A Native Approach to Modeling Timed Behavior in the Pi-Calculus

Kamal Barakat, Stephan Kowalewski
Embedded Software Laboratory
RWTH Aachen University
Aachen, Germany
{*barakat,kowalewski*}*@embedded.rwth-aachen.de*

Thomas Noll
Software Modeling and Verification Group
RWTH Aachen University
Aachen, Germany
noll@cs.rwth-aachen.de

***Abstract*—We introduce a new concept of modeling timed behavior in pi-calculus by representing timed actions (or timers) as interactions between application processes and clock processes. This approach extends the original calculus in a manner such that bisimulation arrangements in pi-calculus remain untouched. We also present a tool to simulate specifications written in our timed version of pi-calculus in order to verify their behavior.**

***Keywords*-Specification and Verification; Embedded and Real-Time Systems**

I. Introduction

The process-algebraic approach has proved to be a powerful and versatile tool to modeling concurrent systems [1]. Process calculi support the high-level description of interaction, communication, and synchronization operations between a collection of independent processes. They also provide algebraic laws that allow process specifications to be manipulated and analyzed, and permit formal reasoning about equivalences between processes. Famous examples of basic process calculi include CSP, CCS, and ACP. The basis of our work is the extension to mobile systems as it is formalized by the π-calculus [2], [3], [4]. This formalism can model communicating processes with message exchange, and specifies mobility through dynamic channel setup and through privatizing names using restriction.

In contrast to its ability to model dynamic aspects caused by mobility, the expression of quantitative timing properties is less developed or too specific for particular applications (timed spi-calculus [5]). For this reason, our aim is to include timing into the π-calculus in a manner which peruses the suitability for mobile systems as far as possible. Such a coherent calculus can serve as an alternative to timed-automaton based approaches such as [6] which provide no inherent features for dynamic channel setup. In non-mobile settings in contrast, this was already accomplished for ACP [7] and CCS [8].

In this paper we model timed behavior as interactions between application and clock processes. We refrain from modeling time steps (cp. [9], [10], [11], [12]), and use the native inter-process communication concept to model timers. This concept is flexible in the sense that the association of clock processes either with the overall application process or with some of its concurrent subprocesses allows to represent both, global and private clocks.

II. The Classic π-Calculus

π-Calculus is an algebraic model that treats mobility in a native way. One can use this model for representing communication protocols, algorithms, programming languages and data structures. Here we only give a sketch; for details we refer to [2], [3], [4]. We start with introducing its syntax. It is parametrized by a set of *agent identifiers*, $\mathcal{A}$, and a set of names, $\mathcal{N}$. The latter serves as both communication channels and data to be transmitted along them. The set of *process expressions* is given by the following context-free grammar

$$P ::= \sum_{\alpha_i \in \mathcal{N}} \alpha_i.P_i \mid P_1 \parallel P_2 \mid new\ x\ P \mid A\langle \vec{y} \rangle$$

Summation of prefixed expressions represents choice, where *action prefixes* α are determined by

$$\alpha ::= x(y) \mid \overline{x}\langle y \rangle \mid \tau \quad \text{where} \quad x, y \in \mathcal{N}$$

The parallel product operator $P_1 \parallel P_2$ composes P_1 and P_2 concurrently. The restriction operator *new* restricts x to the scope of P. An agent call $A \in \mathcal{A}$ where $x, y \in \mathcal{N}$ requires that the agent identifier has been defined by an equation of the form $A(\vec{x}) := P_A$. An equivalence relation, called *structural congruence*, allows to identify processes which differ only in syntactic details. Its definition can be found in [2].

Input actions are also called *positive*, while output actions are *negative*. Each pair of positive and negative actions makes a redex if they carry the same first name. A redex allows these action prefix pairs to *synchronize* their execution (or to *react*), thereby replacing the second name of the receiver by that of the sender. In this sense, the first name represents a communication channel. This results in a silent τ-action. This is formalized by the following *reaction rule*

$$\text{(React)}\ \frac{P \xrightarrow{x(y)} P' \quad Q \xrightarrow{\overline{x}\langle z \rangle} Q'}{P \parallel Q \xrightarrow{\tau} P'[y \mapsto z] \parallel Q'}$$

Again, the complete transition-system semantics of π-calculus can be found in [2].

