Names of the π-Calculus Agents Handled Locally *

Chiara Bodei, Pierpaolo Degano

Dipartimento di Informatica, Università di Pisa

Corso Italia, 40, I-56125 Pisa, Italy

{chiara,degano}@di.unipi.it

and

Corrado Priami

Istituto Policattedra, Università di Verona

Ca' Vignal 2, Strada Le Grazie 1, I-37134 Verona, Italy

priami@sci.univr.it

Abstract

We address the problem of handling names in concurrent and distributed systems made up of mobile processes. We equip processes with local environments. Our structural operational semantics handles these environments so that captures of names are never possible. Our semantics includes the specification of a distributed name manager that conservatively extends standard operational semantics. Bisimulation-based equivalences can be checked on our transition systems. They yield the same equivalence relations as those based on standard interleaving semantics. Finally, we show that our development scales up smoothly to higher-order calculi.

1 Introduction

The specification of systems of mobile processes is gaining more and more attention, see e.g. [20, 22, 25, 29, 7, 13]. Efficiency considerations suggest implementations which provide each sequential process in the system with its own *local* environment. In the π-calculus [20, 18], this amounts to saying that each process has its own space

*Work partially supported by ESPRIT BRA n.8130 - LOMAPS. The first two authors have been also partially supported by the CNR Progetto Strategico *Modelli e Metodi per la Matematica e l'Ingegneria* and by the MURST Progetto *Tecniche Formali per la Specifica, l'Analisi, la Verifica, la Sintesi e la Trasformazione di Sistemi Software.*

Portions of Sections 1, 3, 4 and 5 appeared in *Proceedings of* 23^{rd} *ICALP*, Paderborn, Germany, July 1996, LNCS 1099, (F. Meyer auf der Heide, B. Monien Eds.), Springer-Verlag, 490 - 501.

of private names. Some of these names may be communicated to another process and so they become shared by different local environments.

In this paper, we give a new structural operational semantics to the π-calculus which considers names as being *localized* to their owners. In other words, each sequential process has its local space of names and a local name manager that generates a fresh name, whenever necessary. We use here the adjective *local* in the sense of Algol-like languages, with no connection with the physical localities where processes run. Therefore, our description is less detailed than those which enable their users to explicitly handle the allocation of processes on processors (see e.g. [13, 8, 16, 2]). Indeed here actual distribution is completely transparent to users and only logical separation of processes, induced by parallel composition, is considered. In the conclusions, we suggest a possible extension to our work that also takes care of the actual placement of processes. We only note that management and resolution of names in a distributed setting is orthogonal to allocation of processes.

In our proposal, when a name is exported, it is equipped with the information needed to point back to the local environment where it has been installed as fresh, in the style of static chains that are used to solve non local references in sequential languages with nested environments [23]. More precisely, while deducing a communication (or an extrusion), the name exported is inductively enriched with the path from the receiving process to the one that generated the name (*not* to the sender). We call this path a *relative address*. The reference to the generator of a name is maintained during all the communications by an operator of *address composition*, which updates the relative addresses of names. Names generated by different environments are thus certainly kept distinct. Note that update of names is unnecessary if every process has a unique identifier, e.g. the mail addresses of [1]. But this entails having a *centralized, global* authority that assigns these unique identifiers everytime a new component is added to the system. We can think of our proposal as implementing a *lazy* addressing, while the one that statically associates a unique identifier with each names can be seen as an *eager* addressing. The eager version could result more efficient at run-time, because it requires no dynamic calculation of addresses during communications. However, this gain is lost and our lazy addressing seems more suited than the eager one when some checks are to be made while names are communicated along channels, e.g. for enforcing security properties, like authentication.

We characterize our address composition algebraically. Our proposal makes it useless the global, therefore inefficient, check that a name involved in a transition captures names already in use. Also, α-conversions are no longer necessary to enforce disjointness of local environments (cf. the semantic definitions of the calculi for mobile processes). In this respect, Ambler's work [3] is quite similar to ours, although it is not easy to equip sequential processes with their local environments as in our proposal. This is because [3] uses nesting of binders à la de Bruijn as basic mechanism for naming, while we use parallel composition.

Our transition system for the π-calculus is more concrete and detailed than the original one, because it is nearer to an implementation. However, the two are strongly related. A transition is present in our transition system if and only if a variant of it is present in the original one (see the next Section for the notion of variant). By the way, variants are banned from our transition system, which is therefore *inherently* finitely-branching, rather than finitely-branching up to α-conversions. Our proposal conservatively extends the standard semantics of the π-calculus with local environments. In fact, we adapt the standard notion of bisimulation in such a way that two processes are equivalent (with the new notion of behaviour) if and only if they are equivalent according to the standard one.

Due to the widespread dissemination of systems based on the mobile programming paradigm, we also study how to deal with the names of a process, when its code is sent over a network. When a process migrates, it exports all its names at the same time. Our address composition is applied to all these names. This only entails homomorphically extending our address composition to processes, and our theory scales up smoothly. As a consequence, our model can easily be transferred to real programming languages, such as *Facile* [14, 28] or *CML* [19, 24].

Roughly speaking, we define a distributed version of the π-calculus as far as data are concerned. The last two authors addressed the issue of non centralized control in [12]. The two proposals can be merged, yielding a fully non-interleaving description of the π-calculus and of other languages for mobility. We hope that our proposal might aid the design of language implementations, by suggesting how these can be made truly concurrent.

The paper is organized as follows. The next section briefly surveys the π-calculus. Then, in Section 3 there is an intuitive description of how new names are generated when needed, and how they are represented in different local environments, via relative addresses. A simple example is introduced, and is used in later sections. Section 4 formally introduces our address composition, which keeps the references to generators of names, during their exportation. The *SOS* definition of our version of the late transition system of the π-calculus is in Section 5. Its equivalence (up to α-conversion) to the original one is in Section 6, along with a more global addressing. In Section 7 we slightly modify standard bisimulation to take care of localized names, without changing the equivalence classes of the standard notion. Section 8 lifts the results of the previous sections to higher-order calculi, by taking the $HO\pi$-calculus as a test-bed.

2 The π-calculus

In this section we briefly recall the π-calculus, a model of concurrent communicating processes based on the notion of *naming*. We follow [21].

Definition 2.1 (syntax) *Let $\mathcal{N}$ be a countable, infinite set of* names *ranged over by $a, b, \ldots, x, y, \ldots$, and let τ be a distinguished element such that $\mathcal{N} \cap \{\tau\} = \emptyset$.* Processes *(denoted by $P, Q, R, \ldots \in \mathcal{P}$) are built from names according to the syntax*

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

where π may either be $x(y)$ for input*, or $\overline{x}y$ for* output *or τ for* silent *moves and A is an agent identifier. Hereafter, the trailing* $\mathbf{0}$ *will be omitted (i.e. we will write π instead of $\pi.\mathbf{0}$).*

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. The silent prefix τ denotes an action which is invisible to an external observer of the system. Summation denotes non-deterministic 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. Matching $[x = y]P$ is an `if-then` operator: process P is activated if $x = y$. Finally, each agent identifier A has a unique defining equation of the form $A(y_1, \ldots, y_k) = P$, where the y_i's are the only free names (see below) of P and $y_i \neq y_j$, if $i \neq j$.

The late operational semantics of the π-calculus is defined in *SOS* (Structural Operational Semantics) style, and the labels of transitions are τ for silent actions, $x(y)$ for 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 three cases). We recall the notion of free names $fn(\mu)$, bound names $bn(\mu)$, and names $names(\mu) = fn(\mu) \cup bn(\mu)$ of a label μ. Two functions, *sbj* and *obj*, are also defined giving the subject and the object of input and output actions, respectively.

μ	*Kind*	$fn(\mu)$	$bn(\mu)$	$sbj(\mu)$	$obj(\mu)$
τ	Silent move	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
$\overline{x}y$	Free output	$\{x, y\}$	$\emptyset$	$\{x\}$	$\{y\}$
$x(y), \overline{x}(y)$	Input and bound output	$\{x\}$	$\{y\}$	$\{x\}$	$\{y\}$

Functions *fn*, *bn* and *names* are extended in the obvious way to processes.

Below we assume the *structural congruence* $\equiv_P$ on processes, defined as the least congruence satisfying the following clauses:

- $P \equiv_P Q$ if P and Q are α-equivalent (in symbols $P =_\alpha Q$),
- $(\mathcal{P}/_{\equiv_P}, +, \mathbf{0})$ and $(\mathcal{P}/_{\equiv_P}, |, \mathbf{0})$ are commutative monoids,
- $[x = x]P \equiv_P P$,
- $(\nu\, x)(\nu\, y)P \equiv_P (\nu\, y)(\nu\, x)P$, $(\nu\, x)P \equiv_P P$ if $x \notin fn(P)$,
  $(\nu\, x)(P \,|\, Q) \equiv_P (\nu\, x)P \,|\, Q$ if $x \notin fn(Q)$, and
- $A(a_1, \ldots, a_k) \equiv_P P\{a_1/y_1, \ldots, a_k/y_k\}$, if $A(y_1, \ldots, y_k) = P$.

A *variant* of $P \xrightarrow{\mu}_L Q$ 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.

Table 1 shows the late transition system of the π-calculus. The transition in the conclusion of each rule, as well as in the axiom, stands for all its variants.

$$act : \mu.P \xrightarrow{\mu}_L P$$

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

$$res : \frac{P \xrightarrow{\mu}_L P'}{(\nu x)P \xrightarrow{\mu}_L (\nu x)P'}, x \notin names(\mu) \qquad open : \frac{P \xrightarrow{\overline{x}y}_L P'}{(\nu y)P \xrightarrow{\overline{x}(y)}_L P'}, y \neq x$$

$$close : \frac{P \xrightarrow{\overline{x}(y)}_L P', Q \xrightarrow{x(y)}_L Q'}{P|Q \xrightarrow{\tau}_L (\nu y)(P'|Q')} \qquad com : \frac{P \xrightarrow{\overline{x}y}_L P', Q \xrightarrow{x(z)}_L Q'}{P|Q \xrightarrow{\tau}_L P'|Q'\{y/z\}}$$

Table 1: Late transition system for the π-calculus.

3 Handling names

Consider for a while the binary parallel composition as the main operator of the π-calculus, stipulating now that it is neither associative nor commutative. Then, build abstract syntax trees of processes as binary trees whose nodes are $|$ operators and whose leaves are the sequential components of the whole process. Call them *trees of (sequential) processes* (see Fig. 1). Assume that their left (respectively right) branches denote the left (respectively right) component of parallel compositions, and label their arcs with tag $||_0$ (respectively $||_1$). Therefore, any sequential component of a process is uniquely identified by a string ϑ over $\{||_0, ||_1\}^*$. The string corresponds to a path from the root, the top-level $|$ of the whole process, to a leaf. Intuitively, ϑ is the address of the sequential component relative to the root of the binary tree. In other words, the sequential component is reachable via ϑ from the root. We can also consider trees, with an n-adic parallel operator, as well: just take n tags $||_0, ||_1, \ldots, ||_{n-1}$. Indeed, our results are insensitive to the arity of the parallel operator. Through a set of *SOS* rules, we accumulate strings of tags which are used to specify a distributed name manager that handles names locally to sequential processes. In this way, the space of names of a whole process is partitioned into local environments associated with its sequential sub-processes. Of course, a distributed environment rules out the equations that manage restrictions globally, e.g. $(\nu\, x)(P \mid Q) \equiv_P (\nu\, x)P \mid Q$ if $x \notin \mathit{fn}(Q)$ becomes useless.

In a process, bound names may become free through either input actions or extrusions. When a bound name becomes free, an expensive α-conversion may be needed to prevent other free names from being captured. When such a clash occurs, a fresh name replaces the one that becomes free (see [20] for a detailed discussion), and this requires an extensive search for names in the whole system.

To avoid global management of names, we have to solve two problems. Names have to be generated *locally* and to be brand-new in that local environment. Furthermore, when a name is exported to other local environments via communications or extrusions, we must guarantee that it *captures no* other *free name* around.

First, we introduce a new indexed set of *localized names* (for the sake of simplicity, integers), and we associate a counter with each sequential process. When needed, the first name not in use is taken and the counter is increased. If the firing of a prefix enables new sequential processes, the counter is distributed to all of them. Clearly, this mechanism guarantees that a newly generated name is unique within its generator and does not capture other names therein.

