Theory and Implementation of a Real-Time Extension to the π-Calculus

Ernesto Posse, Juergen Dingel

Theory and implementation of a real-time extension to the π-calculus*

Ernesto Posse Juergen Dingel

School of Computing – Queen's University
Kingston, Ontario, Canada
{eposse,dingel}@cs.queensu.ca

Abstract. We present a real-time extension to the π-calculus and use it to study a notion of time-bounded equivalence. We introduce the notion of timed compositionality and the associated timed congruence which are useful to reason about the timed behaviour of processes under hard constraints. In addition to this meta-theory we develop an abstract machine for our calculus based on event-scheduling and establish its soundness w.r.t. the given operational semantics. We have built an implementation for a realistic language called kiltera based on this machine.

1 Introduction

The π-calculus [8] has become one of the most recognizable formal models of concurrency which allows the description of mobile processes. In order to model real-time mobile systems, a few process algebras have extended the π-calculus with an explicit notion of time including the $TD\pi$-calculus [11], the π_t-calculus [1], and the πRT-calculus [7]. In general, process algebras have been used to identify suitable notions of behavioural equivalence between processes to reason about their behaviour. In the context of real-time systems, time is essential to the comparison of system behaviours. Nevertheless, to the best of our knowledge, suprisingly little attention has been given to time-sensitive process equivalences for timed π-calculi.

Perhaps the most comprehensive study of timed equivalences for timed π-calculi are found in [3], [1], [6] and [2]. The first three study extensions to the π-calculus in which actions are associated with timers over discrete time. The fourth supports dense-time. In [3] and [6] some forms of timed barbed bisimilarity are studied, while [1] presents asynchronous bisimilarities and [2] explores some late bisimilarities. Nevertheless, these equivalences are quite stringent, as they require an exact match in the timing of the transitions of the processes being compared, *for all future behaviours*, and *as far in the future as the processes can run*. Real-time systems often are under *hard constraints* which require a system's response within a certain amount of time T. In this context all late responses are failures and therefore we can restrict our equivalence checking to *equivalence*

* This work has been funded by the Natural Sciences and Engineering Research Council of Canada (NSERC) and IBM Canada.

up-to time T. Furthermore, any reasonable equivalence must address the issue of compositionality: when is it safe to replace one process by another in a timed context? In this paper we define a time-bounded equivalence and show it to be compositional in our timed variant of the π-calculus.

The definition of the semantics of our calculus follows the standard approach and is given in terms of a Plotkin-style structural operational semantics (SOS). However, the purpose of our work is not only to study the theory of timed, mobile systems but also to provide a foundation for a realistic, executable, high-level modelling language for such systems. To this end, we define an abstract machine which complements the SOS of our calculus by describing execution at a level of abstraction more suitable for implementation. We prove the soundness of the machine w.r.t. the SOS. A distinguishing feature of our abstract machine is that it is based on *event-scheduling* as used in discrete-event simulation [15]. Event-scheduling does not iterate over all clock ticks whenever events are far apart in time, unlike the discrete-time approach used by existing implementations of timed π-calculi. We have validated the abstract machine via the implementation of the language kiltera [9] which has been used for teaching (in graduate courses at Queen's and McGill universities) and the modelling and analysis of complex systems such as automobile traffic simulation.

The contributions of this paper are: a process algebra that supports mobility and real-time with higher-level features such as pattern-matching; a formal operational semantics, including a new timed observational equivalence, the notion of timed compositionality and timed congruence; a sound abstract machine based on event-scheduling with a working implementation.

Paper organization Section 2 introduces our calculus, its syntax, its operational semantics. Timed equivalence is studied in Section 3. Section 4 develops the abstract machine. Section 5 concludes. For proofs see [10] and [9].

2 Timed, mobile processes: the π_{klt}-calculus

We define our timed π-calculus, which extends the asynchronous π-calculus with delays, time-value passing and unlike other variants, time observation and pattern-matching.

Definition 1. ***(Syntax)*** *The set $\mathcal{P}$ of π_{klt}* ***terms****, the set $\mathcal{E}$ of* ***expressions*** *and the set of* ***patterns*** *$\mathcal{F}$ are defined by the BNF below. Here P, P_i range over* process terms*, $x, y, ...$ range over the set of* ***(channel/event or variable) names****, A ranges over the set of* ***process names****, E ranges over* expressions*, and F ranges over* patterns*. Process definitions have the form: $A(x_1, ..., x_n) \stackrel{def}{=} P$. n ranges over floating point numbers, s ranges over strings, and f ranges over function names, with function definitions having the form: $f(x_1, ..., x_n) \stackrel{def}{=} E$, and the index set I is a subset $\{1, ..., n\} \subseteq \mathbb{N}$.*

$$P ::= \surd \quad | \quad x!E \quad | \quad \sum_{i \in I} x_i?F_i@y_i.P_i \quad | \quad \nu x.P$$
$$| \quad \Delta E.P \quad | \quad P_1 \parallel P_2 \quad | \quad A(x_1, ..., x_n)$$

$$E ::= \varnothing \quad | \quad n \quad | \quad \mathsf{true} \quad | \quad \mathsf{false} \quad | \quad \text{“}s\text{”} \quad | \quad x$$
$$| \quad \langle E_1, ..., E_m \rangle \quad | \quad f(E_1, ..., E_m)$$

$$F ::= \varnothing \quad | \quad n \quad | \quad \mathsf{true} \quad | \quad \mathsf{false} \quad | \quad \text{“}s\text{”} \quad | \quad x \quad | \quad \langle F_1, ..., F_m \rangle$$

Expressions E are either constants ($\varnothing$ represents the *null* constant), variables (x), tuples of the form $\langle E_1, ..., E_m \rangle$ or function applications $f(E_1, ..., E_m)$. Patterns F have the same syntax as expressions, except that they do not include function applications.

The process $\surd$ simply terminates. The process $x!E$ is a *trigger*; it triggers an event x with the value of E. Alternatively, we can say that it sends the value of E over a channel x. The expression E is optional: $x!$ is shorthand for $x!\varnothing$. A process of the form $\sum_{i \in I} \beta_i.P_i$ is a *listener*, where each β_i is a *guard* of the form $x_i?F_i@y_i$. This process listens to all channels (or events) x_i, and when x_i is triggered with a value v that matches the pattern F_i, the corresponding process P_i is executed with y_i bound to the amount of time the listener waited, and the alternatives are discarded[1]. The suffixes F_i and $@y_i$ are optional: $x?.P$ is equivalent to $x?y@z.P$ for some fresh names y and z. The process $\nu x.P$ hides the name x from the environment, so that it is private to P. Alternatively, $\nu x.P$ can be seen as the creation of a new name, *i.e.*, a new event or channel, whose scope is P. We write $\nu x_1, x_2, ..., x_n.P$ for the process term $\nu x_1.\nu x_2....\nu x_n.P$. The process $\Delta E.P$ is a *delay*: it delays the execution of process P by an amount of time equal to the value of the expression E.[2] The process $P_1 \parallel P_2$ is the parallel composition of P_1 and P_2. We write $\Pi_{i \in I} P_i$ for $P_1 \parallel \cdots \parallel P_n$. The process $A(y_1, ..., y_n)$ creates a new instance of a process defined by $A(x_1, ..., x_n) \stackrel{def}{=} P$, where the ports $x_1, ..., x_n$ are substituted in the body P by the channels (or values) $y_1, ..., y_n$.

