Mobile processes with local clocks *

Pierpaolo Degano, Jean-Vincent Loddo and Corrado Priami

Dipartimento di Informatica, Università di Pisa
Corso Italia, 40, I-56125 Pisa, Italy
{degano,priami}@di.unipi.it

Abstract. We propose a structural operational semantics that expresses temporal aspects of mobile and distributed systems, each sequential component of which has its local clock. Since the run-time support of a programming language implements the operations of the language via some lower-level routines, the same action, put in different contexts, may have different durations. Also the network topology affects these durations, typically when messages are exchanged. We model this through a transition system labelled by actions and their costs, in a discrete time. Then, we define two performance preordings that say when the execution of a process is faster than that of another. The first preorder is similar to those presented in the literature, while the second refines it in that it considers a process faster than another if it is such from some point onwards of its execution. Finally, as an example we compare the performance of a conventional uniprocessor architecture with a prefetch pipeline architecture.

1 Introduction

We propose a timed semantics for mobile processes, in which actions do not occur instantaneously, but require some (discrete) time to be executed. Unlike other proposals, we assume that actions *do not* have a fixed duration. This is a further step towards a more accurate description of the behaviour of processes, because the time an action a consumes depends on the basic operations that the run-time support of the target architecture performs for firing a. Typically, the resolution of a choice imposes some operations on the target architecture such as checking the ready list or implementing fairness policies. Therefore, in practice an action fired after a choice costs more than the same action occurring deterministically. Note that the run-time routine implementing action a is represented in the usual SOS descriptions of processes by the context in which a occurs.

As an example, consider the process

$$(a.P|b.Q) + R$$

where ".", "|" and "+" are interpreted as prefixing, parallel and nondeterministic composition as usual. To fire action a, the run time support spends some time,

* Work partially supported by ESPRIT BRA n. 8130 - LOMAPS.

say t, to choose the sub-process $(a.P|b.Q)$. So, the time needed to complete action b must include also t, even if b is independent of a. Also, in order to tune the cost of actions with the architecture of the network, we establish that communication costs depend on the allocation of processes, on the link used and on the size of the message exchanged. Typically, a communication between processes placed on close nodes is cheaper than a communication involving extensive routing.

We define a structural operational semantics of the π-calculus that associates any sequential component of a system with its own clock, in a truly concurrent fashion. It is a conservative extension of the original semantics given in [20]. Our starting point is an enhanced version of operational semantics from which one smoothly obtains a hierarchy of specifications of concurrent systems closer and closer to implementations. In this way, it is possible to analyse both qualitative and quantitative aspects of systems on tightly related formal descriptions [11]. More precisely, we exploit proved transition systems, a parametric model used in [9, 10] to describe uniformly different qualitative aspects of processes. The transitions of these transition systems are labelled by encodings of their proofs. Then, by inspecting these richer labels we rebuild the operations needed for deducing actions and we can derive their context sensitive costs, reflecting the target architecture. Remarkably, the knowledge of the interacting processes in a communication permits to make their allocation influence costs. In the actual SOS definition of the operational semantics of the π-calculus, the cost function is parametric and can be instantiated to reflect different target architectures. In this way, the same definition can describe the behaviour of a system running on different targets. For simplicity, we consider here networks of homogeneous machines only. The extension of our proposal to heterogeneous networks is only a matter of longer definitions, especially those expressing the costs of operations.

We also define two preorders which say when a process is more efficient than another. Both relations require that the processes to be compared are bisimilar. The first preorder is similar to other proposals (see below) and requires the faster process to be such from the very beginning of its execution. The second preorder refines the first one in that it requires the faster process to perform better from a given point of its execution onwards. Our preorders compare two specifications running on the same architecture. This is the case the literature deals with, and in our framework it requires to keep fixed the function that estimates operation costs, and let only the algorithm change. As our description is parametric, we can also compare a *single* specification when it runs on *different* networks of processors, i.e., we can compare different architectures (operations have different costs) on a specific task.

A brief discussion of some work similar to ours follows. Temporal extensions of *CCS* are proposed in [1, 13, 12, 14]. In these approaches any sequential component of the system has its private clock and any action has a fixed duration expressed by a natural number. Then equivalences are defined over timed processes in the style of the interleaving bisimulation equivalence. Another extension of *CCS* is presented in [7] where the authors define a preordering to establish when a process is more efficient than another. Apart from being parametric and mobility sensitive, our first preorder is essentially that of [7]. Finally, we mention also a temporal model based on continuous time [15], where higher-order transitions and computations are given a geometric description.

Temporal extensions of process algebra not directly related to our proposal may be found in [23, 24, 3, 21, 16, 17, 18]. A survey of these approaches can be found in [22]. Many other models for timed behaviours have been proposed in the literature, e.g. those based on Kripke structures, automata, temporal logics. We do not discuss them here and we refer the interested reader to for instance the proceedings of CONCUR conferences in the LNCS series (for the last editions the volume numbers are 962 and 1119).

This paper is organized as follows. In the next section we recall π-calculus. Its proved semantics is defined in Sect. 3 and it is used to derive the timed semantics in Sect. 4. Section 5 introduces our amortized preordering. An example is discussed in Sect. 6.

2 π-Calculus

In this section we briefly recall π-calculus [20, 19], a model of concurrent communicating processes providing the notion of *naming*.

Definition 1.
Let $\mathcal{N}$ be a countably infinite set of *names* ranged over by $a, b, \ldots, x, y, \ldots$ and let $\mathcal{S} = \{\tau_0, \tau_1, \tau_2, \ldots\}$ be a countably infinite set of invisible actions ranged over by τ_i, with $\mathcal{N} \cap \mathcal{S} = \emptyset$. We also assume a set of *agent identifiers* ranged over by $A, A_1, \ldots$. *Processes* (denoted by $P, Q, R, \ldots \in \mathcal{P}$) are built from names according to the syntax

$$P ::= \mathbf{0} \mid \pi.P \mid P + P \mid P|P \mid (\nu x)P \mid [x = y]P \mid A(y_1, \ldots, y_n)$$

where π may be either $x(y)$ for *input*, or $\overline{x}y$ for *output* (where x is the *subject* and y the *object*, extracted by functions *sbj* and *obj*, respectively) or τ_i for *silent* moves. Hereafter, the trailing $\mathbf{0}$ will be omitted.

The prefix π is the first atomic action that the process $\pi.P$ can perform. The input prefix binds the name y in the prefixed process. Intuitively, some name y is received along the link named x. The output prefix does not bind the name y which is sent along x. A silent prefix τ_i denotes an action which is invisible to an external observer of the system. We use a set of silent prefixes because different internal actions will have different durations. Typically, communications require different times. Summation ($+$) denotes nondeterministic choice. The operator $|$ describes parallel composition of processes. The operator (νx) acts as a static binder for the name x in the process P that it prefixes. In other words, x is a unique name in P which is different from all the external names. Finally, matching $[x = y]P$ is an `if-then` operator: process P is activated if $x = y$. $A(y_1, \ldots, y_n)$ is the definition of constants (hereafter, $\tilde{y}$ denotes $y_1, \ldots, y_n$).

The early operational semantics for the π-calculus is defined in the *SOS* style, and the labels of the transitions are τ for silent actions, xy for free input, $\overline{x}y$ for free output, and $\overline{x}(y)$ for bound output. We will use μ as a metavariable for the labels of transitions (it is distinct from π, the metavariable for prefixes, though it coincides in two cases). We introduce the set $\mathcal{A}$ of visible actions ranged over by α (i.e. either $\mu = \alpha$ and the action is visible or $\mu = \tau$ and is invisible). We sometimes write $(\nu x, y)P$ for $(\nu x)(\nu y)P$. We recall the notion of free names $fn(\mu)$, bound names $bn(\mu)$, and names $n(\mu) = fn(\mu) \cup bn(\mu)$ of a label μ.

μ	*Kind*	$fn(\mu)$	$bn(\mu)$
τ	Silent	$\emptyset$	$\emptyset$
$xy, \overline{x}y$	Free Input and Output	$\{x, y\}$	$\emptyset$
$\overline{x}(y)$	Bound Output	$\{x\}$	$\{y\}$

Functions *fn*, *bn* and *n* are extended in the obvious way to processes. Below we assume that the *structural congruence* $\equiv$ on processes is defined as the least congruence satisfying the following clauses:

- if P and Q are α-equivalent (they only differ in the choice of bound names) then $P \equiv Q$,
- $(\nu x)(\nu y)P \equiv (\nu y)(\nu x)P$, $(\nu x)(R \,|\, S) \equiv (\nu x)R \,|\, S$ if $x \notin fn(S)$, $(\nu x)(R \,|\, S) \equiv R \,|\, (\nu x)S$ if $x \notin fn(R)$, and $(\nu x)P \equiv P$ if $x \notin fn(P)$.