The second problem arises when two different sequential processes, say G and R, have generated two syntactically equal names, say n. However they are semantically distinct. Suppose now that G sends its n to R. It is necessary to distinguish between the two different instances of n. Compare this situation with that of names with limited scopes in Algol-like languages. In the same program it is possible to define the same identifier more than once to denote distinct objects. The mechanism of static chains dynamically solves the problem, keeping track of the local environment in which declarations are made. In our case, in the local environment of R, the name generated by G, will be enriched with the *address* of G *relative* to R, which points back from R to the local environment of G. The relative address can be decomposed into two parts according to the minimal common predecessor P of G and R in the tree. A relative address is a string $\vartheta \bullet \vartheta' \in \{||_0, ||_1\}^* \bullet \{||_0, ||_1\}^*$, where ϑ represents the path from P to R, and ϑ' the path from P to G. Consider Fig. 1, and let G be P_3 and R be P_1. The address of P_3 relative to P_1 is $||_0||_1 \bullet ||_1||_1||_0$ (read the path upwards from P_1 to the root and reverse, then downwards to P_3). We will inductively build relative addresses while deducing transitions. We use the transition system of the π-calculus introduced in [12]. It suffices to record the application of inference rules involving the $|$ in the label of a deduced transition.

A slightly more complex situation arises when a process receives a name and sends it to another process. The name must arrive at the new receiver with the address of the generator (*not* of the sender) relative to the new receiver. This is done by composing relative addresses. Consider again Fig. 1, where P_1 sends to P_2 a name that was generated by P_3 (i.e. with relative address $||_0||_1 \bullet ||_1||_1||_0$). The rules for communication provide us with the address of P_1 relative to P_2, i.e. $||_1||_0 \bullet ||_0||_1$. The composition of the two relative addresses, written $||_1||_0 \bullet ||_0||_1 \star ||_0||_1 \bullet ||_1||_1||_0$, will result in $||_0 \bullet ||_1||_0$, where $\star$ is the address composition defined in the next section.

Relative addresses and their distributed management are worthless if each process gets a unique identifier by some centralized authority. We prefer not to have such a global name generator and our proposal makes it useless.

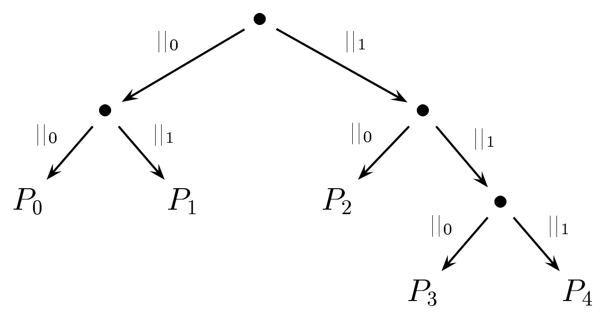

Figure 1: The tree of (sequential) processes of $(P_0|P_1)|(P_2|(P_3|P_4))$.

An example

To make it more intuitive, we instantiate the example above to a simple logical network of a typical educational department (see Fig. 2). Through suitable routines, processes P_i can access their own data-bases, which store information about didactic and administrative matters. In our simplified description, we first divide the departmental network into a didactic and an administrative sub-network identified by tags "did" and "adm", respectively. The didactic branch is further decomposed into a server for teachers (P_0, reachable via "teach"), and one for students (P_1, reachable via "stud"). The administrative staff deal with financial issues and didactic matters. The corresponding sub-networks are identified by "financial" (leading to P_2) and "did". Note that the second "did" is different from the first one, because it occurs within the sub-network for an administration identified by "adm". Finally, within the administration, didactic matters are subdivided into information on exams and lectures (stored in nodes P_3 and P_4, respectively). As an example of a relative address, consider did.teach$\bullet$adm.did.exams, which is P_3's address relative to P_0, the server for teachers (we separate branch identifiers by dots ".").

We consider a simple specification of the flow of information in the departmental network. In Section 6 we shall exemplify our way of handling names in the style of Internet, and in Section 8 the case when processes can be communicated.

Teachers and students can exchange information along two links, about the programs and the timetable of a course. This piece of information is identified by the global name *prog&timetab*; and likewise for lecture notes and exams dates (*notes&dates*). The teacher's link, *t_chan*, is global. The student's link, $\bullet s\_chan$ (shorthand for $\epsilon\bullet\epsilon s\_chan$), has the empty relative address, because it belongs to his/her environment.

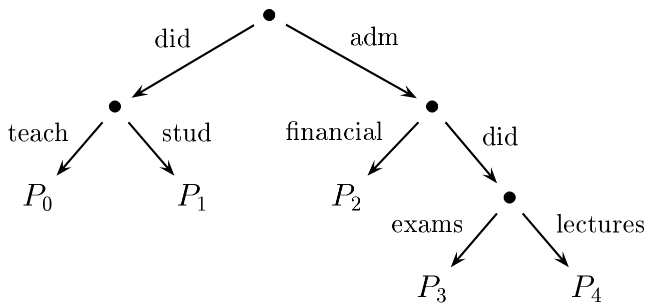

Figure 2: The departmental tree of $(P_0|P_1)|(P_2|(P_3|P_4))$.

A possible specification of the network in the π-calculus follows. Hereafter, names are written in italics, and relative addresses and constants in roman.

$$
\begin{aligned}
&\mathrm{Teacher} = \mathit{t\_chan}(\mathit{req}, y).\\
&\quad [\mathit{req} = \mathit{prog\&timetab}]\overline{y}\langle \bullet \mathit{myprog}, \mathrm{did.teach}\bullet\mathrm{adm.did.lectures}\, \mathit{mytimetab}\rangle.\mathrm{Teacher}\\
&\quad +[\mathit{req} = \mathit{notes\&dates}]\overline{y}\langle \bullet \mathit{mynotes}, \mathrm{did.teach}\bullet\mathrm{adm.did.exams}\, \mathit{mydates}\rangle.\mathrm{Teacher}\\
&\quad +\mathrm{Work}
\end{aligned}
$$

$$
\begin{aligned}
&\mathrm{Stud} = \overline{\mathit{t\_chan}}\langle \mathit{prog\&timetab}, \bullet \mathit{s\_chan}\rangle.\bullet \mathit{s\_chan}(a, b).\overline{\mathit{print}}\langle a, b\rangle.\mathrm{Stud}\\
&\quad +\overline{\mathit{t\_chan}}\langle \mathit{notes\&dates}, \bullet \mathit{s\_chan}\rangle.\bullet \mathit{s\_chan}(a, b).\overline{\mathit{print}}\langle a, b\rangle.\mathrm{Stud}\\
&\quad +\mathrm{Study}.
\end{aligned}
$$

A teacher either works (Work) or waits for a request on channel $\mathit{t\_chan}$ from a student who also communicates the link on which he/she is waiting for the answer ($\mathit{t\_chan}(\mathit{req}, y)$ is an input of two arguments[1]). Then, the teacher answers driven by a matching, and resumes ($\overline{y}\langle z, w\rangle$ is a two-place output). The pointer to the timetable of lectures, *mytimetab*, is prefixed with the address of P_4 relative to P_0. The student waits for the information requested and then prints it.

Two interactions between a teacher and a student, followed by a print action, are depicted in Fig. 3. In the first communication the student asks the teacher for the course program and timetable. The effect is to bind in Teacher the placeholder y to the link $\mathit{s\_chan}$ defined at the address teach$\bullet$stud, and the name *req* to *prog&timetab* (recall that the latter is global). This enables the first output, because the matching $[\mathit{req} = \mathit{prog\&timetab}]$ succeeds. The teacher then uses the student's link, known as $\bullet \mathit{s\_chan}$ in P_1, and teach$\bullet$stud $\mathit{s\_chan}$ in P_0, to output his/her private name $\bullet\, \mathit{myprog}$ and the name *mytimetab* at address did.teach$\bullet$adm.did.lectures. While communicating the two names, the address composition transforms these relative addresses to

[1] For the sake of presentation, in this example we use the polyadic π-calculus; for its translation to the monadic version see [18].

$$
\begin{array}{c}
(\text{Teacher}|\text{Stud})|P \\
\downarrow \tau \\
\overline{\text{teach}\bullet\text{stud}\, s\_chan}\langle \bullet myprog, \text{did.teach}\bullet\text{adm.did.lectures}\, mytimetab\rangle.\text{Teacher} \mid \bullet s\_chan(a,b).\overline{print}\langle a,b\rangle.\text{Stud}|P \\
\downarrow \tau \\
\text{Teacher} \mid \overline{print}\langle \text{stud}\bullet\text{teach}\, myprog, \text{did.stud}\bullet\text{adm.did.lectures}\, mytimetab\rangle.\text{Stud}|P \\
\downarrow \overline{print}\langle \text{stud}\bullet\text{teach}\, myprog, \text{did.stud}\bullet\text{adm.did.lectures}\, mytimetab\rangle \\
(\text{Teacher}|\text{Stud})|P
\end{array}
$$

Figure 3: A computation of (Teacher|Stud)$|P$, where $P = (P_2|(P_3|P_4))$.

stud•teach and did.stud•adm.did.lectures, respectively. The new addresses are then used to print the information required in the last step of the computation.

The similarity of relative addresses with static chains used in Algol-like languages can now be better explained. A relative address qualifying a name (e.g. stud•teach) is a pointer to the local environment where that name (e.g. $\bullet\, myprog$) is defined. Our operational semantics takes care of installing new names when necessary and of updating relative addresses when names are made available as non-local references, via our address composition $\star$.

As an example of how the generation of new names looks like, consider the case in which the student asks for the assignment of two homeworks, on the global channel t_chan. Stud is extended with the summand

$$\overline{t\_chan}\langle hw, \bullet s\_chan\rangle.\bullet s\_chan(a,b).\overline{do}\langle a,b\rangle.\text{Stud}$$

and Teacher with the following summand, where $thehw$ stands for the actual homework to be presented within deadline $\bullet dueto$.

$$[req = hw](\nu\ \ thehw)\overline{y}\langle thehw, \bullet dueto\rangle.\text{Teacher}.$$

A computation involving the new summands is in Fig. 4. First the student asks for an homework. Then the teacher generates a new homework $\bullet hw1$ (counters are initialized to 1) and sends it to the student. Note that the bound name $thehw$ has been replaced by the actual name $\bullet hw1$, local to the teacher's environment. The application of the rule *Close* exports $\bullet hw1$ to Stud and makes it local to (Teacher|Stud). The student solves the exercise via the *do* operation, within stud•teach *dueto*. The next interaction forces the generation of another homework $\bullet hw2$, where 2 is the actual value of the counter, increased by the previous extrusion. The usage and management of counters is formalized in Section 5.

$$
\begin{array}{c}
(\mathrm{Teacher}|\mathrm{Stud})|P \\
\big\downarrow \tau \\
((\nu\, \mathit{thehw})\overline{\mathrm{teach}{\scriptstyle\bullet}\mathrm{stud} s\_chan}\langle \mathit{thehw}, {\scriptstyle\bullet}\mathit{dueto}\rangle.\mathrm{Teacher} \mid s\_chan(a,b).\overline{do}\langle a,b\rangle.\mathrm{Stud})|P \\
\big\downarrow \tau \\
(\nu\, \mathrm{teach}{\scriptstyle\bullet}hw1)(\mathrm{Teacher} \mid \overline{do}\langle \mathrm{stud}{\scriptstyle\bullet}\mathrm{teach}\, hw1, \mathrm{stud}{\scriptstyle\bullet}\mathrm{teach}\, \mathit{dueto}\rangle.\mathrm{Stud})|P \\
\big\downarrow \overline{do}(\mathrm{stud}{\scriptstyle\bullet}\mathrm{teach}\ hw1, \mathrm{stud}{\scriptstyle\bullet}\mathrm{teach}\ \mathit{dueto}) \\
(\mathrm{Teacher}|\mathrm{Stud})|P \\
\big\downarrow \tau \\
((\nu\, \mathit{thehw})\overline{\mathrm{teach}{\scriptstyle\bullet}\mathrm{stud} s\_chan}\langle \mathit{thehw}, {\scriptstyle\bullet}\mathit{dueto}\rangle.\mathrm{Teacher} \mid s\_chan(a,b).\overline{do}\langle a,b\rangle.\mathrm{Stud})|P \\
\big\downarrow \tau \\
(\nu\, \mathrm{teach}{\scriptstyle\bullet}hw2)(\mathrm{Teacher} \mid \overline{do}\langle \mathrm{stud}{\scriptstyle\bullet}\mathrm{teach}\, hw2, \mathrm{stud}{\scriptstyle\bullet}\mathrm{teach}\, \mathit{dueto}\rangle.\mathrm{Stud})|P \\
\big\downarrow \overline{do}(\mathrm{stud}{\scriptstyle\bullet}\mathrm{teach}\ hw2, \mathrm{stud}{\scriptstyle\bullet}\mathrm{teach}\ \mathit{dueto}) \\
(\mathrm{Teacher}|\mathrm{Stud})|P
\end{array}
$$

Figure 4: A computation of (Teacher|Stud)$|P$, where $P = (P_2|(P_3|P_4))$.

4 Address composition

Here we introduce relative addresses and show how these are updated when names are exported.