Timeouts are obtained as a derived construct: the process term $(\sum_{i \in I} \beta_i.P_i) \stackrel{E}{\triangleright} Q$ represents a listener process with a timeout. If after an amount of time determined by the value of the expression E, none of the channels have been triggered, control passes to Q. We define this term as follows:

$$(\Sigma_{i \in I} \beta_i.P_i) \stackrel{E}{\triangleright} Q \stackrel{def}{=} \nu s.((\Sigma_{i \in I} \beta_i.P_i + s?.Q) \parallel \Delta E.s!)$$

The local event s can be thought of as the timeout event. Also, as in the asynchronous π-calculus [5], an output with a continuation $x!E.P$ is syntactic sugar

[1] Note that to enable an input guard it is not enough for the channel to be triggered: the message must match the guard's pattern as well. Pattern-matching of inputs means that the input value must have the same “shape” as the pattern, and if successful, the free names in the pattern are bound to the corresponding values of the input. For example, the value $\langle 3, \mathsf{true}, 7 \rangle$ matches the pattern $\langle 3, x, y \rangle$ with the resulting binding $\{^{\mathsf{true}}/_x, ^{7}/_y\}$. The scope of these bindings is the corresponding P_i.

[2] The value of E is expected to be a non-negative real number. If the value of E is negative, $\Delta E.P$ cannot perform any action. Similarly, terms with undefined values (*e.g.*, $\Delta(1/0).P$) or with incorrectly typed expressions (*e.g.*, $\Delta\mathsf{true}.P$) cause the process to stop. Since the language is untyped we do not enforce these constraints statically.

for $x!E \parallel P$. Other useful extensions include the term $\mathsf{match}\, E\ \mathsf{with}\ F_1 \to P_1 \mid \cdots \mid F_n \to P_n$ which is syntactic sugar for $\nu x.(x!E \parallel x?F_1.P_1 + \cdots + x?F_n.P_n)$, and the conditional term $\mathsf{if}\, E\, \mathsf{then}\, P\, \mathsf{else}\, Q$ which is shorthand for $\mathsf{match}\, E\, \mathsf{with}\ \mathsf{true} \to P \mid \mathsf{false} \to Q$.

The suffix $@y_i$ of input guards is inherited from Timed CSP, but it is absent in all other timed variants of the π-calculus. This construct gives the calculus the power to measure the timing of events, and determine future behaviour accordingly.

An example: testing server response times We illustrate the language with a short example. Consider a simple device to measure a server's response time to some query. To begin the test, the device (D) waits for a signal b from some client. The client provides a maximum response time t and four channels q, a, r, and m. The channels q and a are links to the server, where the device will send the query (q) and where it will expect the answer (a). The channel r is where the client expects to observe the response time. After sending a sample query to the server, the device waits for a response. If the server fails to respond within t seconds, the device will trigger a timeout event (m). We can model this testing system as follows:

$$D(b) \stackrel{def}{=} b?\langle t, q, a, r, m\rangle.q!.(a?@e.r!e.D(b)) \stackrel{t}{\triangleright} m!.D(b)$$

A model of a server, abstracting its internal execution, could be given by $S(q, a, u) \stackrel{def}{=} q?.\Delta u.a!.S(q, a, u)$. A client that uses D to test between two servers succesively and then decides to interact with the fastest is modelled as follows:

$$\begin{aligned} C(q_1, a_1, q_2, a_2, b) \stackrel{def}{=} \nu r_1, m_1, r_2, m_2.(&b!\langle 5, q_1, a_1, r_1, m_1\rangle \parallel b!\langle 5, q_2, a_2, r_2, m_2\rangle \\ &\parallel r_1?e_1.r_2?e_2.\,\mathsf{if}\, e_1 < e_2\, \mathsf{then}\, C'(q_1, a_1) \\ &\qquad\qquad \mathsf{else}\, C'(q_2, a_2) \\ &\parallel m_1?.C'(q_2, a_2) + m_2?.C'(q_1, a_1)) \end{aligned}$$

This client asks the testing device to test two servers (whose channels are parameters to the client). If both servers respond within 5 seconds (r_1 and r_2) the client selects the smaller response time and becomes C' which interacts with the corresponding server only. If it receives a timeout event for either server, it selects the other one. The complete system could be modelled as follows:

$$\nu q_1, a_1, q_2, a_2, b.\big(S(q_1, a_1, 3.2) \parallel S(q_2, a_2, 4.1) \parallel D(b) \parallel C(q_1, a_1, q_2, a_2, b)\big)$$

Operational semantics We now define the semantics formally. Let $\mathcal{N}$ denote the set of all possible names (including channel names). Let $\mathcal{V}$ denote the universe of possible values including booleans, real numbers, strings, tuples of values and channel names, and $\mathcal{B} \subseteq \mathcal{V}$ is the set of basic constants (*i.e.*, non-tuple values). We write $\mathrm{n}(v)$ for the set of all channel names occurring in the value v. To simplify the presentation we assume we have a function $eval : \mathcal{E} \to \mathcal{V}$ that

given an expression returns its value.[3] A sequence of names or values $x_1, ..., x_n$ is abbreviated as $\tilde{x}$. We denote with $\text{fn}(P)$ the set of *free names* of P (*i.e.*, names not bound by either ν or an input guard). A *substitution* is a function $\sigma : \mathcal{N} \to \mathcal{V}$. We write $\{^{V_1}/_{x_1}, ..., ^{V_n}/_{x_n}\}$ or $\{^{\tilde{V}}/_{\tilde{x}}\}$ for the substitution σ where $\sigma(x_1) = V_1$, ..., $\sigma(x_n) = V_n$ and $\sigma(z) = z$ for all $z \notin \{x_1, \ldots, x_n\}$. We write $\text{dom}(\sigma)$ for $\{x_1, ..., x_n\}$. Furthermore, we write $\sigma[^V/_x]$ for substitution update[4]. Substitution is generalized to processes as a function $\sigma : \mathcal{P} \to \mathcal{P}$ in the natural way performing the necessary renamings to avoid capture of free names as usual. We write $P\sigma$ for $\sigma(P)$ denoting the process where all free occurrences of each x in σ have been substituted by $\sigma(x)$. We denote with $\mathcal{M}$ the set of all name substitutions. We write $P \equiv_\alpha Q$ if Q can be obtained from P by renaming of bound names. We use $\mathbb{R}_0^+$ to denote the non-negative reals.

