Recursion vs Replication in Process Calculi: Expressiveness

Catuscia Palamidessi, Frank D. Valencia

Recursion vs Replication in Process Calculi: Expressiveness*

Catuscia Palamidessi
INRIA Futurs and LIX, École Polytechnique
catuscia@lix.polytechnique.fr

Frank D. Valencia
CNRS and LIX, École Polytechnique
fvalenci@lix.polytechnnique.fr

1 Introduction

Process calculi such as CCS [12], the π-calculus [14] and Ambients [6] are among the most influential formal methods for modelling and analyzing the behaviour of concurrent systems; i.e. systems consisting of multiple computing agents, usually called *processes*, that interact with each other. A common feature of these calculi is that they treat processes much like the λ-calculus treats computable functions. They provide a language in which the structure of *terms* represents the structure of processes together with an *operational semantics* to represent computational steps. Another common feature, also in the spirit of the λ-calculus, is that they pay special attention to economy. That is, there are few process constructors, each one with a distinct and fundamental role.

For example, a typical process term is the *parallel composition* $P \mid Q$, which is built from the terms P and Q with the constructor $\mid$ and it represents the process that results from the parallel execution of the processes P and Q. Another typical term is the *restriction* $(\nu x)P$ which represents a process P with a private resource x—e.g., a location, a link, or a name. An operational semantics may dictate that if P can reduce to (or evolve into) P', written $P \longrightarrow P'$, then we can also have the reductions $P \mid Q \longrightarrow P' \mid Q$ and $(\nu x)P \longrightarrow (\nu x)P'$.

Infinite behaviour is ubiquitous in concurrent systems (e.g., browsers, search engines, reservation systems). Hence, it ought to be represented by process terms.

*Supported by the Project Rossignol of the ACI Sécurité Informatique (Ministère de la recherche et nouvelles technologies)

Two standard term representations of them are *recursive process expressions* and *replication*.

Recursive process expressions are reminiscent of the recursive expressions used in other areas of computer science, such as for example Functional Programming. They may come in the form $\mu X.P$ where P may have occurrences of X. The process $\mu X.P$ behaves as P with the (free) occurrences of X replaced by $\mu X.P$. Another presentation of recursion is by using *parametric processes* of the form $A(y_1, \ldots, y_n)$ each assumed to have a unique, possibly recursive, *definition* $A(x_1, \ldots, x_n) \stackrel{\texttt{def}}{=} P$ where the x_i's are pairwise distinct, and the intuition is that $A(y_1, \ldots, y_n)$ behaves as its P with each y_i replacing x_i.

Replication, syntactically simpler than recursion, takes the form $!P$ and it is reminiscent of Girard's bang operator; an operator used to express unlimited number of copies of a given resource in linear-logic [8]. Intuitively, $!P$ means $P \mid P \mid \cdots$; an unbounded number of copies of the process P.

Now, it is not uncommon that a given process calculus, originally presented with one form of defining infinite behavior, is later presented with the other. For example, the π-calculus was originally presented with recursive expressions and later with replication [16]. The Ambient calculus was originally presented with replication and later with recursion [11]. This is reasonable as a variant may simplify the presentation of the calculus or be tailored to specific applications.

From the above intuitive description it should be easy to see that $\mu X.(P \mid X)$ expresses the unbounded parallel behaviour of $!P$. It is less clear, however, whether replication can be used to express the unbounded behaviour of $\mu X.P$. In particular, processes that allows for unboundedly many *nested* restrictions as, for example, in $\mu X.(\nu x)(P \mid X)$ which behaves as $(\nu x)(P \mid (\nu x)(P \mid (\nu x)(P \mid \cdots)))$. In fact, the ability of expressing recursive behaviours via replication depends on the particular process calculus under consideration.

The above discussion raises the issue of *expressiveness*. What does it mean for one variant to be as expressive as another ? The answer to this question is definite in the realm of computability theory via the notion of language equivalence. In concurrency theory, however, this issue is not quite settled.

One approach to comparing expressiveness of two given process calculus variants is by comparing them w.r.t. some standard process equivalence, say $\sim$. If for every process P in one variant there is a Q in the other variant such that $Q \sim P$ then we say that the latter variant is at least as expressive as the former.

Another approach consists in telling two variants apart by showing that in one variant one can solve some fundamental problem (e.g., leader election) while in the other one cannot. It should be noticed that, unlike computability theory, the capability of two variants of simulating Turing Machines does not imply equality in their expressiveness. For example, [18] shows that under some reasonable as-

sumptions the asynchronous version of the π-calculus, which can certainly encode Turing Machines, is strictly less expressive than the original calculus.

In this paper, we shall discuss the work on the relative expressiveness of Recursion and Replication in various process calculi. In particular, CCS, the π-calculus, and the Ambient calculus. We shall begin with the π-calculus, then CCS and then the Ambients calculus. For the simplicity of the presentation we shall consider the polyadic variant of the π-calculus [13]. Finally, we shall also overview the work on this subject in related calculi such as tcc [19] and calculi for Cryptographic Protocols [10].

2 The Polyadic Pi Calculus: pπ

One of the earliest discussions about the relative expressiveness between replication and recursion was in the context of the polyadic π-calculus [13]; one of the main calculi for mobility. It turns out that in this calculus replication is just as expressive as recursion. This results was rather surprising since replication seems such an elementary construct without much control power.

In what follows we shall introduce the polyadic π-calculus and the variants relevant for this paper. The various CCS and Ambients variants will be presented in the next sections as extension/restrictions of the polyadic π-calculus.

2.1 Finite Pi-calculus

Names are the most primitive entities in the π-calculus. We presuppose a countable set of (port, links or channel) *names*, ranged over by $x, y, \ldots$. For each name x, we assume a *co-name* $\overline{x}$ thought of as *complementary*, so we decree that $\overline{\overline{x}} = x$. We shall use $l, l', \ldots$ to range over names and co-names. We use $\vec{x}$ to denote a finite sequence of names $x_1 x_2 \cdots x_n$. The other entity in the π-calculus is a *process.* Process are built from names by the following syntax:

$$\begin{aligned} P, Q, \ldots &:= \sum_{i \in I} \alpha_i.P_i \;\mid\; (\nu x)P \;\mid\; P \mid Q \\ \alpha &:= \overline{x}\vec{y} \;\mid\; x(\vec{y}) \end{aligned} \tag{1}$$

where I is a finite set of indexes.

Let us recall briefly some notions as well as the intuitive behaviour of the various constructs.

The construct $\sum_{i \in I} \alpha_i.P_i$ represents a process able to perform one–but only one–of its α_i's actions and then behave as the corresponding P_i. The actions prefixing the P_i's can be of two forms: An output $\overline{x}y_1 \cdots y_n$ and an input $x(y_1 \cdots y_n)$.