III. Adding Real-Time Features

The central idea of our extension of the π-calculus to a timed setting is the explicit introduction of *clock processes*. The appropriate definition of their scope allows us to model both global and private clocks. In the first case, there will be just one (global) clock process, and all application processes have to synchronize their timed operations with it. In the second case, a (private) clock will be associated with every application process that needs to execute internal timed actions not related to actions in other processes. In order to be able to introduce (timed) behavioral equivalences, we will assume that all timed actions are visible to the environment. We refer to our extended version of the calculus by π^τ-calculus. In the following, $\mathbb{T} := \mathbb{R}_{\geq 0}$ denotes the domain of *time values*.

A. Timed Actions and Clock Processes

Each clock process supports the following *timed actions*:

- $c(x, y)$ where $x, y \in \mathcal{N}$, providing synchronization at time point $x + y$ (with base x and offset y), and
- $\overline{c}\langle t\rangle$ where $t \in \mathbb{T}$, providing the current time t.

The *clock process* definition iterates them in a loop:

$$C(c) := (\text{if } \dot{t} = x + y)\, c(x, y).C\langle c\rangle + \overline{c}\langle \dot{t}\rangle.C\langle c\rangle \qquad (1)$$

where $\dot{t}$ denotes the current absolute time. Thus an application process can use the following complementary actions:

- $\overline{c}\langle v, w\rangle$ with $v, w \in \mathcal{N} \cup \mathbb{T}$, asking the clock process to synchronize at time point $v + w$ (where v and w have to be instantiated by $\mathbb{T}$ values), and
- $c(x)$ with $x \in \mathcal{N}$, inquiring the current time and storing it in x.

Using these features, a process P that uses a private clock can be represented by the process expression $\text{new}\, c\,(P \parallel C\langle c\rangle)$. To formalize the interaction between both processes, we add two transition rules that essentially correspond to Rule (React) from the classic calculus. The difference, however, is that the resulting silent τ actions carry additional information. They come in two shapes:

- **active**: $\tau^{t,u}$ where $t, u \in \mathbb{T}$, indicating a synchronization operation at time point $t + u$, and
- **passive**: τ_x where $x \in \mathcal{N}$, indicating a query of the current time which is stored in x.

The corresponding rules are of the following form ($t, u \in \mathbb{T}$, $x \in \mathcal{N}$).

$$\text{(Sync)}\ \frac{C\langle c\rangle \xrightarrow{c(t,u)} C\langle c\rangle \;\; P \xrightarrow{\overline{c}\langle t,u\rangle} P'}{C\langle c\rangle \parallel P \xrightarrow{\tau^{t,u}} C\langle c\rangle \parallel P'}$$

$$\text{(Query)}\ \frac{C\langle c\rangle \xrightarrow{\overline{c}\langle \dot{t}\rangle} C\langle c\rangle \;\; P \xrightarrow{c(x)} P'}{C\langle c\rangle \parallel P \xrightarrow{\tau_x} C\langle c\rangle \parallel P'[x \mapsto \dot{t}]}$$

We assume that neither the synchronization nor the query operation themselves consume time. Moreover, we assume that these two transition rules are handled with higher priority than those of the untimed π-calculus. Technically this can be achieved by explicitly representing clock synchronization and query steps by another relation, say $\Rightarrow$, and by adding negative premises of the form "$P \not\Rightarrow$" to each of the rules of the classic calculus (for appropriate choices of P). This ensures that timed actions are preferred over non-timed ones.

Similar to the approach in [7], each clock in our calculus increases its value internally and accordingly reacts to timed requests of other processes. It is possible to show that the time operators introduced in that paper, such as *delay*, absolute time-out and initialization, can also be defined in π^τ-calculus. By abstracting the passage of time as an internal activity of the clock process, we save the users of our calculus from the formalities that arise from explicitly defining it (e.g. $\triangleright^t$ in [9]).

B. Example

In the following we demonstrate our formalism using a larger example. Here we abbreviate a sequence of $\overline{c}\langle t, u\rangle$ followed by $c(x)$ as a *bidirectional* timed action $c_x^{t,u}$.