Definition 4.1 (relative addresses) *Let $\vartheta, \vartheta', \vartheta_i, \ldots \in \{||_0, ||_1\}^*$, and let ϵ be the empty string. Then, the set of* relative addresses *is*

$$\mathcal{A} = \{\vartheta_0{\scriptstyle\bullet}\vartheta_1 \mid \vartheta_i = ||_j\vartheta'_i \ \ \textit{implies}\ \vartheta_{i\oplus 1} = ||_{j\oplus 1}\vartheta'_{i\oplus 1}, i, j = 0, 1\}$$

where $\oplus$ is sum modulo 2. We will sometimes omit ϵ in relative addresses.

As we said in the previous section, we use relative addresses to encode paths between pairs of nodes of binary trees of processes, like the one in Fig. 1. Note that the condition $||_0\vartheta'_0{\scriptstyle\bullet}||_1\vartheta'_1$ (and $||_1\vartheta'_0{\scriptstyle\bullet}||_0\vartheta'_1$) makes it explicit that the two components of the relative address describe the two distinct paths going out from the *same* node in a binary tree. This node is the minimal common predecessor of the generator of a name and its user. Also $\epsilon{\scriptstyle\bullet}\vartheta$, $\vartheta{\scriptstyle\bullet}\epsilon$ and $\epsilon{\scriptstyle\bullet}\epsilon$ are relative addresses, meaning that one of the two nodes is a predecessor of the other or that they coincide. Actually, there are only three relative positions of the minimal common predecessors of the possible pairs of the processes that generate (G), receive (R) and send (S) a name in an abstract syntax tree, up to symmetries. These three cases are depicted in Fig. 5. Note that degenerative cases are obtained when some nodes coincide, and symmetries do not alter relative addresses. As a matter of fact, relative addresses determine minimal paths between nodes, as stated below, where ϑ^R means reversing the oriented path ϑ.

Proposition 4.2 *Given two nodes P and P' in a tree, if $\vartheta \bullet \vartheta'$ is the address of P relative to P', then $\vartheta^R \vartheta'$ is the minimal path from P' to P.*

Proof. By definition of relative addresses, ϑ and ϑ' have no part in common. Furthermore, no cycles are possible in a tree. $\square$

We now define address composition. It is partial, but we will make sure later that it is defined whenever used (see Theorem 5.7). Again, Fig. 5 depicts the cases in which we will use the address composition $\star$, that discards the dashed paths.

Definition 4.3 (address composition) *Address composition $\star : (\mathcal{A} \times \mathcal{A}) \rightharpoonup \mathcal{A}$ is defined by the following three exhaustive cases:*

1. $\vartheta_0 \bullet \vartheta \star \vartheta_2 \vartheta \bullet \vartheta_3 = \vartheta_2 \vartheta_0 \bullet \vartheta_3$ *with* $\vartheta_2 \neq \epsilon$
2. $\vartheta_0 \bullet \vartheta_1 \vartheta \star \vartheta \bullet \vartheta_3 = \vartheta_0 \bullet \vartheta_1 \vartheta_3$ *with* $\vartheta_1 \neq \epsilon$
3. $\vartheta' \vartheta_0 \bullet \vartheta \star \vartheta \bullet \vartheta' \vartheta_3 = \vartheta_0 \bullet \vartheta_3$

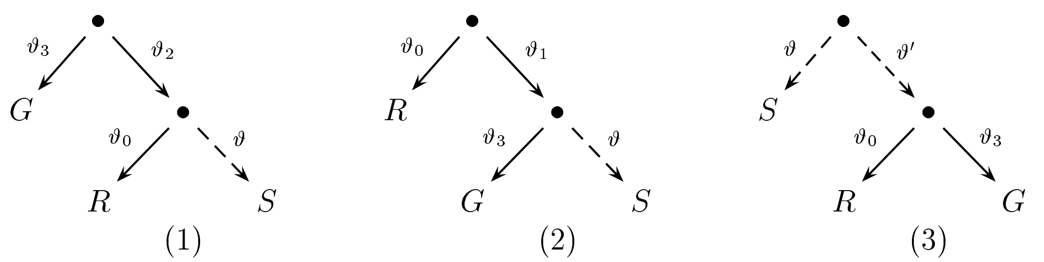

Figure 5: The three possible placements of the generator (G), the sender (S), and the receiver (R) of a name.

Below we prove a few properties of $\star$ which will be useful later. They state that $\langle \mathcal{A} \times \mathcal{A}, \star \rangle$ would be a group, if $\star$ were total. This is quite a natural property of address compositions: given a space of addresses and an interconnection topology, the address composition must always connect two sites in both ways, provided that there is a path between them. The first proposition says that $\star$ has a neutral element and an inverse on $\mathcal{A} \times \mathcal{A}$. Its proof is immediate from Def. 4.3.

Proposition 4.4 $\forall\, \vartheta_i \bullet \vartheta_j \in \mathcal{A}$ *we have that*

1. $\epsilon \bullet \epsilon \star \vartheta_i \bullet \vartheta_j = \vartheta_i \bullet \vartheta_j \star \epsilon \bullet \epsilon = \vartheta_i \bullet \vartheta_j$, *i.e.* $\epsilon \bullet \epsilon$ *is the* neutral element *of* $\star$;
2. $\vartheta_j \bullet \vartheta_i \star \vartheta_i \bullet \vartheta_j = \epsilon \bullet \epsilon$, *i.e. the* inverse *of* $\vartheta_i \bullet \vartheta_j$ *is equal to* $\vartheta_j \bullet \vartheta_i$

Note that $\epsilon \bullet \epsilon$ is the *unique* neutral element of $\star$. We prove that $\star$ is also associative.

Proposition 4.5 *Whenever defined, $\star$ is associative, i.e.*

$$(\vartheta_0 \bullet \vartheta_1 \star \vartheta_2 \bullet \vartheta_3) \star \vartheta_4 \bullet \vartheta_5 = \vartheta_0 \bullet \vartheta_1 \star (\vartheta_2 \bullet \vartheta_3 \star \vartheta_4 \bullet \vartheta_5).$$

Proof. The proof is by case analysis according to the items in Def. 4.3. Three cases are possible: *i)* $\vartheta_2 = \vartheta\vartheta_1$, *ii)* $\vartheta_1 = \vartheta\vartheta_2$, and *iii)* $\vartheta_2 = \vartheta_1$.
Consider case *i)*. We have $(\vartheta_0 \bullet \vartheta_1 \star \vartheta\vartheta_1 \bullet \vartheta_3) \star \vartheta_4 \bullet \vartheta_5 = \vartheta\vartheta_0 \bullet \vartheta_3 \star \vartheta_4 \bullet \vartheta_5$. We now need some hypotheses on ϑ_3 and ϑ_4. We still have three cases: *i')* $\vartheta_4 = \vartheta'\vartheta_3$, *ii')* $\vartheta_3 = \vartheta'\vartheta_4$, both with $\vartheta' \neq \epsilon$ and *iii')* $\vartheta_4 = \vartheta_3$.
Consider case *ii')*, in which, $\vartheta\vartheta_0 \bullet \vartheta'\vartheta_4 \star \vartheta_4 \bullet \vartheta_5 = \vartheta\vartheta_0 \bullet \vartheta'\vartheta_5$. We now prove that the right hand side of the equation in the statement of the theorem yields the same result.

$$\vartheta_0 \bullet \vartheta_1 \star (\vartheta\vartheta_1 \bullet \vartheta'\vartheta_4 \star \vartheta_4 \bullet \vartheta_5) = \vartheta_0 \bullet \vartheta_1 \star \vartheta\vartheta_1 \bullet \vartheta'\vartheta_5 = \vartheta\vartheta_0 \bullet \vartheta'\vartheta_5$$

The other combinations of hypotheses are similar and require only mechanical applications of the definition of $\star$. □

The preceding properties ensure that the identity of names is not lost during exportation (a name always encodes a pointer to its generator). More formally, sending a name from a process S to R and back is an involution, i.e. if $\vartheta_0 \bullet \vartheta_1 \star \vartheta_2 \bullet \vartheta_3$ is defined

$$\vartheta_1 \bullet \vartheta_0 \star (\vartheta_0 \bullet \vartheta_1 \star \vartheta_2 \bullet \vartheta_3) = \vartheta_2 \bullet \vartheta_3.$$

There is a categorical interpretation of the structure seen above (for terminology and definitions, see [17]). In fact every binary tree can be made into a category, where the objects are the nodes and the arrows are the paths in the tree (in our case the minimal ones). In this way the correctness of the address composition $\star$ can be immediately shown.

Proposition 4.6 *A binary tree T gives rise to the category* **Bintree**. *Its* objects *are the nodes of T and for all nodes P, P' there is a (single)* morphism *from P to P' defined as the address of P relative to P'. The morphism* composition *is $\star$; the (unique)* identity *morphism is $\epsilon \bullet \epsilon$.*
Furthermore, **Bintree** *is a groupoid and a total preorder.*

Proof. By Def. 4.1 and Proposition 4.2, relative addresses encode the minimal path between two nodes, that is unique in a tree. Hence there is a single morphism between any pair of nodes P, P' (below we write it as $P \leq P'$). The composition is associative, by Proposition 4.5. By Proposition 4.4.1, for every object P, $\epsilon \bullet \epsilon$ is the unique identity function.
Furthermore **Bintree** is a groupoid, because, by Proposition 4.4.2, every morphism is an isomorphism, having only one inverse. Finally, **Bintree** is a total preorder, because, for every P, $P \leq P$ is the identity and if $P \leq P'$ and $P' \leq P''$ then $P \leq P''$, by definition of $\star$.
□

Proposition 4.6 enables us to prove the following theorem. It says that, when a node S sends a name generated in G to a node R, address composition $\star$ correctly computes the address of R relative to G. In fact, it computes the minimal path between the two.

Theorem 4.7 (correctness of $\star$) *Let G be the generator of a name, and S be the process which receives the name and sends it to the process R. Moreover, let $\vartheta_S \bullet \vartheta_G$ and $\vartheta_R \bullet \vartheta'_S$ be the addresses of G relative to S, and of S relative to R, respectively. Then $\vartheta_R \bullet \vartheta'_S \star \vartheta_S \bullet \vartheta_G$ is the address of G relative to R.*

5 Operational semantics

We define a late operational semantics of the π-calculus which creates and handles names locally, according to the discussion in the previous sections. A few auxiliary definitions and results are helpful. We start with the set of localized names, or simply names. Localized names are prefixed by relative addresses pointing back to their generators, and they are built upon the abstract syntax of processes.

We implicitly assume a maximal degree of distribution: each sequential process is associated with its virtual processor, uniquely identified by nesting of parallel composition. The actual placement of many virtual processors on a physical one does not affect our management of names. Also, it occurs at a different level of abstraction than actual allocation of processes.

Definition 5.1 *Let $\mathcal{N}' = \mathcal{A} \cdot \mathbf{Z} \cup \mathcal{N} \cup \{\tau\}$ be the set of* localized names, *ranged over by $r, s, u, \ldots$, possibly indexed, where $\mathbf{Z}$ is the set of non-zero integer numbers, and "$\cdot$" is the operator of language concatenation.*
The new syntax of the π-calculus is obtained from the one in Def. 2.1, with r, s in place of x, y. In particular, π can either be $r(s)$ or $\overline{r}s$ or τ and a restricted name can be s.
Furthermore, we only assume on the new processes the least congruence induced by the monoidal laws for $+$, and the laws for matching and constant invocation.

The new processes with localized names differ from those in $\mathcal{P}$, because they are *not* α-convertible, and consequently variants of transitions involving them are *banned.* Also, parallel composition is *neither* associative, *nor* commutative and *no* congruence rule for restriction holds any longer. We still keep standard names in $\mathcal{N}$ and we do not prefix them with relative addresses. Of course, an initial pass could make them localized to the root of the tree of processes. We wish to avoid this simple pre-processing, because it is global.

We now start considering the problem of sending names. This requires the composition via $\star$ of the address of the name itself with the address of the sender relative to the receiver, as intuitively discussed in Section 3. First, we lift composition of addresses to exportation of names. Hereafter we write $\vartheta_0 \bullet \vartheta_1 \star \vartheta_2 \bullet \vartheta_3$ only if the composition is defined.

Definition 5.2 *Let $r = \vartheta_r \bullet \vartheta'_r n$ (resp. a) be a name. A name r exported to the relative address $\vartheta \bullet \vartheta'$ is $\vartheta \bullet \vartheta' \star r = (\vartheta \bullet \vartheta' \star \vartheta_r \bullet \vartheta'_r) n$ (resp. $\vartheta \bullet \vartheta' \star a = a$).*

Note that $\vartheta \bullet \vartheta'$ is the address of the sender relative to the receiver. Recall that names also encode a pointer to their generator (above, $\vartheta_r \bullet \vartheta'_r$), unless they belong to $\mathcal{N} \cup \{\tau\}$. This definition makes it explicit that two localized names, syntactically different, that occur in different sequential processes, can denote the *same* name. For instance, both $\bullet s\_chan$ in P_1 and teach$\bullet$studss_chan in P_0 denote the same link between Stud and Teach in the example of Section 3 (or rather, using the relative addresses in $\mathcal{A}$, $\bullet s\_chan$ is the same as $||_0 \bullet ||_1 s\_chan$).