In both cases x is called the *subject* and $y_1 \cdots y_n$ the *object*. The action $\overline{x}\vec{y}$ represents the capability of sending the names $\vec{y}$ on channel x. The action $x(\vec{y})$, with $\vec{y} = y_1, \cdots, y_m$ and no name occurring twice in $\vec{y}$, represents the capability of receiving the names on channel x, say $z_1 \cdots z_m$, and replacing each y_i with z_i in its corresponding continuation.

Furthermore, in $x(\vec{y}).P$ the input actions binds the names $\vec{y}$ in P. The other name binder is the *restriction* $(\nu x)P$ which declares a name x private to P, hence bound in P. Given Q we define in the standard way its *bound names* $bn(Q)$ as the set of variables with a bound occurrence in Q, and its *free names* $fn(Q)$ as the set of variables with a non-bound occurrence in Q.

Finally, the process $P \mid Q$ denotes *parallel composition*; P and Q running in parallel.

Convention 2.1. *We write the summation as* 0 *if* $|I| = 0$, *and drop the "*$\sum_{i \in I}$*" if* $|I| = 1$. *Also we write* $\pi_1.P_1 + \cdots + \pi_n.P_n$ *for* $\sum_{i \in \{1,\ldots,n\}} \pi_i.P_i$.

For simplicity, we omit "()" in processes of the form $x().P$ *as well as the ".*0*" in processes of the form* $x(\vec{y}).0$. *We use* $(\nu x_1 x_2 \cdots x_n)P$ *as an abbreviation* $(\nu x_1)(\nu x_2)\cdots(\nu x_n)P$ *and* $\prod_{i \in I} P_i$, *where* $I = \{i_1, \ldots, i_n\}$, *as an abbreviation of* $P_{i_1} \mid \cdots \mid P_{i_n}$. *Furthermore,* $P\sigma$, *where* $\sigma = \{z_1/y_1, \ldots, z_n/y_n\}$, *denotes the process that results from the substitution in* P *of each* z_i *for* y_i, *applying* α*-conversion wherever necessary to avoid captures.*

Reduction Semantics of Finite Processes. The above intuition about process behaviour is made precise by the rules in Table 1. The *reduction* relation $\longrightarrow$ is the least binary relation on processes satisfying the rules in Table 1. The rules are easily seen to realize the above intuition.

We shall use $\longrightarrow^*$ to denote the reflexive, transitive closure of $\longrightarrow$. A reduction $P \longrightarrow Q$ basically says that P can evolve, after some communication between its subprocesses, into Q. The reductions are quotiented by the *structural congruence* relation $\equiv$ which postulates some basic process equivalences.

Definition 2.2 (Structural Congruence)**.** *Let* $\equiv$ *be the smallest congruence over processes satisfying the following axioms:*

1. $P \equiv Q$ *if* P *and* Q *differ only by a change of bound names (*α*-equivalence).*
2. $P \mid 0 \equiv P$, $P \mid Q \equiv Q \mid P$, $P \mid (Q \mid R) \equiv (P \mid Q) \mid R$.
3. *If* $x \notin fn(P)$ *then* $(\nu x)(P \mid Q) \equiv P \mid (\nu x)Q$.
4. $(\nu x)0 \equiv 0$, $(\nu x)(\nu y)P \equiv (\nu y)(\nu x)P$.

$$\text{REACT:}\ \frac{}{(\cdots + \overline{x}\, z_1 \cdots z_n.P) \ \mid\ (\cdots + x(y_1 \cdots y_n).Q) \ \longrightarrow\ P \ \mid\ Q\{z_1/y_1, \ldots, z_n/y_n\}}$$

$$\text{PAR:}\ \frac{P \longrightarrow P'}{P \mid Q \longrightarrow P' \mid Q} \qquad \text{RES:}\ \frac{P \longrightarrow P'}{(\nu x)P \longrightarrow (\nu x)P'}$$

$$\text{STRUCT:}\ \frac{P \equiv P' \longrightarrow Q' \equiv Q}{P \longrightarrow Q}$$

Table 1: Reductions Rules.

2.2 Infinite Processes in the Polyadic Pi-Calculus

In the literature there are at least two alternatives to extend the above syntax to express infinite behavior. We describe them next.

Pi with Parametric Recursive Definitions: $\mathbf{p}\pi_{\mathtt{D}}$

A typical way of specifying infinite behavior is by using parametric recursive definitions [14]. In this case we extend the syntax of finite processes (Equation 1) as follows:

$$P, Q, \ldots := \ldots \mid A(y_1, \ldots, y_n) \tag{2}$$

Here $A(y_1, \ldots, y_n)$ is an *identifier* (also *call*, or *invocation*) of arity n. We assume that every such an identifier has a unique, possibly recursive, *definition* $A(x_1, \ldots, x_n) \stackrel{\mathtt{def}}{=} P$ where the x_i's are pairwise distinct, and the intuition is that $A(y_1, \ldots, y_n)$ behaves as its P with each y_i replacing x_i. We shall presuppose finitely many such definitions. Furthermore, for each $A(x_1, \ldots, x_n) \stackrel{\mathtt{def}}{=} P$ we require

$$fn(P) \subseteq \{x_1, \ldots, x_n\}. \tag{3}$$

The reduction semantics of the extended processes is obtained simply by extending the structural congruence $\equiv$ in Definition 2.2 with the following axiom:

$$A(y_1, \ldots, y_n) \ \equiv\ P[y_1, \ldots, y_n/x_1, \ldots, x_n] \text{ if } A(x_1, \ldots, x_n) \stackrel{\mathtt{def}}{=} P. \tag{4}$$

As usual $P[y_1 \ldots y_n/x_1 \ldots x_n]$ results from syntactically replacing every free occurrence of x_i with y_i and by applying *name α-conversion*, wherever needed to avoid capture.

We shall use $\mathrm{p}\pi_{\mathrm{D}}$ to denote the polyadic π-calculus with parametric recursive definitions with the above syntactic restrictions.

Pi with Replication: $\mathrm{p}\pi_{!}$

A simple way of expressing infinite behaviour in the π-calculus is by using replication. We shall use $\mathrm{p}\pi_{!}$ to denote the polyadic π-calculus with replication.

In the $\mathrm{p}\pi_{!}$ case, the syntax of finite processes (Equation 1) is extended as follows:

$$P, Q, \ldots := \ldots \mid !P. \tag{5}$$

Intuitively $!P$ behaves as $P \mid P \mid \ldots \mid P \mid !P$; unboundedly many copies of P.