A *variant* of $P \xrightarrow{\mu} Q$ is a transition which only differs in that P and Q have been replaced by structurally congruent processes, and μ has been α-converted, where a name bound in μ includes Q in its scope.

The early transition system for the π-calculus is in Tab. 1. The transition in the conclusion of each rule, as well as in the axiom, stands for all its variants. The table only reports one rule for the binary operators $|$ and $+$; they also have a symmetric rule.

We end this section with the definition of the interleaving weak bisimulation.

Definition 2. A binary relation $\mathcal{R}$ on processes is a *weak simulation* if whenever $(P, Q) \in \mathcal{R}$ implies

- If $Q \xrightarrow{\mu} Q'$, with $bn(\mu) \cap fn(P, Q) = \emptyset$, then for some P', $P \overset{\mu}{\Rightarrow} P'$, and $(P', Q') \in \mathcal{R}$.

where $\overset{\mu}{\Rightarrow}$ is defined as $\overset{\tau}{\Rightarrow} = (\xrightarrow{\tau})^*$ and $\overset{\alpha}{\Rightarrow} = \overset{\tau}{\Rightarrow} \circ \xrightarrow{\alpha} \circ \overset{\tau}{\Rightarrow}$. The relation $\mathcal{R}$ is a *weak bisimulation* if $\mathcal{R}$ and $\mathcal{R}^{-1}$ are weak simulations. We say that P and Q are *weakly equivalent*, in symbols $P \approx Q$, if there exists a weak bisimulation $\mathcal{R}$ such that $(P, Q) \in \mathcal{R}$.

$$Out : \overline{x}y.P \xrightarrow{\overline{x}y} P \quad Ein : x(y).P \xrightarrow{xw} P\{w/y\} \quad Sil : \tau_i.P \xrightarrow{\tau} P$$

$$Par : \frac{P \xrightarrow{\mu} P'}{P|Q \xrightarrow{\mu} P'|Q}, bn(\mu) \cap fn(Q) = \emptyset \qquad Sum : \frac{P \xrightarrow{\mu} P'}{P + Q \xrightarrow{\mu} P'}$$

$$Res : \frac{P \xrightarrow{\mu} P'}{(\nu x)P \xrightarrow{\mu} (\nu x)P'}, x \notin n(\mu) \qquad Open : \frac{P \xrightarrow{\overline{x}y} P'}{(\nu y)P \xrightarrow{\overline{x}(y)} P'}, y \neq x$$

$$Close : \frac{P \xrightarrow{\overline{x}(y)} P', Q \xrightarrow{xy} Q'}{P|Q \xrightarrow{\tau} (\nu y)(P'|Q')},\ y \notin fn(Q) \qquad Com : \frac{P \xrightarrow{\overline{x}y} P', Q \xrightarrow{xy} Q'}{P|Q \xrightarrow{\tau} P'|Q'}$$

$$Match : \frac{P \xrightarrow{\mu} P'}{[x = x]P \xrightarrow{\mu} P'} \qquad Ide : \frac{P\{\tilde{y}/\tilde{x}\} \xrightarrow{\mu} P'}{Q(\tilde{y}) \xrightarrow{\mu} P'}, Q(\tilde{x}) = P$$

Table 1. Early transition system for the π-calculus.

3 Proved Transition System

We enrich the labels of the standard interleaving transition system in order to encode more information, in the style of [8, 6, 9]. Essentially, the label of a transition records the inference rules used during its deduction, besides the action itself. It is then possible to derive different semantic models for the π-calculus by extracting new kinds of labels from the enriched ones. In [10] the first and the last authors studied non interleaving qualitative aspects of the π-calculus. In the next sections we apply similar techniques to derive a machine-dependent temporal model. Processes are given a non interleaving semantics, so it is possible to single out the sequential components of a system of mobile processes, or even groups of them residing on the same processor. This permits to make costs dependent on the topology of the underlying network and to associate each sequential (group of) processes with their own local clocks. We start with the definition of our *enhanced labels*. In addition, we extract only the encoding of the proof from an enhanced label, called *proof term*. Finally, we introduce a function ℓ that takes an enhanced label to the corresponding standard action label.

Definition 3. Let $\mathcal{L} = \{||_0, ||_1\}$ with $\chi \in \mathcal{L}^*$, $\mathcal{O} = \{+_0, +_1, =, ()\} \ni o$ and let $\vartheta \in (\mathcal{L} \cup \mathcal{O})^*$. Then *enhanced labels* (with metavariable θ) are defined by the following syntax

$$\theta ::= \vartheta\alpha \mid \vartheta\tau_i \mid \vartheta\langle ||_0\vartheta_0\alpha_0, ||_1\vartheta_1\alpha_1\rangle$$

with $\alpha_0 = xy$ iff α_1 is either $\overline{x}(y)$ or $\overline{x}y$, or vice versa.

The set $\mathcal{T}$ of *proof terms* is obtained from enhanced labels via the auxiliary function p defined as $p(\vartheta\alpha) = p(\vartheta\tau_i) = \vartheta$, $p(\vartheta\langle\vartheta_0\alpha_0, \vartheta_1\alpha_1\rangle) = \vartheta\langle\vartheta_0, \vartheta_1\rangle$. Function ℓ is defined as $\ell(\vartheta\alpha) = \alpha$, $\ell(\vartheta\tau_i) = \ell(\vartheta\langle\vartheta_0\alpha_0, \vartheta_1\alpha_1\rangle) = \tau$.

Our version of the early transition system for the π-calculus is in Tab. 2, where the symmetric rules for communication (Com_1 and $Close_1$) are omitted. Again, the transitions in the conclusion of each rule stand for all their variants. We call this transition system *proved*, because the labels of the transitions encode portions of their proofs.

$$Act : \pi.P \xrightarrow{\pi} P,\ \pi \text{ not input} \qquad Ein : x(y).P \xrightarrow{xw} P\{w/y\}$$

$$Par_0 : \frac{P \xrightarrow{\theta} P'}{P|Q \xrightarrow{||_0\theta} P'|Q}, bn(\ell(\theta)) \cap fn(Q) = \emptyset \qquad Sum_0 : \frac{P \xrightarrow{\theta} P'}{P + Q \xrightarrow{+_0\theta} P'}$$

$$Par_1 : \frac{P \xrightarrow{\theta} P'}{Q|P \xrightarrow{||_1\theta} Q|P'}, bn(\ell(\theta)) \cap fn(Q) = \emptyset \qquad Sum_1 : \frac{P \xrightarrow{\theta} P'}{Q + P \xrightarrow{+_1\theta} P'}$$

$$Res : \frac{P \xrightarrow{\theta} P'}{(\nu x)P \xrightarrow{\theta} (\nu x)P'}, x \notin n(\ell(\theta)) \qquad Match : \frac{P \xrightarrow{\theta} P'}{[x = x]P \xrightarrow{=\theta} P'}$$

$$Close_0 : \frac{P \xrightarrow{\vartheta\overline{x}(y)} P', Q \xrightarrow{\vartheta' xy} Q'}{P|Q \xrightarrow{\langle ||_0\vartheta\overline{x}(y), ||_1\vartheta' xy\rangle} (\nu y)(P'|Q')},\ y \notin fn(Q) \qquad Open : \frac{P \xrightarrow{\vartheta\overline{x}y} P'}{(\nu y)P \xrightarrow{\vartheta\overline{x}(y)} P'}, x \neq y$$

$$Com_0 : \frac{P \xrightarrow{\vartheta\overline{x}y} P', Q \xrightarrow{\vartheta' xy} Q'}{P|Q \xrightarrow{\langle ||_0\vartheta\overline{x}y, ||_1\vartheta' xy\rangle} P'|Q'} \qquad Ide : \frac{P\{\tilde{y}/\tilde{x}\} \xrightarrow{\theta} P'}{Q(\tilde{y}) \xrightarrow{()\theta} P'}, Q(\tilde{x}) = P$$

Table 2. Early proved transition system for the π-calculus.

The proved transition system differs from the standard one only in the labels of transitions. Rule Par_0 (Par_1) adds to the label a tag $||_0$ ($||_1$) to record that the left (right) component is moving. A tag $+_0$ ($+_1$) means that a nondeterministic choice has been resolved in favour of its left (right) summand. The rules Com_0 and $Close_0$ have in their conclusion a pair instead of a τ to record the components which interacted (and the proof of the relevant transitions). Their symmetric version Com_1 and $Close_1$ are obvious. We record the resolution of a matching through tag $=$. Finally, the invocation of a definition enriches the label of the conclusion of rule *Ide* with the tag $()$.

Hereafter, we will write a transition $P \xrightarrow{\theta} Q$ simply as θ, when unambiguous.