The object, subject, names (free and bound) of an action (still ranged over by μ) are as usual. Below, we extend the needed notions to processes with localized names.

Definition 5.3 *The* free localized names *of a process are defined by induction in the standard way, except for*

$$fn(P_0|P_1) = \{\bullet||_0 \star fn(P_0)\} \cup \{\bullet||_1 \star fn(P_1)\},$$

where the operation $\star$ applies elementwise to its second argument.
Similarly for the bound localized names *of P, $bn(P)$.*
Finally, $names(P) = fn(P) \cup bn(P)$.

Intuitively, $\bullet||_i \star r$ translates a name r from process P_i into $P_0|P_1$, i.e. one level upwards in the tree of processes. To show how this happens, consider

$$fn(\bullet 1(b).(\overline{a}||_0 \bullet 1.\overline{b}\bullet 1 \mid a(x).\overline{||_1 \bullet 1 x})) =$$

$$\{\bullet 1\} \cup (\{\bullet||_0 \star fn(\overline{a}||_0 \bullet 1.\overline{b}\bullet 1)\} \cup \{\bullet||_1 \star fn(a(x).\overline{||_1 \bullet 1 x})\}) \setminus \{b\} =$$
$$\{\bullet 1\} \cup (\{a, \bullet 1, b, \bullet||_0 1\} \cup \{a, \bullet 1\}) \setminus \{b\} = \{\bullet 1, a, \bullet||_0 1\}.$$

The last step but one, determines how the name $||_0 \bullet 1$ in $P_0 = \overline{a}||_0 \bullet 1.\overline{b}\bullet 1$ is known by the whole process. Since P_0 is reachable from the root of the whole tree via $||_0$, we export $||_0 \bullet 1$ to $\bullet||_0$. Similarly for $\bullet 1$ in P_0, known as $\bullet||_0 1$ at the root. Symmetrically for the names in the other parallel component.

To generate new names locally, we enrich processes with a counter, implemented as a family of operators $n \Rightarrow$ (with $n \geq 1$) in the style of causal transition systems [10]. The intuitive meaning of $n \Rightarrow P$ is that P has generated $n-1$ new names and the next one will be n.

Definition 5.4 *Let P be a process and n be a non-zero natural number. The* extended processes *(denoted by $t, t_i, t' \ldots \in \mathcal{T}$), are defined according to the syntax*

$$t ::= n \Rightarrow P |\ (\nu s)t \mid t|t$$

where P is as in Def. 5.1. We assume on $\mathcal{T}$ the least congruence $\equiv$ which satisfies the following clauses

- $n \Rightarrow (\nu\, s)P \equiv (\nu\, s) n \Rightarrow P$ *and*

- $n \Rightarrow (P|Q) \equiv (n \Rightarrow P)|(n \Rightarrow Q)$.

The computations of a process $P \in \mathcal{P}$ start from the extended process $1 \Rightarrow P$.

The standard notions on names (free, bound) are carried over extended processes in the obvious way.[2] Note in passing that the operators $n \Rightarrow$ do *not* distribute neither over prefix nor matching nor $+$.

To exemplify the use of operators $n \Rightarrow$ and of the congruence $\equiv$, suppose to have a computation from the extended process $1 \Rightarrow R$ to the extended process $2 \Rightarrow \overline{a}b.(P|Q)$. Since the counter is 2, one new name has been generated. After the step $\overline{a}b$, one reaches the extended process $2 \Rightarrow (P|Q)$ congruent to $2 \Rightarrow P \mid 2 \Rightarrow Q$. When Q evolves to Q' by generating a new name, one gets the extended process $2 \Rightarrow P \mid 3 \Rightarrow Q'$ (see also the computation in Fig. 6).

We extend the standard substitution of one name for another, in the style of [9]. By abuse of notation we write $-\vartheta \bullet \vartheta' n$ for $\vartheta \bullet \vartheta' - n$ in the following items 2 and 7.

Definition 5.5 *Let $\{-/-\}$ be the standard substitution. Then the* routed substitution $\{\!|\, - / - \,|\!\}$ *is defined by induction as follows:*

1. $(n \Rightarrow P)\{\!|u'/u|\!\} = n \Rightarrow P\{\!|u'/u|\!\}$

2. $((\nu\ s)t)\{\!|u'/u|\!\} = \begin{cases} (\nu\ -u)(t\{\!|-u/u'|\!\})\{\!|u'/u|\!\} & \textit{if } u' = s \\ (\nu\ s)t & \textit{if } u = s \\ (\nu\ s)(t\{\!|u'/u|\!\}) & \textit{otherwise} \end{cases}$

3. $(t_0|t_1)\{\!|u'/u|\!\} = t_0\{\!|u'_0/u_0|\!\}|t_1\{\!|u'_1/u_1|\!\}$, *where* $u'_i = ||_i \bullet \star u'$ *and* $u_i = ||_i \bullet \star u$, *for* $i = 0, 1$

4. $\mathbf{0}\{\!|u'/u|\!\} \ = \mathbf{0}$

5. $(\tau.P)\{\!|u'/u|\!\} = \tau.P\{\!|u'/u|\!\}$

6. $(\overline{r}s.P)\{\!|u'/u|\!\} = (\overline{r}s)\{u'/u\}.P\{\!|u'/u|\!\}$

7. $(r(s).P)\{\!|u'/u|\!\} = \begin{cases} r\{u'/u\}(-u).P\{\!|-u/u'|\!\}\{\!|u'/u|\!\} & \textit{if } u' = s \\ r\{u'/u\}(s).P & \textit{if } u = s \\ r\{u'/u\}(s).(P\{\!|u'/u|\!\}) & \textit{otherwise} \end{cases}$

8. $(P + Q)\{\!|u'/u|\!\} = P\{\!|u'/u|\!\} + Q\{\!|u'/u|\!\}$

9. $[r = s]P\{\!|u'/u|\!\} = [r\{u'/u\} = s\{u'/u\}]P\{\!|u'/u|\!\}$

10. $A(r_1, \ldots, r_k)\{\!|u'/u|\!\} = A(r_1\{u'/u\}, \ldots, r_k\{u'/u\})$

[2] More precisely, $names(t) = names(FC(t))$, where FC is introduced in Def. 6.1.

For the sake of brevity, we consider only the cases $(\nu s)t$ and $t_0|t_1$ in items 2 and 3 above, although one needs also the cases $(\nu s)P$ and $P_0|P_1$. Their definitions are just the same, so we omit them. Essentially, the rooted substitution distributes over the counters $n \Rightarrow$ and also over the operators of processes, and then acts as the standard one. While descending within the components of a parallel composition, the rooted substitution inductively updates both the localized names u and u'. This amounts to pushing these names one level downwards in the tree of processes (in Def. 5.3 they were lifted one level upwards, instead). Two crucial items are 2 and 7, when $u' = s$. We will prove that the substitutions $\{\!| -u/u' |\!\}$ actually are local α-conversions, because the negative name $-u$ is fresh in t and in P (see Proposition 5.8). In passing, negative names occur only bound in extended processes. The α-conversions implemented by items 2 and 7 are the only ones occurring in our setting, together with those possibly called by the congruence rule for constant invocation. Rooted substitutions will be used in the operational semantics. We will make sure later that, whenever applied, rooted substitution is a conservative extension of the standard one to localized names. Also, by inspecting Tab. 2, it will be clear that, if $u' \in \mathcal{N}$ then $u \in \mathcal{N}' \setminus \mathcal{N}$.

We will now work out an application of routed substitution. Consider for instance

$$(2 \Rightarrow \overline{x}y.w(z).\overline{z}z \mid 2 \Rightarrow x(z).(\nu\, x)\overline{w}x.(x(y) \mid \overline{x}z))\{\!|\bullet 1/x|\!\}.$$

First, item 3 applies, that distributes the substitution $\{\!|\bullet 1/x|\!\}$ to the two components. The name $\bullet 1$ is enriched with the tag $||_0$ (respectively $||_1$) in the left (respectively right) component needed to point back to the environment where $\bullet 1$ has been created. Indeed, $\bullet 1$ exported to $||_i\bullet$ is $||_i\bullet \star \bullet 1 = ||_i\bullet 1$. Instead, the name x to be substituted is not enriched with a relative address, because $x \in \mathcal{N}$. So, one obtains

$$\begin{aligned}
&(2 \Rightarrow \overline{x}y.w(z).\overline{z}z\{\!|\ ||_0\bullet 1/x|\!\}) \mid (2 \Rightarrow x(z).(\nu\, x)\overline{w}x.(x(y) \mid \overline{x}z)\{\!|\ ||_1\bullet 1/x|\!\}) = \\
&\quad (2 \Rightarrow \overline{||_0\bullet 1}y.w(z).\overline{z}z) \mid (2 \Rightarrow ||_1\bullet 1(z).((\nu\, x)\overline{w}x.(x(y) \mid \overline{x}z))\{\!|\ ||_1\bullet 1/x|\!\}) = \\
&\quad (2 \Rightarrow \overline{||_0\bullet 1}y.w(z).\overline{z}z) \mid (2 \Rightarrow ||_1\bullet 1(z).(\nu\, x)\overline{w}x.(x(y) \mid \overline{x}z))
\end{aligned}$$

Note that, the substitution takes the binders of names into account correctly. In fact, the distribution of $\{\!|\bullet 1/x|\!\}$ stops when the new binder $(\nu\, x)$ is encountered, according to the second case of item 2. This example will be used just after Proposition 5.8.

We show now an example where an α-conversion is forced. Suppose to have $(n+1) \Rightarrow (\nu\, a)\ \overline{a}\ \bullet n\{\!|a/\bullet n|\!\}$.[3] Then, the first rule of item 2 applies and exploits the fresh negative name $\bullet - n$, yielding:

$$\begin{aligned}
&(n+1) \Rightarrow (\nu \bullet - n)\ \overline{a}\ \bullet n\{\!|\bullet - n/a|\!\}\{\!|a/\bullet n|\!\} = \\
&(n+1) \Rightarrow (\nu \bullet - n)\ \overline{\bullet - n}\ \bullet n\{\!|a/\bullet n|\!\} = (n+1) \Rightarrow (\nu \bullet - n)\ \overline{\bullet - n}\ a.
\end{aligned}$$

For notational convenience, we introduce two forms of selective substitutions within a sub-process reachable from the whole process via a path ϑ.

[3]This is an intermediate step in the deduction of the communication of the process $(\nu\, b)(m \Rightarrow\ \overline{b}a|n \Rightarrow b(x)(\nu\, a)\overline{a}x)$, according to Tab. 2.

Definition 5.6 *Let t be an extended process and $\vartheta \in \{||_0, ||_1\}^*$. Then, the* selective rooted substitution $t\{\!|/|\!\}_{@\vartheta}$ *is defined by induction as*

- $(t_0|t_1)\{\!|r/s|\!\}_{@\vartheta} = \begin{cases} t_0\{\!|r/s|\!\}_{@\vartheta'}|t_1 & \text{if } \vartheta = ||_0\vartheta' \\ t_0|t_1\{\!|r/s|\!\}_{@\vartheta'} & \text{if } \vartheta = ||_1\vartheta' \end{cases}$

- $t\{\!|r/s|\!\}_{@\epsilon} = t\{\!|r/s|\!\}$

The selective increment *of counters $t\{n+1 \Rightarrow\ /n \Rightarrow\}_{@\vartheta}$ is defined as follows, by stipulating that $(n+1\Rightarrow)$ and $(n\Rightarrow)$ are considered as tokens*

- $(t_0|t_1)\{n+1\Rightarrow\ /n\Rightarrow\}_{@||_i\vartheta} = t_i\{n+1\Rightarrow\ /n\Rightarrow\}_{@\vartheta}$

- $t\{n+1\Rightarrow\ /n\Rightarrow\}_{@\epsilon} = t\{n+1\Rightarrow\ /n\Rightarrow\}$.

The selective rooted substitution first travels downwards a tree of processes to single out the sub-process P' reachable via ϑ. Then, it applies the rooted substitution to P'. Note that the names s and r are *not* updated while descending the tree, because when this selective substitution is called (rules Com_i and $Close_i$ in Tab. 2), they are already localized in P'. The selective increment simply adds 1 to the counter of the process reachable via ϑ. It will be used in rule *Open*, only. Both $t\{\!|-/-|\!\}_{@\vartheta}$ and $t\{-/-\}_{@\vartheta}$ are partial. It will be evident that they are indeed defined, whenever used in the rules for the operational semantics.