The reduction semantics for $\mathrm{p}\pi_{!}$ is obtained simply by extending the structural congruence $\equiv$ in Definition 2.2 with the following axiom:

$$!P \equiv P \mid !P. \tag{6}$$

Barbed Bisimilarity

We shall often state expressiveness results by claiming the existence of a process in one calculus which is equivalent to some given process in another calculus. For this purpose, here we recall a standard way of comparing processes. We shall use $\mathrm{p}\pi_{!}$ to denote the calculus with replication.

Let us begin by recalling a basic notion of observation for the π-calculus. Intuitively, given $l = x$ ($l = \overline{x}$) we say that (the barb) l can be *observed* at P, written $P \downarrow_l$, iff P can have an input (output) with subject x. Formally,

Definition 2.3 (Barbs)**.** *Define $P \downarrow_{\overline{x}}$ iff $\exists \vec{z}, \vec{y}, R : P \equiv (\nu \vec{z})(\overline{x}\vec{y}.Q \mid R)$ and x is not in $\vec{z}$. Similarly, $P \downarrow_x$ iff $\exists \vec{z}, \vec{y}, Q, R : P \equiv (\nu \vec{z})(x(\vec{y}).Q \mid R)$ and x is not in $\vec{z}$. Furthermore, $P \Downarrow_l$ iff $\exists Q : P \longrightarrow^* Q \downarrow_l$.*

Let us now recall the notion of barbed (weak) bisimilarity and congruence. Remember that a process *context* C in a given calculus is an expression with a hole [.] such that placing a process in the hole produces a well-formed process term in the calculus.

For technical purposes, we shall use $\mathrm{p}\pi_{\mathrm{D}+!}$ as the calculus whose process syntax arises from extending the syntax of finite processes (Equation 1) with both replication and recursive definitions. The reduction semantics of $\mathrm{p}\pi_{\mathrm{D}+!}$ of the extended processes is obtained by extending the structural congruence $\equiv$ in Definition 2.2 with the axioms in Equations 4 and 6.

Definition 2.4 (Barbed Bisimilarity)**.** *A* (weak) barbed-simulation *is a binary relation $\mathcal{R}$ satisfying the following:* $(P, Q) \in \mathcal{R}$ *implies that:*

1. *if* $P \longrightarrow P'$ *then* $\exists Q' : Q \longrightarrow^* Q' \wedge (P', Q') \in \mathcal{R}$.
2. *if* $P \downarrow_l$ *then* $Q \Downarrow_l$.

The relation $\mathcal{R}$ is a barbed bisimulation *iff both $\mathcal{R}$ and its converse $\mathcal{R}^{-1}$ are barbed -simulations. We say that P and Q are* (weak) barbed bisimilar *iff* $(P, Q) \in \mathcal{R}$ *for some barbed bisimulation $\mathcal{R}$. Furthermore, we say that P and Q are barbed congruent, written* $P \approx Q$*, iff for each context* $C[\cdot]$ *in* $\mathrm{p}\pi_{\mathrm{D}+!}$, $C[P] \sim C[Q]$.

2.3 Recursive Definitions vs Replication in Pi

Here we recall a result stating that the variants $\mathrm{p}\pi_!$ and $\mathrm{p}\pi_{\mathrm{D}}$ can be regarded as being equally expressive w.r.t (weak) barbed congruence $\approx$ given in Definition 2.4. More precisely, the expressiveness criteria w.r.t to barbed congruence we shall use in this section can be stated as follows.

Criteria 2.5. *We say that a π-calculus variant is as expressive as another iff for every process P in the second variant one can construct a process $[\![P]\!]$ in the first variant such that $[\![P]\!]$ is (weakly) barbed congruent to P.*

All the results presented in this section are consequences of the expressiveness results in [20].

From $\mathrm{p}\pi_{\mathrm{D}}$ to $\mathrm{p}\pi_!$ and back: Encodings

We shall now provide encodings from one variant into the other and state their correctness. We shall say that a map $[\![]\!]$ is a *homomorphism for parallel composition* iff $[\![P \mid Q]\!] = [\![P]\!] \mid [\![Q]\!]$. The notion of homomorphism for the other operators is defined analogously.

Definition 2.6. *Let $[\![\cdot]\!]_0$ be the map from $\mathrm{p}\pi_{\mathrm{D}}$ processes and recursive definitions into $\mathrm{p}\pi_!$ processes given by:*

$$\begin{array}{lcl} [\![0]\!]_0 & = & 0, \\ [\![A_i(\vec{x_i}) \stackrel{\mathtt{def}}{=} P_i]\!]_0 & = & !\, a_i(\vec{x_i}).[\![P_i]\!]_0, \\ [\![A_i(\vec{y_i})]\!]_0 & = & \overline{a_i}\vec{y_i}, \end{array}$$

and for all other processes $[\![\cdot]\!]_0$ is a homomorphism.

Let P be an arbitrary $\mathrm{p}\pi_{\mathrm{D}}$ process with $\{ A_1(\vec{x}_i) \stackrel{\mathtt{def}}{=} P_1, \ldots, A_n(\vec{x}_n) \stackrel{\mathtt{def}}{=} P_n \}$ as the set of recursive definitions of its process identifiers. The encoding *of P, denoted $[\![P]\!]$, is defined as*

$$[\![P]\!] = (\nu a_1 \cdots a_n)([\![P]\!]_0 \mid \prod_{i\in\{1,\ldots,n\}} [\![A_i(\vec{x}_i) \stackrel{\mathtt{def}}{=} P_i]\!]_0)$$

where $a_1, \ldots, a_n \notin fn(P)$.

Intuitively, each $A(\vec{y})$, with $A(\vec{x}) \stackrel{\mathtt{def}}{=} P$, is translated into a particle $\overline{a}\vec{y}$ which excites a copy of P (with $\vec{y}$ substituted for $\vec{x}$) by interacting with a replicated resource, a provider of instances of P, of the form $!\, a(\vec{x}).[\![P]\!]$. The correctness of the encoding is stated below.

Theorem 2.7. *Let $[\![\cdot]\!]$ be the encoding in Definition 2.6. For each P in $\mathrm{p}\pi_{\mathrm{D}}$, $P \approx [\![P]\!]$.*

Let us now give an encoding of $\mathrm{p}\pi_!$ into $\mathrm{p}\pi_{\mathrm{D}}$. The idea is simple: Each $!P$ is translated into a process A_P, recursively defined as $A_P(\vec{x}) \stackrel{\mathtt{def}}{=} P \mid A_P(\vec{x})$ which can provide an unbounded number of copies of P.