Example 1 (Sensor network) *The sensor node S generates data every three seconds and sends it to the receiver R over channel d, while the receiver immediately replies over the same channel with an acknowledgment or sends a reset message over secure channel r - no message loss - if the sensor fails to provide data in four Seconds. Upon reception of a reset message, the sensor resets its timer and starts over. The malicious process M starts after 15Sec to attack the channel d causing message loss. We model the sensor process as an automaton of type $TYPES$:* ← begin TYPES

$$\begin{aligned} SEN1 &= r.c_t.SEN1 + c_t^{t,3Sec}.SEN2 \\ SEN2 &= r.c_t.SEN1 + \overline{d}\langle data\rangle.SEN3 \\ SEN3 &= r.c_t.SEN1 + d(x).SEN1 \end{aligned}$$

and define the process S from this type: ← end

$$S := TYPES$$

We define the receiver process R as the following:

$$\begin{aligned} R &:= c_s^{s,4Sec}.\overline{r}.R + d(y).c_s.R' \\ R' &:= c_s^{s,4Sec}.\overline{r}.R + \overline{d}\langle ack\rangle.R \end{aligned}$$

The malicious process M represents an external failure source that randomly absorbs *messages on d:*

$$\begin{aligned} M &:= c_u.M' \\ M' &:= (\text{if } u \geq 15Sec)\, d(z).\tau.M + \tau.M \end{aligned}$$

We will address constraints in our timed system in section IV. The system process consists of all above processes running in parallel in addition to the clock process $C(c)$ as described in (1)*:*

$$SYS := S \parallel R \parallel M \parallel C\langle c\rangle \qquad \blacksquare$$

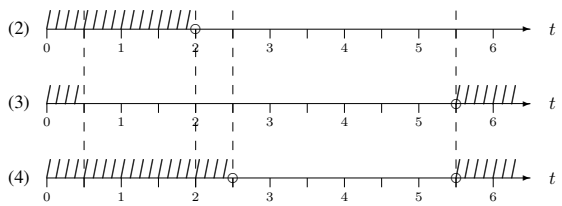

Figure 1. Evaluating the Ω function (Example 2)

IV. Extension by Time Constraints

A. Time Constraints

We extend the classic conditional prefixing of process expressions as suggested by Parrow [3]. The standard *match* and *mismatch* constructs only refer to the (in-)equality of names. In our extension names can also be instantiated by time values. Therefore, we must extend these constraints to do simple arithmetic comparisons on these values.

Definition 1 (Time constraints) *Let $x, y \in \mathcal{N}$, $t \in \mathbb{T}$, and let $\bowtie \in \{<, >, \leq, \geq, =, \neq\}$. Let $\delta ::= x - y \mid x - t \mid x$. A* (time) constraint *is a list of (in-)equations of the form $C = C_1 C_2 \dots C_n$ where $C_i = (\text{if } \delta \bowtie 0)$. A constraint with all names instantiated by values in $\mathbb{T}$* holds *if all of its (in-)equations are satisfied.*

If a timed action is constrained, its execution proceeds if the constraint holds. Each timed action prefix defines one or more points in time where it can fire. Constraints restrict this set to a subset of possible execution opportunities. In the following we address this point formally.

B. Semantics of Constraints

A mechanism is needed to dynamically quantify available execution points in time of timed action prefixes. We therefore introduce the Ω function. For a given constraint C and a subsequent timed action η, Ω yields the set of time points (that is, a subset of $\mathbb{T}$) at which $C\eta$ executes. See [13] for it's formal definition; here we consider an example.

Example 2 *In this example, $A(s,t)$ is a sequential composition of $A_1(s)$ and $A_2(t)$. Figure 1 shows the evaluation using the Ω function on each of those timed processes.*

$$A_1(t) := (\text{if } t > 2)\, \overline{c}\langle 0, t\rangle \quad (2)$$

$$A_2(s) := (\text{if } s < 5)\, c_s^{s,0.5}. \quad (3)$$

$$A(s,t) := (\text{if } t > 2)\, \overline{c}\langle 0, t\rangle.(\text{if } s < 5)\, c_s^{s,0.5} \quad (4)$$

Evaluating Ω on $A_1(t)$ (for arbitrary t) produces $]2, \infty[$, while evaluating Ω on $A_2(s)$ yields $[0.5, 5.5[$. The composition of both processes evaluates to $]2.5, 5.5[$. ■

Combining restricted action prefixes reduces the total number of execution times of them as a sequence. In extreme cases, a sequence of constrained timed actions can block.