The late transition system for the π-calculus with localized names is in Tab. 2. Following the ideas of [11, 12], we encode in the labels of transitions the parallel structure of processes to identify the sequential component that moves. More precisely a label is

$$\mu@\vartheta \quad \text{(or simply } \mu \text{ if } \vartheta = \epsilon \text{ or } \mu = \tau),$$

where ϑ encodes the nesting of the process which performed the μ action, as far as the parallel structure of the system is concerned. In fact, a tag $||_0$ (respectively $||_1$) is added whenever a rule Par_0 (respectively Par_1) is applied. Rules In and $Open$ generate a new name $\bullet n$ and increment the counter of the sequential component that moves. In the case of In, the new name is distributed to the residual through the routed substitution $\{\!|\bullet n/x|\!\}$ which enriches $\bullet n$ with the correct relative address. As for *Open*, the new name must be distributed to t, the whole process under restriction. Thus, we use the name $\bullet\vartheta n$ localized in t, i.e. $\bullet\vartheta \star \bullet n$. Also, the counter of the sub-process reachable via ϑ is increased by the operation of selective increment, because this sub-process generated a fresh name. The rules for communication *locally* check whether the channel is the same. The receiver then distributes the value read to the sequential component t_i reachable via ϑ_i (which fired the input) by using the information encoded in the @-part of the label of the transition. Indeed, s_1 is s_0 exported to $||_1\vartheta_1\bullet||_0\vartheta_0$ in Com_0; and symmetrically for Com_1. Note that rules $Close_i$ introduce the correct restriction on the name computed by sending the object of the output to the minimal common predecessor of t_0 and t_1 in the tree of processes.

Note that our rules have *no global* conditions on names. In particular, we drop the side conditions of the original operational semantics of the π-calculus on free and bound names. The only rule which applies substitutions to a context larger than a sequential process is *Open*. However, only the operand of ν is affected. This context is the minimum needed to notify the owners of a name which for them is no longer private.

We now derive a transition. Consider the extended process occurring in Fig. 6

$$2 \Rightarrow w(z).\overline{z}z \mid 3 \Rightarrow (\nu\, x)\overline{w}x.(x(||_0\bullet - 2) \mid \overline{x}y).$$

We deduce a communication between $w(z)$ and $\overline{w}x$, by using rule $Close_1$.

$$\frac{2 \Rightarrow w(z).\overline{z}z \xrightarrow{w(\bullet 2)} 3 \Rightarrow \overline{\bullet 2}\bullet 2, \quad \dfrac{3 \Rightarrow \overline{w}x.(x(||_0\bullet - 2) \mid \overline{x}y) \xrightarrow{\overline{w}x} (3 \Rightarrow x(||_0\bullet - 2) \mid 3 \Rightarrow \overline{x}y)}{(\nu\, x)(3 \Rightarrow \overline{w}x.(x(||_0\bullet - 2) \mid \overline{x}y) \xrightarrow{\overline{w}(\bullet 3)} (4 \Rightarrow ||_0\bullet 3(||_0\bullet - 2) \mid 4 \Rightarrow \overline{||_1\bullet 3}y)}}{2 \Rightarrow w(z).\overline{z}z \mid 3 \Rightarrow (\nu\, x)\overline{w}x.(x(||_0\bullet - 2) \mid \overline{x}y) \xrightarrow{\tau} (\nu\, \bullet||_1 3)(3 \Rightarrow \overline{||_0\bullet||_1 3}\, ||_0\bullet||_1 3 \mid (4 \Rightarrow ||_0\bullet 3(||_0\bullet - 2) \mid 4 \Rightarrow \overline{||_1\bullet 3}y)}$$

Consider the target of the transition above, in which $||_0\bullet 3$, $||_1\bullet 3$ and $\bullet||_1 3$ occur in different sub-processes but denote the same thing. Thus $(\nu\, \bullet||_1 3)$ enforces communication between the rightmost processes. Also, $||_0\bullet||_1 3$, the name extruded, is an alias for $\bullet||_1 3$, hence the leftmost process is stuck.

The following proposition ensures that no name in Tab. 2 is left unknown and that every call to address composition $\star$ is well defined.

Proposition 5.7 *All names in Tab. 2 are defined, i.e. every call to address composition $\star$ is defined.*

PROOF. When names are in $\mathcal{N}$, the proof is obvious. Otherwise, note that if $t \xrightarrow{\mu @ \vartheta} t'$ and $\vartheta''\bullet\vartheta' m \in \mathit{names}(\mu)$, then either ϑ'' is a suffix of ϑ or vice versa. Indeed, let P be the minimal common predecessor of $\mu @ \vartheta$ and of G, the generator of name $\bullet m$. Assume that in the derivation of the transition $t \xrightarrow{\mu @ \vartheta} t'$ P has been reached, i.e. P is a sub-term of t. Then certainly ϑ'' is a suffix of ϑ, because ϑ'' is the path from P to the sub-process of t that performs action μ, i.e. reachable via ϑ. Otherwise, if P has not been yet reached then ϑ is a suffix of ϑ''.
Now it is routine checking that all the calls to $\star$ are well defined (see Def. 4.3). $\square$

We now show that the routed substitution behaves as the standard one, whenever applied. The only difference is that relative addresses are updated and that α-conversions are made explicitly.

Proposition 5.8 *Whenever applied in Tab. 2, the routed substitution $\{\!|\, - / - \,|\!\}$ conservatively extends the standard substitution $\{-/-\}$.*

$$Out : n \Rightarrow \overline{r}s.P \xrightarrow{\overline{r}s} n \Rightarrow P \qquad In : n \Rightarrow r(s).P \xrightarrow{r(\bullet n)} (n+1) \Rightarrow P\{\!|\bullet n/s|\!\}$$

$$Tau : n \Rightarrow \tau.P \xrightarrow{\tau} n \Rightarrow P$$

$$Par_0 : \frac{t_0 \xrightarrow{\mu @ \vartheta} t'_0}{t_0|t_1 \xrightarrow{\mu @ ||_0 \vartheta} t'_0|t_1} \quad Par_1 : \frac{t_1 \xrightarrow{\mu @ \vartheta} t'_1}{t_0|t_1 \xrightarrow{\mu @ ||_1 \vartheta} t_0|t'_1} \quad Sum : \frac{n \Rightarrow P \xrightarrow{\mu @ \vartheta} t'}{n \Rightarrow (P+Q) \xrightarrow{\mu @ \vartheta} t'}$$

$$Res : \frac{t \xrightarrow{\mu @ \vartheta} t'}{(\nu u)t \xrightarrow{\mu @ \vartheta} (\nu u)t'},\ \mu \neq \tau \ \Rightarrow \begin{cases} u \notin \{r, s\} \\ r = \bullet\vartheta \star sbj(\mu) \\ s = \bullet\vartheta \star obj(\mu) \end{cases}$$

$$Com_0 : \frac{t_0 \xrightarrow{\overline{r_0}s_0 @ \vartheta_0} t'_0,\ t_1 \xrightarrow{r_1(\bullet m) @ \vartheta_1} t'_1}{t_0|t_1 \xrightarrow{\tau} t'_0|t'_1\{\!|s_1/\bullet m|\!\}_{@\vartheta_1}},\ \begin{cases} s_1 = ||_1\vartheta_1\bullet||_0\vartheta_0 \star s_0 \\ r_1 = ||_1\vartheta_1\bullet||_0\vartheta_0 \star r_0 \end{cases}$$

$$Com_1 : \frac{t_0 \xrightarrow{r_1(\bullet m) @ \vartheta_1} t'_0,\ t_1 \xrightarrow{\overline{r_0}s_0 @ \vartheta_0} t'_1}{t_0|t_1 \xrightarrow{\tau} t'_0\{\!|s_1/\bullet m|\!\}_{@\vartheta_1}|t'_1},\ \begin{cases} s_1 = ||_0\vartheta_1\bullet||_1\vartheta_0 \star s_0 \\ r_1 = ||_0\vartheta_1\bullet||_1\vartheta_0 \star r_0 \end{cases}$$

$$Open : \frac{t \xrightarrow{\overline{r}s @ \vartheta} t'}{(\nu u)t \xrightarrow{\overline{r}(\bullet n) @ \vartheta} (t'\{n+1 \Rightarrow /n \Rightarrow\}_{@\vartheta})\{\!|\bullet\vartheta n/u|\!\}},\ \begin{cases} u = \bullet\vartheta \star s \\ r \neq s \end{cases}$$

$$Close_0 : \frac{t_0 \xrightarrow{\overline{r_0}(s_0) @ \vartheta_0} t'_0,\ t_1 \xrightarrow{r_1(\bullet m) @ \vartheta_1} t'_1}{t_0|t_1 \xrightarrow{\tau} (\nu u)(t'_0|t'_1\{\!|s_1/\bullet m|\!\}_{@\vartheta_1})},\ \begin{cases} s_1 = ||_1\vartheta_1\bullet||_0\vartheta_0 \star s_0 \\ r_1 = ||_1\vartheta_1\bullet||_0\vartheta_0 \star r_0 \\ u = \bullet||_0\vartheta_0 \star s_0 \end{cases}$$

$$Close_1 : \frac{t_0 \xrightarrow{r_1(\bullet m) @ \vartheta_1} t'_0,\ t_1 \xrightarrow{\overline{r_0}(s_0) @ \vartheta_0} t'_1}{t_0|t_1 \xrightarrow{\tau} (\nu u)(t'_0\{\!|s_1/\bullet m|\!\}_{@\vartheta_1}|t'_1)},\ \begin{cases} s_1 = ||_0\vartheta_1\bullet||_1\vartheta_0 \star s_0 \\ r_1 = ||_0\vartheta_1\bullet||_1\vartheta_0 \star r_0 \\ u = \bullet||_1\vartheta_0 \star s_0 \end{cases}$$

Table 2: Late transition system of the π-calculus with localized names.

Proof. Recall that a computation starts from $1 \Rightarrow P$, with $P \in \mathcal{P}$ and that any extended process t reachable from it only has extended sub-processes $n \Rightarrow t'$ with $n \geq 1$. By Proposition 5.7, the names occurring in the rooted substitutions are all defined. Then, note that a rooted substitution preserves the identity of an object referenced to by a localized name $\vartheta \bullet \vartheta' n$ when this is updated to $||_i \bullet \star \vartheta \bullet \vartheta'$, by item 3 of Def. 5.5. Thus, it suffices to prove that a rooted substitution implements an α-conversion, whenever the standard one would. This occurs whenever $\{\!|u'/u|\!\}_{@\vartheta}$ is applied to t in the conclusion of a rule in Tab. 2, and u does not occur free in the sub-process t reachable via ϑ. In these cases, items 2 or 7 of Def. 5.5, with $u' = s$, apply. We proceed by induction of the derivation of transitions in these cases.

In: holds vacuously.

Com_i and $Close_i$: $\bullet m$ has been just generated in the sub-process t_1' reachable via ϑ, hence all negative names $\vartheta \bullet \vartheta' - k$ therein, if any, are such that $k < m$. Therefore $\bullet - m$ is fresh.

$Open$: $\bullet n$ is fresh in $n \Rightarrow P$, i.e. the sub-process of t reachable via ϑ, consequently also $\bullet \vartheta n$ is fresh in t' and in all its sub-processes, because updated by the rooted substitution, whenever needed.

$\square$

$$1 \Rightarrow a(x).(\overline{x}y.w(z).\overline{z}z \mid x(z).(\nu\, x)\overline{w}x.(x(y) \mid \overline{x}z))$$
$$\Big\downarrow a(\bullet 1)$$
$$2 \Rightarrow \overline{||_0 \bullet 1}y.w(z).\overline{z}z \mid 2 \Rightarrow ||_1 \bullet 1(z).(\nu\, x)\overline{w}x.(x(y) \mid \overline{x}z)$$
$$\Big\downarrow \tau$$
$$2 \Rightarrow w(z).\overline{z}z \mid 3 \Rightarrow (\nu\, x)\overline{w}x.(x(||_0 \bullet - 2) \mid \overline{x}y)$$
$$\Big\downarrow \tau$$
$$(\nu \bullet ||_1 3)(3 \Rightarrow \overline{||_0 \bullet ||_1 3}\, ||_0 \bullet ||_1 3 \mid (4 \Rightarrow ||_0 \bullet 3(||_0 \bullet - 2) \mid 4 \Rightarrow \overline{||_1 \bullet 3}y))$$

Figure 6: A computation of $1 \Rightarrow a(x).(\overline{x}y.w(z).\overline{z}z \mid x(z).(\nu\, x)\overline{x}v.(x(y) \mid \overline{x}z))$.

An example of how extended processes evolve is in the computation depicted in Fig. 6. Its first transition shows the generation of the new name $\bullet 1$. The application of the routed substitution $\{\!|\bullet 1/x|\!\}$ introduced by rule In is in the example after Def. 5.5. The counter $2 \Rightarrow$ is distributed to the components of the parallel composition. The second transition is a communication along a channel called $||_0 \bullet 1$ by the sender, and called $||_1 \bullet 1$ by the receiver. Indeed, $||_0 \bullet 1$ is an alias for $||_1 \bullet 1$. Also, the counter of the residual of the receiver is increased. In fact, rule In generates a new name $\bullet 2$ that will be replaced with y by substitution $\{\!|y/\bullet 2|\!\}$ in the conclusion of rule Com_0. Also there is an α-conversion, because $(x(y)|\overline{x}\ ||_1 \bullet - 2)\{\!|y/\bullet 2|\!\} = x(||_0 \bullet - 2)|\overline{x}y$. The derivation of the last transition is the one worked out before Proposition 5.7.