Pattern matching is formally defined by a function $match : \mathcal{F} \times \mathcal{V} \times \mathcal{M} \to \mathcal{M} \uplus \{\bot\}$ which takes as input a pattern, a datum (*i.e.*, a concrete value) and a substitution and returns either a new substitution which extends the original substitution with the appropriate bindings, or $\bot$ if the datum does not match the pattern. The substitution provided as input is used to ensure that all occurrences of a variable in a tuple match the same data. For a formal definition of this function see [10].

Any well-defined semantics must ensure that processes which are structurally equivalent behave in the same way. We now define such an equivalence relation, called *structural congruence*.

Definition 2. ***(Structural congruence over process terms)*** *The relation $\equiv \subseteq \mathcal{P} \times \mathcal{P}$ is defined to be the smallest congruence over $\mathcal{P}$ which satisfies the following axioms: 1) if $P \equiv_\alpha P'$ then $P \equiv P'$; 2) $\nu x.\surd \equiv \surd$; 3) $\nu x.\nu y.P \equiv \nu y.\nu x.P$; 4) $(\mathcal{P}, \|, \surd)$ is an abelian monoid; 5) if $x \notin fn(P)$ then $P \parallel \nu x.Q \equiv \nu x.(P \parallel Q)$; and 6) if $A(x_1, ..., x_n) \stackrel{def}{=} P$ then $A(y_1, ..., y_n) \equiv P\{^{y_1}/_{x_1}, ..., ^{y_n}/_{x_n}\}$.*

A *timed labelled transition system* or *TLTS*, is a transition system in which we distinguish between transitions due to actions and evolution (passage of time). Formally, a TLTS is a tuple $(\mathcal{S}, \mathcal{L}, \to, \rightsquigarrow)$ where $\mathcal{S}$ is a set of states, $\mathcal{L}$ is a set of *labels*, $\to \subseteq \mathcal{S} \times \mathcal{L} \times \mathcal{S}$ is a *transition relation* and $\rightsquigarrow \subseteq \mathcal{S} \times \mathbb{R}_0^+ \times \mathcal{S}$ is an *evolution relation*. A *rooted* TLTS $(\mathcal{S}, s_0, \mathcal{L}, \to, \rightsquigarrow)$ is a TLTS with a distinguished *initial state* s_0. We write $s \xrightarrow{a} s'$ for $(s, a, s') \in \to$ and $s \overset{d}{\rightsquigarrow} s'$ for $(s, d, s') \in \rightsquigarrow$. We write $s \xrightarrow{a}$ to mean that $\exists s' \in \mathcal{S}.\, s \xrightarrow{a} s'$.

Definition 3. ***(Process transitions and evolution)*** *The meaning of a π_{klt} term P_0 is a rooted TLTS $(\mathcal{P}, P_0, \mathcal{A}, \to, \rightsquigarrow)$ where $\mathcal{A}$ is the set of action labels described below and the relations $\to \subseteq \mathcal{P} \times \mathcal{A} \times \mathcal{P}$ and $\rightsquigarrow \subseteq \mathcal{P} \times \mathbb{R}_0^+ \times \mathcal{P}$ are the smallest relations satisfying the inference rules in Table 1. The elements of $\mathcal{A}$ are actions of the form τ (silent action), $x?u$ (reception), $x!u$ (trigger), or $x!\nu u$ (bound trigger) where u is a value. We let α range over $\mathcal{A}$. We write* $\text{bn}(\alpha)$ *for*

[3] We do not need a name environment, as all expressions will be closed, since the appropriate substitutions of free names are performed before evaluation takes place.

[4] $\sigma[^V/_x](x) \stackrel{def}{=} V$ and $\sigma[^V/_x](y) \stackrel{def}{=} \sigma(y)$ if $x \neq y$.

(TRIG) $x!E \xrightarrow{x!eval(E)} \surd$ (DEL) if $eval(E) = 0$ then $\Delta E.P \xrightarrow{\tau} P$

(CH) if $\sigma = match(F_i, v, \emptyset) \neq \bot$ then $\sum_{i\in I} x_i?F_i@y_i.P_i \xrightarrow{x_i?v} P_i\sigma[0/y_i]$

(NEW) if $P \xrightarrow{\alpha} P'$ and $x \notin \mathrm{n}(\alpha)$ then $\nu x.P \xrightarrow{\alpha} \nu x.P'$

(PAR) if $P \xrightarrow{\alpha} P'$ and $\mathrm{bn}(\alpha) \cap \mathrm{fn}(Q) = \emptyset$ then $P \parallel Q \xrightarrow{\alpha} P' \parallel Q$

(COMM) if $P \xrightarrow{x!v} P'$ and $Q \xrightarrow{x?v} Q'$ then $P \parallel Q \xrightarrow{\tau} P' \parallel Q'$

(OPEN) if $P \xrightarrow{x!u} P'$ and $x \notin \mathrm{n}(u)$ then $\nu\tilde{u}.P \xrightarrow{x!\nu u} P'$ with $\tilde{u} = \mathrm{n}(u)$

(CLOSE) if $P \xrightarrow{x!\nu u} P'$ and $Q \xrightarrow{x?v} Q'$ then $P \parallel Q \xrightarrow{\tau} \nu\tilde{u}.(P' \parallel Q')$ with $\tilde{u} = \mathrm{n}(u)$

(CNGR) if $P \xrightarrow{\alpha} P'$, $P \equiv Q$ and $P' \equiv Q'$ then $Q \xrightarrow{\alpha} Q'$

(TIDLE) $\surd \overset{d}{\rightsquigarrow} \surd$ (TTRIG) $x!E \overset{d}{\rightsquigarrow} x!E$

(TCH) $\sum_{i\in I} x_i?F_i@y_i.P_i \overset{d}{\rightsquigarrow} \sum_{i\in I} x_i?F_i@y_i.P_i\{{}^{y_i+d}/_{y_i}\}$

(TNEW) if $P \overset{d}{\rightsquigarrow} P'$ then $\nu x.P \overset{d}{\rightsquigarrow} \nu x.P'$

(TDEL) if $0 \leqslant d \leqslant eval(E)$ then $\Delta E.P \overset{d}{\rightsquigarrow} \Delta(E - d).P$

(TPAR) if $P \overset{d}{\rightsquigarrow} P'$ and $Q \overset{d}{\rightsquigarrow} Q'$ then $P \parallel Q \overset{d}{\rightsquigarrow} P' \parallel Q'$

(TCNGR) if $P \overset{d}{\rightsquigarrow} P'$, $P \equiv Q$ and $P' \equiv Q'$ then $Q \overset{d}{\rightsquigarrow} Q'$

Table 1. Process transitions and evolution.

the set of bound names of the action α*, namely* $\mathrm{bn}(x?u) = \mathrm{bn}(x!\nu u) \overset{def}{=} \mathrm{n}(u)$, $\mathrm{bn}(x!u) = \mathrm{bn}(\tau) \overset{def}{=} \emptyset$*. We impose an additional constraint on the TLTS to guarantee maximal progress (urgency of internal actions):*

$$\textit{if } P \xrightarrow{\tau} \textit{ then } P \overset{d}{\not\rightsquigarrow} \textit{ for all } d > 0$$

The rules (CH) and (TCH) in Table 1 are of particular interest. The rule (TCH) states that a listener can let time pass by, incrementing the variables y_i according to the elapsed time. The rule (CH) states that when an event x_i is triggered with a value that matches the corresponding pattern, the process P_i is selected with the appropriate bindings, in particular y_i is bound to 0 as any waiting time has already been taken into account and added by previous applications of the (TCH) rule.