Definition 2 (Non-blocking sequence) *Let $C_1 c_{x_1}^{p_1,q_1} \dots C_k c_{x_k}^{p_k,q_k}$ be a sequence of timed actions and their constraints. This sequence is called* non-blocking *if, for all $1 \leq i \leq k$, $\Omega(C_i, c_{x_i}^{p_i,q_i})$ is non-empty and, for all $1 \leq i < k$, $p_i + q_i \leq p_{i+1} + p_{i+1}$.*

This property means in effect that a concatenation of timed actions and their constraints allows the consecutive execution of all its actions; no action prefix finishes after the start schedule of the next one and no constraint completely blocks the execution of its timed action prefix. We suggest an algorithm for reducing sequences of timed action prefixes. This is necessary because it is easier to compare two different sequences by comparing their reductions.

Lemma 1 (Reduction of non-blocking sequences) *Any non-blocking sequence $C_1\eta_1 \dots C_k\eta_k$ can be reduced to some $C\eta$.*

Proof sketch: Assume the sequence $C_1\eta_1.C_2\eta_2$. The basic idea is to shift the constraints of the second action prefix, C_2, one position to the left after performing necessary alpha-conversion. This produces a structurally congruent sequence $C_1 C_2'\eta_1.\eta_2$. We merge next η_1 and η_2 into one action prefix using structural congruence rules and alpha-conversion. This is possible because of the assumption that the overall sequence is non-blocking. ■

If the number of timed actions in the original sequence is greater than two, we apply the algorithm recursively starting by the two leftmost elements, and proceed by replacing them with the result of the reduction. A formal description of the algorithm is given in [13].

Example 3 *Assume a sequence of four elements:*

$$P := c_s.(\text{if } s > 10)\, c_s^{s,2}.c_s.(\text{if } s < 14)\, c^{s,4}$$

We go through the reduction step by step:

1) *Create a new sequence P' by inserting a passive timer with some new name, say $\tilde{t}$, in front of P so that:*

$$P' := c_{\tilde{t}} c_s.(\text{if } s > 10)\, c_s^{s,2}.c_s.(\text{if } s < 14)\, c^{s,4}$$

$\tilde{t}$ becomes a reference to the start of execution, while P' still retains the timed behavior of P because passive timers cause no delay, hence $P \equiv P'$. This step simplifies comparing reductions of different sequences and has to be done once in the first iteration.

2) *Passive consecutive timers execute instantly when unguarded and consume no execution time. We merge them together in one action prefix:*

$$P' \equiv c_{\tilde{t},s}(\text{if } s > 10)\, c_s^{s,2}.c_s.(\text{if } s < 14)\, c^{s,4}$$

3) *Shift the constraint $(\text{if } s > 10)$ one position to the left by alpha-converting s in it to its equivalent $\tilde{t}$:*

$$P' \equiv (\text{if } \tilde{t} > 10) c_{\tilde{t},s}.c_s^{s,2}.c_s.(\text{if } s < 14)\, c^{s,4}$$

4) *Transfer information from $c_{\tilde{t},s}$ to $c_s^{s,2}$ by alpha-converting all shared names between the set of bound*

names of $c_{\tilde{t},s}$ and the set of the free names in $c_s^{s,2}$, and then merge c_s with the result:

$$P' \equiv (\text{if } \tilde{t} > 10)c_s^{\tilde{t},2}.(\text{if } s < 14)\, c^{s,4}$$

5) *Similar to steps 3 & 4, the s in $(\text{if } s < 14)$ and in the superscript of $c^{s,4}$ is alpha-converted to $\tilde{t} + 2$:*

$$P' \equiv (\text{if } \tilde{t} > 10)(\text{if } \tilde{t} + 2 < 14)c^{\tilde{t}+2,4} \qquad \blacksquare$$

C. Structural Congruence of Timed Actions

Our timer system makes the consequent arrangements to structural congruence simple.

Definition 3 (Timed structural congruence) *Let both $C_1\eta_1 \ldots C_k\eta_k$ and $C'_1\eta'_1 \ldots C'_l\eta'_l$ be sequences of constrained timed actions.*

1) *If both sequences are blocking, then $C_1\eta_1 \ldots C_k\eta_k \equiv C'_1\eta'_1 \ldots C'_l\eta'_l \equiv \mathbf{0}$.*
2) *Otherwise, let $Cc_x^{p,q}$ and $C'c_{x'}^{p',q'}$ be the corresponding reductions according to Lemma 1. If $p + q = p' + q'$ and $\Omega(C, c_x^{p,q}) = \Omega(C', c_{x'}^{p',q'})$, then $C_1\eta_1 \ldots C_k\eta_k \equiv C'_1\eta'_1 \ldots C'_l\eta'_l$.*

Rule 1 defines deadlocking and stresses the continuity and universality of time in π^τ-calculus. If the current time exceeds the threshold of a timed action, this action becomes permanently inaccessible and can be *garbage collected* by utilizing the congruence with the empty process. The including context will deadlock if no other execution paths are available. Rule 2 says that if reductions of sequences coincide and none of them is blocking then they are structurally congruent, e.g., $c_s^{0,2}.\overline{c}\langle s, 1\rangle$ and $\overline{c}\langle 0, 3\rangle$.