Our transition system is a more concrete version of the original one in Tab. 1. However, there is a tight connection between them that we will formalize at the end of the next section. Intuitively, our transitions are "rooted" variants of the original ones.

6 A Global View of Names

The names occurring in the label of $t \xrightarrow{\mu @ \vartheta} t'$ have a local meaning. This enables processes to exchange names only knowing that they run in parallel, with *no* knowledge of the *global* context in which they are plugged. Indeed, if $\mu = \overline{r}(s)$, the actual reference of r has to be computed following the relative address of r, composed with ϑ; similarly for s. In this section we show how to globalize names in the style of Internet addressing, though they are generated and handled locally. We propose to transform a name as it is known at the root of the tree of processes. We thus have a conversion table of names, but *not* a global manager of names.

The logical addresses of Internet form an n-adic tree, such that all the arcs at the same level have distinct names, each identifying a different domain. When a new address is assigned in a domain (local environment), for instance in a local area network, it is unique. Of course the same name can be assigned to hosts of other domains. They will be distinguished when prefixed with the paths to their domains. See [15] for a technique similar to ours that handles names locally in this setting.

Rephrasing the above in our case, a new local name is generated by the local counter of the sequential component that needs it, while uniqueness is ensured by the relative address of the generator with respect to the root of the system. The correctness of this proposal follows from the fact that a name is uniquely identified by a suitable composition of its relative address and of the access path from the root of the whole system to the sub-process that actually uses it.

Our localized names are converted by the following transition relation which uses the relation defined in Tab. 2 as an auxiliary arrow. By abuse of notation, we will write $\vartheta \bullet \vartheta' \star \mu$ for $\vartheta \bullet \vartheta' \star sbj(\mu)\ \ \vartheta \bullet \vartheta' \star obj(\mu)$ if $\mu \neq \tau$, and for τ otherwise.[4]

$$\boxed{\text{if } t \xrightarrow{\mu @ \vartheta} t' \text{ then } t \xmapsto{\bullet \vartheta \star \mu} t'}$$

To illustrate the new style of naming, we consider the last step of the computation in Fig. 3. The label of the output transition in $\longrightarrow$ was

$$\mu = \overline{print}\langle \text{stud}\bullet\text{teach}\, myprog, \text{did.stud}\bullet\text{adm.did.lectures}\, mytimetab\rangle @\text{did.stud},$$

and an easy calculation shows that the actual label in the output transition in $\mapsto$ is

$$\bullet\text{did.stud} \star \mu = \overline{print}\langle \bullet\text{did.teach}\, myprog, \bullet\text{adm.did.lectures}\, mytimetab\rangle.$$

[4] See also Def. 8.1. Of course, parentheses and overbars occurring in μ are preserved in $\vartheta \bullet \vartheta' \star \mu$.

The names that are object of the output are now displayed with their accessing paths from the root of the system to the processes they belong to.

Now we can show that our transition systems are in bijection with the original one of [21], up to a sort of rooted renaming of their transitions. Essentially, a transition $t \xrightarrow{\mu'@\vartheta} t'$ can be seen as a variant of $P \xrightarrow{\mu} P'$, where μ, P and P' are obtained from μ', t and t' replacing localized names with fresh names in $\mathcal{N}$. Also counters $n \Rightarrow$ are discarded from extended processes by the following auxiliary function FC.

Definition 6.1
The auxiliary function FC is the homomorphic extension of $fc(n \Rightarrow P) = P$ to extended processes.

The correspondence between the standard transition for the π-calculus and ours is now easy to state. In its statement, given suitable $A \subseteq \mathcal{N}$ and $U \subseteq (\mathcal{N}' \setminus \mathcal{N})$, we denote by $\{\!|A/U|\!\}$ the simultaneous rooted substitution $\{\!|a_1/u_1, \ldots, a_k/u_k|\!\}$ for all $u_i \in U$, with $a_i \neq a_j$ if $i \neq j$.

Theorem 6.2

$$P_0 \xrightarrow{\mu}_L P_1 \text{ if and only if } t_0 \xrightarrow{\mu'@\vartheta} t_1,$$

with $\mu = (\bullet\vartheta \star \mu')\{\!|A/U|\!\}$ and for $i = 0, 1$, $P_i =_\alpha FC(t_i\{\!|A/U|\!\})$,

where $U \supseteq (names(t_0, t_1) \cup obj(\mu')) \cap (\mathcal{N}' \setminus \mathcal{N})$ and $A \cap fn(t_0, t_1) = \emptyset$

Furthermore, $P_0 \xrightarrow{\mu}_L P_1$ if and only if $t_0 \xmapsto{\mu''} t_1$, where $\mu'' = \bullet\vartheta \star \mu'$.

Proof. We obtain a new set of rules for the standard π-calculus, equivalent to that in Tab. 1, as follows. Split axiom *act* in three axioms (*tau* for τ, *in* for input and *out* for output); remove the congruence rules for restriction; remove the monoidal laws for $|$ and add the symmetrical rules for this operator. The new rules are in bijection with those in Tab 2. Therefore, the deduction of the transitions $Q \xrightarrow{\mu} Q'$ and $t \xrightarrow{\mu@\vartheta'} t'$, with $Q = FC(t)$ are also in bijection, because the counters $n \Rightarrow$ do not affect the deduction.

Only if) Take $t_0 = n \Rightarrow P_0$ and proceed by induction on the derivation of $P_0 \xrightarrow{\mu}_L P_1$.

[*Base step*] Trivial if $\mu = \tau.P$ (rule *tau*) or $\mu = \overline{x}y.P$ (rule *out*). If the axiom is *in*, $P_0 = x(y).P_1$ and

$$n \Rightarrow P_0 \xrightarrow{x(\bullet n)} (n+1) \Rightarrow P_1\{\!|\bullet n/y|\!\}.$$

Since $\{\!|\bullet n/y|\!\}$ calls no α-conversion as $\bullet n$ is free in P_1 (and is the unique name in $\mathcal{N}' \setminus \mathcal{N}$ therein), it suffices to take $A = \{w\}$ for $w \notin names(P_0, P_1)$.

[*Inductive step*] If the last rule applied is *sum* or *res*: immediate by inductive hypothesis as no rooted substitution is needed. Similarly for the new rules par_i, because the rules in Tab. 2 require no rooted substitutions and $\{\!|A/U|\!\}$ takes care of the $||_i$ prefixed to ϑ.

The cases for the new rules for communication are symmetric, so we consider only com_0. The transition $t_0 \xrightarrow{\mu'@\vartheta} t_1$ reads as

$$n \Rightarrow P_0|P_1 \xrightarrow{\tau} t'_0|t'_1\{\!|y/\bullet m|\!\}_{@\vartheta_1},$$

and it is deduced because of $n \Rightarrow P_0 \xrightarrow{\overline{x}y@\vartheta_0} t'_0$ and $n \Rightarrow P_1 \xrightarrow{x(\bullet n)@\vartheta_1} t'_1$, which by inductive hypothesis correspond to $P_0 \xrightarrow{\overline{x}y}_L P'_0$ and to $P_1 \xrightarrow{x(z)}_L P'_1$. If $y \in bn(P'_1)$ the substitution $P'_1\{y/z\}$ may require α-conversions, replacing the bound occurrences of y with, say, $w \notin A \cup fn(t_0, t_1)$. Then

$$FC(t'_1\{\!|y/\bullet m|\!\}_{\vartheta_1})\{\!|A \cup \{w\}/U \cup \{\bullet\vartheta_1 \star \bullet - m\}|\!\} =_\alpha P'_1\{y/z\}.$$

For rule *open*, the transition $t_0 \xrightarrow{\mu'@\vartheta} t_1$ actually is

$$n \Rightarrow (\nu y)P_0 \xrightarrow{\overline{x}(\bullet n)@\vartheta} t'_1 = (t_1\{n+1 \Rightarrow /n \Rightarrow\}_\vartheta)\{\!|\bullet\vartheta n/u|\!\},$$

and is deduced because of $n \Rightarrow P \xrightarrow{\overline{x}(y)@\vartheta} t'_1$. Now, $\bullet\vartheta n$ is the only localized name not in $\mathcal{N}$ of t'_1. For some fresh w, thus $FC(t'_1\{\!|w/\bullet\vartheta n|\!\}) =_\alpha P_1$.

Similarly for the new rules $close_i$, because the objects of the bound output $\overline{x}(\bullet n)@\vartheta_{out}$ and of the input $x(\bullet m)@\vartheta_{in}$ are the only localized names not in $\mathcal{N}$ of the processes in the premises, and the rooted substitution requires no introduction of negative names.

If) The deduction Δ of $t_0 \xrightarrow{\mu'@\vartheta} t_1$ induces a corresponding deduction Δ' of

$$1 \Rightarrow FC(t_0\{\!|A/U|\!\}) \xrightarrow{\mu'\{\!|A/U|\!\}@\vartheta} t'_1 = t_1\{\!|A/U|\!\},$$

because counters $n \Rightarrow$ do not affect deductions. t'_1 has at most two localized names not in $\mathcal{N}$, i.e. $names(t'_1) \cap \{\mathcal{N}' \setminus \mathcal{N}\} \subseteq \{\bullet 1, \bullet - 1\} = U'$. Now, for $w, z \notin names(t_0, t'_1) \cup A$, the required P_0 is $FC(t_0\{\!|A \cup \{w,z\}/U \cup U'|\!\}) = FC(t_0\{\!|A/U|\!\})$. Now the transition $P_0 \xrightarrow{\mu}_L P_1$ can be derived mimicking Δ', where P_1 is $FC(t_1\{\!|A \cup \{w,z\}/U \cup U'|\!\})$ and $\mu = (\bullet\vartheta \star \mu')\{\!|A \cup \{w,z\}/U \cup U'|\!\}$.

The second claim follows immediately, by definition of $\longmapsto$. □

The new transition relation $t \xmapsto{\mu'} t'$ permits a global observer to follow the evolution of a system, by abstracting from its internal structure. In the next section we will easily adapt standard bisimulation to our case, relying on this new arrow $\longmapsto$. Note also that all the computations in our new setting are fresh, in the sense of [4]. Roughly speaking, in a fresh computation all names exported by an extrusion or imported by an input have never been used in the preceding transitions.

Finally, note that here the space of logical addresses is binary because the parallel operator of the π-calculus is such. Recall that our address composition requires no modification if trees are n-adic. As a consequence, our proposal is flexible enough to handle n-adic trees of logical addresses. For example, we model the insertion of a domain D' in a given domain D with n sub-domains, by adding a new arc from D to D', labelled with $||_{n+1}$ (see Fig. 7). In fact, the rule defining the new transition $\longmapsto$ shows that no modification is needed on the "new" names occurring in D', as well as on the "old" ones before the insertion.

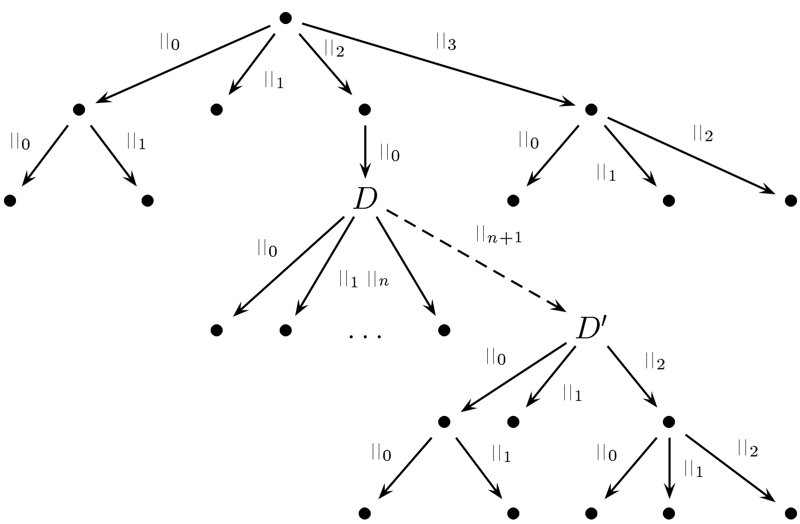

Figure 7: Adding a new domain D' as a sub-domain of D in a tree of processes.

7 Equivalences

The equivalences on the π-calculus are usually based on bisimulations. Their standard definitions compare the behaviour of a computational step in one system with that of another and check the target systems for bisimilarity. We first recall the definition of late bisimulation [21], and then we accommodate it to fit our framework.

