The *timed trace for hole i* of γ is defined to be the pair

$$t\text{-}trace(\gamma, i) \triangleq (((\eta_1(i), t_{1,i})(\eta_2(i), t_{2,i})\cdots)\lceil(vis(T, col(T, i)) \times \mathsf{R}^{\geq 0}), hltime(i, \gamma)).$$

4.3.3. Zeno-respecting property

The following definition is needed for the substitutivity results. A timed action transducer T is *Zeno-respecting* if for each admissible execution

$$\gamma = s_0 a_1 \eta_1 s_1 a_2 \eta_2 s_2 \cdots$$

of T, the following condition holds: for each hole i, either $hltime(i, \gamma) = \infty$, or there is an index j such that $i \notin active(T, s_k)$ for all $k \geq j$.

Thus, if a Zeno-respecting timed action transducer advances time to infinity then, for each hole, either the local time also advances to infinity, or the hole becomes permanently inactive from some point on.

4.3.4. Combining timed action transducers and timed automata

Let T be a timed action transducer. A *timed automaton assignment* for T is an automaton assignment for T that maps each colour to a timed automaton.

Lemma 4.2. Suppose T is a timed action transducer and ζ is a timed automaton assignment for T. Then $T(\zeta)$ is a timed automaton.

Proof We show that $T(\zeta)$ satisfies axioms **S1**-**S2**. Let $Z \triangleq \lambda i.\zeta(col(T,i))$.

For axiom **S1**, assume $(s',z') \xrightarrow{d}_{T(\zeta)} (s'',z'')$ and $(s'',z'') \xrightarrow{d'}_{T(\zeta)} (s,z)$. We must prove $(s',z') \xrightarrow{d+d'}_{T(\zeta)} (s,z)$. By the assumption and the definition of composition, there exist triggers η and η' such that

1. $s' \xrightarrow[\eta]{d}_T s''$
2. $\forall i : [\mathbf{if}\ \eta(i) = 0\ \mathbf{then}\ z'(i) = z''(i)\ \mathbf{else}\ z'(i) \xrightarrow{\eta(i)}_{Z(i)} z''(i)]$
3. $s'' \xrightarrow[\eta']{d'}_T s$
4. $\forall i : [\mathbf{if}\ \eta'(i) = 0\ \mathbf{then}\ z''(i) = z(i)\ \mathbf{else}\ z''(i) \xrightarrow{\eta'(i)}_{Z(i)} z(i)]$

Now it is routine to check that

1. $s' \xrightarrow[\eta+\eta']{d+d'}_T s$
2. $active(T,s') = active(T,s'')$
3. $i \in active(T,s')$ implies $z'(i) \xrightarrow{\eta(i)+\eta'(i)}_{Z(i)} z(i)$
4. $i \notin active(T,s')$ implies $z'(i) = z(i)$

Together this implies the validity of axiom **S1**.

For axiom **S2**, assume $(s',z') \xrightarrow{d}_{T(\zeta)} (s,z)$. We must prove that there exists a transducer trajectory for $(s',z') \xrightarrow{d} (s,z)$. By the assumption and the definition of composition, there exists a trigger η such that

1. $s' \xrightarrow[\eta]{d}_T s$
2. $\forall i : [\mathbf{if}\ \eta(i) = 0\ \mathbf{then}\ z'(i) = z(i)\ \mathbf{else}\ z'(i) \xrightarrow{\eta(i)}_{Z(i)} z(i)]$

Choose a transducer trajectory w, tt_i ($i \in holes(T)$) for $s' \xrightarrow[\eta]{d} s$. Next, choose for each $i \in active(T,s')$ a trajectory w_i for $z'(i) \xrightarrow{\eta(i)}_{Z(i)} z(i)$. For $i \notin active(T,s')$, let w_i be the function with domain $[0,0]$ given by $w_i(0) = z'(i)$. Let w' be the function with domain $[0,d]$ given by $w'(t) \triangleq (w(t), z_t)$, where $z_t = \lambda i.w_i(tt_i(t))$. We claim that w' is a transducer trajectory for $(s',z') \xrightarrow{d} (s,z)$. For this, first observe that:

$$\begin{aligned} w'(0) &= (w(0), z_0) \\ &= (s', \lambda i.w_i(tt_i(0))) \end{aligned}$$

$$\begin{aligned} &= (s', \lambda i.w_i(0)) \\ &= (s, \lambda i.z'(i)) \\ &= (s', z'). \end{aligned}$$

By similar reasoning $w'(d) = (s, z)$. Now assume $t, t' \in [0, d]$ with $t < t'$. It is routine to check

1. $w(t) \ {}_{\lambda i.tt_i(t')-tt_i(t)}\!\!\xrightarrow{t'-t} w(t')$
2. $i \notin active(T, s')$ implies $z_t(i) = z_{t'}(i)$
3. $i \in active(T, s')$ implies $z_t(i) \xrightarrow{tt_i(t')-tt_i(t)} z_{t'}(i)$

Together this implies $w'(t) \xrightarrow{t'-t}_{T(\zeta)} w'(t')$. This completes the proof that w' is a transducer trajectory, and thereby the proof of the lemma. ∎

The next lemma is analogous to Lemma 2.1 in the untimed case, and plays an important role in the substitutivity result for timed action transducers in the next section.

Lemma 4.3. Suppose T is a Zeno-respecting timed action transducer, ζ is a timed automaton assignment for T, and $\alpha = (s_0, z_0)a_1(s_1, z_1)a_2(s_2, z_2)\cdots$ is a non-Zeno execution of $T(\zeta)$ with trigger sequence $\eta_1\eta_2\cdots$. Let $Z(i) = \zeta(col(T, i))$ for each hole i.

Then $\gamma = s_0a_1\eta_1s_1a_2\eta_2s_2\cdots$ is a non-Zeno execution of T, $t\text{-}trace(\gamma) = t\text{-}trace(\alpha)$, and for each hole i, $t\text{-}trace(\gamma, i) \in t\text{-}traces(Z(i))$.

Proof By Lemma 2.1, we know that γ is an execution of T. Because α is non-Zeno γ is non-Zeno as well, and $t\text{-}trace(\gamma) = t\text{-}trace(\alpha)$. Fix a hole i. Define α' to be the sequence obtained by taking the sequence $z_0(i)\eta_1(i)z_1(i)\eta_2(i)z_2(i)\cdots$ and removing all subsequences $\eta_j(i)z_j(i)$ with $\eta_j(i) = 0$. Then, by definition of $T(\zeta)$, α' is an execution of $Z(i)$. Because T is Zeno-respecting, α' is non-Zeno. Let

$$\begin{aligned} t_{0,i} &= 0, \\ t_{j+1,i} &= \textbf{if } \eta_{j+1}(i) \in \mathsf{R}^+ \textbf{ then } t_{j,i} + \eta_{j+1}(i) \textbf{ else } t_{j,i}. \end{aligned}$$

Then

$$\begin{aligned} t\text{-}trace(\alpha') &= (((\eta_1(i), t_{0,i})(\eta_2(i), t_{1,i})\cdots)\lceil(vis(Z(i)) \times \mathsf{R}^{\geq 0}), \sup_j t_{j,i}) \\ &= t\text{-}trace(\gamma, i), \end{aligned}$$

which implies $t\text{-}trace(\gamma, i) \in t\text{-}traces(Z(i))$. ∎

4.4. Substitutivity

We are now ready to state and prove our substitutivity results for timed action transducers. Our results require the hypothesis that the action transducers are Zeno-respecting. Without this hypothesis, it might happen that an admissible execution of a composition includes a Zeno execution of some argument. Since timed trace inclusion does not imply inclusion of the sets of Zeno traces, this means that $\leq^{\mathrm{t}}$ need not be substitutive for such action transducers.