Definition 2.8. *Let $[\![\cdot]\!]_0$ be the map from $\mathrm{p}\pi_!$ processes into $\mathrm{p}\pi_!$ processes given by:*

$$\begin{array}{lcl} [\![0]\!] & = & 0, \\ [\![!P]\!] & = & A_P(\vec{x}) \text{ where } A_P(\vec{x}) \stackrel{\mathtt{def}}{=} P \mid A_P(\vec{x}) \text{ and } fn(P) \subseteq \{\vec{x}\} \end{array}$$

and for all other processes $[\![\cdot]\!]_0$ is a homomorphism.

We can now state the correctness with respect to barbed congruence.

Theorem 2.9. *Let $[\![\cdot]\!]$ be the encoding in Definition 2.8. For each P in $\mathrm{p}\pi_!$, $P \approx [\![P]\!]$.*

2.4 Recursion vs Replication in the Private Pi Calculus

The Private π-calculus [20] is a sub-calculus with a restricted form of communication. The idea is that only *bound-outputs* are allowed; i.e, outputs of the form $(\nu\vec{z})\overline{x}\vec{z}.P$. Such bound-outputs are usually abbreviated as $\overline{x}(\vec{z})$ assuming that no name occur more than once in $\vec{z}$.

The above syntactic restriction results in a pleasant symmetry between input and outputs in that they both can be seen as binders. Moreover, the restriction ensures that α-conversion is the only kind of substitution required in the calculus. In fact, the rule REACT in Table 1, which applies a substitution to the continuation of the input, can be replaced by the following rule:

$$\overline{x}(\vec{z}).P \mid x(\vec{z}).P \longrightarrow (\nu\vec{z})(P \mid Q) \tag{7}$$

Let us denote by $\mathrm{Priv}\mathrm{p}\pi_!$ the calculus that results from applying to $\mathrm{p}\pi_!$ the syntactic restriction mentioned above. The $\mathrm{Priv}\mathrm{p}\pi_\mathrm{D}$ calculus is analogously defined as a restriction on $\mathrm{p}\pi_\mathrm{D}$ except that we need an extra-condition to ensure that α-conversion is the only substitution needed in the calculus: In every invocation $A(\vec{z})$, no name may occur more than once in the vector $\vec{z}$.

Now, if we wish an encoding $[\![\cdot]\!]$ from $\mathrm{Priv}\mathrm{p}\pi_!$ into $\mathrm{Priv}\mathrm{p}\pi_\mathrm{D}$ such that $[\![P]\!] \approx P$, we can simply take that of Definition 2.8 restricted to the $\mathrm{Priv}\mathrm{p}\pi_!$ case. As shown below, however, the above restriction makes impossible the existence of an encoding from $\mathrm{Priv}\mathrm{p}\pi_\mathrm{D}$ into $\mathrm{Priv}\mathrm{p}\pi_\mathrm{D}$.

Consider for example the process $P = A(z_0)$ where

$$A(x) \stackrel{\mathtt{def}}{=} \overline{x}(z).A(z).$$

The process P, in parallel with a suitable R, can perform a sequence of actions where the subject of an action is the object of the next one. This kind of sequences are called *logical threads* [20]. Moreover, P can perform the infinite logical thread $\overline{z_0}(z_1).\overline{z_1}(z_2).\ldots$.

Interestingly, as an application of the type theory for $\mathrm{Priv}\mathrm{p}\pi_!$, the results in [20] state that *no process in* $\mathrm{Priv}\mathrm{p}\pi_!$ *can exhibit an infinite logical thread*. Together with P above, this property of $\mathrm{Priv}\mathrm{p}\pi_!$ can be used to prove the following result.

Theorem 2.10. *There is a process P in* $\mathrm{Priv}\mathrm{p}\pi_\mathrm{D}$ *such that $P \not\approx Q$ for every Q in* $\mathrm{Priv}\mathrm{p}\pi_!$.

Therefore, we cannot have an expressiveness result of the kind we have for $\mathrm{p}\pi_\mathrm{D}$ and $\mathrm{p}\pi_!$ in the previous section. I.e., there is no encoding $[\![\cdot]\!]$ from $\mathrm{Priv}\mathrm{p}\pi_\mathrm{D}$ processes into $\mathrm{Priv}\mathrm{p}\pi_!$ processes such that $[\![P]\!] \approx P$.

3 The Calculus of Communicating Systems (CCS)

Undoubtedly CCS [12], a calculus for synchronous communication, remains as a standard representative of process calculi. In fact, many foundational ideas in the theory of concurrency have sprung from this calculus. In the following we shall consider some variants of CCS without relabelling operations.

3.1 Finite CCS

The finite CCS processes can be obtained as a restriction of the finite processes of the Polyadic π-calculus by requiring all inputs and outputs to have empty subjects

only. Intuitively, this means that in CCS there is no sending/receiving of links but synchronization on them. (Notice that the ability of transmitting names is used for the encoding of recursion into replication in Definition 2.6.) More, precisely, the syntax of finite CCS processes is obtained by replacing the second line of Equation (1) with

$$\alpha \;::=\; \overline{x} \;\mid\; x \;\mid\; \tau \tag{8}$$

where τ represents a distinguished action; the *silent* action, with the decree that $\overline{\tau} = \tau$.

The (unlabelled) reduction relation $\longrightarrow$ for finite CCS processes can be obtained from that for the π-calculus given in the previous section. However, since α-conversion does not hold for one of the CCS variants we consider next, we find it convenient to define $\longrightarrow$ in terms of labelled reduction of CCS given in Table 2. A transition $P \xrightarrow{\alpha} Q$ says that P can perform an action α and evolve into Q. The reduction relation is then defined as $\longrightarrow \stackrel{\texttt{def}}{=} \xrightarrow{\tau}$.

$$\text{SUM}\ \frac{}{\sum_{i\in I}\alpha_i.P_i \xrightarrow{\alpha_j} P_j}\ \text{if } j \in I \qquad \text{RES}\ \frac{P \xrightarrow{\alpha} P'}{(\nu x)P \xrightarrow{\alpha} (\nu x)P'}\ \text{if } \alpha \notin \{x, \overline{x}\}$$

$$\text{PAR}_1\ \frac{P \xrightarrow{\alpha} P'}{P \mid Q \xrightarrow{\alpha} P' \mid Q} \qquad \text{PAR}_2\ \frac{Q \xrightarrow{\alpha} Q'}{P \mid Q \xrightarrow{\alpha} P \mid Q'} \qquad \text{COM}\ \frac{P \xrightarrow{\alpha} P' \;\; Q \xrightarrow{\overline{\alpha}} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q'}$$

$$\text{RED}\ \frac{P \xrightarrow{\tau} Q}{P \longrightarrow Q}$$

Table 2: An operational semantics for finite CCS.