The standard interleaving semantics is obtained from the proved transition system by relabelling each transition through function ℓ in Def. 3, as stated by the following proposition. Its proof is by straightforward induction.

Proposition 4. *$P \xrightarrow{\mu} P'$ is an early transition if and only if there exists θ with $\ell(\theta) = \mu$ such that $P \xrightarrow{\theta} P'$ is an early proved transition.*

4 Timed semantics

In this section we derive a temporal model of the π-calculus from the proved transition system. The new semantics takes into account the target architecture on which a system runs. Essentially, we relabel the proved transitions with new labels, consisting of a pair $\langle \alpha, t \rangle$, where α is an action whose execution is completed at time t. The label for invisible actions is simply τ, yet care is taken to record the time needed to execute it. Since our model is truly concurrent, the execution of actions on different processors may overlap in time. We give an example of overlapping actions in the conclusions. We will see below how local clocks are associated with the sequential components of a system.

As mentioned in the Introduction, we wish to reflect the different duration of actions due to the different needs of their run-time support routines. Since the structural operational semantics of a language specifies its abstract machine, the context in which an action a occurs in the program represents the operations that the target machine performs for firing a. Accordingly, a suitable linearization of the deduction of a transition represents the execution of the corresponding run-time support routines on the target machine. Following this intuition, we assign a cost (in discrete time) to each inference rule of the operational semantics via a function denoted by \$. Of course, this comprises an estimate of the duration of the actions. In other words, the occurrence of a transition receives a duration time computed according to its deduction.

As a matter of fact, we do not fix function \$, and we let it be a parameter to the definition of our timed transition system. In this way, it is possible to estimate the performance of different architectures, each with its own cost function.

More formally, function \$ has the following type

$$\$: \mathcal{A} \cup \mathcal{S} \cup \mathcal{C} \cup \mathcal{O} \to I\!N$$

where $\mathcal{C} = \mathcal{N} \times \mathcal{N} \times \mathcal{L}^* \times \mathcal{L}^*$.

When applied to a visible action $\alpha \in \mathcal{A}$, function \$ returns the time needed to perform the very basic, low-level operations corresponding to α, independently of the context in which α occurs. Similarly for the invisible actions τ_i that are not communications. Different actions in $\mathcal{S}$ are given different costs, just to model different delays a system may exhibit. This is why we introduced a set of indexed internal actions. For communications, we propose a more complex cost function. A call has the format $\$(x, y, \chi_0, \chi_1)$. The first argument is the link $x \in \mathcal{N}$, whose bandwidth affects the cost of a communication. Also the size of the datum $x \in \mathcal{N}$ does, so the message exchanged is the second argument. The last two arguments are used to record which sub-processes actually communicate, rather than the locations where they reside. Indeed, the distance between them is a relevant parameter for computing the time needed to exchange a message. Collecting the above considerations, a typical function assigning costs to communications may be as follows

$$\$(x, y, \chi_0, \chi_1) = bandwidth(x) \times size(y) \times distance(\chi_0, \chi_1).$$

For encoding locations, we use here $\chi_0, \chi_1 \in \{||_0, ||_1\}^*$. To see why, consider the binary abstract syntax tree of a process, when the parallel composition $|$ is the only syntactic operator. Then, a sequence χ can be seen as the access path from the root, i.e. from the whole process, to a leaf, i.e. to a sub-process. So, the two last arguments, together with allocation tables, can be used to determine where the two communicating processes have been actually placed. Function \$ applied to an operator $o \in \mathcal{O}$ estimates the cost needed to execute the run-time support routines implementing o. For example, a typical definition of $\$(())$ should take into account the access to the definition of the identifier P, the binding of the actual parameters $\tilde{y}$ to the formal ones $\tilde{x}$ and the operations on the control stack.

The configurations of our timed transition systems are pairs made up of a process and a function, that we call *timer*, that associates a local time to each sequential component of the process. This is analogous to the common notion of configuration used in operational semantics of imperative languages, made up of a program and a store. More precisely, a *timer* is a total function

$$\sigma : \mathcal{L}^* \to I\!N.$$

For example, $\sigma(||_0||_1) = 7$ records that the actions of the sub-process Q of $(P|Q)|R$ can only start at time 8. In a sense, $\sigma(||_0||_1)$ *is* the local clock of Q, as $||_0||_1$ uniquely identifies Q within the running process. A timer can then be seen as the collection of the local clocks associated with all the active sub-processes of a given system, plus some local clocks that however are useless and will never be used improperly, as we show later on. For example, consider again the process above, in which the location $||_0$ corresponds to no sub-process any more. However, $\sigma(||_0)$ can also have a value, because, say, $(P|Q)|R$ has been reached from $\alpha.(P|Q)|R$ after firing α. Analogously, all the locations not yet used, e.g. of the form $||_0||_1||_0\chi$, will receive a value, even if it will never be accessed. This is for technical reasons made clear below, and ensures that our timers are total.

We now define the initial timer and two further operations on timers. Since we assume that timed computations start at time 0, we define below the initial timer as the constant zero function. The other operations are needed to modify timers. The first updates timers, and is analogous to the standard one for stores, $[value/identifier]$. The second one simply adds up the values of timers on corresponding locations.

Definition 5. The set of timers Σ, ranged over by $\sigma : \mathcal{L}^* \to I\!N$, is the smallest set that contains the initial timer and is closed under the (primitive recursive) operations of update and sum defined hereafter.

The *initial* timer is the constant function $\mathbf{0}_\sigma :\to \Sigma$, defined as

$$\mathbf{0}_\sigma = \lambda\chi.\, 0.$$

The *update* function $\cdot[\cdot/\cdot] : \Sigma \times \mathcal{N} \times \mathcal{L}^* \to \Sigma$ is defined as

$$\sigma[t/\chi_0] = \lambda\chi.\, \texttt{if } \chi = \chi_0\chi_1 \texttt{ then } t \texttt{ else } \sigma(\chi).$$

The *sum* function $\cdot + \cdot : \Sigma \times \Sigma \to \Sigma$ is defined as

$$\sigma + \sigma' = \lambda\chi.\sigma(\chi) + \sigma'(\chi).$$

A few comments are in order. First, note that the above definition is effective. To explain the update function, consider again the process $\alpha.(P|Q)|R$ introduced above. Assume a timer such that $\sigma(||_0) = 3$, and let action α require 4 time units to occur. After firing α, P and Q become active in two different locations, namely $||_0||_0$ and $||_0||_1$. Also, their local clocks should change their value to 7. This is made by calling the update function as follows: $\sigma[7/||_0]$, where $||_0$ is the location where α actually occurred. As noted above, all the locations with prefix $||_0$ will be updated, including $||_0||_0$ and $||_0||_1$.

The following is a simple auxiliary function. Roughly, it extracts the locations $\chi \in \{||_0, ||_1\}^*$ of sub-processes involved in a proved transition from its enhanced label.

Definition 6. The function $\underline{\cdot} : \{\mathcal{L} \cup \mathcal{O}\}^* \to \mathcal{L}^*$ is inductively defined as follows.

$$\underline{\epsilon} = \epsilon, \quad \underline{||_i\vartheta} = ||_i\underline{\vartheta} \quad \underline{o\vartheta} = \underline{\vartheta}$$

We now define by induction an auxiliary function $\mathcal{F}$ that computes the time elapsed to execute the run-time support routines implementing an action. It produces a timer given a proof term and a location. Recall that a proof term encodes the context in which an action occurs, so it represents the calls to the run-time support routines needed to enable the occurrence of that action. We will shortly comment on the next definition below.

Definition 7. The function $\mathcal{F} : \mathcal{T} \times \mathcal{L}^* \to \Sigma$ is defined as

1. $\mathcal{F}(\epsilon, \chi) = \mathbf{0}_\sigma$
2. $\mathcal{F}(o\varsigma, \chi) = \mathbf{0}_\sigma[\$(o)/\chi] + \mathcal{F}(\varsigma, \chi)$,
3. $\mathcal{F}(||_i\varsigma, \chi) = \mathcal{F}(\varsigma, \chi||_i)$,
4. $\mathcal{F}(\langle\vartheta_0, \vartheta_1\rangle, \chi) = \mathcal{F}(\vartheta_0, \chi) + \mathcal{F}(\vartheta_1, \chi)$

When there is no context, function $\mathcal{F}$ returns the initial timer, as no run-time support is invoked. When the proof term begins with an operation o, say a matching or a non deterministic choice, its cost is accumulated in the timer, say σ', resulting from processing the rest of the proof term ς, at the position determined by the location χ. Argument χ singles out the (sub-)process involved in the action. Technically, the cost of the operation o is assigned in the initial timer at χ, and then added, at the same location, to σ'. Instead, if the proof term begins with a $||_i$, then the location that has to be affected is the left component of the current one. So, in item (3), the relevant tag is added to the current location for future use, and no cost is charged to the location, because we assume that different sequential sub-processes be allocated on different processors. Finally, when the proof term is a pair recording the contexts of two partners of a communication, the costs of both contexts is computed, and the resulting timers are merged via the sum function.