Theorem 4.4. The relations $\leq_*^t$ and $\leq^t$ on timed automata are substitutive for all Zeno- and τ-respecting timed action transducers.

Proof Similar to the proof of Theorem 2.2. Let T be a Zeno- and τ-respecting timed action transducer. We show that $\leq^t$ is substitutive for T. The proof that $\leq_*^t$ is substitutive for T is similar.

Suppose ζ, ζ' are timed automaton assignments for T such that, for all c, $\zeta(c) \leq^t \zeta'(c)$, and suppose that $\beta \in$ *t-traces*$(T(\zeta))$. We have to prove that $\beta \in$ *t-traces*$(T(\zeta'))$. For this, let $Z \triangleq \lambda i.\zeta(col(T,i))$ and $Z' \triangleq \lambda i.\zeta'(col(T,i))$. Then $Z(i) \leq^t Z'(i)$ for each hole i.

Since $\beta \in$ *t-traces*$(T(\zeta))$, $T(\zeta)$ has a non-Zeno execution

$$\alpha \quad = \quad (s_0, z_0)a_1(s_1, z_1)a_2(s_2, z_2)\cdots$$

with *t-trace*$(\alpha) = \beta$. Let $\eta = \eta_1\eta_2\cdots$ be a trigger sequence for α, and let

$$\gamma \quad = \quad s_0 a_1 \eta_1 s_1 a_2 \eta_2 s_2 \cdots$$

By Lemma 4.3, γ is a non-Zeno execution of T, *t-trace*$(\gamma) = \beta$, and for each hole i,

$$\beta_i \quad \triangleq \quad \textit{t-trace}(\gamma, i) \in \textit{t-traces}(Z(i)).$$

Since $Z(i) \leq^t Z'(i)$, we obtain $\beta_i \in$ *t-traces*$(Z'(i))$, for all i. Therefore, $Z'(i)$ has, for each i, a non-Zeno execution α_i with *t-trace*$(\alpha_i) = \beta_i$. Let γ_0 be the sequence obtained from γ by removing all clearing steps. Then γ_0 is a non-Zeno execution of T and *t-trace*$(\gamma_0) = \beta$. As in the untimed case, our job is to "paste" together γ_0 and the α_i to obtain an execution of $T(\zeta')$. We construct an automaton A that describes several allowable ways to do this pasting and that generates executions of $T(\zeta')$ with the required properties. The set of states of A consists of all valuations of the following state variables in their domains:

- a variable *frag* ranging over the set of execution fragments of T. This variable denotes the part of γ_0 that still has to be dealt with. The initial value of *frag* is γ_0.
- for each hole i, a variable $frag_i$ ranging over execution fragments of $Z'(i)$. This variable denotes the part of α_i's that still has to be pasted together with (the remainder of) γ_0. The initial value of $frag_i$ is α_i.
- a variable *exec* ranging over finite executions of $T(\zeta')$. The limit of the values of *exec* will be the execution of $T(\zeta')$ in which we are interested. The initial value of *exec* is the trivial execution consisting of the state composed from the first states of γ_0 and the first states of the α_i.
- a variable *delay* ranging over $\mathsf{R}^{\geq 0}$.
- a vector $w, tt_i (i \in holes(T))$ of variables ranging over transducer trajectories of T.
- for each hole i, a variable w_i ranging over trajectories of $Z'(i)$.

Automaton A has actions *CLEARING*, *TIME* and *BASIC*, which correspond to the three different types of actions of $T(\zeta')$: clearing steps, time-passage steps, and the remaining "basic" steps. The transitions of A are defined using precondition/effect style in Figure 3. The intuition is that, while building an execution of $T(\zeta')$, automaton A peels off initial steps of γ_0 and the α_i. If the remainder of γ_0 starts with a non-time-passage step a, and, for each hole i that participates in this step, the remainder of α_i starts with the action required for hole i, then

BASIC
Precondition
$\wedge$ *frag* begins with $s' \xrightarrow[\eta]{a} s$
$\wedge$ $a \notin \mathsf{R}^+$
$\wedge$ for all holes i that participate in the first step of *frag*:
  $frag_i$ begins with an $\eta(i)$ step
Effect
remove the first step from *frag*;
for each hole i that participates in the first step of *frag* **do**
  remove the first step from $frag_i$;
append to *exec* an a followed by the state of $T(\zeta')$ composed
  from the first states of *frag* and all the $frag_i$

CLEARING
Precondition
$\wedge$ *frag* contains at least one step
$\wedge$ hole i_0 participates in the first step of *frag*
$\wedge$ $frag_{i_0}$ begins with a τ step
Effect
remove the first step from $frag_{i_0}$;
append to *exec* a τ followed by the state of $T(\zeta')$ composed
  from the first states of *frag* and all the $frag_i$

TIME
Precondition
$\wedge$ *frag* begins with $s' \xrightarrow[\eta]{d} s$
$\wedge$ for all holes i that are active is s': $frag_i$ begins with $s'_i \xrightarrow{d_i} s_i$
Effect
$w, tt_i(i \in holes(T)) :=$ any transducer trajectory for $s' \xrightarrow[\eta]{d} s$;
for each hole i that is active in s' **do**
  $w_i :=$ any trajectory for $s'_i \xrightarrow{d_i} s_i$;
$delay := \min(\{d\} \cup \{tt_i^{-1}(d_i) \mid i \text{ is active in } s' \text{ and } d_i \leq \eta(i)\})$;
if $delay = d$ **then** remove first step from *frag*
  else replace first step $s' \xrightarrow[\eta]{d} s$ of *frag* by $s'' \xrightarrow[\eta']{d'} s$,
  where $s'' = w(delay)$, $d' = d - delay$ and $\eta' = \eta - \lambda i.tt_i(delay)$;
for each hole i that is active in s' **do**
  if $tt_i(delay) = d_i$ **then** remove first step from $frag_i$
    else replace first step $s'_i \xrightarrow{d_i} s_i$ of $frag_i$ by $s''_i \xrightarrow{d'_i} s_i$,
    where $s''_i = w_i(tt_i(delay))$ and $d'_i = d_i - tt_i(delay)$;
append to *exec* the real-value of *delay* followed by the state of $T(\zeta')$ composed
  from the first states of *frag* and all the $frag_i$

Fig. 3. Algorithm for pasting together γ_0 and the α_i.

a *BASIC* step is taken by A. If, for some hole i, the remainder of α_i starts with a τ step then A can do a corresponding *CLEARING* action, provided that i participates in the next step of γ_0. The most complicated part of the definition of A is the description of the *TIME* step. Here the intuition is that if the remainder of γ_0 starts with a time passage step and, for each hole i that participates in this step, the remainder of α_i also starts with a time passage step, automaton A nondeterministically chooses trajectories corresponding to all these steps, and then determines the *maximal* progress it can make along these trajectories without passing beyond the limit time of any of them. More specifically, suppose that the remainder of γ_0 begins with a step $s' \xrightarrow[\eta]{d} s$ with transducer trajectory

$w, tt_i(i \in holes(T))$. Suppose further that for all holes i that are active is s', the remainder of α_i begins with $s'_i \xrightarrow{d_i} s_i$. Then the maximal global increase in time is d. For each active hole i the maximal local increase of time is the minimum of d_i and $\eta(i)$. In order to translate this to a global increase in time, observe that the inverse mapping of tt_i is defined, since this function is both continuous and strictly monotonic. Therefore the requirement that the local increase in time for hole i is at most $\min(d_i, \eta(i))$ is equivalent to the requirement that the global increase in time is at most $\min(tt_i^{-1}(d_i), d)$.