3.2 Infinite CCS Processes

Both recursion and replication are found in the CCS literature in the forms we saw for the polyadic π-calculus. Nevertheless, as recursion in CCS comes in other forms. Some forms of recursion exhibit *dynamic* name scoping while others, as in the π-calculus, have *static* name scoping. By dynamic scoping we mean that, unlike the static case, the occurrence of a name can get dynamically (i.e., during execution) captured under a restriction. Surprisingly, this will have an impact on their relative expressiveness.

In the literature there are at least four alternatives to extend the above syntax to express infinite behavior. We describe them next.

CCS with Parametric Definitions: $\mathrm{CCS_p}$

The processes of $\mathrm{CCS_p}$ calculus are the finite CCS processes plus recursion using parametric definition exactly as in $\mathrm{p}\pi_{\mathrm{D}}$. So in particular we have the restriction on parametric definitions in Equation 3. The calculus is the variant in [14]. The rules for $\mathrm{CCS_p}$ are those in Table 2 plus the rule:

$$\text{CALL}\ \frac{P_A[y_1,\ldots,y_n/x_1,\ldots,x_n] \xrightarrow{\alpha} P'}{A(y_1,\ldots,y_n) \xrightarrow{\alpha} P'}\ \text{if } A(x_1,\ldots,x_n) \stackrel{\mathtt{def}}{=} P_A \tag{9}$$

As usual $P[y_1 \ldots y_n/x_1 \ldots x_n]$ results from syntactically replacing every free occurrence of x_i with y_i renaming bound names, i.e., *name α-conversion*, wherever needed to avoid capture. (Of course if $n = 0$, $P[y_1 \ldots y_n/x_1 \ldots x_n] = P$).

As shown in [14] in $\mathrm{CCS_p}$ we can identify process expression differing only by renaming of bound names; i.e., *name α-equivalence*—hence $(\nu x)P$ is the same as $(\nu y)P[y/x]$.

Constant Definitions: $\mathrm{CCS_k}$

We now consider the CCS alternative for infinite behavior given in [12]. We refer to identifiers with arity zero and their corresponding definitions as *constant* and *constant* (or *parameterless*) definitions, respectively. We omit the "( )" in $A(\,)$.

Given $A \stackrel{\mathtt{def}}{=} P$, requiring all names in $fn(P)$ to be formal parameters, as we did in $\mathrm{p}\pi_{\mathrm{D}}$ (Equation 3), would be too restrictive—P would not have visible actions. Consequently, let us drop the requirement to consider a fragment allowing *only* constant definitions but *with possible occurrence of free names in their bodies*. The rules for this fragments are those of $\mathrm{CCS_p}$. We shall refer to this fragment as $\mathrm{CCS_k}$. In this case Rule CALL, which for $\mathrm{CCS_k}$ we prefer to call CONS, takes the form

$$\text{CONS}\ \frac{P \xrightarrow{\alpha} P'}{A \xrightarrow{\alpha} P'}\ \text{if } A \stackrel{\mathtt{def}}{=} P \tag{10}$$

i.e., there is no α-conversion involved; thus allowing name captures. As illustrated in the next section, this causes scoping to be dynamic and α-equivalence not to hold. This is also the reason we cannot just take the reduction relation $\longrightarrow$ of the π-calculus restricted to $\mathrm{CCS_k}$ processes as such a relation assumes α-conversion due to the structural rule.

Recursion Expressions: CCS_μ

Hitherto we have seen process expressions whose recursive behavior is specified in an underlying set of definitions. It is often convenient, however, to have expressions which can specify recursive behavior on their own. Let us now extend the finite CCS processes to include such recursive expressions. The extended syntax is given by:

$$P, Q, \ldots := \ldots \mid X \mid \mu X.P \tag{11}$$

Here $\mu X.P$ binds the occurrences of the *process variable* X in P. As for bound and free names, the *bound variables* of P, $bv(P)$ are those with a bound occurrence in P, and the *free variables* of P, $fv(P)$ are those with a non-bound occurrence in P. An expression generated by the above syntax is said to be a *process (expression)* iff it is closed (i.e., it contains no free variables). The process $\mu X.P$ behaves as P with the free occurrences of X replaced by $\mu X.P$. Applying *variable* α-conversions wherever necessary to avoid captures. The semantics $\mu X.P$ is given by the rule:

$$\text{REC}\ \frac{P[\mu X.P/X] \xrightarrow{\alpha} P'}{\mu X.P \xrightarrow{\alpha} P'} \tag{12}$$

We call CCS_μ the resulting calculus. From [7] it follows that in CCS_μ we can identify processes up-to name α-equivalence.

Remark 3.1 (**Static and Dynamic Scope: Preservation of α-Equivalence).** An interesting issue of the substitution $[\mu X.P/X]$ applied to P is whether it *also requires* the renaming of *bound names* in P to avoid captures (i.e., *name α-conversion*). Such a requirement seems necessary should we want to identify process up-to *α-equivalence*. In fact, the requirement gives CCS_μ *static* scope of names. Let us illustrate this with an example.

Example 3.2. Consider $\mu X.P$ with $P = (x \mid (\nu x)(\bar{x}.t \mid X))$. First, let us assume we perform name α-conversions to avoid captures. So, $[\mu X.P/X]$ in P renames the bound X by a fresh name, say z, thus avoiding the capture of $P's$ free z in the replacement: I.e,

$$P[\mu X.P/X] = (x \mid (\nu z)(\bar{z}.t \mid \mu X.P)) = (x \mid (\nu z)(\bar{z}.t \mid \mu X.(x \mid (\nu x)(\bar{x}.t \mid X))))$$

The reader may care to verify (using the rules in Table 2 plus Rule REC) that t will not be performed; i.e., there is no $\mu X.P \xrightarrow{\alpha_1} P_1 \xrightarrow{\alpha_2} \ldots$ s.t. $\alpha_i = t$.

Now let us assume that the substitution makes no name α-conversion, thus causing a free occurrence of x in P, shown in a box below, to get bound, *dynamically in the scope* of the outermost restriction: I.e.,

$$P[\mu X.P/X] = (x \mid (\nu x)(\bar{x}.t \mid \mu X.P)) = (x \mid (\nu x)(\bar{x}.t \mid \mu X.(\boxed{x} \mid (\nu x)(\bar{x}.t \mid X)))).$$

The reader can verify that now t can eventually be performed. Such an execution of t cannot be performed by $\mu X.Q$ where Q is $(x \mid (\nu z)(\bar{z}.t \mid X))$ i.e, P with the binding and bound occurrence of x syntactically replaced with z. This shows that name α-*equivalence does not hold in this dynamic scope case*. □

It should be pointed out that using recursive expressions with no name α-conversion is in fact equivalent to using instead constant definitions as in the previous calculus CCS_{k}. In fact, in presenting CCS, [12] uses alternatively both kinds of constructions; using Rule REC, with no name α-conversion, for one and Rule CONS for the other. For example, by taking $A \stackrel{\text{def}}{=} P$ with P as in Example 3.2 one can verify that in CCS_{k}, A exhibits exactly the same dynamic scoping behavior illustrated in the above example. So, *name* α-*equivalence does not hold in CCS*. Notice that the above observations imply some semantics differences between CCS and the π-calculus. The former does not satisfy name α-equivalence because of the dynamic nature of name scoping—see Example 3.2. The latter uses static scoping and satisfies α-equivalence. □

Replication: $\text{CCS}_{!}$

The processes of $\text{CCS}_{!}$ are those finite CCS processes plus replication exactly as in $\text{p}\pi_{!}$. This variant is presented in [2]. In the context of CCS, this operators are studied in [2, 3, 9].