We now give three examples of applications of function $\mathcal{F}$. The three timers computed here will be used in the examples after the definition of the timed operational semantics.

Example 1. The timer resulting from the application of function $\mathcal{F}$ to the proof term $= ||_0{+}_0$ and the empty location ϵ is computed as follows.

$$\begin{aligned}
&\mathcal{F}(= ||_0{+}_0, \epsilon) = \\
&\quad \mathbf{0}_\sigma[\$(=)/\epsilon] + \mathcal{F}(||_0{+}_0, \epsilon) = \\
&\quad \mathbf{0}_\sigma[\$(=)/\epsilon] + \mathcal{F}(+_0, ||_0) = \\
&\quad \mathbf{0}_\sigma[\$(=)/\epsilon] + \mathbf{0}_\sigma[\$(+_0)/||_0] + \mathcal{F}(\epsilon, ||_0) = \\
&\quad \mathbf{0}_\sigma[\$(=)/\epsilon] + \mathbf{0}_\sigma[\$(+_0)/||_0] + \mathbf{0}_\sigma = \\
&\quad \mathbf{0}_\sigma[\$(=)/\epsilon] + \mathbf{0}_\sigma[\$(+_0)/||_0] = \tilde{\sigma}
\end{aligned}$$

For instance $\tilde{\sigma}(||_0||_1) = \$(=) + \$(+_0)$ and $\tilde{\sigma}(||_1) = \$(=)$.

Consider now the proof term $\langle ||_0, ||_1{+}_0 \rangle$ originated by a communication. The application of $\mathcal{F}$ to it and to ϵ yields

$$\begin{aligned}
&\mathcal{F}(\langle ||_0, ||_1{+}_0 \rangle, \epsilon) = \\
&\quad \mathcal{F}(||_0, \epsilon) + \mathcal{F}(||_1{+}_0, \epsilon) = \\
&\quad \mathcal{F}(\epsilon, ||_0) + \mathcal{F}(+_0, ||_1) = \\
&\quad \mathbf{0}_\sigma + \mathbf{0}_\sigma[\$(+_0)/||_1] + \mathcal{F}(\epsilon, ||_1) = \\
&\quad \mathbf{0}_\sigma[\$(+_0)/||_1] + \mathbf{0}_\sigma = \\
&\quad \mathbf{0}_\sigma[\$(+_0)/||_1]
\end{aligned}$$

The third proof term we consider is $+_0\langle ||_0{+}_0, ||_1 = \rangle$ that encodes a communication within the scope of a nondeterministic choice.

$$\begin{aligned}
&\mathcal{F}(+_1\langle ||_0{+}_0, ||_1 = \rangle, \epsilon) = \\
&\quad \mathbf{0}_\sigma[\$(+_1)/\epsilon] + \mathcal{F}(\langle ||_0{+}_0, ||_1 = \rangle = \\
&\quad \mathbf{0}_\sigma[\$(+_1)/\epsilon] + \mathcal{F}(||_0{+}_0, \epsilon) + \mathcal{F}(||_1 =, \epsilon) = \\
&\quad \mathbf{0}_\sigma[\$(+_1)/\epsilon] + \mathcal{F}(+_0, ||_0) + \mathcal{F}(=, ||_1) = \\
&\quad \mathbf{0}_\sigma[\$(+_1)/\epsilon] + \mathbf{0}_\sigma[\$(+_0)/||_0] + \mathcal{F}(\epsilon, ||_0) + \mathbf{0}_\sigma[\$(=)/||_1] + \mathcal{F}(\epsilon, ||_1) = \\
&\quad \mathbf{0}_\sigma[\$(+_1)/\epsilon] + \mathbf{0}_\sigma[\$(+_0)/||_0] + \mathbf{0}_\sigma[\$(=)/||_1]
\end{aligned}$$

We now have all the ingredients to define our timed transition for the π-calculus. As mentioned above, its configurations, ranged over by $S, S_0, S_1, \ldots$, have the form

$$\langle P, \sigma \rangle,$$

where $P \in \mathcal{P}$ is a process and $\sigma \in \Sigma$ is a timer. A timed transition is written as

$$\langle P, \sigma \rangle \xmapsto{\beta}_{\$} \langle P', \sigma' \rangle,$$

where the label β is either τ for invisible actions or $\langle \alpha, t \rangle$ for the visible ones ($\alpha \in \mathcal{A}$ and $t \in I\!N$). The index \$ makes explicit the dependency of the components of labels expressing time on the cost function chosen. Hereafter, we feel free to omit the index \$ when no ambiguity can arise. The timed transition relation of the π-calculus is defined by the rules in Tab. 3. Their premises use proved transitions, defined in Tab. 2.

$$Ext: \frac{P \xrightarrow{\vartheta\alpha} P'}{\langle P, \sigma \rangle \xmapsto{\alpha, \sigma'(\underline{\vartheta})}_{\$} \langle P', \sigma' \rangle}, \ \sigma' = \sigma + \mathcal{F}(\vartheta, \epsilon) + \mathbf{0}_{\sigma}[\$(\alpha)/\underline{\vartheta}]$$

$$Sil: \frac{P \xrightarrow{\vartheta\tau_i} P'}{\langle P, \sigma \rangle \xmapsto{\tau}_{\$} \langle P', \sigma' \rangle}, \ \sigma' = \sigma + \mathcal{F}(\vartheta, \epsilon) + \mathbf{0}_{\sigma}[\$(\tau_i)/\underline{\vartheta}]$$

$$Int: \frac{P \xrightarrow{\vartheta\langle\vartheta_0\alpha_0, \vartheta_1\alpha_1\rangle} P'}{\langle P, \sigma \rangle \xmapsto{\tau}_{\$} \langle P', \sigma' \rangle}, \ \begin{cases} \sigma'' \ = \ \sigma + \mathcal{F}(\vartheta\langle\vartheta_0, \vartheta_1\rangle, \varepsilon) \\ t = max_i \ \sigma''(\underline{\vartheta\vartheta_i}) + \$(sbj(\alpha_0), obj(\alpha_0), \underline{\vartheta\vartheta_0}, \underline{\vartheta\vartheta_1}) \\ \sigma' = \sigma''[t/\underline{\vartheta\vartheta_0}, \underline{\vartheta\vartheta_1}] \end{cases}$$

Table 3. Early timed transition system for the π-calculus.

Now we briefly comment on the three inference rules of Tab. 3. Rule Ext is for visible actions. So, we compute two timers that express the two components of the actual cost of α, performed by the sub-process at the location $\underline{\vartheta}$. The timer $\mathbf{0}_{\sigma}[\$(\alpha)/\underline{\vartheta}]$ stores the cost needed to perform action α in isolation. The other is timer $\mathcal{F}(\vartheta, \epsilon)$ and reflects the time spent in solving the constraints imposed by the context in which α occurs. Note that the correct values of time have been assigned to all the sub-processes actively involved in traversing the context, as exemplified after the definition of function $\mathcal{F}$. The two timers are then added to that of the source of the transition, which records the local clocks at the beginning of the execution of α. Rule Sil for internal actions is quite similar. The only difference here concerns the label, which is simply τ and carries no information on the elapsed time. However, the time spent in silent actions, both internal and communications, is recorded in the new timer σ' of the target state of the transition. The last rule is a little more complex. For readability, we introduce the intermediate timer σ''. It results from the sum of the timer at the beginning of the communication and of the timer collecting the costs of the context. Actually, the last timer is the sum of two timers, one for each the components at location $\underline{\vartheta\vartheta_i}$. Then, the selection of the time $max_i\ \sigma''(\underline{\vartheta\vartheta_i})$ of the slower process mimics that the faster process is delayed till the slower one is ready to synchronize. Finally, the actual timer σ' is obtained by adding to the above also the cost of the communication. Recall that we stipulated it dependent on the channel ($sbj(\alpha_0)$ which is equal $sbj(\alpha_1)$), the data exchanged ($obj(\alpha_0) = obj(\alpha_1)$) and the distance between the locations ($\underline{\vartheta\vartheta_i}$) where the partners reside.

The following example shows two timed computations. As usual, a computation is just a sequence of transitions, such that the source and the target of two consecutive transitions match.