Example Let us revisit the server response time example. Consider the execution of the device $D(b)$ when it receives a request to test the first server with links q_1 and a_1. In the given example this server takes 3.2 seconds to respond. Then the testing device will have the following execution (in this example we explicitly expand the timeout construct):

$$\begin{aligned}
D(b) &\xrightarrow{b?\langle 5,q_1,a_1,r_1,m_1\rangle} q_1!.(a_1?@e.r_1!e.D(b)) \overset{5}{\triangleright} m_1!.D(b)\\
&\xrightarrow{q_1!\varnothing} (a_1?@e.r_1!e.D(b)) \overset{5}{\triangleright} m_1!.D(b)\\
&\equiv \nu s.((a_1?@e.r_1!e.D(b) + s?.m_1!.D(b)) \parallel \Delta 5.s!)\\
&\overset{3.2}{\rightsquigarrow} \nu s.((a_1?@e.r_1!(e + 3.2).D(b) + s?.m_1!.D(b)) \parallel \Delta(5 - 3.2).s!)\\
&\xrightarrow{a_1?\varnothing} \nu s.(r_1!(0 + 3.2).D(b) \parallel \Delta(5 - 3.2).s!)\\
&\xrightarrow{r_1!3.2} \nu s.(D(b) \parallel \Delta(5 - 3.2).s!)\\
&\overset{1.8}{\rightsquigarrow} \nu s.(D(b) \parallel \Delta 0.s!)\\
&\xrightarrow{\tau} \nu s.(D(b) \parallel s!) \equiv D(b)
\end{aligned}$$

3 Timed equivalence

As explained in the introduction, all behaviours that violate hard response-time constraints of real-time systems are considered failures. Therefore we can weaken our comparison criteria to behaviours up to a given deadline. Consider the following processes: $A_1 \stackrel{def}{=} (a?.P) \overset{3}{\triangleright} Q$ and $A_2 \stackrel{def}{=} (a?.P) \overset{5}{\triangleright} Q$. Before time 3, both A_1 and A_2 have exactly the same transitions ($a?$) and evolutions. If we have a hard constraint requiring interaction before 3 time units, we don't care about their behaviour beyond time 3, and so it makes sense to identify the two processes up-to time 3. Nevertheless, these systems cannot be identified under standard notions of bisimilarity. To see this, recall the definition of timeout. We can see that A_1 has the following execution: $A_1 \overset{3}{\rightsquigarrow} (a?.P) \overset{0}{\triangleright} Q \xrightarrow{\tau} \xrightarrow{\tau} Q$ but this cannot be matched by A_2: $A_2 \overset{3}{\rightsquigarrow} (a?.P) \overset{2}{\triangleright} Q \not\xrightarrow{\tau}$. Hence, A_1 and A_2 cannot be identified by any of the existing timed bisimilarities for timed π-calculi, such as those in [3], [1], [6] or [2] or bisimilarities that match evolution directly (*i.e.*, $P \overset{d}{\rightsquigarrow} P'$ implies $Q \overset{d}{\rightsquigarrow} Q'$ with P' bisimilar to Q').

In [13], Schneider introduced a notion of *timed bisimilarity up-to time T* to compare the behaviour of Timed CSP processes. A good notion of observational equivalence is one which satisfies the property that whenever two processes are identified, no observer or context can distinguish between them. Such a property is satisfied by an equivalence relation which is preserved by all combinators or operators of the language, in other words, by a congruence relation (compositionality). Unfortunately Schneider's equivalence is not a congruence in the context of timed π-calculi, because, as ground bisimilarity, it is not preserved by listeners (input). In the theory of the π-calculus several alternative definitions of bisimilarity have been explored to ensure compositionality. Sangiorgi's *open bisimilarity* [12] has the desired feature: it is a congruence for all π-calculus operators. This suggests the following equivalence for dense-time π-calculi which combines Schneider's timed bisimilarity with Sangiorgi's open bisimilarity.

Definition 4. ***(Open timed-bisimulation)*** *Let $\mathcal{S}$ be a set of terms in some language equipped with a notion of substitution, where substitutions are functions $\sigma : \mathcal{S} \to \mathcal{S}$. Let $(\mathcal{S}, \mathcal{L}, \to, \rightsquigarrow)$ be a TLTS over $\mathcal{S}$. A relation $B \subseteq \mathcal{S} \times \mathbb{R}_0^+ \times \mathcal{S}$, is called an **open timed-simulation** if for all $t \in \mathbb{R}_0^+$, whenever $(P, t, Q) \in B$ then, for any substitution $\sigma : \mathcal{S} \to \mathcal{S}$ and any $d \in \mathbb{R}_0^+$ such that $d < t$:*

1. $\forall \alpha \in \mathcal{L}. \forall P' \in \mathcal{S}. P\sigma \xrightarrow{\alpha} P' \Rightarrow \exists Q' \in \mathcal{S}. Q\sigma \xrightarrow{\alpha} Q' \wedge (P', t, Q') \in B$
2. $\forall P' \in \mathcal{S}. P\sigma \overset{d}{\rightsquigarrow} P' \Rightarrow \exists Q' \in \mathcal{S}. Q\sigma \overset{d}{\rightsquigarrow} Q' \wedge (P', t-d, Q') \in B$

*If B and B^{-1} are open-timed simulations, then B is called an **open-timed-bisimulation**. Let $\leftrightarrows \stackrel{def}{=} \{(P, u, Q) \in \mathcal{S} \times \mathbb{R}_0^+ \times \mathcal{S} \mid \exists B. B$ is an open timed-bisimulation $\& (P, u, Q) \in B\}$. For any given $t \in \mathbb{R}_0^+$, let $\leftrightarrows_t \stackrel{def}{=} \{(P, Q) \in \mathcal{S} \times \mathcal{S} \mid (P, t, Q) \in \leftrightarrows\}$.*

Remark 1. For any $t \in \mathbb{R}_0^+$, $\leftrightarrows_t$ is an equivalence relation and $\leftrightarrows$ is the largest open timed-bisimulation.

With this definition we can now establish that $A_1 \leftrightarrows_3 A_2$ for the processes A_1 and A_2 defined above.