D. Strong and Weak Bisimulation

Since timers in π^τ-calculus are mere I/O actions, our extension does not impose changes to native bisimulation in π-calculus. If the application process synchronizes its timer(s) with a global clock, the external observer detects these timer reactions and strong bisimulation can be deduced about different implementations of application processes. If the application process, however, synchronizes some or all of its timers with a private clock, the external observer perceives a silent (timed) τ-action as the result of the interaction between those time agents, hence weak bisimulation applies. See [13] for examples on timed bisimulation.

V. Tool Implementation

Our tool implementation aims to provide a π^τ-calculus simulating mechanism to evaluate timed telecommunication protocols at an abstract level. Our simulation tool[1] assumes that specifications are written in LaTeX. The parts of the LaTeX document that contain π^τ-calculus relevant syntax are redefined math environments and commands to which our compiler is sensitive. Process definitions are translated into metadata which the interpreter understands. In the metadata names are replaced with unique number IDs, which simplifies many aspects of the execution like alpha-conversion and parameter passing and facilitates exporting the specification to platforms such as UPPAAL. The interpreter's output is a log file that registers all inter-process communications with their time-stamps.

[1] Available at http://web.embedded.rwth-aachen.de/pical/.

Acknowledgement: This work was funded by the DFG Cluster of Excellence on Ultra-high Speed Information and Communication (UMIC), German Research Foundation grant DFG EXC 89.

References

[1] J. Bergstra, A. Ponse, and S. Smolka, *Handbook of Process Algebra*. Elsevier Science Ltd, 2001.

[2] R. Milner, *Communicating and mobile systems: the π-calculus*. Cambridge Univ. Press, 1999.

[3] J. Parrow, "An introduction to the π-calculus," in *Handbook of Process Algebra*, J. A. Bergstra, A. Ponse, and S. A. Smolka, Eds. Elsevier Science Ltd, 2001, ch. 8, pp. 479–543.

[4] D. Sangiorgi and D. Walker, *The π-calculus: a Theory of Mobile Processes*. Cambridge University Press, 2001.

[5] C. Haack and A. Jeffrey, "Timed spi-calculus with types for secrecy and authenticity," in *CONCUR 2005 – Concurrency Theory*, ser. Lecture Notes in Computer Science, vol. 3653, 2005, pp. 202–216.

[6] N. Lynch, R. Segala, and F. Vaandrager, "Hybrid I/O automata," *Information and Computation*, no. 1, pp. 105 – 157.

[7] J. Baeten and C. Middelburg, "Process algebra with timing: real time and discrete time," in *Handbook of Process Algebra*, J. A. Bergstra, A. Ponse, and S. A. Smolka, Eds. Elsevier Science Ltd, 2001, ch. 10, pp. 627–684.

[8] M. Hennessy and T. Regan, "A process algebra for timed systems," *Information and Computation*, no. 2, pp. 221 – 239.

[9] M. Berger and N. Yoshida, "Timed, distributed, probabilistic, typed processes," *Programming Languages and Systems*, pp. 158–174, 2009.

[10] W. Jin, H. Wang, and M. Zhu, "Modeling MARTE sequence diagram with timing pi-calculus," in *14th IEEE Int. Symp. on Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC 2011)*. IEEE, 2011, pp. 61–66.

[11] Q. Wang, C. Du, C. Ma, and G. Li, "Extension of TD-pi calculus in real-time distributed virtual-test system description," in *Int. Conf. on Computer Science and Software Engineering*, vol. 3. IEEE, Dec. 2008, pp. 363–369.

[12] J. Lee and J. Zic, "On modeling real-time mobile processes," in *Proc. of 25th Australasian Conference on Computer Science (ACSC 2002)*, ser. CRPIT, vol. 4, Citeseer. ACS, pp. 139–147.

[13] K. Barakat, "Introducing timers to π-calculus," RWTH Aachen, Tech. Rep. AIB-2011-18, Aug.