Example 2. Consider the process

$$R = [x = x]((((\nu y)\overline{x}y.\overline{y}z.P + P') \mid (y(w).Q + Q'))$$

It has the following proved computation

$$R \stackrel{=||_0+_0\overline{x}(y)}{\longrightarrow} \overline{y}z.P|(y(w).Q + Q') \stackrel{\langle ||_0\overline{y}z, ||_1+_0yz\rangle}{\longrightarrow} P|Q\{z/w\}$$

The derived timed computation is

$$\langle R, \mathbf{0}_\sigma\rangle \stackrel{\overline{x}(y),\sigma'(||_0)}{\longmapsto} \langle \overline{y}z.P|(y(w).Q + Q'), \sigma'\rangle \stackrel{\tau}{\longmapsto} \langle P|Q\{z/w\}, \sigma''\rangle$$

Using the value of $\mathcal{F}(= ||_0+_0, \epsilon)$ computed in the example above, the timer

$$\sigma' = \mathbf{0}_\sigma + \mathcal{F}(= ||_0+_0, \epsilon) + \mathbf{0}_\sigma[\$(\overline{x}(y))/||_0]$$

becomes

$$\begin{aligned}&\mathbf{0}_\sigma[\$(=)/\epsilon] + \mathbf{0}_\sigma[\$(+_0)/||_0] + \mathbf{0}_\sigma[\$(\overline{x}(y))/||_0] \ = \\ &\ \mathbf{0}_\sigma[\$(=)/\epsilon] + \mathbf{0}_\sigma[\$(+_0) + \$(\overline{x}(y))/||_0]\end{aligned}$$

Similarly,

$$\sigma'' = \sigma'''[max\ \{\sigma'''(||_0), \sigma'''(||_1)\} + \$(y, z, ||_0, ||_1)/||_0, ||_1],$$

where

$$\sigma''' = \sigma' + \mathcal{F}(\langle ||_0, ||_1+_0\rangle, \epsilon) = \mathbf{0}_\sigma[\$(=)/\epsilon] + \mathbf{0}_\sigma[\$(+_0) + \$(\overline{x}(y))/||_0] + \mathbf{0}_\sigma[\$(+_0)/||_1].$$

Since $\sigma'''(||_0) = \sigma'''(||_1) + \$(\overline{x}(y))$, $max\{\sigma'''(||_0), \sigma'''(||_1)\} = \sigma'''(||_0)$. Therefore

$$\sigma'' = \mathbf{0}_\sigma[\$(=)/\epsilon] + \mathbf{0}_\sigma[\$(+_0) + \$(\overline{x}(y)) + \$(y, z, ||_0, ||_1)/||_0, ||_1].$$

We now give another example of computation. It consists of a single communication, one partner of which is within the scope of a nondeterministic choice. Consider the proved computation

$$R + ((\overline{x}y.P + P')|[x = x]x(w).Q) \stackrel{+_1\langle ||_0 +_0, ||_1 = \rangle}{\longrightarrow} P|Q\{y/w\}$$

The derived timed computation is

$$\langle R + ((\overline{x}y.P + P')|[x = x]x(w).Q), \mathbf{0}_\sigma\rangle \stackrel{\tau}{\longmapsto} \langle P|Q\{y/w\}, \sigma'\rangle$$

using the value $\mathcal{F}(+_1\langle ||_0 +_0, ||_1 = \rangle, \epsilon)$ computed above, the auxiliary timer is

$$\sigma'' = \mathbf{0}_\sigma[\$(+_1)/\epsilon] + \mathbf{0}_\sigma[\$(+_0)/||_0] + \mathbf{0}_\sigma[\$(=)/||_1]$$

from which

$$\sigma' = \sigma''[t/||_0, ||_1],$$

where $t = \$(+_1) + max\{\$(=), \$(+_0)\} + \$(x, y, ||_0, ||_1)$.
Note that the communication has synchronized the local clocks of both P and $Q\{y/w\}$ to time t.

Above, we anticipated that a timer can be seen as the collection of the local clocks associated with the *active* sub-processes only, although it is a total function on locations. Also, we claimed that no reference is made to local clocks of sub-processes no longer there or not yet present in the whole system. A simple inspection to the rules defining the timed operational semantics justifies the above statements. In fact, in a transition $\langle P, \sigma\rangle \stackrel{\alpha,t}{\longmapsto} \langle P', \sigma'\rangle$, the time t is the value of timer σ' at the location where action α occurred (cfr. rule Ext). Similarly when a silent transition is derived via rule Sil. In rule Int, the auxiliary timer σ'' is accessed twice, at the locations where the input and the output action are performed.

We end this section with a proposition that establishes the correspondence between interleaving and timed transitions, and shows that the latter conservatively extend the former. Its proof relies on the correspondence between the above relations with the proved one. First we need a function that maps timed labels to actions.

Definition 8. The function $\ell_T(\cdot) : (\mathcal{A} \times I\!N) \cup \{\tau\} \to \mathcal{A} \cup \{\tau\}$ is defined as

$$\ell_T(\langle\alpha, t\rangle) = \alpha \quad \text{and} \quad \ell_T(\tau) = \tau$$

Proposition 9. *Given a process P and a timer σ, $P \stackrel{\mu}{\longrightarrow} P'$ is an early transition if and only if there exists β with $\ell_T(\beta) = \mu$ and $\langle P, \sigma\rangle \stackrel{\beta}{\longmapsto} \langle P', \sigma'\rangle$, is a timed transition.*

5 Performance preorders

In this section we introduce two preorderings between processes that express their relative efficiency and that require their behaviour to be the same from a functional point of view. More precisely, if P and P' are in a preordering relation, we say that P is faster than P'. This will provide us with the basis for choosing the specification more suited to the architecture at hand, or the architecture better suited to a given specification.

Before introducing our first preordering, we define a weak timed transition relation and the notion of viable and stuck configurations. The new relation is used to abstract from the internal actions of a system and it is quite standard. Viable (stuck) configurations may (may not) originate visible transitions.

Definition 10. Given a cost function $\$$ and the transition relation $\longmapsto_{\$}$ in Tab. 3, then the timed weak transition relation $\overset{\beta}{\Longmapsto}_{\$}$ is

$$\overset{\tau}{\Longmapsto}_{\$} = (\overset{\tau}{\longmapsto}_{\$})^* \quad \text{and} \quad \overset{\beta}{\Longmapsto}_{\$} = \overset{\tau}{\Longmapsto}_{\$} \circ \overset{\beta}{\longmapsto}_{\$} \circ \overset{\tau}{\Longmapsto}_{\$}$$

A process P is *viable* if there exists a process P' such that $P \overset{\alpha}{\Rightarrow} P'$. Otherwise, it is said *stuck*. A configuration $\langle P, \sigma \rangle$ is viable (stuck) if its process part is.

We define below a simulation, called *efficiency simulation*, saying that a process S_0 simulates S_1 and is faster than it. Similarly, we define an *inefficiency simulation* by imposing that S_0 simulates S_1 and is slower than it. Finally, we define the *performance preordering* that relates S_0 and S_1 if S_0 simulates efficiently S_1 and S_1 simulates inefficiently S_0. This relation turns out to be a refinement of the classical interleaving bisimulation equivalence.

Definition 11. Let $\$_0$ and $\$_1$ be two cost functions. Then, a binary relation $\mathcal{R}$ on configurations is an *efficiency simulation* if whenever $(S_0, S_1) \in \mathcal{R}$ implies either

- S_0, S_1 are stuck, or
- if $S_1 \overset{\beta_1}{\longmapsto}_{\$_1} S_1'$, with $bn(\beta_1) \cap fn(S_0, S_1) = \emptyset$, then for some S_0', $S_0 \overset{\beta_0}{\Longmapsto}_{\$_0} S_0'$, and $\beta_0 \leq \beta_1$ and $(S_0', S_1') \in \mathcal{R}$.

where $\beta_0 \leq \beta_1$ iff $\beta_0 = \beta_1 = \tau$ or $\beta_0 = \langle \alpha, t_0 \rangle$ and $\beta_1 = \langle \alpha, t_1 \rangle$ with $t_0 \leq t_1$.

Inefficiency simulation is obtained by replacing $\beta_0 \leq \beta_1$ above with $\beta_1 \leq \beta_0$.

The relation $\mathcal{R}$ is a *performance preordering* if $\mathcal{R}$ is an efficiency simulation and $\mathcal{R}^{-1}$ is an inefficiency simulation.

Finally, we say that a process P is more efficient than a process Q (written $P \preceq Q$) if there exists a performance preordering $\mathcal{R}$ such that $(\langle P, \mathbf{0}_\sigma \rangle, \langle Q, \mathbf{0}_\sigma \rangle) \in \mathcal{R}$.

When the indexing cost functions $\$_0$ and $\$_1$ are equal, preordering $\preceq$ coincides with the analogous relation defined on CCS usually called performance preordering in the literature [7]. It relates two processes running on the same target machine and on the same network if and only if one is faster than the other for any action that it can perform. In other words, the faster process must simulate more efficiently the other step by step from the very beginning.