We leave it to the reader to check that the definition of A is type correct, in the sense that each variable is only assigned values in its domain. Note that in the effect part of the *TIME* action the argument of the min operator is always a nonempty, finite set of positive real numbers: by Lemma 4.1, the number of holes that participate in a time passage step of T is finite.

Pick an arbitrary maximal execution $\delta = u_0 b_1 u_1 b_2 u_2 \cdots$ of A. Since the three actions of A only append values to variable *exec*, we can define α' as the limit of the values of *exec* along δ. By construction, α' is an execution of $T(\zeta')$. We claim that α' is non-Zeno and $t\text{-}trace(\alpha') = \beta$.

In order to see this, we first establish that A satisfies the following invariant properties. Here we write $u.v$ for the value of state variable v of A in state u.

1. For all reachable states u, $t\text{-}trace(u.exec)\ ;\ t\text{-}trace(u.frag) = \beta$.
2. For all reachable states u and holes i, $t\text{-}trace(u.frag_i) = t\text{-}trace(u.frag, i)$.

Proof By simple inductive arguments. ■

Using Invariants 1 and 2, we next prove three claims.

Claim 1. Suppose that u is a reachable state of A and $u.frag$ is not a single state execution fragment. Then u has an outgoing step.

Proof Let $s' \xrightarrow[\eta]{a} s$ be the first step of $u.frag$. If, for some hole i that participates in this first step, $u.frag_i$ begins with a τ-step, then a *CLEARING* action is enabled in u. So suppose that for all holes i that participate in the first step $u.frag_i$ does not begin with a τ-step. We consider two cases.

1. Suppose $a \notin \mathsf{R}^+$. It follows by Invariant 2 that, for each hole i that participates in the first step of $u.frag$, $frag_i$ starts with an $\eta(i)$ step. But this means that a *BASIC* action is enabled,
2. Suppose $a \in \mathsf{R}^+$. If hole i participates in the first step, then it follows by axiom **T2** that $\eta(i) \in \mathsf{R}^+$. Since $u.frag_i$ does not begin with a τ-step, Invariant 2 implies that it begins with a time passage step. Because this is the case for each hole i that participates in the first step, a time passage action is enabled in state u.

■

Claim 2. Execution δ has no infinite suffix that consists of *CLEARING* steps only.

Proof Analogous to the corresponding proof in the untimed case. ■

Claim 3. If δ contains an infinite suffix that consists of *CLEARING* and *TIME* steps only, then $ltime(\alpha') = \infty$.

Proof The proof is by contradiction. Suppose δ has an infinite suffix with *CLEARING* and *TIME* steps only, but $ltime(\alpha')$ is finite.

Suppose $u' \stackrel{TIME}{\longrightarrow} u$ is a step of A, d is the label of the first step of $u'.frag$ and, for each i that participates in the first step of $u'.frag$, d_i is the label of the first step of $u'.frag_i$. Then we say that $u' \stackrel{TIME}{\longrightarrow} u$ is *full* if $u.delay = d$, and *i-full* for hole i if $u.tt_i(u.delay) = d_i$. By definition, each *TIME* step is either full or i-full for at least one hole i.

If δ contains infinitely many full *TIME* steps then $ltime(\alpha') = \infty$, because γ_0 is non-Zeno. So we may assume that δ contains only finitely many full *TIME* steps. This means that δ has an infinite suffix δ' that consists of *CLEARING* and non-full *TIME* steps only. By Claim 2, δ' contains infinitely many non-full *TIME* steps. If in A there is a non-full *TIME* step from u' to u, s' is the first state of $u'.frag$ and s is the first state of $u.frag$, then $active(T, s') = active(T, s)$ by axiom **T3**. Also, if in A there is a *CLEARING* step from u' to u, then the first state of $u'.frag$ equals the first state of $u.frag$. Therefore, there is a fixed collection of holes that participate in the non-full *TIME* steps of δ'. By Lemma 4.1 we know, moreover, that this collection is finite. So, the execution fragment δ' contains infinitely many i-full *TIME* steps for some hole i. This means that α_i is infinite; then since it is non-Zeno α_i is admissible.

For $u' \stackrel{TIME}{\longrightarrow} u$ a step of A, $u.tt_i(u.delay)$ gives the amount of time that has passed for hole i during that step. Because α_i is admissible, the sum of the time-passage actions for hole i along δ increases without bound:

$$\lim_{k\to\infty} \sum_{\{j \mid 1 \le j \le k,\ b_j = TIME\}} u_j.tt_i(u_j.delay) = \infty.$$

But this contradicts the fact that δ' contains no full *TIME* steps: if u_l is the first state of δ' and $u_l.frag$ begins with a step $s' \xrightarrow[\eta]{d} s$, then for all $k > l$:

$$\sum_{\{j \mid l < j \le k,\ b_j = TIME\}} u_j.tt_i(u_j.delay) < \eta(i).$$

■

We return to the proof that α' is non-Zeno and $t\text{-}trace(\alpha') = \beta$. Again we consider cases.

1. Suppose δ is finite, with final state u_n. Then, by Claim 1, $u_n.frag$ consists of a single state execution fragment. In combination with Invariant 1, this gives $t\text{-}trace(u_n.exec) = \beta$. But α' is defined as the limit of $u_n.exec$, so $\alpha' = u_n.exec$. Hence α' is finite (and hence non-Zeno) and $t\text{-}trace(\alpha') = \beta$.
2. Suppose δ is infinite and contains infinitely many *BASIC* actions. Since *frag* is initially γ_0, and each *BASIC* step removes a step from γ_0, it follows that γ_0 is infinite. But since γ_0 is non-Zeno, it is in fact admissible. Because there are infinitely many *BASIC* steps in δ, it follows by construction of A that the limit as $j \to \infty$ of $ltime(u_j.exec)$ is ∞, and that hence α' is admissible (and hence non-Zeno). By Invariant 1, $t\text{-}trace(u_j.exec)$ is a prefix of β for each j. Since the limit α' of the executions u_j is admissible, $t\text{-}trace(\alpha') = \beta$.
3. Suppose δ is infinite and contains only finitely many *BASIC* actions. Then δ has an infinite suffix with *CLEARING* and *TIME* actions only. Combination

of this fact with Claim 3 gives that α' is admissible (and hence non-Zeno). Now we use the same argument as in the previous case. By Invariant 1, $t\text{-}trace(u_j.exec)$ is a prefix of β for each j. Since the limit α' of the executions u_j is admissible, $t\text{-}trace(\alpha') = \beta$.

The fact that α' is non-Zeno and $t\text{-}trace(\alpha') = \beta$ implies $\beta \in t\text{-}traces(T(\zeta'))$, as required. ■

5. A Timed Process Algebra

In this section, we give examples of operations that can be expressed as timed action transducers. Together, these operations form a language that we will call $\mathcal{L}_t$. Paraphrasing Alur and Henzinger [AH94], we can summarise the main idea behind $\mathcal{L}_t$ as:

real-time process algebra = untimed process algebra + timers.

After the definition of the operators of $\mathcal{L}_t$ in Section 5.1, we will discuss the expressivity of the language in Section 5.2.