Proposition 1. *For any TLTS $M = (\mathcal{S}, \mathcal{L}, \rightarrow, \rightsquigarrow)$, any $t, u \in \mathbb{R}_0^+$, and any $P, P', P'' \in \mathcal{S}$:*

1. *If $P \leftrightarrows_t P'$ then for any $u \leqslant t$, $P \leftrightarrows_u P'$; and*
2. *if $P \leftrightarrows_t P'$ and $P' \leftrightarrows_u P''$ then $P \leftrightarrows_{\min\{t,u\}} P''$.*

Timed compositionality Now we focus on the compositionality properties of open timed-bisimilarity. First, we have that it is closed under substitutions:

Lemma 1. *For any substitution σ, and any $t \in \mathbb{R}_0^+$, if $P \leftrightarrows_t Q$ then $P\sigma \leftrightarrows_t Q\sigma$.*

As mentioned above, a good observational equivalence should be a congruence. However, as we have argued, we only care about the observable behaviour up-to some time T, which means that the equivalence must be preserved by our operators only up to that time. Hence we are after a notion of *timed-compositionality*, characterized by a *timed-congruence*, which we now formally define:

Definition 5. ***(Timed-congruence)*** *Given some set $\mathcal{S}$, and a ternary relation $R \subseteq \mathcal{S} \times \mathbb{R}_0^+ \times \mathcal{S}$, we define R's t-**projection** to be the binary relation $R_t \stackrel{def}{=} \{(p, q) \in \mathcal{S} \times \mathcal{S} \mid (p, t, q) \in R\}$. R is called a t-**congruence** iff R_t is a congruence. R is called a **timed-congruence** if it is a t-congruence for all $t \in \mathbb{R}_0^+$. R is called a **timed-congruence up-to** u iff it is a t-congruence for all $0 \leqslant t \leqslant u$.*

Open timed-bisimilarity satisfies the following stronger property which we obtain using Lemma 1:

Lemma 2. *For any $P, P', Q, Q_1, ..., Q_n \in \mathcal{P}$, and any $t \in \mathbb{R}_0^+$, if $P \leftrightarrows_t P'$ then:*

1. *$\Delta E.P \leftrightarrows_{t+e} \Delta E.P'$ where $e = eval(E)$*
2. *$\nu x.P \leftrightarrows_t \nu x.P'$*
3. *$P \parallel Q \leftrightarrows_t P' \parallel Q$*
4. *$x?F@y.P + \sum_{i=1}^n \beta_i.Q_i \leftrightarrows_t x?F@y.P' + \sum_{i=1}^n \beta_i.Q_i$ where each β_i* is of the form $x_i?F_i@y_i$.

The immediate consequence, which follows from Proposition 1 and Lemma 2, is timed-compositionality:

Theorem 1. *$\leftrightarrows_t$ is a timed-congruence up-to t and $\leftrightarrows$ is a timed-congruence.*

We also obtain the following properties as a consequence of Lemma 2, which guarantees equivalence up to the least upper bound of the pairwise time-equivalences:

Corollary 1. *For any families of terms $\{P_i \in \mathcal{P}\}_{i \in I}$ and $\{Q_i \in \mathcal{P}\}_{i \in I}$, if for each $i \in I$, $P_i \leftrightarrows_{t_i} Q_i$ then*

1. *$\prod_{i \in I} P_i \leftrightarrows_{\min\{t_i \mid i \in I\}} \prod_{i \in I} Q_i$*
2. *$\sum_{i \in I} x_i?F_i@y_i.P_i \leftrightarrows_{\min\{t_i \mid i \in I\}} \sum_{i \in I} x_i?F_i@y_i.Q_i$*

4 An abstract machine for the π_{klt}-calculus

We now turn our attention to the executable semantics of π_{klt}.

4.1 Abstract machine specification

Our abstract machine is similar to Turner's abstract machine for the π-calculus [14], but unlike Turner's, we have to take evolution over real-time and pattern-matching into account. As mentioned in the introduction, the abstract machine is based on event-scheduling, which, unlike discrete-time algorithms, does not require idle iteration cycles at times when no events are scheduled. The key idea is to treat each π_{klt} term as a *simulation event* to be executed by an event-scheduler (and not to be confused with a *communication event* in the language itself). Such event scheduler forms the heart of our abstract machine.

The global queue The event-scheduler contains a queue of simulation events (terms) to be executed, but rather than store them all in a single linear queue, we divide them into *time-slots*, *i.e.*, sequences of all simulation events to be executed at a given instant in time. Hence the global event queue is a time-ordered queue of time-slots, each of which is a queue of terms. We describe the operation of our abstract machine by showing how it evolves in these two "dimensions" of time: the "vertical dimension" which corresponds to the execution of all terms in a single time-slot, and the "horizontal dimension", which corresponds to the advance in time, *i.e.*, the progress of the global queue.

Definition 6. ***(Global queue)*** *The set $\mathcal{R}$ of **global queue states**, ranged over by R, is defined by the following BNF, where T ranges over the set $\mathcal{T}$ of **time-slots**:*

$$\begin{aligned} R &::= (t_1, T_1) \cdot (t_2, T_2) \cdot \cdots \cdot (t_n, T_n) \quad | \quad \langle\rangle \\ T &::= P_1 :: P_2 :: \cdots :: P_m \quad | \quad \epsilon \end{aligned}$$

where each $P_i \in \mathcal{P}$, each $t_i \in \mathbb{R}_0^+$, and for each $i \geqslant 1$, $t_i < t_{i+1}$.

Event observers and the heap Multiple processes can trigger and/or listen to the same channel. Hence we need to keep track of each of these requests in an *observer set*[5], containing *observers, i.e.,* requests to either send a message over a channel or listen to it. We also define the *heap*, which is a map associating each channel name to its observer set.

Definition 7. ***(Channel observers and heap)*** *The set of **channel observers**, ranged over by O, **observer sets**, ranged over by Q and the set $\mathcal{H}$ of **heaps**, ranged over by H, are defined by the following BNFs:*

$$\begin{aligned} O &::= !v \quad | \quad ?(F, y, P, t, c) \\ Q &::= \{O_1, O_2, \cdots, O_n\} \quad | \quad \emptyset \\ H &::= x_1 \mapsto Q_1, x_2 \mapsto Q_2, ..., x_m \mapsto Q_m \quad | \quad \epsilon \end{aligned}$$

[5] Analogous to "channel queues" in Turner's terminology.

where $v \in \mathcal{V}$, $F \in \mathcal{F}$, $P \in \mathcal{P}$, *and* $t \in \mathbb{R}_0^+$. *We denote* $H\{x \mapsto Q\}$ *for the heap where the entry for* x *is updated to* Q. *We extend this notation to indexed sets:* $H\{x_i \mapsto Q_i\}_{i\in I}$ *stands for* $H\{x_1 \mapsto Q_1\}\cdots\{x_n \mapsto Q_n\}$ *for* $I = \{1, ..., n\}$.

Observers of the form $!v$ are *output observers*, and denote an attempt to send a value v over the given channel. Observers of the form $?(F, y, P, t, c)$ denote *input observers*, with a pattern F to match, elapsed-time variable y, body P to execute when a message arrives and which start listening at time t. This tag t is necessary in order to assign the correct elapsed-time to y once interaction occurs. Such time-stamp is not present in Turner's machine, since his is "time agnostic". The last item, c, is a tag used to identify the original π_{klt} listener, so that each branch of a listener $\sum_{i\in I} x_i?F_i@y_i.P_i$ will have an observer $?(F_i, y_i, P_i, t, c)$ sharing the same identifier c. This is also absent from Turner's machine, since he does not implement the choice operator.