When instead the cost functions $\$_0$ and $\$_1$ are different, the advantages of the parametricity of our timed transition system show up. For example, we can compare the same algorithm on different target machines. More precisely, let $S_0 = S_1$ in Def. 11 and define the cost functions $\$_0 \neq \$_1$ in order to express the costs of execution instructions on two different architectures.

Performance preordering gives a rough estimation of the efficiency of processes. Indeed, it may consider incomparable processes which should sensibly be comparable. Consider for instance a process P that is slower than another, say Q, in its initial steps (e.g. because it initializes complex data structures) and then performs better than Q (e.g. because its sophisticated data structures help a lot). In this case, we should say that the process P is more efficient than Q, while according to the performance preordering the two processes are simply unrelated.

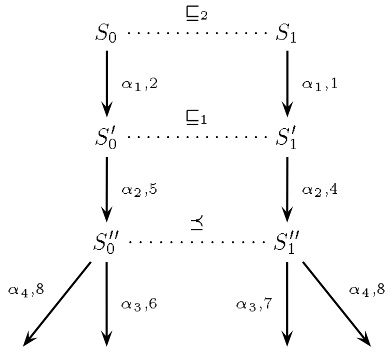

Fig. 1. Comparing performance and amortized preordering.

As an example consider the processes S_0 and S_1 depicted in Fig. 1, and assume that $S_0'' \preceq S_1''$. After an initial delay on actions α_1 and α_2, S_0 is always faster than S_1. To overcome the weakness of performance preordering and to gain the relation between the processes above we define an *amortized preordering*, in the style of amortized complexity. Essentially, it relates two processes if one is faster than the other from a given point on of the simulation. This new preordering is defined through a family of indexed relations of the form $\sqsubseteq_i$. The intuition of $P \sqsubseteq_i Q$ is that the process P will always be faster than Q after i

steps.

Definition 12. Given two cost functions $\$_0$ and $\$_1$, the family of binary relations $\sqsubseteq_i$ over configurations is defined as follows

- $\sqsubseteq_0 = \preceq$
- $\sqsubseteq_{i+1}$ is the smallest relation including $\sqsubseteq_i$ such that
  - if $S_1 \stackrel{\beta_1}{\longmapsto}_{\$_1} S_1'$, with $bn(\beta_1) \cap fn(S_0, S_1) = \emptyset$, then for some S_0', $S_0 \stackrel{\beta_0}{\Longmapsto}_{\$_0} S_0'$, $\ell_T(\beta_0) = \ell_T(\beta_1)$ and $S_0' \sqsubseteq_i S_1'$;
  - if $S_0 \stackrel{\beta_0}{\longmapsto}_{\$_0} S_0'$, with $bn(\beta_0) \cap fn(S_0, S_1) = \emptyset$, then for some S_1', $S_1 \stackrel{\beta_1}{\Longmapsto}_{\$_1} S_1'$, $\ell_T(\beta_0) = \ell_T(\beta_1)$ and $S_0' \sqsubseteq_i S_1'$.

The *amortized preordering* $\sqsubseteq$ between configurations is defined as

$$S_0 \sqsubseteq S_1 \text{ if and only if } \exists i \,:\, S_0 \sqsubseteq_i S_1$$

We say that two processes P and Q are in the amortized preordering (written $P \sqsubseteq Q$) iff $\langle P, \mathbf{0}_\sigma \rangle \sqsubseteq \langle Q, \mathbf{0}_\sigma \rangle$, and we let $P \sqsubseteq_i Q$ iff $\langle P, \mathbf{0}_\sigma \rangle \sqsubseteq_i \langle Q, \mathbf{0}_\sigma \rangle$.

Consider again Fig. 1. We can establish the amortized preordering between S_0 and S_1, because $S_0 \sqsubseteq_2 S_1$ due to the assumption $S_0'' \preceq S_1''$.

We end this section by establishing the relations of our amortized preordering with the performance preordering and with the interleaving weak equivalence. We first report a useful lemma. Also, we need the following auxiliary definition.

Definition 13. Let $\mathcal{T}_c$ and $\mathcal{T}_p$ be the sets of binary relations over configurations and over processes, respectively. Then, function $\phi : \mathcal{T}_c \to \mathcal{T}_p$ is defined as

$$\begin{aligned}\phi(\mathcal{R}) = \{&(P, Q) \in \mathcal{P} \times \mathcal{P} \mid (\langle P, \sigma \rangle, \langle Q, \sigma' \rangle) \in \mathcal{R}\} \cup \\ &\{(P, Q) : P, Q \in \mathcal{P} \text{ and they are stuck}\}\end{aligned}$$

Lemma 14.

- *$\{(P, Q) : P, Q \in \mathcal{P}$ and they are stuck$\}$ is a weak bisimulation.*
- *if $\mathcal{R}$ is*
  - *an efficiency or an inefficiency simulation then $\phi(\mathcal{R})$ is a weak simulation*
  - *a performance preordering then $\phi(\mathcal{R})$ is a weak bisimulation.*
- *$\phi(\sqsubseteq_i)$ is a weak bisimulation.*

Proof. The proof of the first two items is immediate by definition. The last item follows by definition and from the preceeding items.

It is now easy to prove that our semantic model conservatively extends the original semantics of the π-calculus.

Theorem 15.

- *$P \preceq Q$ implies but is not implied by $P \approx Q$,*
- *$P \sqsubseteq_i Q$ implies but is not implied by $P \approx Q$.*

Proof. We first prove the inclusions. Let $\mathcal{R}$ be either $\preceq$ or $\sqsubseteq_i$ and $\mathcal{S}$ be either of their obvious extension to configurations. Given $(P, Q) \in \mathcal{R}$, by definition $(\langle P, \mathbf{0}_\sigma, Q, \mathbf{0}_\sigma \rangle) \in \mathcal{S}$. By Lemma 14 there exists a weak bisimulation $\phi(\mathcal{S})$ that contains (P, Q). Strictness of implication relies on lack of symmetry in preorders.

The following corollary is immediate by noting that $\sqsubseteq_0 = \preceq$ and that $\sqsubseteq_i \subset \sqsubseteq$.

Corollary 16.

- *$P \preceq Q$ implies but is not implied by $P \sqsubseteq Q$,*
- *$P \sqsubseteq Q$ implies but is not implied by $P \approx Q$.*

6 A case study

In this section we use our mathematical framework to compare the efficiency of two architectures that run sequential programs. We consider a conventional uniprocessor and a prefetch pipeline machine. For the sake of simplicity, in computing the time of the fetch-execute cycle we assume that all instructions take the same time.

We start with the specification of the uniprocessor architecture. It consists of two modules: a memory M and a processor P. The memory fetches an instruction i and sends it to the processor for execution along the channel $fetch$; then, it resumes. The processor P receives the instruction, executes it and then restarts. Their specifications are

$$M = \overline{fetch}\, i.M \quad \text{and} \quad P = fetch(x).x.P$$

The overall system hides the channel *fetch* and is

$$SYS_c = (\nu\, fetch)(M \mid P).$$

The prefetch pipeline architecture has two modules in place of P above: an instruction unit IU and a execution unit EU. The unit IU interacts with M to fetch instructions. The unit EU executes them. We specify the three units composing the system as follows.

$$M = \overline{fetch}\, i.M, \quad IU = fetch(x).\overline{exec}\, x.IU \quad \text{and} \quad EU = exec(s).s.EU.$$

The behaviour of the prefetch pipeline machine is described by

$$SYS_p = (\nu\, fetch, exec)(M \mid (IU \mid EU)).$$

Figure 2 shows the proved transition systems of the two architectures. Note that both are finite state; in passing, we remark that the proved transition system for a process has exactly the same states as the standard interleaving transition system. As we will see later, this property allows us to derive closed forms for the costs of executing the k^{th} instruction of a program.