5.1. Operators

5.1.1. The patient construction

An important collection of timed action transducers can be obtained from untimed action transducers. In this subsection we present a simple but important construction, inspired by Nicollin and Sifakis [NS92], that transforms an untimed action transducer into a timed one, by simply inserting arbitrary time-passage steps. Suppose T is an (untimed) action transducer with $\mathsf{R}^+ \cap acts(T) = \emptyset$ and $\mathsf{R}^+ \cap acts(T, c) = \emptyset$, for all c. Then $patient(T)$ is the timed action transducer T' that has exactly the same components as T, except:

- $acts(T') = acts(T) \cup \mathsf{R}^+$,
- for all c, $acts(T', c) = acts(T, c) \cup \mathsf{R}^+$,
- $steps(T') = steps(T) \cup$
  $\{s \xrightarrow[\eta]{d} s \mid s \in states(T),\ d \in \mathsf{R}^+,\ \eta = \lambda i.\textbf{if } i \in active(T, s) \textbf{ then } d \textbf{ else } 0\}$.

It is straightforward to check that $patient(T)$ is indeed a timed action transducer. However, $patient(T)$ need not be Zeno-respecting. For example, consider an action transducer T that activates and deactivates the same hole i infinitely many times in one execution. The action transducer $patient(T)$ can intersperse the activations of i time-passage steps, in such a way that all the time-passage occurs when i is inactive. This problematic behaviour is not possible with the action transducers of Section 3, since these activate and deactivate each hole at most once during an execution. In general, $patient(T)$ need also not be τ-respecting even if T is τ-respecting. For instance, the variant of the external choice operation $\Box$ with an infinite index set is τ-respecting, but its patient timed version is not. The problem with infinitary external choice is that in the initial state infinitely many holes are active. Since in a timed action transducer all active holes participate in time-passage steps, this means that the patient version of the action transducer does not satisfy the third condition in the definition of τ-respecting, which requires that in each step only finitely many holes participate. The following simple lemma characterises the situations in which the *patient* operation preserves the property of being τ-respecting, and returns a timed action transducer that is Zeno-respecting.

Lemma 5.1. Suppose T is an action transducer. Then

1. *patient*(T) is Zeno-respecting iff T can activate and deactivate each hole at most finitely many times in each execution.
2. *patient*(T) is τ-respecting iff T is τ-respecting and in each state of T only finitely many holes are active.

The characterisation in the first part of Lemma 5.1 looks a little less than satisfying because it is expressed in terms of executions rather than the basic action transducer definition. However, this seems unavoidable.

All the patient timed versions of the operators in the language $\mathcal{L}_u$ are Zeno- and τ-respecting, by Lemma 5.1. Thus, by Theorem 4.4, the timed trace preorders $\leq^{\mathsf{t}}_*$ and $\leq^{\mathsf{t}}$ are substitutive for the *patient* variants of all these operations. The timed action 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 action transducers of $\mathcal{L}_u$.

5.1.2. Clocks

Timed action transducers that are obtained via the *patient* construction do not impose time constraints on their arguments. One way to impose such constraints is by using explicit *clock variables*, as advocated in [AD94, AH94]. In this subsection, we show how clock variables can be expressed using timed action transducers. The unary timed action transducer CLOCK_x models the effect of adding a clock variable x to a system.

We consider a set X of *clock variables*, ranged over by $x, y, \ldots$. The set of *clock constraints* ϕ is defined inductively by (here t ranges over $\mathsf{R}^{\geq 0}$):

$$\phi \quad ::= \quad x{<}t \mid x{=}t \mid \phi \wedge \phi' \mid \neg\phi.$$

Note that constraints such as true, $5{<}4$, $x{\geq}0$, and $x{\in}[2,5)$ can be defined as abbreviations. A *time assignment* ξ assigns a nonnegative real value $\xi(x)$ to each clock variable x. A time assignment ξ *satisfies* a clock constraint ϕ, denoted by $\xi \models \phi$, 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 $\phi[t/x]$ the formula obtained from ϕ by replacing all occurrences of x by t.

The state set of action transducer CLOCK_x is $\mathsf{R}^{\geq 0}$, with 0 as the initial state. There is a single hole called 1. Time proceeds at the same rate for the action transducer and its argument. The argument automaton can reset the value of the clock variable x at any moment by performing an action containing the label *reset*(x). In addition, the argument automaton can use clock constraints as labels to test the value of the clock variable. In order to define the step relation formally it is convenient to define some auxiliary functions. Let x be a clock variable, $t \in \mathsf{R}^{\geq 0}$ and a a set of labels. Then $a[t/x]$ is the label set obtained from a by replacing each clock constraint ϕ in a by $\phi[t/x]$. We say $a[t/x]$ is *satisfiable* if all time constraints contained in it are satisfiable. We also define

$$\mathcal{V}(x,t,a) \quad \triangleq \quad \textbf{if } \textit{reset}(x) \in a \textbf{ then } 0 \textbf{ else } t.$$

Now the steps of CLOCK_x can be defined by:

$$\begin{array}{rl} t \xrightarrow[\{(1,d)\}]{d} t+d & \text{if } d>0, \\ t \xrightarrow[\{(1,a)\}]{b} \mathcal{V}(x,t,a) & \text{if } a \notin \mathsf{R}^+ \text{ and } b = a[t/x] \text{ satisfiable.} \end{array}$$

As an example, let the actions a, b, c be given by $a \triangleq \{\mathit{sw\_off}, x{\in}(9,10]\}$, $b \triangleq \{\mathit{sw\_off}, 9.5{\in}(9,10]\}$ and $c \triangleq \{\mathit{sw\_off}, 1{\in}(9,10]\}$ Then CLOCK_x has a step

$$9.5 \xrightarrow[\{(1,a)\}]{b} 9.5,$$

but not a step

$$1 \xrightarrow[\{(1,a)\}]{c} 1,$$

because in the second case the clock constraint $x{\in}(9,10]$ is violated. CLOCK_x is trivially Zeno- and τ-respecting. Thus relations $\leq_*^{\mathrm{t}}$ and $\leq^{\mathrm{t}}$ are substitutive for this action transducer.

Our definition of clocks directly follows the one proposed in [AD94, AH94]. In fact, it is possible to encode each (finite state) *clock-constrained system* in the sense of [AH94] within our language: by Theorem 3.1 we can encode the underlying finite automaton (with the clock constraints viewed as part of the transition labels), and if we then apply a CLOCK operator for each of the clock variables that is used, the resulting expression will generate the same timed traces as the clock constrained system that it encodes. We suppose that, for some applications, it will be useful to have a more general notion of clock. One can, for instance, extend the set of clock constraints with formulas like $x+y{<}1$, or allow for assignments of the form $x{:=}y+4$, or introduce labels that ask the clock to emit its current time. The important point here is that explicit clocks constitute an important and useful construct in real-time process algebra. Our specific choice of clock operations is just an example, subject to modification.

5.1.3. Bounds

None of the timed action transducers introduced so far constrain the passage of time; in particular, all action transducers we have defined are willing to advance time by any amount d. However, in order to express that a certain event is *guaranteed* to occur before or at a given time, for instance in the specification of a timeout, we need an operator which (under certain conditions) can block time. In this subsection we give an example of such an operator.