The operational rules for $\text{CCS}_{!}$ are those in Table 2 plus the following rule:

$$\text{REP}\ \frac{P \mid\ !P \stackrel{\alpha}{\longrightarrow} P'}{!P \stackrel{\alpha}{\longrightarrow} P'} \tag{13}$$

From [14] we know that in $\text{CCS}_{!}$ one can identify processes up to name α-equivalence.

3.3 Expressiveness Results for CCS

In this section we report results from [2, 3, 9] on the expressiveness for the CCS variants above.

The following theorem summarizes the expressiveness of the various calculi and it is an immediate consequence of the results in [2] and [9]. As for the π-calculus we compare expressiveness w.r.t. barbed congruence with the obvious restriction to CCS contexts (see Criteria 2.5).

Theorem 3.3. *The following holds for the CCS variants:*

1. *CCS_{k} is exactly as expressive as CCS_{p} w.r.t to barbed congruence.*
2. *CCS_{μ} is exactly as expressive as $CCS_{!}$ w.r.t to barbed congruence.*
3. *The divergence problem (i.e., whether a given process P has an infinite sequence of $\longrightarrow$ reductions) is undecidable for the calculi in (1) but decidable for those in (2).*

The results (1-3) are summarized in Figure 1. Let us now elaborate on the significance and implications of the above results. A noteworthy aspect of (1) is that any finite set of parametric (possibly mutually recursive) definitions can be replaced by a set, *finite* as well, of parameterless definitions . This arises as a result of the restricted nature of communication in CCS (e.g., absence of mobility). Related to this result is that of [12] which shows that, in the context of value-passing CCS, a parametric definition can be encoded using an set of constant definitions and infinite sums. However, this set is *infinite*.

Regarding (1) some readers may feel that given a process P with a parametric definition D, one could simply create as many constant definitions as permutations of possible parameters w.r.t. the finite set of names in P and D. This would not work for $\mathrm{CCS}_{\mathrm{p}}$; the unfolding of call to D within a restriction may need α-conversions to avoid name captures, thus generating new names (i.e., names not in P nor D) during execution.

Regarding (2), we wish to recall the encoding $[\![\cdot]\!]$ of CCS_{μ} into $\mathrm{CCS}_{!}$ which resembles that of Definition 2.6 in the context of the π-calculus.

Definition 3.4. *The encoding $[\![\cdot]\!]$ of CCS_{μ} processes into $CCS_{!}$ is homomorphic over all operators in the sub-calculus defining finite behavior and is otherwise defined as follows:*

$$\begin{array}{lcl} [\![X_i]\!] & = & \bar{x}_i \\ [\![\mu X_i.P]\!] & = & (\nu x_i)(!x_i.[\![P]\!] \mid \bar{x}_i) \end{array}$$

where the names x_i's are fresh.

The above encoding is correct w.r.t. barbed congruence, i.e., $[\![P]\!] \approx P$. It is important to notice that it would not be correct had we adopted dynamic scoping

in the Rule REC for CCS_k (see Remark 3.1). The $\mu X.P$ in Example 3.2 actually gives us a counter-example.

Another noteworthy aspect of the results mentioned above is the distinction between static and dynamic name scoping for the calculi under consideration. Static scoping renders the calculus with recursion decidable, *w.r.t. the divergence problem*, and no more expressive than the calculus with replication. In contrast, dynamic scoping renders the calculus with constant definitions undecidable and as expressive as that with parametric definitions. This is interesting since as discussed in Section 3.2 the difference between the calculi with static or dynamic scoping is very subtle. Using static scoping for recursive expressions was discussed in the context of ECCS [7], an extension of CCS whose ideas lead to the design of the π-calculus [14].

It should be noticed that preservation of divergence is not a requirement for equality of expressiveness w.r.t to barbed congruence since *barbed congruence does not preserve divergence*. Hence, although the results in [2] prove that divergence is decidable for $CCS_!$ (and undecidable for CCS_p), it does not follow directly from the arrows in Figure 1 that it is also decidable for CCS_μ. The decidability of the divergency problem for $CCS_!$ is proven in [9]

Finally, it is worth pointing out that, as exposed in [15], decidability of divergence does not imply lack of *Turing* expressiveness. In fact the authors in [3] show that $CCS_!$ is Turing-complete. They do this by showing how construct, given a two-counter machine, a process that can nondeterministically simulate such a machine. Two-counter machines are standard Turing-complete devices.

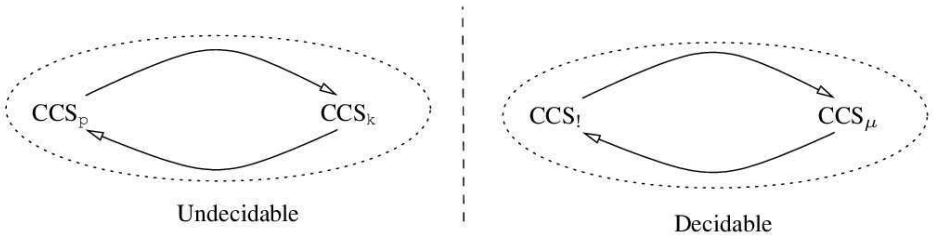

Figure 1: Classification of CCS variants. An arrow from X to Y indicates that for every P in Y one can construct a process $[\![P]\!]$ in X which is barbed congruent to P. (Un)decidability is meant w.r.t. the existence of divergent computations

4 The Mobile Ambients Calculus

The calculus of Mobile Ambients is a formalism for the description of distributed and mobile systems in terms of *ambients*; i.e. a named collection of active processes and nested sub-ambients.

The work in [4] studies the expressiveness of recursion versus replication in Mobile Ambients. In particular, the authors of [4] study the expressive power of ambient mobility in the (Pure) Mobile Ambients variants with replication and recursion.

4.1 Finite Processes of Ambients

The Pure Ambient Calculus focuses on ambient and processes interaction. Unlike the π-calculus, it abstracts away from process communication.