Assume that the cost functions of the conventional and the prefetch pipeline processors are $\$_c$ and $\$_p$, respectively. Furthermore, we assume that there is no

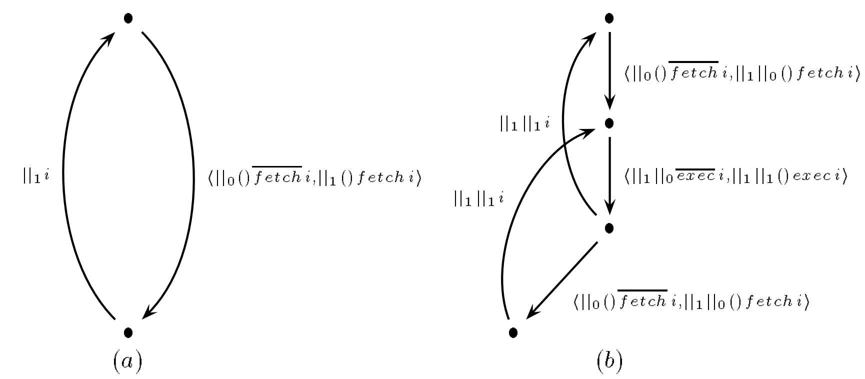

Fig. 2. Proved transition system of the conventional (a) and the prefetch pipeline uniprocessor (b).

appreciable difference between the time required by the two architectures for fetching, as well as for executing instructions. This amounts to say that

$$\$_c(fetch, i, ||_0, ||_1) = \$_p(fetch, i, ||_0, ||_1||_0) = \$(fetch) \text{ and } \$_c(i) = \$_p(i) = \$(i),$$

where the arguments $||_0$ and $||_1$ of $\$_c$ are the locations of M and P, and the arguments $||_0$ and $||_1||_0$ of $\$_p$ are the locations of M and of IU.

In any prefetch pipeline processor the cost of sending the (decoded) instruction from IU to EU is always smaller than fetching (and decoding) it. Hence

$$\$_p(exec, i, ||_1||_0, ||_1||_1) \leq \$_p(fetch),$$

where $||_1||_1$ is the location of EU.

Suppose that the machines we are comparing have to interpret a set of CISC instructions. Generally, both of them take more time in the execution of an instruction than in its fetch, so we let

$$\$(fetch) \leq \$(i).$$

Usually, the architectures we are considering have a firmware run-time support, and resolve calls to routines, in our case invocations of identifiers, through a jump instruction. Then, we neglect in what follows the cost of invoking identifiers M, P, IU and EU and let

$$\$_c(()) = \$_p(()) = 0.$$

In fact, assigning a cost different than 0 to invocations of identifiers only makes the cost function a bit more complex, but does not at all change our results.

The above assumptions reflect the features of real conventional and prefetch pipeline architectures. Under them, the cost of the execution of a single instruction i in the conventional architecture is

$$t_c = \$(fetch) + \$(i).$$

When $k-1$ instructions are executed, the timer associated with P has value $(k-1) \times t_c$. Hence, we can express the label of the transition representing the execution of the k^{th} instruction as $\langle i, T_c(k) \rangle$, where

$$T_c(k) = k \times t_c.$$

Consider now the prefetch pipeline architecture. The cost of executing the first instruction is $\$(fetch) + t_p$, where

$$t_p = \$_p(exec, i, ||_1||_0, ||_1||_1) + \$(i).$$

The time for completing the next instruction, call it i' for a while, only involves that of the communication between IU and EU and that of the actual execution of i'. This is because the fetch of the current instruction i' can overlap in time with the execution of the preceding one. Figure 2(b) shows clearly that the transitions corresponding to a fetch of an instruction can occur independently of the execution of the previous one. In fact, the action of sending, of receiving and of executing an instruction are performed in different locations, namely in $||_0, ||_1||_0$ and $||_1||_1$, respectively. Therefore, we can express the label of the transition representing the execution of the k^{th} instruction as $\langle i, T_p(k) \rangle$, where

$$T_p(k) = \$(fetch) + k \times t_p.$$

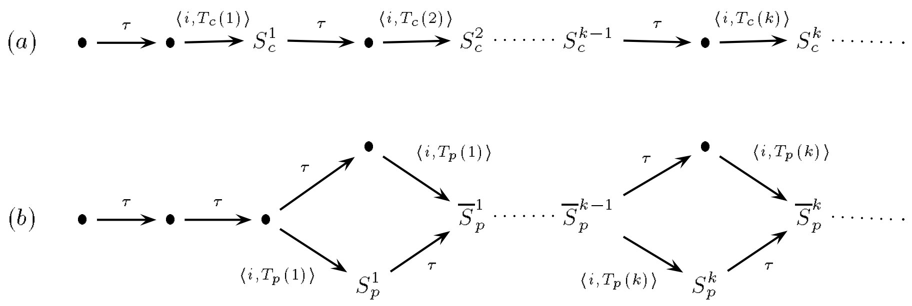

Fig. 3. Timed behaviour of the conventional (a) and the prefetch pipeline uniprocessor (b).

Now, it is easy to draw the timed transition systems for SYS_c and SYS_p, relying on functions T_c and T_p. They are depicted in Fig. 3, and have a denumerable set of states. Nevertheless, we can compare the efficiency of the two architectures. Indeed, for determining the preordering relations between SYS_c and SYS_p, we first establish a weak bisimulation $\mathcal{B}$ that relates the nodes of the two timed transition systems. Of course, $\mathcal{B}$ should only consider the actions in labels, i.e. $\ell_T(\beta)$. Determining $\mathcal{B}$ is easy, because all the nodes of one timed transition system are weakly bisimilar to those of the other one, as one can check by inspecting Fig. 2, modified by replacing the enhanced labels θ with standard actions $\ell(\theta)$. In particular, we consider below the pairs of bisimilar states (S_c^k, S_p^k) and $(S_c^k, \overline{S}_p^k)$, as we called them in Fig. 3 (index k records that k instructions have been executed).

Now, to establish a preordering relation, we only need to compare the temporal part of the labels of the transitions with target S_c^k and S_p^k, $\overline{S}_p^k$. In other words, it suffices to compare the values $T_c(k)$ and $T_p(k)$. Since

$$\langle SYS_c, \mathbf{0}_\sigma \rangle \overset{i,T_c(1)}{\Longmapsto} S_c^1 \quad \text{and} \quad \langle SYS_p, \mathbf{0}_\sigma \rangle \overset{i,T_p(1)}{\Longmapsto} S_p^1,\ \langle SYS_p, \mathbf{0}_\sigma \rangle \overset{i,T_p(1)}{\Longmapsto} \overline{S}_p^1$$

and $T_c(1) < T_p(1)$, we derive that $SYS_p \not\preceq SYS_c$. Also, $SYS_c \not\preceq SYS_p$, because after some initial steps the prefetch pipeline architecture is faster than the conventional one. In fact, there exists $\tilde{k}$ such that, for all $k \geq \tilde{k}$, the transitions

$$S_c^{k-1} \overset{i,T_c(k)}{\Longmapsto} S_c^k \quad \text{and} \quad \overline{S}_p^{k-1} \overset{i,T_p(k)}{\Longmapsto} S_p^k,\ \overline{S}_p^{k-1} \overset{i,T_p(k)}{\Longmapsto} \overline{S}_p^k$$

satisfy the inequality $T_p(k) < T_c(k)$. Hence, $SYS_c \not\preceq SYS_p$. Actually, the number $\tilde{k} \in I\!N$ is the smallest k such that

$$k \geq 1/(1 - (\$_p(exec, i, ||_1||_0, ||_1||_1)/\$(fetch))).$$

Also, for all $k \geq \tilde{k}$, the inequality $T_p(k) < T_c(k)$ implies that $S_p^k \preceq S_c^k$ and $\overline{S}_p^k \preceq S_c^k$. Therefore $SYS_p \sqsubseteq_k SYS_c$, from which $SYS_p \sqsubseteq SYS_c$.

To give an idea of the value of $\tilde{k}$, we consider the following two cases. If the internal communication between IU and EU takes 50% of the time needed for a fetch, the value of $\tilde{k}$ is 2, hence $SYS_p \sqsubseteq_2 SYS_c$. When the percentage above is 90%, $\tilde{k} = 10$ and hence $SYS_p \sqsubseteq_{10} SYS_c$. This shows that the prefetch pipeline architecture is faster than a conventional uniprocessor if the interpreted assembler program has more than 1 instruction in the first case, and more than 9 in the second one. The amortized preordering gives formal evidence that a prefetch pipeline architecture is (almost) always more efficient than the conventional uniprocessor one.

The above argument can be applied when the considered set of instructions to be interpreted by our architectures is of type RISC, rather than CISC as done above. In this case the fetch is usually slower then the execution of an instruction, so we let

$$\$(i) \leq \$(fetch).$$

Under this assumption, to get the new cost function, it is sufficient to exchange the role of $\$(i)$ and $\$(fetch)$ in the definitions above. More precisely, we define the cost of executing a single instruction as

$$t_p^R = \$_p(exec, i, ||_1||_0, ||_1||_1) + \$(fetch)$$

and that of the transition representing the execution of the k^{th} instruction as

$$T_p^R(k) = \$(i) + k \times t_p^R.$$

Also in the RISC case, simple calculations determine a value $\tilde{k}$, from which onwards $SYS_p \sqsubseteq_k SYS_c$. Summing up, we have formally shown that the pipeline-based architecture is better.