For any clock variable x, the unary timed action transducer BOUND_x^b ensures that the value of x does not advance beyond a given upper bound in $\mathsf{R}^{\geq 0} \cup \{\infty\}$, initially b. The state set of this action transducer is $\mathsf{R}^{\geq 0} \times (\mathsf{R}^{\geq 0} \cup \{\infty\})$, with $(0,b)$ as the initial state. The first state component gives the current value of x, and the second component gives a bound on the value of x.[7] There is a single hole called 1. The value of x can be reset at any moment by an action with label

[7] For simplicity, we do not consider strict bounds. Such bounds can be imposed by parameterising the action transducer with an additional boolean that tells whether the time bound

reset(x); similarly the bound can be modified via an action with label $x{:}{\leq}u$, for $u \in \mathsf{R}^{\geq 0} \cup \{\infty\}$. For x a clock variable, $u \in \mathsf{R}^{\geq 0} \cup \{\infty\}$ and a a finite set of labels,

$$\mathcal{B}(x, u, a) \quad \triangleq \quad \textbf{if } \{u' \mid x{:}{\leq}u' \in a\} = \emptyset \textbf{ then } u \textbf{ else } \min\{u' \mid x{:}{\leq}u' \in a\}.$$

Now the steps of BOUND_x^b can be defined by:

$$\begin{array}{lll} (t, u) \xrightarrow[\{(1,d)\}]{d} (t + d, u) & & \text{if } 0 < d \leq u - t, \\ (t, u) \xrightarrow[\{(1,a)\}]{a} (\mathcal{V}(x, t, a), \mathcal{B}(x, u, a)) & & \text{if } a \notin \mathsf{R}^{+}. \end{array}$$

Thus there is, for instance, a step

$$(1, 10) \stackrel{8.5}{\rightarrow} (9.5, 10),$$

but not a step

$$(1, 10) \stackrel{9.5}{\rightarrow} (10.5, 10),$$

because that would violate the time bound. Clearly, BOUND_x^b is Zeno- and τ-respecting. Thus relations $\leq_*^{\mathsf{t}}$ and $\leq^{\mathsf{t}}$ are substitutive for this action transducer.

In the literature many other proposals can be found on how to constrain the passage of time: [AD94] uses a Büchi style acceptance criterion for this purpose, [HNSY92] advocates the use of program invariants, [AH94] proposes the related notion of delay predicates, [MP93] uses so-called important events, and [BPV94] uses stability with respect to linear inequalities. It is not clear to us how these approaches can be transferred to a process algebraic setting, where automata are built up step by step and not given a priori. Our approach to use BOUND operators can be viewed as a special case of the invariant approach of [HNSY92], with a fixed invariant stating that the values of the clock variables never exceed the values of the corresponding bound variables.

5.1.4. Timers

In applications, we will mostly want to use the clock and bound action transducers in combination. Furthermore, we typically want to hide the assignment labels outside the scope of these action transducers, where they are no longer needed. Finally, it is convenient to do a "garbage collection" and remove vacuous constraints like 4<7 that are generated by clock action transducers. For these reasons, we define the following derived operation TIMER_x^u, for any clock variable x and initial bound $u \in \mathsf{R}^{\geq 0} \cup \{\infty\}$:

$$\mathsf{TIMER}_x^u(A) \quad \triangleq \quad (\mathsf{CLOCK}_x(\mathsf{BOUND}_x^u(A))) \backslash (T \cup L_x),$$

where T is the set of all tautologies and L_x is the set of all assignments to x.

Example. We define a timed version of the automatic switch-off mechanism we described in Section 3. 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.

---

is strict or not. Alternatively, one can follow a suggestion of Abadi and Lamport [AL92], 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.

$$\begin{aligned} SWITCH' \;\triangleq\; & \mathsf{TIMER}_x^{\infty}((\{sw\_on, reset(x), x{:}{\leq}10\}\ ;\\ & (\{sw\_on, reset(x)\}^* \{sw\_off, x{\in}[9,10], x{:}{\leq}\infty\}))^{\omega}). \end{aligned}$$

Example. To illustrate the use of multiple, nested clocks we specify the process of having breakfast. Breakfast should be both started and finished after 6 am and before 9 am. The whole breakfast should take at least 15 minutes, and, since fresh bread is only available at 7.50 am, the end of the breakfast should be situated after 8 am.

$$\begin{aligned} BREAKFAST \;\triangleq\; & \mathsf{TIMER}_x^{8\frac{3}{4}}(\mathsf{TIMER}_y^{\infty}(\{start, x{\geq}6, x{:}{\leq}9, reset(y)\}\ ;\\ & \{finish, x{\geq}8 \wedge y{\geq}\frac{1}{4}, x{:}{\leq}\infty\}\ ;\ \mathsf{STOP})). \end{aligned}$$

5.1.5. Changing speed

Thus far, in all timed action transducers that we have considered, time advances with the same rate for the action transducer and all the (active) holes. However, the framework of timed action transducers allows us to define, quite easily, operators that change the speed of processes.

For all $l, u \in \mathsf{R}^+$ with $l \leq r$, we define a unary timed action transducer $\mathsf{RATE}_{[l,u]}$. The action transducer has a single state s, which is also the initial state. Both the action transducer and its argument have the same set of actions, and in fact the action transducer allows the argument to perform any non time-passage action a at any time. However, the rate at which the local time changes relative to the global time lies in the interval $[l, u]$.

$$s \xrightarrow[\{(1,a)\}]{a} s \quad \text{if} \quad a \notin \mathsf{R}^+,$$

$$s \xrightarrow[\{(1,d')\}]{d} s \quad \text{if} \quad \frac{d'}{d} \in [l, u].$$

It is routine to verify that $\mathsf{RATE}_{[l,u]}$ is a timed action transducer. RATE action transducers can be used both to speed up clocks and to make them drift. For $r > 1$, $\mathsf{RATE}_{[r,r]}$ speeds its argument up by a factor r. For $\Delta \ll 1$, $\mathsf{RATE}_{[1-\Delta,1+\Delta]}$ introduces a tolerance of Δ on all timing of its argument. We think that RATE action transducers can be useful in the process algebraic description of protocols that involve drifting clocks, such as the audio control protocol analyzed in [BPV94].

An interesting property of the RATE action transducers is that in general they do not preserve Wang's [Yi90] axiom of *time determinism*. This axiom, which is valid for all timed process algebras that we have encountered in the literature, states that the resulting state after a time step is uniquely determined by the amount of time that has passed:

$$s \xrightarrow{d} s' \wedge s \xrightarrow{d} s'' \quad \Rightarrow \quad s' = s''.$$

5.2. Expressivity of $\mathcal{L}_t$

We define $\mathcal{L}_t$ as the language consisting of (1) the timed action transducers obtained by applying the *patient* operation to the untimed operations of the language $\mathcal{L}_u$, (2) the CLOCK, BOUND, TIMER and RATE operators.

The operations from $\mathcal{L}_t$ are sufficiently expressive to define — as derived operators — all the constructs that we have encountered in the literature on timed process algebras, except those that involve binding mechanisms (like the integration construct of [BB91]) and those that are not compatible with timed trace inclusion (like the + from CCS). In this section, we give some of these derived operators. Also, we show how one can encode within $\mathcal{L}_t$ the finite state fragment of the *timed-bounded automata model* of [MMT91].

5.2.1. Wait constructs

Using a single timer, we can program the process $\mathsf{WAIT}\, d$ of Timed CSP [RR88, DS89], which waits time d and then terminates successfully.

$$\mathsf{WAIT}\, d \quad \triangleq \quad \mathsf{TIMER}^d_x(x{=}d).$$

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

$$\mathsf{WAIT}\, [l, u] \quad \triangleq \quad \mathsf{TIMER}^u_x(x{\geq}l).$$

5.2.2. Urgency

Using a timer, we can force any action a to be performed immediately: we define the *urgent action* $\overline{a}$ by

$$\overline{a} \quad \triangleq \quad \mathsf{TIMER}^0_x(a),$$

where x is a clock variable to which a does not refer. With urgent actions it becomes trivial to define the urgent prefixing operators of TCCS [MT90] and ATP [NS94]: $a.A \triangleq \overline{a}\ ;\ A$. Urgent actions are also useful for defining the *timeout* construct of Timed CSP. For a given delay d this operator is defined, as in [DS89], by

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

where *abort* is a fresh label, not in the label set of A and B. If, at time d, A has not performed any visible action, an interrupt occurs and automaton B is started. Note the use of the auxiliary label *abort* to force the choice between A and B at time d.