Unlike Turner's machine, any given non-empty observer set can contain both inputs and outputs simultaneously, because it is possible that the value sent over a channel does not match any of the patterns of the available input observers and thus the corresponding output observer must be suspended with the existing inputs on the same observer set. Hence the following auxiliary definitions will be useful to extract the relevant observers from a set.[6]

Definition 8. *Given an observer set* Q, *we denote:*

$inputs(Q) \stackrel{def}{=} \{O \in Q \mid O \text{ is of the form } ?(F, y, P, t, c)\}$

$outputs(Q) \stackrel{def}{=} \{O \in Q \mid O \text{ is of the form } !v\}$

$patt(?(F, y, P, t, c)) \stackrel{def}{=} F \qquad tag(?(F, y, P, t, c)) \stackrel{def}{=} c \qquad val(!v) \stackrel{def}{=} v$

$inms(v, Q) \stackrel{def}{=} \{(O, \sigma) \mid O \in inputs(Q), \sigma = match(patt(O), v, \emptyset),\ \sigma \neq \bot\}$

$outms(F, Q) \stackrel{def}{=} \{(O, \sigma) \mid O \in outputs(Q), \sigma = match(F, val(O), \emptyset),\ \sigma \neq \bot\}$

The last two functions give us the set of observers and bindings for successful matches between a value v (resp. a pattern F) and the patterns (resp. values) available as input (resp. output) observers in the set.

We also need the ability to remove input observers from all branches of a listener once a branch has been triggered. To this end, we define the following which removes all c tagged inputs from an observer set Q:

$withdraw(Q, c) \stackrel{def}{=} \{O \in inputs(Q) \mid tag(O) \neq c\} \cup outputs(Q)$

which we use to define the following function that removes all such inputs anywhere in the heap[7]:

$rall(\epsilon, c) \stackrel{def}{=} \epsilon$ and

$rall((H, x \mapsto Q), c) \stackrel{def}{=} rall(H, c), x \mapsto withdraw(Q, c)$

[6] We assume the standard set theoretical operations for observer sets: *e.g.*, $Q \cup \{O\}$ denotes the observer set that adds O to Q, $Q \backslash O$ is the set that results from removing O from Q, $O \in Q$ tests for membership, etc.

[7] This definition is inefficient since it traverses the entire heap. In practice, the input observers contain a list of pointers to the relevant heap entries to remove the alternatives efficiently.

Executing time-slots We now describe the "vertical dimension" of time, *i.e.*, the execution of terms within one time-slot.

Definition 9. ***(Time-slot execution)*** *The behaviour of time-slots at a time* $t \in \mathbb{R}_0^+$ *is defined as the smallest relation* $\rightarrow_t \subseteq (\mathcal{H} \times \mathcal{T}) \times (\mathcal{H} \times \mathcal{T})$ *satisfying the rules below:*

(NIL) $(H, \surd :: T) \rightarrow_t (H, T)$

(RES) $(H, \nu x.P :: T) \rightarrow_t (H\{k \mapsto \emptyset\}, P\{^k/_x\} :: T)$ *with* k *fresh*

(SP$_1$) $(H, (P_1 \parallel P_2) :: T) \rightarrow_t (H, T :: P_1 :: P_2)$

(SP$_2$) $(H, (P_1 \parallel P_2) :: T) \rightarrow_t (H, T :: P_2 :: P_1)$

(OUT-F) *if* $H(x) = Q$ *and* $inms(eval(E), Q) = \emptyset$ *then*
$(H, x!E :: T) \rightarrow_t (H\{x \mapsto Q \cup \{!eval(E)\}\}, T)$

(OUT-S) *if* $H(x) = Q$ *and* $(?(F, y, P, u, c), \sigma) \in inms(eval(E), Q) \neq \emptyset$ *then*
$(H, x!E :: T) \rightarrow_t (rall(H, c), P\sigma[^{t-u}/_y] :: T)$

(INP-F) *if* $\forall i \in I.\, H(x_i) = Q_i$, $outms(F_i, Q_i) = \emptyset$ *and* c *fresh, then*
$(H, \sum_{i \in I} x_i?F_i@y_i.P_i :: T) \rightarrow_t (H\{x_i \mapsto Q_i \cup \{?(F_i, y_i, P_i, t, c)\}\}_{i \in I}, T)$

(INP-S) *if* $\exists i \in I.\, H(x_i) = Q_i$ *and* $(O, \sigma) \in outms(F_i, Q_i) \neq \emptyset$ *then*
$(H, \sum_{i \in I} x_i?F_i@y_i.P_i :: T) \rightarrow_t (H\{x_i \mapsto Q_i \backslash O\}, P_i\sigma[^0/_{y_i}] :: T)$

(INST) *if* $A(\tilde{x}) \stackrel{def}{=} P$ *then* $(H, A(\tilde{v}) :: T) \rightarrow_t (H, T :: P\{^{\tilde{v}}/_{\tilde{x}}\})$

The rule (NIL) simply ignores a terminated process. The (SP) rules break a parallel composition into its components and spawn their execution in the current time-slot in an arbitrary order. Unlike Turner's machine, this is non-deterministic. The (RES) rule allocates a new spot for the new channel, and initializes its observer set to empty.

The (OUT-F) rule describes the case when a message is sent over x and there is no matching listener in x's observer set (output failure). In this case we simply add a new trigger observer to x's observer set and continue. The (OUT-S) rule (output success) applies when there is a matching observer O with σ being the resulting bindings. In this case, we remove all observers belonging to the matching listener (those tagged with c), and then execute the body P of the observer, applying the substitution σ extended with the binding of the elapsed time variable y to the difference between the current time t and the time u when the receiver started listening. Note that since there might be more than one successful match, the choice is non-deterministic, contrasting again with Turner's machine.

The rules (INP-S) and (INP-F) are the dual of (OUT-S) and (OUT-F). In rule (INP-F), when attempting to execute a listener, if there are no matching triggers in the relevant observer sets, we simply add the appropriate observers to the corresponding observer sets. Note that the added observers are tagged with the current time t, and with the same tag c. On the other hand, in rule (INP-S), one of the branches succeeds in matching the pattern with an observer O and binding σ. In this case, we remove the output observer from the event's observer set and execute the body of the corresponding branch, applying the substitution σ and binding y_i to 0, since the listener did not have to wait.

Finally, the rule (INST) deals with process instantiations. This simply assumes the set of process definitions is available, and schedules the execution of the body

of the definition by replacing its parameters by the arguments provided by the instantiation.

Note that there is no rule associated with the Δ operator. We specify its behaviour in the description of the global scheduler below.

Global event scheduler Now we can define the behaviour of the global event scheduler. For this purpose we will assume we have a function $insort : \mathbb{R}_0^+ \times \mathcal{P} \times \mathcal{R} \to \mathcal{R}$ (formally defined in [10]) which, given a time, inserts a term in the appropriate time-slot in the global queue, preserving the order of time-slots w.r.t. their time-stamps.