The syntax of the finite processes can be derived from those of the $\mathrm{p}\pi$-calculus by (1) introducing ambients, and the actions for ambient and processes interaction, (2) eliminating the action for process communication and (3) restricting summations to have arity at most one. In summary, we obtain the following syntax:

$$\begin{array}{rcl} P, Q, \ldots & := & 0 \mid \alpha.P \mid n[P] \mid (\nu x)P \mid P \mid Q \\ \alpha & := & in\ x \mid out\ x \mid open\ x \end{array} \tag{14}$$

The intuitive behaviour of the ambient $n[P]$ and α actions is better explained after presenting the reduction semantics of Ambients. The intuitive behaviour of the others constructs can be described exactly as in the π-calculus.

Reduction Semantics of Finite Processes. The *reduction* relation $\longrightarrow$ for Ambients can be obtained by adding the axiom $(\nu n)(m[P]) \equiv m[(\nu n)P]$ if $m \neq n$ to the structural congruence in Definition 2.2 and the following rules for ambients and process interaction to the rules of the $\mathrm{p}\pi$-calculus in Table 1:

1. $n[in\ m.P \mid Q] \mid m[R] \longrightarrow m[n[P \mid Q] \mid R]$
2. $m[n[out\ m.P \mid Q] | R] \longrightarrow n[P \mid Q] \mid m[R]$
3. $open\ n.P \mid n[Q] \longrightarrow P \mid Q$
4. $\dfrac{P \longrightarrow Q}{n[P] \longrightarrow n[Q]}$

Rules (1-3) describe ambients and their actions and Rule (4) simply says that reduction can occur underneath ambients. Rule (1) describes how, by using the *in* action, an ambient named n can enter another ambient named m. Similarly, Rule (2) describes how an ambient named n can exit another ambient named m by using the *out* action. Finally Rule (3) describes how a process can dissolve an ambient boundary to access its contents by performing the *open* action over the name n of the ambient.

4.2 Infinite Process of Ambients

Infinite behaviour in Ambients can be represented by using replication as in $\mathrm{p}\pi_!$ or recursive expressions of the form $\mu X.P$.

The $MA_!$ calculus

The calculus $MA_!$ extends the syntax of the finite Ambients processes with $!P$. Its reduction semantics $\longrightarrow$ is obtained by adding the structural axiom $!P \equiv P \mid !P$ to the structural axioms of finite Ambients processes.

The MA_r calculus

The calculus MA_r extends the syntax of the finite Ambients processes with recursive expression of the form $\mu X.P$ exactly as in CCS_μ (Section 3.2). Its reduction semantics $\longrightarrow$ is obtained by adding the structural axiom $\mu X.P \equiv P[\mu X.P/X]$ to the structural axioms of finite Ambients processes.

Notice that the issue of the substitution $[\mu X.P/X]$ applied to P we discussed in Section 3.2 arises again: Whether the substitution *also requires* the renaming of *bound names* in P to avoid captures (i.e., *name α-conversion*). Such a requirement seems necessary should we want to identify process up-to *α-equivalence*–which is included in the structural congruence $\equiv$ for Ambients. The CCS examples in Section 3.2 (see Remark 3.1) can easily be adapted here to illustrate that we obtain dynamic scoping of names if we do not perform the α-conversion in the substitution.

It should be noticed that the above has not been completely clarified in the literature of Ambients. In fact, it raises a technical issue in the results on expressiveness which we shall recall in the next section.

Expressiveness Results

To isolate the expressiveness of restriction and ambient actions in $MA_!$ and MA_r, [4] considers the following fragments of MA_c with $c \in \{!, r\}$: (1) $MA_c^{-\nu}$, the MA_c

calculus without the restriction constructor $(\nu x)P$, (2) MA_c^{-mv}, the MA_c calculus without the *in* and *out* actions, and finally (3) $MA_c^{-mv,\nu}$, the corresponding calculus with no *in/out* action nor restriction.

The separation results in [4] among the various calculi are given in terms of the decidability of *termination*; i.e., the problem of whether given a process P does not have any infinite sequence of reductions. Obviously, if the question is decidable in a given calculus then we know that there is no termination-preserving encoding of Turing Machines into the calculus. The results in [4] are summarized in Figure 2.

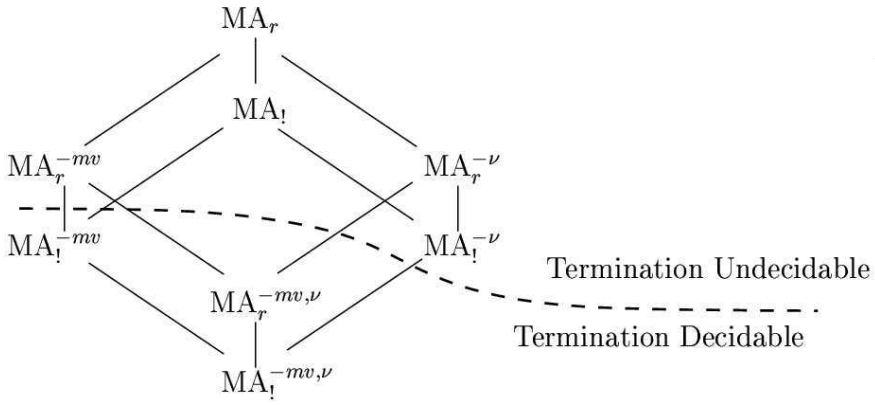

Figure 2: Hierarchy of Ambient Calculi.

Remark 4.1. The undecidability of process termination for MA_r^{-mv} is obtained by a reduction from termination of RAM machines, a Turing Equivalent formalism. First [4] uses a CCS fragment with recursion and *dynamic scope of names* to provide a termination-preserving encoding of RAMs. Then the CCS fragment is claimed to be a sub-calculus of MA_r^{-mv}. The undecidability of process termination for MA_r^{-mv} follows immediately.

Nevertheless, as illustrated in Section 3.2 Remark 3.1 such dynamic scope causes α-equivalence not to be preserved. In principle, this may cause a technical problem in the proof of the result since MA_r^{-mv} requires α-equivalence to be preserved; i.e., the CCS fragment used to simulate RAMs is not a sub-calculus of MA_r^{-mv}.

One way to deal with the above problem is to use a more involved notion of α-conversion in MA_r^{-mv} [5]. Another way would be to consider parametric recursion in MA_r, as in $\mathrm{CCS_p}$ or $\mathrm{p}\pi_{\mathrm{D}}$, and then use $\mathrm{CCS_p}$ as the sub-calculus of MA_r^{-mv} to encode RAMs. Nevertheless, either way we will be changing the original semantics of MA_r^{-mv} given in [11] which treats α-conversion and recursion as in CCS_μ [21].