Example. We consider a simple resource-granting system described in [LA92]. The system consists of two components, a *watch* and a *manager*. The watch ticks at an approximately-predictable rate, and the manager counts ticks in order to decide when to grant a resource. The watch is modeled as an automaton $WATCH$ that does *tick* actions, such that the times between successive *tick* actions, and the time of the first *tick* action are in the interval $[c_1, c_2]$:

$$WATCH \quad \triangleq \quad (\mathsf{WAIT}\, [c_1, c_2]\ ;\ \overline{tick})^\omega.$$

Automaton *MANAGER* models the manager: it waits a particular number $k > 0$ of *tick* actions before it does a *grant* action, counting from the beginning or from the last preceding *grant*. We assume that a *grant* action occurs within l time units after it has been enabled, for some $l < c_1$.

$$\begin{aligned} MANAGER &\triangleq (MANAGER_k)^\omega \\ MANAGER_i &\triangleq tick \; ; MANAGER_{i-1} \quad \text{for } 0 < i \leq k \\ MANAGER_0 &\triangleq \mathsf{WAIT}\,[0,l] \; ; \overline{grant} \end{aligned}$$

The full system can now be described as the parallel composition of automata *WATCH* and *MANAGER*, with the *tick* action hidden:

$$SYSTEM \triangleq (WATCH \| MANAGER) \backslash \{tick\}.$$

Essentially, the result about the resource-granting system proved in [LA92] is that

$$SYSTEM \leq^{\mathrm{t}} (\mathsf{WAIT}\,[k \cdot c_1 - l, k \cdot c_2 + l] \; ; \overline{grant})^\omega.$$

Example. Another example, taken from [BB91], is a watch that is perfect, except for some fluctuations of the ticks:

$$\begin{aligned} WATCH' \triangleq\; & \mathsf{WAIT}\,0.5 \; ; \\ & ((\mathsf{WAIT}\,[0.5-\epsilon, 0.5+\epsilon] \; ; \overline{tick} \; ; \mathsf{STOP}) \wedge \mathsf{WAIT}\,1)^\omega. \end{aligned}$$

5.2.3. Execution delay

The *execution delay* operator of ATP [NSY93, NS94] is given by:

$$\lceil A \rceil^d(B) \triangleq (\mathsf{TIMER}_x^d((A \wedge (abort \; ; B)) \parallel C)) \backslash \{abort, cancel\},$$

where

$$C \triangleq (cancel \;\Box\; \{abort, x{=}d\}) \; ; x{:}{\leq}\infty \; ; \mathsf{STOP}.$$

$\lceil A \rceil^d(B)$ behaves as A until time d; at time d, A is interrupted and B is started. However, if A performs an action with the label *cancel*, then the interrupt is cancelled and A can continue to run forever. The process C takes care that once A has done a *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. The labels *cancel* and *abort* are hidden so that they cannot synchronise with any action of the environment. 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.

5.2.4. MMT-automata

It is possible to encode within $\mathcal{L}_t$ each finite state *timed-bounded automaton* in the sense of [MMT91]. We will refer to time-bounded automata as *MMT-automata*, derived from the names of the authors of [MMT91]. The MMT-automata model is an extension with real-time of the I/O automata model of [LT87]. It has been used extensively in [LA92, SALL93] for verification purposes.

An *MMT-automaton* B consists of[8]:

- an (untimed) automaton A,
- a partitioning of $acts(A)$ into three sets of *input*, *output* and *internal* actions, respectively; it is required that input actions are *enabled* in each state, i.e., for each state s' and for each input action a there exists a state s such that $s' \xrightarrow{a}_A s$,
- a partition $\{C_1, \ldots, C_n\}$ of the locally controlled (output and internal) actions into equivalence classes,
- for each class C_i, a lower time bound $b_l(C_i) \in \mathsf{R}^{\geq 0}$ and an upper time bound $b_u(C_i) \in \mathsf{R}^+ \cup \{\infty\}$, such that $b_l(C_i) \leq b_u(C_i)$.

Intuitively, in a real-time execution of B we just take steps from A, but the times at which these steps may occur are constrained by the bound maps b_l and b_u. Suppose that during execution a class C_i becomes enabled at time t. Then b_l and b_u specify that if C_i stays enabled, an action from C_i must be executed in the time interval $[t + b_l(C_i), t + b_u(C_i)]$. If C_i becomes disabled, then the timing constraints on C_i are removed.

Without loss of generality, we may assume that A has only a single start state: if there are $n > 1$ start states then the encoding of A can be defined as the disjoint union of the encodings of n copies of A in which the set of start states is restricted to a singleton. In our encoding of A, we assume for each class C_i a corresponding clock variable x_i.

As an intermediate step, we define an auxiliary automaton A^+, which is identical to A except that the labels of the transitions have been enriched with extra information: the set of labels of A^+ consists of the input and output actions of A, together with the set of clock constraints and assignments that refer to $x_1, \ldots, x_n$. For each step $s' \xrightarrow{a} s$ of A, automaton A^+ contains a corresponding step

$$s' \stackrel{b \cup \{\phi\} \cup S}{\longrightarrow} s,$$

where b is empty if a is an internal action and equal to $\{a\}$ otherwise, ϕ is a clock constraint that is equal to true if a is an input action and equal to $x_j \geq b_l(C_j)$ if a is a locally controlled action that belongs to class C_j, and S is a label set consisting of:

- a label $reset(x_j)$ if a is a locally controlled action in C_j,
- labels $reset(x_j)$ and $x_j{:}\leq b_u(C_j)$ for those classes C_j that are not enabled in s' but are enabled in s,
- a label $x_j{:}\leq\infty$ for those classes C_j that are enabled in s' but not in s.

Under the assumption that A (and hence A^+) is finite there exists, by Theorem 3.1, an $\mathcal{L}_u$-expression $expr(A^+)$ denoting A^+ up to tree equivalence. Using this auxiliary expression, we define the $\mathcal{L}_u$-expression $expr(B)$ by

$$expr(B) \quad \triangleq \quad \mathsf{TIMER}_{x_1}^{u_1}(\cdots \mathsf{TIMER}_{x_n}^{u_n}((expr(A^+)))\cdots),$$

[8] Here we follow the definition from [LA92], which is slightly more restrictive than the original definition of [MMT91] because it does not allow for strict bounds. This restriction is not crucial, but only convenient.

where u_i equals $b_u(C_i)$ if C_i is enabled in the start state, and ∞ otherwise. Without proof, we claim that $expr(B)$ generates exactly the same timed behaviours as the MMT-automaton B according to the definition of [MMT91].

5.3. Counterexamples

Although the converse of Theorem 4.4 does not hold, our result appears to be quite sharp: for many examples of timed action transducers that are not Zeno- and τ-respecting, the timed trace preorders are indeed not substitutive.

The timed trace preorders $\leq_*^{\mathsf{t}}$ and $\leq^{\mathsf{t}}$ are for instance not substitutive for the operation of infinitary external choice. It is easy to see that $\mathsf{WAIT}\,2 \equiv_*^{\mathsf{t}} \mathsf{WAIT}\,1\,;\,\mathsf{WAIT}\,1$: both processes wait time 2 and then terminate successfully. However, for infinite I,

$$\Box_{i\in I}\ (\mathsf{WAIT}\,2) \quad \not\leq_*^{\mathsf{t}} \quad \Box_{i\in I}\ (\mathsf{WAIT}\,1\,;\,\mathsf{WAIT}\,1)$$

because, unlike the first process, the second process will never manage to do a successful termination action at time 2 since it has to do an infinite number of τ actions at time 1.