Definition 10. ***(Scheduler)*** *The behaviour of the scheduler is given by the smallest relations* $\hookrightarrow_0, \hookrightarrow_1, \hookrightarrow_2 \subseteq (\mathcal{H} \times \mathcal{R}) \times (\mathcal{H} \times \mathcal{R})$ *which satisfy the rules below:*

(TS) *if* $(H, T) \to_t (H', T')$ *and* $T \neq \epsilon$ *then* $(H, (t, T) \cdot R) \hookrightarrow_0 (H', (t, T') \cdot R)$

(ADV) $(H, (t, \epsilon) \cdot R) \hookrightarrow_1 (H, R)$

(SCH) $(H, (t, \Delta E.P :: T) \cdot R) \hookrightarrow_2 (H, insort(t + eval(E), P, (t, T) \cdot R))$

The rule (TS) states that as long as there are terms in the current time-slot then they are executed. The (ADV) rule states that when the current time-slot is empty, execution moves on to the next available time-slot. Finally, the (SCH) rule describes the behaviour of the delay operator: to execute $\Delta E.P$, the value d of E is computed and P is inserted at time $t + d$ (where t is the current time). Note that P is inserted in $(t, T) \cdot R$ because the value of E may be 0. This may create a new time-slot, if there was none at time $t + d$.

Example We illustrate the abstract machine with a sample execution. Consider the processes $Q \stackrel{def}{=} \Delta 3.2.x!1$ and $P \stackrel{def}{=} x?1@e.P_1$ where $P_1 \stackrel{def}{=} \Delta(5 - e).P_2$ for some P_2. Suppose that the current time is, for example, 7. The execution of the time-slot containing only $\nu x.(P \parallel Q)$, assuming the heap is initially empty (just to simplify notation) is:

$$
\begin{array}{ll}
(\epsilon, (7, \nu x.(P \parallel Q))) & \\
\hookrightarrow_0 \ (k \mapsto \emptyset, (7, (P \parallel Q)\{^k/_x\})) & (\text{RES})+(\text{TS}) \\
\equiv \quad (k \mapsto \emptyset, (7, (P\{^k/_x\} \parallel Q\{^k/_x\}))) & \\
\hookrightarrow_0 \ (k \mapsto \emptyset, (7, P\{^k/_x\} :: Q\{^k/_x\})) & (\text{SP}_1)+(\text{TS}) \\
\hookrightarrow_0 \ (k \mapsto \{?(1, e, P_1\{^k/_x\}, 7, c)\}, (7, Q\{^k/_x\})) & (\text{INP-F})+(\text{TS}) \\
\equiv \quad (k \mapsto \{?(1, e, P_1\{^k/_x\}, 7, c)\}, (7, \Delta 3.2.k!1)) & \\
\hookrightarrow_2 \ (k \mapsto \{?(1, e, P_1\{^k/_x\}, 7, c)\}, insort(7 + 3.2, k!1, (7, \epsilon))) & (\text{SCH}) \\
\equiv \quad (k \mapsto \{?(1, e, P_1\{^k/_x\}, 7, c)\}, (7, \epsilon) \cdot (10.2, k!1)) & \\
\hookrightarrow_1 \ (k \mapsto \{?(1, e, P_1\{^k/_x\}, 7, c)\}, (10.2, k!1)) & (\text{ADV}) \\
\hookrightarrow_0 \ (k \mapsto \emptyset, (10.2, P_1\{^k/_x\}\{^{10.2-7}/_e\})) & (\text{OUT-S})+(\text{TS}) \\
\equiv \quad (k \mapsto \emptyset, (10.2, \Delta(5 - 3.2).P_2\{^{3.2}/_e\}\{^k/_x\})) & \\
\hookrightarrow_2 \ (k \mapsto \emptyset, (10.2, \epsilon) \cdot (12, P_2\{^{3.2}/_e\}\{^k/_x\})) & (\text{SCH}) \\
\hookrightarrow_1 \ (k \mapsto \emptyset, (12, P_2\{^{3.2}/_e\}\{^k/_x\})) & (\text{ADV})
\end{array}
$$

Here we used (SP_1) which selected the left process P to be executed first. This resulted in registering the input observer in k's observer set. If we had used (SP_2) instead, then Q would have been executed first, resulting in the scheduling happening first, but P would remain in the time-slot for time 7, and thus it would be executed before advancing in time.

4.2 Soundness

We now establish that reductions in our abstract machine correspond to valid π_{klt} executions, following the same approach from [14]. To do this, we first encode the states of our abstract machine as π_{klt} terms, in particular we need to encode the heap, its observer sets, time-slots and the global queue. We use $x \in H$ to denote that there is an entry for x in the heap H.

Definition 11. ***(Encoding the machine state)*** *The set of* ***triggers in the heap entry for*** x *and the set of* ***all triggers in the heap*** *are given by:*

$$triggers(Q, x) \stackrel{def}{=} \{x!v \mid !v \in outputs(Q)\}$$

$$alltriggers(H) \stackrel{def}{=} \bigcup_{x \in H} triggers(H(x), x)$$

The set of ***branches of a listener with tag*** c *is given by:*

$$branches(H, c, t) \stackrel{def}{=} \bigcup_{x \in H} alts(H(x), c, x, t) \text{ where}$$

$$alts(Q, c, x, t) \stackrel{def}{=} \{x?F@y.P\{^{y+(t-u)}/_{y}\} \mid ?(F, y, P, u, c) \in inputs(Q)\}$$

The set of ***all listeners in the heap*** *is given by:*

$$alllisteners(H, t) \stackrel{def}{=} \{\sum branches(H, c, t) \mid c \in alltags(H)\}$$

where

$$alltags(H) \stackrel{def}{=} \bigcup_{x \in H} tags(H(x)) \text{ and}$$

$$tags(Q) \stackrel{def}{=} \{tag(O) \mid O \in inputs(Q)\}$$

The ***encoding of the heap*** H ***at time*** t*, is given by:*

$$[\![H]\!]_t \stackrel{def}{=} (\prod alllisteners(H, t)) \parallel (\prod alltriggers(H))$$

The ***encoding of the time-slot*** $T = P_1 :: P_2 :: \cdots :: P_n$ *is:*

$$[\![T]\!] \stackrel{def}{=} P_1 \parallel P_2 \parallel \cdots \parallel P_n$$

The ***encoding of a heap/time-slot pair at time*** t *is:*

$$[\![(H, T)]\!]_t \stackrel{def}{=} [\![H]\!]_t \parallel [\![T]\!]$$

The ***encoding of the global queue*** $R = (t_1, T_1) \cdot (t_2, T_2) \cdot ...$ *is given by:*

$$[\![R]\!] \stackrel{def}{=} [\![T_1]\!] \parallel \Delta(t_2 - t_1).[\![T_2]\!] \parallel \cdots \parallel \Delta(t_n - t_1).[\![T_n]\!]$$