5 Recursion vs Replication in Other Calculi

Here, we shall briefly survey work studying the relative expressive power of Recursion vs Replication in other process calculi.

In the context of calculi for security protocols, the work in [10] uses a process calculus to analyze the class of ping-pong protocols introduced by Dolev and Yao. The author show that all nontrivial properties, in particular reachability, become undecidable for a very simple recursive variant of the calculus. The recursive variant is capable of an implicit description of the active intruder, including full analysis and synthesis of messages . The authors then show that the variant with replication renders reachability decidable.

In the context of calculi for Timed Reactive System, the work in [17] studies the expressive power of some variants of Timed concurrent constraint programming (tcc). The tcc model is a process calculus introduced in [19] aimed at specifying timed systems, following the paradigms of Synchronous Languages [1]. The work states that: (1) recursive procedures with parameters can be encoded into parameterless recursive procedures with dynamic scoping, and vice-versa. (2) replication can be encoded into parameterless recursive procedures with static scoping, and vice-versa. (3) the languages from (1) are strictly more expressive than the languages from (2). Furthermore, it states that behavioral equivalence is undecidable for the languages from (1), but decidable for the languages from (2). The undecidability result holds even if the process variables take values from a fixed finite domain.

The reader may have noticed the strong resemblance of the work on tcc and that of CCS described in the previous section; e.g., static-dynamic scoping issue w.r.t recursion. In fact, [17] had a great influence in the work we described in this paper for CCS. In particular, in the discovery of the dynamic name scoping exhibited by the CCS presentation in [12].

6 Final Remarks

The expressiveness differences between recursion and replication we have surveyed in this paper may look surprising to those acquainted with the π-calculus where recursion is a derived operation. Our interpretation of this difference is that the link mobility of the π-calculus is a powerful mechanism which makes up for the weakness of replication.

The expressiveness of the replication $!P$ arises from unbounded parallel behaviour, which with recursion can be defined as $\mu X.(P \mid X)$. The additional expressive power of recursion arises from the unbounded nested scope of $\mu X.P$ as in $R = \mu X.(\nu x)(P \mid X)$ which behaves as $(\nu x)(P \mid (\nu x)(P \mid (\nu x)(P \mid \cdots)))$.

This, in general, cannot be simulated with replication. However, suppose that the unfolding of recursion applies α-conversion to avoid captures as we saw in Section 3.2. For example for the process R above we will have the unfolding $(\nu x_1)(P[x_1/x] \mid (\nu x_2)(P[x_2/x] \mid (\nu x_3)\cdots)))$ and each x_i will only occur in $P[x_i/x]$. It is easy to see the replication $!(\nu x)P$ captures the behaviour of R. Therefore, R does not really exhibit (significant) unbounded nesting of scope.

All in all, the ability of expressing recursive behaviours via replication in a given process calculus may depend on the mechanisms of the calculus to compensate for the restriction of replication as well as on how meaningful the unbounded nesting of the recursive expressions are.

References

[1] G. Berry and G. Gonthier. The ESTEREL synchronous programming language: design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, November 1992.

[2] N. Busi, M. Gabbrielli, and G. Zavattaro. Replication vs. recursive definitions in channel based calculi. In *ICALP: Annual International Colloquium on Automata, Languages and Programming*, 2003.

[3] N. Busi, M. Gabbrielli, and G. Zavattaro. Comparing recursion, replication, and iteration in process calculi. In *ICALP: Annual International Colloquium on Automata, Languages and Programming*, 2004.

[4] N. Busi and G. Zavattaro. On the expressive power of movement and restriction in pure mobile ambients. *Theoretical Computer Science*, 322(3):477–515, September 2004.

[5] N. Busi and G. Zavattaro. *Personal Communication*, May 2005.

[6] L. Cardelli and A. Gordon. Mobile Ambients. In M. Nivat, editor, *Proc. of Foundations of Software Science and Computation Structures (FoSSaCS), European Joint Conferences on Theory and Practice of Software (ETAPS'98)*, volume 1378 of *Lecture Notes in Computer Science*, pages 140–155, Lisbon, Portugal, 1998. Springer-Verlag, Berlin.

[7] U. Engberg and M. Nielsen. A calculus of communicating systems with label-passing. Technical report, University of Aarhus, 1986.

[8] Jean-Yves Girard. Linear logic. *Theor. Comput. Sci.*, 50:1–102, 1987.

[9] P. Giambiagi, G. Schneider, and F. Valencia. On the expressiveness of infinite behavior and name scoping in process calculi. In *FoSSaCS*, pages 226–240, 2004.

[10] H. Huttel and J. Srba. Recursion vs. replication in simple cryptographic protocols. In *Proceedings of the 31st Annual Conference on Current Trends in*

Theory and Practice of Informatics (SOFSEM'05), volume 3381 of *LNCS*, pages 175–184. Springer-Verlag, 2005.

[11] Francesca Levi and Davide Sangiorgi. Mobile safe ambients. *ACM Transactions on Programming Languages and Systems*, 25(1):1–69, January 2003.

[12] R. Milner. *Communication and Concurrency*. International Series in Computer Science. Prentice Hall, 1989. SU Fisher Research 511/24.

[13] R. Milner. The polyadic π-calculus: A tutorial. In F. L Bauer, W. Brauer, and H. Schwichtenberg, editors, *Logic and Algebra of Specification*, pages 203–246. Springer-Verlag, Berlin, 1993.

[14] R. Milner. *Communicating and Mobile Systems: the π-calculus*. Cambridge University Press, 1999.

[15] S. Maffeis and I. Phillips. On the computational strength of pure ambient calculi. In *EXPRESS'03*, 2003.

[16] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, Part I + II. *Information and Computation*, 100(1):1–77, 1992.

[17] M. Nielsen, C. Palamidessi, and F. Valencia. On the expressive power of concurrent constraint programming languages. In *Proc. of the 4th International Conference on Principles and Practice of Declarative Programming (PPDP 2002)*, pages 156–167. ACM Press, October 2002.

[18] C. Palamidessi. Comparing the expressive power of the synchronous and the asynchronous pi-calculus. In ACM Press, editor, *POPL'97*, pages 256–265, 1997.

[19] V. Saraswat, R. Jagadeesan, and V. Gupta. Foundations of timed concurrent constraint programming. In *Proc. of the Ninth Annual IEEE Symposium on Logic in Computer Science*, pages 71–80, 4–7 July 1994.

[20] D. Sangiorgi and D. Walker. *The $\pi-$calculus: A Theory of Mobile Processes*. Cambridge University Press, 2001.

[21] D. Sangiorgi. *Personal Communication*, May 2005.