7 Conclusions and Future Work

The basis of our work is an enhanced structural operational semantics that uniformly describes qualitative as well as quantitative aspects of mobile and distributed systems. More precisely, we use proved transition systems, whose transitions are labelled by (encodings of) their derivations. A simple relabelling then suffices for recovering the many different aspects of systems.

Taking advantage of this, here we defined a timed semantics for π-calculus that associates a local clock, with discrete values, with any sequential component of a process. Besides considering mobile processes rather than processes in calculi without value passing as done in the literature, we refined previous work on timed semantics on three aspects. First, we established a cost for any operator of the language to reflect that the run-time support executes some low-level routines to implement an operator of the language. Also, we determine the cost of communications according to the placement of processes on the underlying interconnection network, to the links used and to the size of the messages transferred. Second, we parametrized our transition relation with a cost function. In this way, one can instantiate the same semantic definition to take into account different target architectures. Finally, we extended the machinery introduced in the literature to compare the performance of processes. As usual, we required the two processes to be (interleaving) bisimilar as far as only actions are concerned. Then, we refined the known notion of efficiency pre-orderings, stipulating that a process is faster than another if it is such from a given point of its execution onwards.

The mathematical framework introduced can be used to estimate and compare the performance of two different processes when they run on a given architecture. This could provide potential benefits in the construction of optimizing compilers for concurrent and distributed systems. Due to parametricity, we can also select the architecture more suited to a specific system of mobile processes, by comparing its performance when it is executed on different architectures. In fact, the costs of the run-time support routines are machine sensible, as well as those of communications depend on the network topology and on its technology.

Our proposal is interleaving in style. Nevertheless, our timed transition systems describes, to some extent, non interleaving features. In fact, it may happen that in a computation the occurrence of two transitions overlap in time. A simple example follows. Suppose we have the process $\alpha.\beta.P \mid \gamma.Q$ and that the cost function is $\$(\alpha) = 3, \$(\beta) = 2, \$(\gamma) = 4$. Then, a possible computation, starting with the initial timer identically equal to 0, is

$$\langle \alpha.\beta.P \mid \gamma.Q, \mathbf{0}_\sigma \rangle \xrightarrow{\alpha,3} \langle \beta.P \mid \gamma.Q, \sigma \rangle \xrightarrow{\beta,5} \langle P \mid \gamma.Q, \sigma' \rangle \xrightarrow{\gamma,4} \langle P \mid Q, \sigma'' \rangle,$$

where $\sigma(\beta.P) = 3$, $\sigma(\gamma.Q) = \sigma'(\gamma.Q) = 0$, $\sigma'(P) = \sigma''(P) = 5$, $\sigma''(Q) = 4$ (for clarity, we are using processes in place of locations, so we write $\gamma.Q$ in place of $||_1$). In the computation above, the execution of γ starts at time 0 and ends at time 4, and during this interval α has been completed and β started. Therefore, timed computations convey some information about causality. In our example, clearly α and γ are not causally related. It would be interesting to place the timed semantics in the spectrum interleaving versus non interleaving descriptions. We already established that our semantics conservatively extends the original, interleaving semantics of π-calculus [20], but its exact relations with causal or local semantics (see, e.g. [10]) are still unknown.

Another research direction is about real time (see, e.g., [2, 15]). The extension is easy as fas as transitions are concerned. It suffices to consider cost functions that take value over the positive reals. Much work has to be done instead for studying the performance of concurrent systems in this case.

Here, we compared the idealized specifications of a traditional and of a pipeline architectures (both CISC and RISC), and show that the latter behaves more efficiently almost always. Admittedly, the size of our example is small. We feel that only an extensive testing of our proposal on real size examples can assess its validity. We plan to take examples from both the side of parallel architectures and of networks of asynchronous processors. We are confident that our techniques will scale up without major problems. In fact, the existing programming language Facile [5] has been equipped with a proved operational semantics. From this, we derived information about qualitative aspects, used then sucessfully to debug Facile programs [5]. Thus, we hope the same with quantitative descriptions. A minor point concerns the labels of the proved transitions used to derive the transitions of interest, qualitative or quantitative.. They may be long wired and difficult to deal with, but we handle them mechanically. We already experimented on this [4] in derived models that express qualitative aspects of processes, like causality and locality. Another minor problem arises from our assumption to have an unbounded number of processors available. In fact, we assign a new processor/location to a sequential sub-process as soon as needed. E.g., from one processor running process $\alpha.(P|Q)$, we pass to two processors after α has been executed. More realistically, we can allocate more than one sequential sub-process on the same processor. Technically, this is easily achieved by imposing an equivalence relation over locations.

Together with other papers on the qualitative descriptions of concurrent systems mentioned above, this work encourages us to develop further the idea of having a single detailed model from which many others can be coherently derived and fruitfully used for the analysis of systems.

References

1. L. Aceto and D. Murphy. Timing and causality in process algebra. Technical Report 9/93, University of Sussex, 1993.
2. R. Alur and D. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–236, 1994.
3. J.C.M Baeten and J.A. Bergstra. Real-time process algebra. Technical Report CS-R9053, Centre for Mathematics and Computer Science, Amsterdam, 1990.
4. A. Bianchi, S. Coluccini, P. Degano, and C. Priami. An efficient verifier of truly concurrent properties. In V. Malyshkin, editor, *Proceedings of PaCT'95, LNCS 964*, pages 36–50. Springer-Verlag, 1995.
5. Roberta Borgia, Pierpaolo Degano, Corrado Priami, Lone Leth, and Bent Thomsen. Understanding mobile agents via a non-interleaving semantics for facile. Technical Report ECRC-96-4, European Computer-Industry Research Centre, 1996.
6. G. Boudol and I. Castellani. A non-interleaving semantics for CCS based on proved transitions. *Fundamenta Informaticae*, XI(4):433–452, 1988.
7. F. Corradini, R. Gorrieri, and M. Roccetti. Performance preordering: ordering processes with respect to speed. In *Proceedings of MFCS95, LNCS 969*, 1995.
8. P. Degano, R. De Nicola, and U. Montanari. Partial ordering derivations for CCS. In *Proceedings of FCT, LNCS 199*, pages 520–533. Springer-Verlag, 1985.
9. P. Degano and C. Priami. Proved trees. In *Proceedings of ICALP'92, LNCS 623*, pages 629–640. Springer-Verlag, 1992.
10. P. Degano and C. Priami. Causality for mobile processes. In *Proceedings of ICALP'95, LNCS 944*, pages 660–671. Springer-Verlag, 1995.
11. P. Degano and C. Priami. Enhanced operational semantics. *ACM Computing Surveys*, 28(2):352–354, 1996.
12. G. Ferrari and U. Montanari. Dynamic matrices. In *Proceedings of AMAST'95, LNCS 936*. Springer-Verlag, 1995.
13. R. Gorrieri and M. Roccetti. Towards performance evaluation in process algebras. In *Proceedings of AMAST'93, LNCS*. Springer Verlag, 1993.
14. R. Gorrieri, M. Roccetti, and E. Stancapiano. A theory of processes with durational actions. *Theoretical Computer Science*, (140), 1994.
15. E. Goubault. Durations for truly concurrent transitions. In *Proceedings of ESOP96, LNCS 1058*, pages 173–187, 1995.
16. M. Hennessy and T. Regan. A temporal process algebra. Technical Report 2/90, University of Sussex, 1990.
17. Andrea Maggiolo-Schettini and Adriano Peron. A graph rewriting framework for statecharts semantics. In J. Cuny, H. Ehrig, G. Engels, and G. Rozenberg, editors, *Proc. Fifth Intl. Workshop on Graph Grammars and Their Application to Comp. Sci.*, volume 1073 of *Lecture Notes in Computer Science*, pages 107–121. Springer, 1996.
18. Andrea Maggiolo-Schettini and Józef Winkowski. Towards an algebra for timed behaviours. *Theeoretical Computer Science*, 103(2):335–363, 1992.
19. R. Milner. The polyadic π-calculus: a tutorial. Technical Report ECS-LFCS-91-180, University of Edinburgh, 1991.

20. R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes (I and II). *Information and Computation*, 100(1):1–77, 1992.
21. F. Moller and C. Tofts. A temporal calculus of communicating processes. In *Proceedings of CONCUR'90, LNCS 458*. Springer-Verlag, 1990.
22. X. Nicollin and J. Sifakis. An overview and synthesis on timed process algebras. In *Real Time: Theory in Practice, LNCS 600*, pages 526–548. Springer-Verlag, 1991.
23. G.M. Reed and A.W. Roscoe. A timed model for communicating sequential processes. *Theoretical Computer Science*, pages 249–261, 1988.
24. Y. Wang. Real-time behaviour of asynchronous agents. In J.C.M. Beaten and J.W. Klop, editors, *Proceedings of CONCUR'90, LNCS 458*. Springer Verlag, 1990.

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