Another example is the choice operator $+$ that plays a dominant role in many real-time process calculi (TCCS [MT90], the timed extension of CCS proposed in [Yi90], ATP [NS94], and ACPρ [BB91]). This operator is just the *patient* version of the choice operator from CCS, which has three states s, s_1, s_2, with s start state, and steps (for $i \in \{1,2\}$, and all actions a and b of the first and second argument, respectively):

$$s \xrightarrow[\{(i,a)\}]{a} s_i \qquad s_i \xrightarrow[\{(i,a)\}]{a} s_i$$

Relation $\leq_*^{\mathsf{t}}$ is not substitutive for the patient version of $+$ because, for instance,

$$\mathsf{WAIT}\,2 + \mathsf{WAIT}\,1.5 \quad \not\leq_*^{\mathsf{t}} \quad (\mathsf{WAIT}\,1\,;\,\mathsf{WAIT}\,1) + \mathsf{WAIT}\,1.5.$$

The first process terminates at time 1.5, whereas the second process terminates at time 2.

The loss of substitutivity may be viewed as a problem for a process algebra with CCS choice based on trace equivalence (it is not a problem if certain other equivalence are used, such as observational congruence [MT92]). Via Lemma 5.1 we have identified a general class of operations for which trace equivalence is a congruence and with patient versions for which timed trace equivalence is a congruence. Even though we advocate in this paper the use of timed trace equivalence, we think it will be quite interesting to extend Van Glabbeek's [Gla93] lattice of process equivalences with a real-time dimension, and to study the impact of the *patient* construction on congruence properties for other equivalences as well.

5.4. Remarks

Some untimed operators display undesired behaviour when transformed into timed operators via the *patient* construction. We give an example. In a timed process algebra, one typically wants to have the identity

$$\mathsf{WAIT}\,1\,;\,\mathsf{WAIT}\,1 \quad = \quad \mathsf{WAIT}\,2.$$

In order for this equation to be valid it is essential that in the action transducer for the untimed sequential composition operator ";", the second argument is not active in the initial state. In [GV92], a sequential composition operator is described for which this is not the case:

$$s_1 \xrightarrow[\{(1,a)\}]{a} s_1 \qquad \text{if } \surd \neq a$$
$$s_1 \xrightarrow[\{(1,\surd),(2,b)\}]{b} s_2$$
$$s_2 \xrightarrow[\{(2,b)\}]{b} s_2$$

For the patient version of this operator we obtain the undesired identity

$$\mathsf{WAIT}\,1\,;\,\mathsf{WAIT}\,1 \quad = \quad \mathsf{WAIT}\,1.$$

A very interesting issue that we can only touch upon in this paper, is the impact of *patient* construction on the validity of algebraic laws. All the laws that we have checked and that are valid for $\mathcal{L}_u$ up to trace equivalence, remain valid for $\mathcal{L}_t$ up to timed trace equivalence. However, in general it is not the case that the *patient* construction preserves validity of algebraic laws. For instance, the law

$$A^* B \quad = \quad A\,;(A^* B) \,\Box\, B$$

holds (in a semantics based on $\equiv_*$) for the variant of the iteration operator in which only a single copy is made of the second argument, but does not hold after *patient* has been applied (in a semantics based on $\equiv_*^t$).

6. Discussion

The main result of this paper is the characterisation in terms of action transducers of a very general class of operations that preserve inclusion of timed traces. For the untimed case, several substitutivity results for classes of operations have been reported in the literature (see, for instance, [Sim85, BIM88, GV92]). We believe our result to be the first one of this kind for the timed case. The combined complexity of multiple start states, infinitely many arguments, copying, activation and deactivation of arguments, internal actions, and different rates makes the proof of our result rather involved. It looks like that we have now reached a point at which any obvious generalisation of the class of operations violates the substitutivity property.

We think that many other equivalences and preorders for timed systems that have been proposed in the literature, such as the timed bisimulation equivalence of [Klu93], are also preserved by our class of action transducers. We expect that the situation in the timed case will be largely analogous to the one in the untimed linear time – branching time spectrum of [Gla93] where, roughly speaking, we see that the finer the behavioural equivalence, the larger the class of operations for which it is substitutive. However, results in this area still need to be worked out.

An obvious question left open in this paper is to find a sound and complete axiomatisation of timed trace inclusion for the language $\mathcal{L}_t$ or a fragment of it. Results of [AD94] can be adapted to show that, even if we exclude the RATE operator and only allow for rational numbers in clock constraints and bounds, deciding timed trace inclusion for $\mathcal{L}_t$ is Π_1 hard. Hence there does not exist a finite equational axiomatisation of timed trace inclusion for the full language $\mathcal{L}_t$. However, it may be possible to find interesting partial results: axioms that allow the elimination of certain operators in favor of others, or complete axiomatisations of subcalculi. For this it might be necessary to add to the language auxiliary operators such as the integration construct of timed ACP [BB91].

Before it can become practically useful, the language $\mathcal{L}_t$ will have to be extended with a more powerful mechanism for recursion, and with the possibility to parameterise processes and actions with data. Such extensions are standard, however, and one could simply follow the approaches taken in process algebras such as Extended LOTOS [Bri88] or μCRL [GP93].

We do not believe that one single approach, assertional or process algebraic, can solve all problems regarding the specification and verification of timed systems. A solution has to be sought rather in a smooth combination of various formalisms. Use of process algebraic notation often allows one to give compact, intuitive *specifications* of timed systems. Thus far, however, process algebraic techniques cannot claim much success when it comes to *verification* of timed systems. Here assertional methods appear to perform much better (see, for instance, [SALL93, HL94, BPV94]). Because the notion of explicit timers fits rather well with assertional proof techniques for real-time (see [AL92, BPV94]), we hope that it will be not too difficult to use these techniques, and in particular the simulation proof methods of [LV92, LV93], in the setting of our language $\mathcal{L}_t$. Together with a limited repertoire of algebraic laws, this may then form the basis of a methodology in which the benefits of algebraic and assertional methods can be combined.

Acknowledgement Roberto Segala pointed out to us the importance of making an infinite number of copies of the second argument in the action transducer for Kleene's star operator. Pedro D'Argenio and Jan Tretmans discovered a mistake in an earlier version.

References

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

[AH94] R. Alur and T.A. Henzinger. Real-time system = discrete system + clock variables. In T. Rus and C. Rattray, editors, *Theories and Experiences for Real-Time System Development — Papers presented at First AMAST Workshop on Real-Time System Development,* Iowa City, Iowa, November 1993, pages 1–29. World Scientific, 1994.

[AL92] M. Abadi and L. Lamport. An old-fashioned recipe for real time. In de Bakker et al. [dBHRR92], pages 1–27.

[BB91] J.C.M. Baeten and J.A. Bergstra. Real time process algebra. *Formal Aspects of Computing*, 3(2):142–188, 1991.

[BBP94] J.A. Bergstra, I. Bethke, and A. Ponse. Process algebra with iteration and nesting. *The Computer Journal*, 37(4):243–258, 1994.

[BIM88] 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,* San Diego, California, pages 229–239, 1988. Full version available as Technical Report 90-1150, Department of Computer Science, Cornell University, Ithaca, New York, August 1990. Accepted to appear in *Journal of the ACM.*

[BK90] J.C.M. Baeten and J.W. Klop, editors. *Proceedings CONCUR 90,* Amsterdam, volume 458 of *Lecture Notes in Computer Science.* Springer-Verlag, 1990.