Definition 7.1 (late bisimulation) *A binary relation $\mathcal{S}$ on processes is a* late simulation *if $P\,\mathcal{S}\,Q$ implies that*

- *If $P \xrightarrow{\mu}_L P'$ and μ is τ, $\overline{x}y$, or $\overline{x}(y)$ with $y \notin fn(P,Q)$, then for some Q', $Q \xrightarrow{\mu}_L Q'$ and $P'\,\mathcal{S}\,Q'$*
- *If $P \xrightarrow{x(y)}_L P'$ and $y \notin fn(P,Q)$, then for some Q', $Q \xrightarrow{x(y)}_L Q'$ and for all $w \in \mathcal{N}$, $P'\{w/y\}\mathcal{S}Q'\{w/y\}$.*

The relation $\mathcal{S}$ is a late bisimulation *if both $\mathcal{S}$ and $\mathcal{S}^{-1}$ are late simulations. P is* late bisimilar *to Q (written $P \approx Q$) if there exists a late bisimulation $\mathcal{S}$ such that $P\,\mathcal{S}\,Q$.*

We now define a late bisimulation over our transition system. There is no difference with the standard definition as far as output and silent transitions are concerned. Even in the case of input, we need no change (although the operational rule In calls for a rooted substitution). Indeed, the argument of the input is only a placeholder that must be instantiated with any name $w \in \mathcal{N}$, as done in Def. 7.1. We only need some care when generating new names through extrusions. In this case, too, we replace the object of the bound output $\vartheta\bullet\vartheta' n$ with a new name $b \in \mathcal{N}$ within the two processes.

The derivatives obtained in this way are then checked for bisimilarity. More formally, we have the following definition.

Definition 7.2 (name late bisimulation) *A binary relation $\mathcal{R}$ on extended processes is a* name late simulation *if $t_0 \,\mathcal{R}\, t_1$ implies that*

- *If $t_0 \stackrel{\mu}{\longmapsto} t'_0$ and μ is τ, $\overline{x}y$, then for some t'_1, $t_1 \stackrel{\mu}{\longmapsto} t'_1$ and $t'_0 \,\mathcal{R}\, t'_1$*
- *If $t_0 \stackrel{x(r)}{\longmapsto} t'_0$ then for some t'_1, $t_1 \stackrel{x(s)}{\longmapsto} t'_1$ and for all $w \in \mathcal{N}$, $t'_0\{\!|w/r|\!\}\mathcal{R}t'_1\{\!|w/s|\!\}$*
- *If $t_0 \stackrel{\overline{x}(r)}{\longmapsto} t'_0$ then for some t'_1, $t_1 \stackrel{\overline{x}(s)}{\longmapsto} t'_1$ and for a name $b \in \mathcal{N}$, $b \notin names(t_0, t_1)$, $t'_0\{\!|b/r|\!\}\mathcal{R}t'_1\{\!|b/s|\!\}$.*

The relation $\mathcal{R}$ is a name late bisimulation *if both $\mathcal{R}$ and $\mathcal{R}^{-1}$ are name late simulations. t_0 is* name late bisimilar *to t_1 (written $t_0 \approx_N t_1$) if there exists a name late bisimulation $\mathcal{R}$ such that $t_0 \,\mathcal{R}\, t_1$.*

Note that there is no need for the condition $r, s \notin fn(t_0, t_1)$ in the last two items, because both rules *In* and *Open* generate fresh localized names. We now prove that the name late equivalence, induced by the bisimulation above, coincides with the standard late equivalence.

Theorem 7.3

$$P \approx P' \text{ if and only if } t \approx_N t',$$

with $FC(t) =_\alpha P$ and $FC(t') =_\alpha P'$.

Proof. *If*) By the second claim of Theorem 6.2, there exists a bijection, say ϕ, between the state space of P and t. Similarly, let ϕ' be the bijection between the state space of P' and t'. Given a late bisimulation $\mathcal{R}$ containing $\langle P, P'\rangle$, let

$$\mathcal{R}_n = \{\langle \phi(P_i), \phi'(P'_i)\rangle \mid \langle P_i, P'_i\rangle \in \mathcal{R}\}.$$

To show that $\mathcal{R}_n$ is name late bisimulation we only need to note that the condition in the third item of Def. 7.2, i.e. $b \notin names(t_0, t_1)$ amounts to taking a variant of both transitions $t_0 \stackrel{\overline{x}(r)}{\longmapsto} t'_0$ and $t_1 \stackrel{\overline{x}(s)}{\longmapsto} t'_1$. This is obvious because b is fresh in both processes under consideration.

Only if) Let $\mathcal{R}_n$ be a localized late bisimulation containing $\langle t, t'\rangle$. Then,

$$\mathcal{R} = \{\langle FC(t_i), FC(t'_i)\rangle \mid \langle t_i, t'_i\rangle \in \mathcal{R}_n\}$$

is a late bisimulation by the second claim of Theorem 6.2 and because α-convertible processes are congruent. The correspondence in the case of the *Open* transitions relies on the same argument used in the *If* part. $\square$

$$
\begin{array}{cc}
t_0 = 1 \Rightarrow\ (\nu\, a)(\nu\, c)\overline{x}a.[\overline{c}b \mid c(z).a(y).\overline{y}z] & t'_0 = 5 \Rightarrow\ (\nu\, a)\overline{x}a.\tau.a(y).\overline{y}b \\
\big\downarrow \scriptstyle{\overline{x}(\bullet 1)} & \big\downarrow \scriptstyle{\overline{x}(\bullet 5)} \\
t_1 = 2 \Rightarrow\ (\nu\, c)[\overline{c}b \mid c(z).||_1 \bullet 1(y).\overline{y}z] & t'_1 = 6 \Rightarrow\ \tau.\bullet 5(y).\overline{y}b \\
\big\downarrow \scriptstyle{\tau} & \big\downarrow \scriptstyle{\tau} \\
t_2 = (\nu\, c)[2 \Rightarrow\ \mathbf{0} \mid 3 \Rightarrow\ ||_1 \bullet 1(y).\overline{y}b] & t'_2 = 6 \Rightarrow\ \bullet 5(y).\overline{y}b \\
\big\downarrow \scriptstyle{\bullet 1(\bullet 3)} & \big\downarrow \scriptstyle{\bullet 5(\bullet 6)} \\
t_3 = (\nu\, c)[2 \Rightarrow\ \mathbf{0} \mid 4 \Rightarrow\ \bullet\overline{3}b] & t'_3 = 7 \Rightarrow\ \bullet\overline{6}b \\
\big\downarrow \scriptstyle{\bullet\overline{3}b} & \big\downarrow \scriptstyle{\bullet\overline{6}b} \\
t_4 = (\nu\, c)[2 \Rightarrow\ \mathbf{0} \mid 4 \Rightarrow\ \mathbf{0}] & t'_4 = 7 \Rightarrow\ \mathbf{0}
\end{array}
$$

Figure 8: The computations of t_0 and t'_0.

We end this section with an example. Consider the extended processes

$$t_0 = 1 \Rightarrow\ (\nu\, a)(\nu\, c)\ \overline{x}a.(\overline{c}b \mid c(z).a(y).\overline{y}z) \text{ and } t'_0 = 5 \Rightarrow\ (\nu\, a)\ \overline{x}a.\tau.a(y)\overline{y}b$$

which are clearly late bisimilar once counters have been erased. The computations of the two processes are depicted in Fig. 8.

To show that t_0 can simulate t'_0 we build a simulation $t_0 \mathcal{R} t'_0$. We choose $d \notin names(t_0, t'_0)$, and we show that $t_1\{d/\bullet 1\}\mathcal{R}t'_1\{d/\bullet 5\}$. The processes after the bound outputs are $2 \Rightarrow\ (\nu\, c)(\overline{c}b \mid c(z).d(y).\overline{y}z)$ and $6 \Rightarrow\ \tau.d(y).\overline{y}b$ and both perform a τ. Hence, we prove that $t_2 \mathcal{R} t'_2$. This holds if $\forall w \in \mathcal{N}, t_3\{w/\bullet 3\}\mathcal{R}t'_3\{w/\bullet 6\}$, i.e. if $(\nu\, c)(2 \Rightarrow\ \mathbf{0} \mid 4 \Rightarrow\ \overline{w}b)\ \mathcal{R}\ 7 \Rightarrow\ \overline{w}b$. Now the counters are immaterial and the simulation is obvious. The relation $\mathcal{R}^{-1}$ is built likewise.

8 Higher-Order Mobile Processes

Some higher-order calculi have recently been defined for modelling mobile systems. A communication may cause processes to migrate. Examples of such calculi are the higher-order π-calculus *HOπ* [25], *Plain CHOCS* [27], *Facile* [14], and *CML* [24, 19]. We show the robustness of our approach to the semantics of mobile processes, by taking a late version of the *HOπ*-calculus as a test-bed.

When a process P migrates, it exports all its names at the same time. We need to extend the notions on names introduced in Section 5 to processes, considered as name containers. For the sake of brevity, we will not give the definitions in detail, but only show what has to be added or changed when the object of an action is a process.

As far as syntax is concerned, we only add process variables, ranged over by K, and we assume that $K \in \mathcal{N}$. Also, we enclose in angled brackets a process object of an output, e.g. $\overline{a}\langle \overline{b}x.(c(z).\overline{z}z|\overline{d}x)\rangle$.

We re-use our example in Section 3 to help intuition. Assume that a student asks a teacher for the date of an exam. Suppose also that there is a clash of dates (this teacher is often quite absent minded), and the student asks if the old date od can be changed to the new one nd. The teacher sends a process Update(od,nd) which performs the change to be run by P_3. This change of date is modelled by defining processes P_0 and P_3 as follows, where u_chan is a global channel.

$$P_0 = \text{Teacher} + \overline{u\_chan}\langle \text{Update}(\text{did.teach}_\bullet\text{adm.did.exams}\, od, \text{teach}_\bullet\text{stud}\, nd)\rangle.P_0,$$

$$P_3 = u\_chan(K).K.P_3 + \text{Other\_exams}.$$

After the interaction between P_0 and P_3, the placeholder K in P_3 is instantiated to the process Update, and P_3 becomes

$$\text{Update}({}_\bullet\, od, \text{adm.did.exams}_\bullet\text{did.stud}\, nd).P_3.$$

As for the semantic definitions, we first lift the address composition of Def. 4.3 to the more powerful process address composition, $\star^H$. It re-locates all the names of a process, distributing the address composition for names onto the communicated process P. In other words $\star^H$ composes the address of the sender of P relative to its receiver with the addresses of all the names contained in P. It is inductively defined on the syntax of processes and has localized names as base case, to which $\star$ applies. The only item of the following definition that deserves a comment is the fifth one. Suppose that a process $P = (P_0|P_1)$ has to be exported to some process Q, and that its address relative to Q is $\vartheta_\bullet\vartheta'$. When $(P_0|P_1)$ replaces the placeholders in Q, carrying all its names, the address of the *old* instance of P_i in P, relative to the *new* instance of P_i in Q is $\vartheta||_i{}_\bullet\vartheta'||_i$, for $i = 0, 1$. In the last item below, Y_i stands for either a name $r \in \mathcal{N}'$ or a process, possibly with localized names.