The ***encoding of the machine state*** (H, R) *is defined as:*

$$[\![(H, R)]\!] \stackrel{def}{=} \nu x_1, ..., x_k.([\![H]\!]_{curtime(R)} \parallel [\![R]\!])$$

where $\{x_1, ..., x_k\}$ *are the names of entries in the heap* H *and* $curtime((t, T) \cdot R) \stackrel{def}{=} t$ *is the time-stamp of the first time-slot.*

Note that to create a single listener we have to quantify over the possible entries in the heap. This is because each branch of a listener may listen to different channels, and therefore, the corresponding input observers may be dispersed over multiple heap entries. The use of tag c allows us to identify all input observers

which belong to the same listener. Also, note that the definition $alts$ which gives us an alternative branch of a listener, substitutes $y + (t - u)$ for y in P, where t is the current time and u is the time-stamp of the input observer, *i.e.*, when the listener began listening. This is because the encoding corresponds to taking a "snapshot" of the machine's state at time t, but each listener may have registered at some time $u \leqslant t$ so we have to take the already elapsed time into account. The encoding $[\![R]\!]$ for the global queue considers the first time-slot to represent the current one, so all future time-slots are delayed relative to the current time t_1.

Now we establish our result (we write $\Longrightarrow$ for $(\xrightarrow{\tau}) \equiv \cup \equiv$).

Lemma 3. *If* $(H, T) \rightarrow_t (H', T')$ *then* $[\![(H, T)]\!]_t \Longrightarrow [\![(H', T')]\!]_t$

Theorem 2. ***(Abstract machine soundness)***

1. *If* $(H, R) \hookrightarrow_0 (H', R')$ *then* $[\![(H, R)]\!] \Longrightarrow [\![(H', R')]\!]$
2. *If* $(H, R) \hookrightarrow_1 (H', R')$ *then* $[\![(H, R)]\!] \overset{d}{\rightsquigarrow} [\![(H', R')]\!]$ with $d = t_2 - t_1$ where $R = (t_1, T_1) \cdot (t_2, T_2) \cdot ...$
3. *If* $(H, R) \hookrightarrow_2 (H', R')$ *then* $[\![(H, R)]\!] \equiv [\![(H', R')]\!]$

5 Conclusions

We have introduced the π_{klt}-calculus, a timed extension to the asynchronous π-calculus which adds some high-level features such as pattern-matching. We have given an operational semantics in terms of timed-labelled transition systems, and developed a basic theory of time-bounded congruence. We developed an abstract machine for the calculus and established its soundness with respect to the operational semantics. To the best of our knowledge this is the first use of event-scheduling (as used in simulation) as a language interpreter. We have implemented the π_{klt}-calculus in a language called kiltera (available at `http://www.kiltera.org`) which is based on the described abstract machine. Moreover, it extends π_{klt} with primitives for distributed computing, allowing processes to be sent to remote sites and have site-dependent behaviour.

Aside from differences in the particular choice of operators, the most closely related work, to the best of our knowledge, is found in [3], [1], [6] and [2]. As mentioned before the first three consider only discrete-time variants of the π-calculus and all deal with stringent notions of timed equivalence which do not take into account time bounds. Furthermore, the equivalences in [3] are not shown to be congruences. In fact, we suspect that these equivalences are not congruences, for the same reasons that strong, ground bisimilarity is not a congruence in the π-calculus, or (non-open) timed-bisimilarity is not a congruence in π_{klt}: they are insensitive to values over channels. The other papers deal with congruences, but they are sensitive to behaviours beyond time-bounds, distinguishing processes that should be identified when considering hard constraints.

In terms of execution, to the best of our knowledge, there is no other abstract machine for a timed π-calculus, and the only other implementation is that of the $TD\pi$-calculus [4]. This implementation is quite different from ours, both in terms

of the execution model and architecture. Firstly, it uses of a clock-tick model rather than event-scheduling. Secondly, instead of using an abstract machine, it translates the source language (TiMo) to Java code, and thus depends on the Java run-time system and additional libraries. There are further differences regarding distributed execution, but these fall outside the scope of this paper. For a more detailed comparison with this and other related work, we refer the reader to [10].

There are several possible future lines of research including the development of a type system, weaker notions of equivalence and refinement relations with appropriate axiomatizations, as well as symbolic methods to help analyze systems. The mentioned extension of π_{klt} and kiltera to distribution will be described in a future paper.

References

1. M. Berger. Basic Theory of Reduction Congruence for Two Timed Asynchronous π-Calculi. In *Proc. of CONCUR'04*, volume 3170 of *LNCS*. Springer, 2004.
2. J. Chen. A proof system for weak congruence in timed π-calculus. Tech. Report. 2004-13, LIFO, Université d'Orléans, 2004.
3. G. Ciobanu. Behaviour Equivalences in Timed Distributed π-calculus. In *Software-intensive systems and new computing paradigms - challenges and visions*, volume 5380 of *LNCS*, pages 190–208. Springer, 2008.
4. G. Ciobanu and C. Juravle. MCTools: A Software Platform for Mobility and Timed Interaction. Tech. Report FML-09-01, Formal Methods Laboratory – Romanian Academy – Iasi Branch, February 2009.
5. K. Honda and M. Tokoro. An object calculus for asynchronous communication. In *Proc. of ECOOP '91*, volume 512 of *LNCS*, pages 133 – 147. Springer, 1991.
6. H. Kuwabara, S. Yuen, and K. Agusa. Congruence Properties for a Timed Extension of the π-Calculus. In *Proc. of DSN2005 Workshop: Dependable Software, Tools and Methods*, pages 207 – 214, 2005.
7. J. Y. Lee and J. Zic. On modeling real-time mobile processes. In *Proc. of ACSC'02*, pages 139–147, January 2002.
8. R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, parts I and II. Reports ECS-LFCS-89-85 and ECS-LFCS-89-86 86, Computer Science Dept., University of Edinburgh, March 1989.
9. E. Posse. *Modelling and Simulation of dynamic structure, discrete-event systems*. Ph.d. thesis, School of Computer Science. McGill University, August 2008.
10. E. Posse. A real-time extension to the π-calculus. Tech. Report 2009-557, School of Computing – Queen's University, http://www.cs.queensu.ca, 2009.
11. C. Prisacariu and G. Ciobanu. Timed Distributed π-Calculus. Tech. Report FML-05-01, Institute of Computer Science, Romanian Academy, 2005.
12. D. Sangiorgi. A theory of bisimulation for the π-calculus. Tech. Report ECS-LFCS-93-270, University of Edinburgh, 1993.
13. S. Schneider. An operational semantics for Timed CSP. *Information and Computation*, 1995.
14. D. N. Turner. *The polymorphic Pi-calculus: Theory and Implementation*. Ph.d. thesis, Univ. of Edinburgh, 1996.
15. B. P. Zeigler, H. Praehofer, and T. G. Kim. *Theory of modeling and simulation*. Academic Press, second edition, 2000.