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

[Bri88] E. Brinksma. *On the design of Extended LOTOS – a specification language for open distributed systems.* PhD thesis, Department of Computer Science, University of Twente, 1988.

[BW90] J.C.M. Baeten and W.P. Weijland. *Process Algebra.* Cambridge Tracts in Theoretical Computer Science 18. Cambridge University Press, 1990.

[dBHRR92] J.W. de Bakker, C. Huizing, W.P. de Roever, and G. Rozenberg, editors. *Proceedings REX Workshop on Real-Time: Theory in Practice,* Mook, The Netherlands, June 1991, volume 600 of *Lecture Notes in Computer Science.* Springer-Verlag, 1992.

[DS89] J. Davies and S. Schneider. An introduction to Timed CSP. Technical Monograph PRG-75, Oxford University Computing Laboratory, Programming Research Group, August 1989.

[FZ94] W.J. Fokkink and H. Zantema. Basic process algebra with iteration: Completeness of its equational axioms. *The Computer Journal*, 37(4):259–267, 1994.

[Gla93] R.J. van Glabbeek. The linear time – branching time spectrum II (the semantics of sequential systems with silent moves). In E. Best, editor, *Proceedings CONCUR 93,* Hildesheim, Germany, volume 715 of *Lecture Notes in Computer Science*, pages 66–81. Springer-Verlag, 1993.

[GP93] J.F. Groote and A. Ponse. Proof theory for μCRL: A language for processes with data. In D.J. Andrews, J.F. Groote, and C.A. Middelburg, editors, *Proceedings of the International Workshop on Semantics of Specification Languages*, Workshops in Computer Science, pages 231–250. Springer-Verlag, 1993.

[GSSL94] R. Gawlick, R. Segala, J.F. Søgaard-Andersen, and N. 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.

[GV92] J.F. Groote and F.W. Vaandrager. Structured operational semantics and bisimulation as a congruence. *Information and Computation*, 100(2):202–260, October 1992.

[HL94] C. Heitmeyer and N.A. Lynch. The generalized railroad crossing — a case study in formal verification of real-time systems. In *Proceedings 15th IEEE Real-Time Systems Symposium,* San Juan, Puerto Rico, pages 120–131, December 1994.

[HNSY92] T.A. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. Symbolic model checking for real-time systems. In *Proceedings 7^{th} Annual Symposium on Logic in Computer Science,* Santa Cruz, California, pages 394–406. IEEE Computer Society Press, 1992.

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

[JSV93] A.S.A. Jeffrey, S.A. Schneider, and F.W. Vaandrager. A comparison of additivity axioms in timed transition systems. Report CS-R9366, CWI, Amsterdam, November 1993.

[Kle56] S.C. Kleene. Representation of events in nerve nets and finite automata. In *Automata Studies*, pages 3–41. Princeton University Press, 1956.

[Klu93] A.S. Klusener. *Models and axioms for a fragment of real time process algebra.* PhD thesis, Department of Mathematics and Computing Science, Technical University of Eindhoven, December 1993.

[LA92] N.A. Lynch and H. Attiya. Using mappings to prove timing properties. *Distributed Computing*, 6(2):121–139, 1992.

[Lam94] L. Lamport. How to write a long formula. *Formal Aspects of Computing*, 6(5):580–584, 1994.

[LLSA93] B.W. Lampson, N.A. Lynch, and J.F. Søgaard-Andersen. Correctness of at-most-once message delivery protocols. In *FORTE'93 - Sixth International Conference on Formal Description Techniques,* Boston, MA, October 1993, pages 387–402, 1993.

[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, August 1987. A full version is available as MIT Technical Report MIT/LCS/TR-387.

[LV92] N.A. Lynch and F.W. Vaandrager. Forward and backward simulations for timing-based systems. In de Bakker et al. [dBHRR92], pages 397–446.

[LV93] N.A. Lynch and F.W. Vaandrager. Forward and backward simulations – part II: Timing-based systems. Report CS-R9314, CWI, Amsterdam, March 1993. Also, MIT/LCS/TM-487.b, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA. To appear in *Information and Computation*.

[LX90] K.G. Larsen and L. Xinxin. Compositionality through an operational semantics of contexts. In M. Paterson, editor, *Proceedings 17^{th} ICALP,* Warwick, volume 443 of *Lecture Notes in Computer Science*, pages 526–539. Springer-Verlag, July 1990. An extended version appeared as: Report R89-13, The University of Aalborg, Dept. of Mathematics and Computer Science, Aalborg, Denmark, May 1989.

[Mil85] G.J. Milne. CIRCAL and the representation of communication, concurrency, and time. *ACM Transactions on Programming Languages and Systems*, 7(2):270–298, 1985.

[Mil89] R. Milner. *Communication and Concurrency.* Prentice-Hall International, Englewood Cliffs, 1989.

[MMT91] M. Merritt, F. Modugno, and M. Tuttle. Time constrained automata. In J.C.M. Baeten and J.F. Groote, editors, *Proceedings CONCUR 91,* Amsterdam, volume 527 of *Lecture Notes in Computer Science*, pages 408–423. Springer-Verlag, 1991.

[MP93] Z. Manna and A. Pnueli. Verifying hybrid systems. In R.L. Grossman, A. Nerode, A.P. Ravn, and H. Rischel, editors, *Hybrid Systems*, volume 736 of *Lecture Notes in Computer Science*, pages 4–35. Springer-Verlag, 1993.

[MT90] F. Moller and C. Tofts. A temporal calculus of communicating systems. In Baeten and Klop [BK90], pages 401–415.

[MT92] F. Moller and C. Tofts. Behavioural abstraction in TCCS. In W. Kuich, editor, *Proceedings 19^{th} ICALP,* Vienna, volume 623 of *Lecture Notes in Computer Science*. Springer-Verlag, 1992.

[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 3rd International Workshop on Computer Aided Verification,* Aalborg, Denmark, volume 575 of *Lecture Notes in Computer Science*, pages 376–398. Springer-Verlag, 1992.

[NS94] X. Nicollin and J. Sifakis. The algebra of timed processes ATP: Theory and application. *Information and Computation*, 114(1):131–178, 1994.

[NSY93] X. Nicollin, J. Sifakis, and S. Yovine. From ATP to timed graphs and hybrid systems. *Acta Informatica*, 30(2):181–202, 1993.

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

[RR88] G.M. Reed and A.W. Roscoe. A timed model for communicating sequential processes. *Theoretical Computer Science*, 58:249–261, 1988.

[SALL93] J.F. Søgaard-Andersen, B.W. Lampson, and N.A. Lynch. Correctness of communication protocols – a case study. Technical Report MIT/LCS/TR-589, Laboratory for Computer Science, MIT, Cambridge, MA, November 1993.

[Sim85] R. de Simone. Higher-level synchronising devices in MEIJE–SCCS. *Theoretical Computer Science*, 37:245–267, 1985.

[Vaa91] F.W. Vaandrager. On the relationship between process algebra and input/output automata (extended abstract). In *Proceedings 6^{th} Annual Symposium on Logic in Computer Science,* Amsterdam, pages 387–398. IEEE Computer Society Press, 1991.

[Vaa93] F.W. Vaandrager. Expressiveness results for process algebras. In J.W. de Bakker, W.P. de Roever, and G. Rozenberg, editors, *Proceedings REX Workshop on Semantics: Foundations and Applications,* Beekbergen, The Netherlands, June 1992, volume 666 of *Lecture Notes in Computer Science*, pages 609–638. Springer-Verlag, 1993.

[Yi90] Wang Yi. Real-time behaviour of asynchronous agents. In Baeten and Klop [BK90], pages 502–520.