Definition 8.1 *Process address composition* $\star^H$ *is defined by*

- $\vartheta_\bullet\vartheta' \star^H r = \vartheta_\bullet\vartheta' \star r$
- $\vartheta_\bullet\vartheta' \star^H \mathbf{0} = \mathbf{0}$
- $\vartheta_\bullet\vartheta' \star^H \pi.P = \begin{cases} \vartheta_\bullet\vartheta' \star sbj(\pi) \;\; \vartheta_\bullet\vartheta' \star^H obj(\pi).\; \vartheta_\bullet\vartheta' \star^H P & \textit{if } \pi \neq \tau \\ \tau.\; \vartheta_\bullet\vartheta' \star^H P & \textit{otherwise} \end{cases}$
- $\vartheta_\bullet\vartheta' \star^H (P_0 + P_1) = \vartheta_\bullet\vartheta' \star^H P_0 \; + \; \vartheta_\bullet\vartheta' \star^H P_1$
- $\vartheta_\bullet\vartheta' \star^H (P_0|P_1) = \vartheta||_0{}_\bullet\vartheta'||_0 \star^H P_0 \;\;|\;\; \vartheta||_1{}_\bullet\vartheta'||_1 \star^H P_1$
- $\vartheta_\bullet\vartheta' \star^H (\nu\, s)P = (\nu\; \vartheta_\bullet\vartheta' \star s)\; \vartheta_\bullet\vartheta' \star^H P$
- $\vartheta_\bullet\vartheta' \star^H [r = s]P = [\vartheta_\bullet\vartheta' \star r = \vartheta_\bullet\vartheta' \star s]\; \vartheta_\bullet\vartheta' \star^H P$
- $\vartheta_\bullet\vartheta' \star^H A(Y_1, \ldots, Y_k) = A(\vartheta_\bullet\vartheta' \star^H Y_1, \ldots, \vartheta_\bullet\vartheta' \star^H Y_1)$

We apply below the process address composition to a simple example, by reporting only the relevant steps.

$$||_1\bullet||_0 \star^H (||_0\bullet 1(K).b(a).K \;\mid\; \overline{||_1\bullet 2}\;\langle(\nu\, a)\overline{a}\;\; ||_1\bullet 1\rangle) =$$

$$||_1||_0\bullet||_0||_0 \star^H (||_0\bullet 1(K).b(a).K) \;\mid\; ||_1||_1\bullet||_0||_1 \star^H (\overline{||_1\bullet 2}\;\langle(\nu\, a)\overline{a}\;\; ||_1\bullet 1\rangle) =$$

$$(||_1||_0\bullet||_0||_0 \star ||_0\bullet 1)(K).b(a).K \;\mid\; ||_1||_1\bullet||_0||_1 \star \overline{||_1\bullet 2}\;(||_1||_1\bullet||_0||_1 \star^H \langle(\nu\, a)\overline{a}\;\; ||_1\bullet 1\rangle) =$$

$$||_1||_0\bullet||_0 1(K).b(a).K \;\mid\; \overline{||_1||_1\bullet||_0 2}\;\langle(\nu\, a)\overline{a}\;\; ||_1||_1\bullet||_0||_1 \star ||_1\bullet 1\rangle =$$

$$||_1||_0\bullet||_0 1(K).b(a).K \;\mid\; \overline{||_1||_1\bullet||_0 2}\;\langle(\nu\, a)\overline{a}\;\; ||_1||_1\bullet||_0 1\rangle$$

$$\begin{array}{c}
1\Rightarrow (d(a).d(e).\overline{c}\langle a(K).b(a).K \mid \overline{e}a\rangle \mid c(K').K') \\
\big\downarrow{\scriptstyle d(\bullet 1)@||_0} \\
2\Rightarrow d(e).\overline{c}\langle ||_0\bullet 1(K).b(a).K \mid \overline{e}\; ||_1\bullet 1\rangle \mid 1\Rightarrow c(K').K' \\
\big\downarrow{\scriptstyle d(\bullet 2)@||_0} \\
3\Rightarrow \overline{c}\langle ||_0\bullet 1(K).b(a).K \mid \overline{||_1\bullet 2}\; ||_1\bullet 1\rangle \mid 1\Rightarrow c(K').K' \\
\big\downarrow{\scriptstyle \tau} \\
3\Rightarrow \mathbf{0} \mid 2\Rightarrow (||_1||_0\bullet||_0 1(K).b(a).K \mid \overline{||_1||_1\bullet||_0 2}\;\; ||_1||_1\bullet||_0 1)
\end{array}$$

Figure 9: A computation of $1\Rightarrow (d(a).d(e).\overline{c}\langle a(K).b(a).K \mid \overline{e}a\rangle \mid c(K').K')$.

It is straightforward to check that the process address composition homomorphically extends the address composition of names. Thus we can safely extend the results given for names to processes. In particular $\star^H$ is always defined, whenever used in the operational rules.

As sketched above, sending a process corresponds to sending all its names, using the process address composition.

Definition 8.2 *A process P exported to the relative address $\vartheta\bullet\vartheta'$ is $\vartheta\bullet\vartheta' \star^H P$.*

Similarly to what we did for $\star^H$, we enrich routed substitution to deal with processes, and we denote it by $\{\!| - / - |\!\}^H$. Its definition is exactly the same as Def. 5.5, except that names may now denote processes. For example, item 3 becomes

$$(t_0|t_1)\{\!|K'/K|\!\}^H = t_0\{\!|K'_0/K_0|\!\}^H|t_1\{\!|K'_1/K_1|\!\}^H,\; K'_i = ||_i\bullet \star^H K',\; K_i = ||_i\bullet \star^H K.$$

The definition of $\{\!| - / - |\!\}^H_{@\vartheta}$ is straightforward.

At this point it is easy to define a late transition system of the $HO\pi$-calculus with localized names (see [25] for the details on the standard operational semantics and [12] for its proved version). It is obtained from the one in Tab. 2, by replacing substitutions and address composition $\star$ with their higher-order versions. We briefly discuss two rules only, where we consider the relevant names to denote processes, in the case of input. Table 3 shows instances of the $HO\text{-}Com_0$ and $HO\text{-}In$ rules. The rule *HO-In* still has a $\bullet n$ which, in this case, acts as a place-holder of the process to be received. As usual, $HO\text{-}Com_0$ checks whether the channel is the same and then replaces the placeholder $\bullet m$ with the process K_1, i.e. K_0 updated as needed, within the residual of the process which fired the input. In fact, all the names in process K_0 are correctly updated by routed substitution, via $\star^H$.

Figure 9 shows an example of computation.

$$HO\text{-}In : n\Rightarrow r(K).P \xrightarrow{r(\bullet n)} (n+1)\Rightarrow P\{\!|\bullet n/K|\!\}^H$$

$$HO\text{-}Com_0 : \frac{t_0 \xrightarrow{\overline{r_0}K_0 @\vartheta_0} t_0', t_1 \xrightarrow{r_1(\bullet m) @\vartheta_1} t_1'}{t_0|t_1 \xrightarrow{\tau} t_0'|t_1'\{\!|K_1/\bullet m|\!\}^H_{@\vartheta_1}}, \begin{cases} K_1 = ||_1\vartheta_1\bullet||_0\vartheta_0 \star^H K_0 \\ r_1 = ||_1\vartheta_1\bullet||_0\vartheta_0 \star r_0 \end{cases}$$

Table 3: Two rules for the higher-order case.

9 Conclusions

We have presented a late *SOS* semantics of the π-calculus which specifies local environments of names. Our results can be carried over the early version, by adapting the ideas of [9]. Our proposal may help to design efficient implementations for concurrent languages. In fact, a unique, global manager of names avoids clashes, but it may result in a bottleneck, as all the components of a system must communicate with this centralized authority. Instead, in our setting the management of local environments is made locally. The correct exportation of localized names is made by an operator which composes them. Its characterization, which is factorized out of the semantic descriptions, is crucial in our work. In addition, we give a global view of names, which are handled on a local basis. Our proposal extends the original semantics of the π-calculus [21], in that it gives a more detailed description of system evolution, but it preserves the equivalence classes, induced by bisimulation. Furthermore, we show that our proposal can easily be lifted to the higher-order π-calculus. All this helps to show that our approach can scale up to a real programming language like *Facile*, which has also a proved operational semantics [5].

Our description of the name manager uses a family of operators $n\Rightarrow$ which extends the syntax of the language. These operators are quite similar to those used to define the causal [10] and locational semantics of CCS [6] and of the π-calculus [26, 4]. We can thus use the technique presented in [4] to encode our name handler into the pure π-calculus.

The relative addresses that we use to handle names locally are defined according to the abstract syntax of systems considering the parallel composition as the main operator. Here we only consider binary trees as the logical architecture, and argue that the extension to n-adic trees is immediate. Of course, taking an n-adic parallel operator affects the *SOS* rules. Only slight and technical adjustments are needed to Tab. 2, once the rules involving the new parallel operator have replaced the corresponding rules in Tab. 1. Also, there is no difficulty in reflecting a different addressing mechanism when a more basic description of the topology of the network is available, as well as the actual placement of processes on processors. It suffices to take the space of relative addresses, quotient the relation induced by the topology and by the allocation. This relation will consider the logical access paths to the processes running on the same processor to be equivalent. Then, the address composition can be left unchanged and the quotient made after the needed calculations. This suggests a way to define a hierarchy of descriptions, which is increasingly nearer the actual implementation. Another possible benefit of our approach concerns security in mobile agent environments. Relative addresses can keep track of the origin of mobile agents in multi-hop travel on the network, thus they may help in the authentication of mobile code.

For those interested in studying truly concurrent aspects of mobile systems, a distributed specification of local environments like ours can be of interest. Also, to describe distribution of control, one can exploit the techniques presented in [12], where some non-interleaving semantics are given to the π-calculus. The two approaches can easily be coupled, because both rely on proved transition systems, resulting in a description oriented towards distribution of both data and control.

Admittedly, our localized names are sometimes long-wired and difficult to read. However, they should be used as *internal* names for specifications nearer to implementations, rather than in high-level specifications, where a global space of names and global checks are quite acceptable. They are thus likely to be handled by mechanical tools that help system designers.

Acknowledgment The authors thank the referees for their helpful and precise remarks and comments, in particular for the suggestion concerning the possible use of our technique in the security setting.

References

[1] G.A. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, 1988.

[2] R.M. Amadio. An asynchronous model of locality, failure, and process mobility. In *Proceedings of COORDINATION'97, LNCS 1282*, 1997.

[3] J.S. Ambler. A de Bruijn notation for the π-calculus. Technical Report 569, Dept. of Computer Science, Queen Mary & Westfield College, London, 1991.

[4] M. Boreale and D. Sangiorgi. A fully abstract semantics of causality in the π-calculus. In *Proceedings of STACS'95, LNCS.* Springer Verlag, 1995.

[5] R. Borgia, P. Degano, C. Priami, L Leth, and B. Thomsen. Understanding mobile agents via a non-interleaving semantics for Facile. In R. Cousot and D.A. Schmidt, editors, *Proceedings of SAS'96*, LNCS 1145, pages 98–112. Springer-Verlag, 1996.

[6] G. Boudol, I. Castellani, M. Hennessy, and A. Kiehn. A theory of processes with localities. *Theoretical Computer Science*, 114, 1993.

[7] L. Cardelli. A language with distributed scope. *Computing Systems*, 8(1):27–59, 1995.

[8] L. Cardelli and A.D. Gordon. Mobile ambients. *Foundations of Software Science and Computational Structures, Maurice Nivat (Ed.), LNCS 1378*, pages 140–155, 1998.

[9] R. Cleaveland and D. Yankelevich. An operational framework for value-passing processes. In *Proceedings of POPL'94*, pages 326–338, 1994.

[10] Ph. Darondeau and P. Degano. Causal trees. In *Proceedings of ICALP'89, LNCS 372*, pages 234–248. Springer-Verlag, 1989.

[11] P. Degano and C. Priami. Proved trees. In *Proceedings of ICALP'92, LNCS 623*, pages 629–640. Springer-Verlag, 1992.

[12] P. Degano and C. Priami. Non interleaving semantics for mobile processes. Extended abstract. In *Proceedings of ICALP'95, LNCS 944*, pages 660–671. Springer-Verlag, 1995. To appear in TCS.

[13] C. Fournet and Gonthier. The reflexive chemical abstract machine and the join calculus. In *Proceedings of 23rd ACM Symposium on Principles of Programming Languages*, 1996.

[14] A. Giacalone, P. Mishra, and S. Prasad. Operational and algebraic semantics for Facile: A symmetric integration of concurrent and functional programming. In *Proceedings ICALP'90, LNCS 443*, pages 765–780. Springer-Verlag, 1990.

[15] D. Gilbert and C. Palamidessi. A calculus of process migration. Technical report, 1997.

[16] K. Honda and M. Tokoro. A small calculus for concurrent objects. *OOPS Messenger*, 2(2):50–54, 1991.

[17] S. Mac Lane. *Categories for the Working Mathematician*. Springer-Verlag, 1971.

[18] R. Milner. The polyadic π-calculus: a tutorial. Technical Report ECS-LFCS-91-180, University of Edinburgh, 1991.

[19] R. Milner, D. Berry, and D. Turner. A semantics for ML concurrency primitives. In *Proceedings of POPL'92*, 1992.

[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] R. Milner, J. Parrow, and D. Walker. Modal logics for mobile processes. *Theoretical Computer Science*, 114:149–171, 1993.

[22] F. Orava and J. Parrow. An algebraic verification of a mobile network. *Formal Aspects of Computing*, pages 497–543, 1992.

[23] T.W. Pratt. *Programming Languages: Design and Implementation*. Prentice Hall, 1984.

[24] J. Reppy. *Higher order concurrency*. PhD thesis, Cornell University, TR 92-1285, 1992.

[25] D. Sangiorgi. *Expressing Mobility in Process Algebras: First-Order and Higher-Order Paradigms*. PhD thesis, University of Edinburgh, 1992.

[26] D. Sangiorgi. Locality and non-interleaving semantics in calculi for mobile processes. In *Proceedings of TACS'94, LNCS 789*. Springer-Verlag, 1994.

[27] B. Thomsen. Plain CHOCS: a second generation calculus for higher order processes. *Acta Informatica*, 30(1):1–59, 1993.

[28] B. Thomsen, L. Leth, S. Prasad, T.-M. Kuo, A. Kramer, F. Knabe, and A. Giacalone. Facile Antigua Release Programming Guide. Technical Report ECRC-93-20, European Computer-Industry Research Centre, 1993.

[29] D. Walker. Objects in the π-calculus. *Information and Computation*, 116(2):253–271, 1995.